Capítulo 19 — Fechas: de la etiqueta del calendario al número de día

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en variables y tipos (`02`), operadores (`03`), `if` y bucles (`04`), funciones (`05`), arreglos y cadenas (`06`), punteros (`07`), `struct` (`08`), tipos como `size_t` y `time_t` (`16`), entrada/salida (`14`), archivos (`15`) y tu librería de cadenas (`18`).*
>
> *Acá vas a dejar de pelear con un calendario lleno de reglas raras (meses de distinto largo, años bisiestos, husos horarios) y vas a **construir tu propia librería de fechas**: un conjunto de funciones que validan, comparan, suman y formatean fechas **convirtiéndolas primero a un simple número de días**. Al final vas a poder imprimir un calendario entero, calcular vencimientos y entender por qué existe el "efecto 2038".*

Cómo leer este capítulo

Este es el capítulo donde "sé usar `time()`" se convierte en "entiendo **qué** es una fecha, **por qué** es tan incómoda, y cómo domarla". No es teoría: cada idea viene con **código para mirar**, y cada bloque cierra con **práctica** para que la ejercites 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 calendario es una convención	Una fecha es una **etiqueta humana**, no un número: tiene reglas raras (meses de distinto largo y años bisiestos).
2	El paradigma: linealizar	Convertí la fecha a **un solo número de día** y toda la aritmética se vuelve trivial.
3	Aritmética de fechas	Sumar días, restar días, comparar y ubicar el día de la semana: todo con el número de día.
4	El reloj del sistema	`time_t`, `struct tm`, `mktime` y por qué tu computadora **no sabe** en qué huso horario estás hasta que se lo decís.
5	Formatear y parsear	Mostrar una fecha como texto (a tu búfer o **prestada**) y volver del texto a la fecha (`strftime`, `sscanf`, `strptime`).
6	El catálogo completo + jefe	`time_t`, UTC vs local, `difftime`... y un impresor de calendarios para cerrar.

Los **ejercicios** se numeran en orden: `19.1`, `19.2`, `19.3`, 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.*

Tu librería objetivo

Antes de arrancar, mirá el plano de la casa que vas a construir. Estas son las funciones que vas a tener escritas **vos** al terminar el capítulo. Acá van **solo las declaraciones** (el *contrato*: qué recibe y qué devuelve cada una), no el cuerpo. Es tu mapa de ruta.

```c
/* ==== date.h — el contrato de tu librería (solo prototipos) ==== */
#include <stddef.h>   /* size_t */
#include <time.h>     /* time_t, struct tm */

typedef struct {
    int year;    /* año completo, ej. 2024            */
    int month;   /* 1 = enero ... 12 = diciembre      (1-based) */
    int day;     /* 1 = primero ... 31                (1-based) */
} Date;

/* --- Calendario --- */
int  date_is_leap(int year);                      /* ¿año bisiesto? */
int  date_days_in_month(int year, int month);     /* 28, 29, 30 o 31 */
int  date_is_valid(Date d);                       /* ¿existe esa fecha? */

/* --- Linealización (el paradigma) --- */
long date_to_day_number(Date d);                  /* fecha → número de día */
Date date_from_day_number(long n);                /* número de día → fecha */

/* --- Aritmética --- */
Date date_add_days(Date d, long n);               /* fecha + n días */
long date_diff_days(Date a, Date b);              /* días entre a y b */
int  date_compare(Date a, Date b);                /* -1, 0 o 1 */
int  date_day_of_week(Date d);                    /* 0 = domingo ... 6 = sábado */
int  date_day_of_year(Date d);                    /* 1 = 1 de enero ... 365/366 */

/* --- Reloj del sistema --- */
time_t date_to_epoch(Date d, int hour, int min, int sec);
Date   date_today(void);
void   date_now_local(struct tm *out);            /* "ahora" en tu huso */

/* --- Formato y parseo --- */
size_t      date_format(Date d, const char *fmt, char *out, size_t cap);  /* a TU búfer
*/
const char *date_to_string(Date d, const char *fmt);   /* búfer interno PRESTADO (¡ojo!)
*/
int         date_parse(const char *s, const char *fmt, Date *out);        /* 1 = ok */
```

> ***Fijate en tres decisiones de diseño.*** *(1) El año, el mes y el día van* ***1-based*** *(enero es* `1`*, no* `0`*): es lo natural para un humano y evita la trampa de* `struct tm`*. (2) Hay* ***dos mundos****: tu* `Date` *(para lógica de calendario pura, sin husos horarios) y* `time_t` */* `struct tm` *(para hablar con el reloj del sistema). Los vas a conectar en la Parte 4. Mantenerlos separados es lo que te va a salvar de los bugs de huso horario. (3) El formato tiene* ***dos contratos****:* `date_format` *escribe en* ***tu*** *búfer (seguro, tuyo);* `date_to_string` *te* ***presta*** *un búfer interno (cómodo, pero se pisa en la próxima llamada). Vas a ver por qué en la Parte 5.*

PARTE 1 — El calendario es una convención

Antes de escribir una línea, hay que ver **por qué** las fechas son tan molestas. No es casualidad: es el resultado de siglos de humanos tratando de hacer coincidir su calendario con el cielo. Si entendés eso, todas las reglas raras dejan de ser magia.

1.1 Una fecha no es un número: es una etiqueta

Pensá en el **calendario de la pared de la cocina**. Cuando decís "14 de octubre de 2026", no estás diciendo un número: estás leyendo una **etiqueta** en una grilla. Y hay una trampa enorme escondida: **esas etiquetas no se pueden sumar directamente**.

- `2026 + 1` **no** es "el año que viene": es… `2027`, que casualmente *podría* serlo, pero solo porque el año coincide con la etiqueta del año.
- `14 + 1` **no** es "mañana": es `15`, y `15` de octubre es mañana, pero solo porque el mes tiene más de 14 días.
- `10 + 3` (mes + 3) **no** es "tres meses después": `10 + 3 = 13`, y no existe el mes 13.

Moraleja: una fecha es una **etiqueta de tres piezas** (año, mes, día) que vive en una estructura irregular. Para operar con fechas vas a necesitar **otra representación**. Eso es la Parte 2. Primero, veamos por qué el calendario es tan irregular.

1.2 El calendario gregoriano: un edificio con cuartos de distinto tamaño

El calendario que usás todos los días se llama **gregoriano** (por el papa Gregorio XIII, que lo implantó en 1582). Es un edificio con **doce cuartos**, pero **no todos del mismo tamaño**:

Mes	Días	Mes	Días
enero	31	julio	31
febrero	**28 o 29**	agosto	31
marzo	31	septiembre	30
abril	30	octubre	31
mayo	31	noviembre	30
junio	30	diciembre	31

Mirá el desorden: meses de 31, meses de 30, y **febrero**, el mes raro que cambia de tamaño cada tanto. No hay ninguna fórmula elegante detrás: es **historia**. Los romanos tenían otro calendario, y cuando agregaron meses y movieron días, quedó este enredo. Y lo vas a tener que codificar tal cual.

> ***El truco para no memorizar.*** *La tabla es la ley, pero hay un patrón nemotécnico:* ***31 - 30 - 31 - 30****, salvo el último par (julio y agosto, ambos de 31, por vanidad de dos emperadores). Y febrero, aparte, con su regla propia (1.3). Guardá la* ***tabla****; la nemotecnia es solo para humanos.*

1.3 El año bisiesto: por qué febrero a veces tiene 29

Acá está la parte que a todo el mundo le enseñaron mal. La regla **completa** es:

> *Un año es* ***bisiesto*** *si es divisible por* ***4****,* ***excepto*** *si es divisible por* ***100****,* ***salvo*** *que sea divisible por* ***400****.*

O sea: **2024** es bisiesto (divisible por 4). **1900 no** es bisiesto (divisible por 100, y no por 400). **2000 sí** es bisiesto (divisible por 400). Febrero de un año bisiesto tiene **29** días; en el resto, **28**.

¿Por qué semejante regla? Porque el **año solar** —el tiempo real que tarda la Tierra en dar una vuelta al Sol— dura unos **365,2422 días**, no 365 exactos:

```
año solar ≈ 365,2422 días
calendario simple = 365 días     → se atrasa 0,2422 días por año
0,2422 × 4 ≈ 0,97 días           → cada 4 años, casi un día de atraso
→ agregamos 1 día cada 4 años    (el "bisiesto")
pero ese "casi" deja un sobrante → corregimos con las reglas del 100 y del 400
```

Cada regla **corrige** el error de la anterior: el 4 agrega días de más, el 100 los quita, el 400 los vuelve a poner. Es una **cascada de correcciones** para que el calendario no se desincronice de las estaciones (si no, en unos siglos la Navidad caería en verano).

> ***Por qué te importa de verdad.*** `date_days_in_month` *(y medio mundo de la aritmética de fechas) dependen de acertar el bisiesto. Y acertarlo* ***mal*** *es el bug de fechas más famoso: hay sistemas reales que asumieron "cada 4 años" y se rompieron el* ***29 de febrero de 1900*** *o el de* ***2100*** *(fines de siglo que* ***no*** *son bisiestos). La regla de los 400 no es un detalle: es la que casi todos olvidan.*

> ***Nota histórica fina.*** *El gregoriano no rige desde siempre: se adoptó en 1582. Las fechas* ***anteriores*** *a esa época, en el calendario gregoriano, se llaman* ***prolépticas*** *(o sea: "si el calendario hubiera existido"). Para este capítulo uses fechas modernas, la distinción no te cambia nada; pero es bueno saber que "año 0" o "año 1" son convenciones, no hechos.*

1.4 Validar: no toda etiqueta existe

Con las reglas de arriba ya podés responder la pregunta más básica: **¿esta fecha existe?** `2024-02-29` sí; `2023-02-29` no; `2024-04-31` tampoco (abril tiene 30); `2024-13-01` no (no hay mes 13); `2024-00-15` no (los meses van del 1 al 12). Validar **antes** de calcular es la primera defensa contra bugs raros.

```
¿existe?
2024-02-29  →  sí   (2024 es bisiesto)
2023-02-29  →  no   (2023 no es bisiesto → febrero tiene 28)
1900-02-29  →  no   (1900 divisible por 100 y no por 400)
2000-02-29  →  sí   (2000 divisible por 400)
2024-04-31  →  no   (abril tiene 30)
2024-12-31  →  sí
```

Práctica de la Parte 1

Ejercicio 19.1 — El detector de bisiestos (`date_is_leap`)

Conceptos · operadores `%` , `&&` , `||` , reglas de divisibilidad, `int`

Contexto · Medio calendario depende de saber si un año es bisiesto. Simplificación: una función que responde sí/no a esa pregunta, con la regla **completa** (no la de "cada 4 años").

Consigna · Implementá `date_is_leap(int year)` : devuelve **distinto de cero** si `year` es bisiesto y `0` si no. Usá la regla de los 4 / 100 / 400. Comprobá con los **casos de fin de siglo**, que son los que separan una solución correcta de una que "casi" anda.

Entrada · Un año (`int`). **Salida** · "Bisiesto" o "No bisiesto".

Escenario de prueba

```
2024  → Bisiesto
2023  → No bisiesto
1900  → No bisiesto      (divisible por 100, no por 400)
2000  → Bisiesto         (divisible por 400)
2100  → No bisiesto
1600  → Bisiesto
```

Identificadores sugeridos · `date_is_leap` , `year` **Criterios de aceptación**

- [ ] Se usa `%` para las tres divisibilidades.
- [ ] `1900` da **no** y `2000` da **sí** (la parte que casi todos olvidan).
- [ ] `2024` da **sí** y `2023` da **no**.
- [ ] El retorno es "verdadero/falso" al estilo C (distinto de cero / cero).

Ejercicio 19.2 — ¿Cuántos días tiene el mes? (`date_days_in_month`)

Conceptos · tabla de largos de mes, reusar `date_is_leap` , `switch` o arreglo

Contexto · Para validar fechas y para armar calendarios necesitás el largo de cada mes, que depende de febrero y del bisiesto. Simplificación: una función que devuelve 28, 29, 30 o 31.

Consigna · Implementá `date_days_in_month(int year, int month)` : devuelve la cantidad de días de `month` en `year` . Los meses de 31 y de 30 son fijos (tabla); febrero es `29` si `date_is_leap(year)` y `28` si no. Podés usar un `switch` , un arreglo, o la lógica que prefieras. Decidí y documentá qué devolvés si `month` está fuera de `1..12` .

Entrada · Un año y un mes. **Salida** · La cantidad de días.

Escenario de prueba

```
2024, 2   → 29
2023, 2   → 28
1900, 2   → 28
2000, 2   → 29
2024, 4   → 30
2024, 1   → 31
2024, 12  → 31
```

Identificadores sugeridos · `date_days_in_month` , `year` , `month` , `days`
Criterios de aceptación

- ☐ Los meses de 31 y 30 dan lo correcto (no memorizados a mano en cada `case` sin orden).
- ☐ Febrero reusa `date_is_leap` (no repite la regla).
- ☐ Los cuatro casos de febrero del escenario dan lo esperado.
- ☐ El caso `month` fuera de rango está decidido y documentado.

Ejercicio 19.3 — El validador de fechas (`date_is_valid`)

Conceptos · `struct` , validación por rangos, reusar `date_days_in_month`

Contexto · Antes de confiar en una fecha leída del usuario o de un archivo, hay que verificar que exista. Simplificación: una función que dice si una `Date` es real.

Consigna · Definí la estructura `Date` (con `year` , `month` , `day`). Implementá `date_is_valid(Date d)` : devuelve "verdadero" solo si

1. el `month` está entre `1` y `12` ;
2. el `day` está entre `1` y `date_days_in_month(d.year, d.month)` .

Comprobá con varios casos, incluidos los que **parecen** válidos pero no lo son.

Entrada · Una `Date` (posiblemente inválida). **Salida** · "Válida" o "Inválida".

Escenario de prueba

```
2024-02-29  → Válida
2023-02-29  → Inválida     (febrero de 2023 tiene 28)
2024-04-31  → Inválida     (abril tiene 30)
2024-13-01  → Inválida     (no hay mes 13)
2024-00-10  → Inválida     (los meses empiezan en 1)
2024-06-00  → Inválida     (los días empiezan en 1)
2024-12-31  → Válida
```

Identificadores sugeridos · `Date` , `date_is_valid` , `d` **Criterios de aceptación**

- [ ] Se valida el **rango del mes** antes de usar el día.
- [ ] El largo del mes sale de `date_days_in_month` (no está hardcodeado).
- [ ] Los siete casos del escenario dan lo esperado.
- [ ] No se accede a datos de la fecha sin haberla validado antes de usarla.

PARTE 2 — El paradigma: linealizar la fecha

Este es el corazón del capítulo, y la idea que te va a volar la cabeza. En el capítulo `18` cambiaste "caminar hasta el `'\0'` " por "guardar el largo". Acá vas a cambiar "operar con tres piezas irregulares" por **"operar con un solo número"**.

2.1 El problema: restar dos fechas es un infierno

Imaginate que querés saber **cuántos días pasaron** entre `2024-02-27` y `2024-03-02`. Con las etiquetas no podés hacer `02 - 27` (da negativo y sin sentido) ni `03 - 02` (ignora los meses). Tenés que: restar partes, ajustar por fin de mes, sumar los días de los meses del medio, y acordarte del bisiesto. Un festival de casos borde.

```
2024-02-27  →  2024-03-02
¿cuántos días?

feb 2024 es bisiesto → tiene 29 días
27 → 29 :    2 días
29 → 01 mar: 1 día (o al revés: 29 feb + 1 = 1 mar)
01 → 02 mar: 1 día
total:       4 días        ← ¡y esto fue un caso FÁCIL!
```

Ahora imaginate hacer lo mismo con **años** de por medio, o comparar dos fechas, o calcular el día de la semana. Cada operación es una receta distinta, llena de reglas. Es insoportable.

2.2 La solución: un número de día

Acá está el truco. Elegí un **día cero** (un *epoch*, un origen) y **contá los días** desde ahí. Cada fecha pasa a ser **un solo número entero**:

```
elijo origen: 1970-01-01  →  día 0

1969-12-31  →  -1
1970-01-01  →   0
1970-01-02  →   1
2000-01-01  →   10957
2024-01-01  →   19723
```

Y ahora, mágicamente, **toda la aritmética se vuelve de primaria**:

```
días entre a y b        =  b Día − a Día          (una resta)
fecha + n días          =  fecha Día + n          (una suma)
comparar dos fechas     =  comparar dos números   (un <)
día de la semana        =  (Día + constante) % 7  (un módulo)
```

Ese número se llama **número de día serial** (o *Rata Die*, o *Julian Day Number*, según el origen que elijas). El nombre importa poco: lo que importa es que **una fecha irregular se convierte en un punto en una recta**, y en la recta todo es fácil.

> ***Pensalo como el "kilómetro 0".*** *Es como cuando medís distancias por ruta: no decís "entre la ciudad A y la B hay 3 provincias y 2 peajes", decís* ***"hay 250 km"****. El número de día es el kilometraje del calendario. Y elegir el origen (1970, el año 1, lo que sea) es elegir dónde ponés el mojón 0: no cambia las distancias, solo la numeración.*

> ***El paralelo con el capítulo `18`.*** *Allá, "caminar la calle hasta el `'\0'`" era lento y frágil; guardar el `count` lo resolvía. Acá, "operar con tres piezas irregulares" es frágil;* ***linealizar a un número de día*** *lo resuelve. Misma filosofía:* ***cambiá la representación para que la operación sea trivial****.*

2.3 Convertir ida y vuelta

Necesitás **dos** funciones: una que lleve fecha → número, y otra que vuelva número → fecha.

De fecha a número (la ida) es "contar". La forma directa: contar los días de **todos los años completos** anteriores, más los días de los **meses completos** del año actual, más los días del mes que ya pasaron.

```
número_de_día(2024-03-02)
= (días de 1970..2023 completos)          ← años enteros
+ (días de ene 2024 + días de feb 2024)   ← meses enteros (¡feb respeta bisiesto!)
+ (2 - 1)                                  ← días ya pasados de marzo
```

Cada año aporta **365** o **366** (según bisiesto); cada mes aporta lo suyo. Es una suma ordenada.

De número a fecha (la vuelta) es "descontar". Vas restando bloques: primero grandes (siglos, y mejor todavía: bloques de 400 años, que tienen una cantidad **fija** de días), después años, después meses, y lo que sobra es el día del mes.

> ***El detalle que hace la diferencia: el bloque de 400 años.*** *El calendario gregoriano se* ***repite*** *cada 400 años:* ***400 años = 146097 días*** *exactos, con 97 bisiestos. Ese bloque fijo es la clave para hacer la vuelta sin adivinar. (Por eso el 400 de la regla del bisiesto no es casualidad: es lo que hace que todo cierre cada 400 años.)*

> ***No hace falta que memorices la fórmula.*** *En este capítulo el* ***algoritmo es tuyo****: podés contar con bucles simples (año a año, mes a mes). Va a ser más lento que la fórmula cerrada, pero es* ***correcto y transparente****, que para aprender es lo que importa. Si después querés la versión rápida, la vas a poder derivar vos.* ***La clave es que la ida y la vuelta sean inversas****: si* `date_from_day_number(date_to_day_number(d))` *te devuelve* `d`*, ganaste.*

Práctica de la Parte 2

Ejercicio 19.4 — Linealizar (`date_to_day_number`)

Conceptos · contar años/meses/días, reusar `date_is_leap` y `date_days_in_month`, `long`

Contexto · Es la operación que habilita toda la aritmética posterior. Simplificación: convertir una fecha a la cantidad de días transcurridos desde el **1 de enero de 1970**.

Consigna · Implementá `date_to_day_number(Date d)`, usando como origen el **1970-01-01 = día 0**. Contá los días de los años **completos** anteriores a `d.year`, más los días de los meses completos de `d.year` antes de `d.month`, más `d.day - 1`. El retorno es `long` (puede ser negativo para fechas anteriores al origen).

Entrada · Una `Date` válida. **Salida** · El número de día serial.

Escenario de prueba

```
1970-01-01  →  0
1970-01-02  →  1
1969-12-31  → -1
2000-01-01  → 10957
2000-03-01  → 11017
2024-01-01  → 19723
```

> ***Cómo comprobar 2000-01-01 = 10957:*** *son 30 años desde 1970 hasta 2000, con los bisiestos* `1972, 1976, …, 1996` *(7 en total).* `30 × 365 + 7 = 10957`*. Y* `2000-03-01` *agrega enero (31) + febrero bisiesto (29) = 60 →* `10957 + 60 = 11017`*.*

Identificadores sugeridos · `date_to_day_number` , `d` , `days` , `y` , `m` **Criterios de aceptación**

- [ ] Se usa `long` (o más ancho) para el resultado (puede ser negativo).
- [ ] Cada año suma 365 o 366 según `date_is_leap` .
- [ ] Cada mes suma lo que dice `date_days_in_month` (febrero respeta el bisiesto).
- [ ] Los seis valores del escenario salen exactos, incluido el negativo.

Ejercicio 19.5 — Volver atrás (`date_from_day_number`)

Conceptos · descontar bloques, bucles, condición de corte, ida y vuelta

Contexto · Para mostrar una fecha ("dentro de 45 días") o para sumar/restar, hay que poder recorrer el camino inverso. Simplificación: la operación recíproca de 19.4.

Consigna · Implementá `date_from_day_number(long n)` (mismo origen: `0` = 1970-01-01). Descontá años (365 o 366 según bisiesto) hasta que no quepa uno más; después descontá meses enteros; lo que sobre, `+1` , es el día del mes. Comprobá que es **inversa** de 19.4: para varias fechas, `date_from_day_number(date_to_day_number(d))` devuelve exactamente `d` .

Entrada · Un número de día (puede ser negativo). **Salida** · La `Date` correspondiente.

Escenario de prueba

```
0       → 1970-01-01
1       → 1970-01-02
-1      → 1969-12-31
10957   → 2000-01-01
11017   → 2000-03-01
19723   → 2024-01-01
20740   → 2026-10-14
```

> ***Prueba de ida y vuelta (la que importa):*** *para cada fecha del escenario de 19.4,* `date_from_day_number(date_to_day_number(d)) == d` *. Si eso se cumple en muchos casos (incluidos fines de mes y fines de año), tus dos funciones están bien.*

Identificadores sugeridos · `date_from_day_number` , `n` , `year` , `month` **Criterios de aceptación**

- [ ] Maneja números **negativos** (fechas anteriores a 1970).
- [ ] Es **inversa** de `date_to_day_number` en los casos probados.
- [ ] Los fines de mes y de año se calculan bien (no hay "mes 13" ni "día 32").
- [ ] Devolvió una `Date` **válida** en todos los casos (`date_is_valid` da verdadero).

PARTE 3 — Aritmética de fechas

Con el número de día, esta parte es casi una formalidad: cada operación es **dos lines** de código. Pero son las funciones que más vas a usar.

3.1 Sumar y restar días

"Vence en 30 días", "hace una semana", "3 días hábiles después". Todo eso es:

```
fecha + n días   =   date_from_day_number( date_to_day_number(fecha) + n )
```

Nada más. El número de día hace que "cruzar un fin de mes", "cruzar un año bisiesto", "cruzar un fin de año" sean **exactamente el mismo caso**: sumar `n` . Los casos borde que antes eran un infierno, ahora **no existen**.

3.2 Diferencia en días

Cuántos días entre dos fechas = **restar los números de día**. El orden importa: `b - a` da cuántos días hay que sumarle a `a` para llegar a `b` . Si `a` es posterior, da negativo.

```
diff(2024-01-01, 2024-03-01) = 11017 − ... = 60 días
diff(2024-03-01, 2024-01-01) = -60 días
```

> ***Decidí el signo explícitamente.*** *¿Querés el resultado* ***con signo*** *(te dice cuál es anterior) o siempre* ***positivo*** *(la distancia "a secas")? Los dos son válidos; lo que* ***no*** *es válido es no saber qué devolvés. Documentalo.*

3.3 Comparar fechas

Comparar `a < b` es comparar sus números de día. Devolvés `-1`, `0` o `1` (el mismo contrato que `strcmp`, capítulo `06`, para que sea familiar).

3.4 Día de la semana y día del año

Estos dos salen **gratis** del número de día:

- **Día de la semana:** como sabés qué día fue el origen (1970-01-01 fue **jueves**), el resto de dividir por 7 te da el día. En la convención de C (`struct tm`), **domingo = 0**. Entonces

  ```
  día_semana = (número_de_día + 4) mod 7      (el +4 alinea el jueves con el 4)
  ```

  Cuidado con el `mod` de números **negativos** en C: `-1 % 7` da `-1`, no `6`. Para que dé siempre `0..6`, la fórmula segura es `((x % 7) + 7) % 7`.

- **Día del año:** cuántos días pasaron desde el 1 de enero (1 = 1 de enero, 366 = 31 de diciembre en bisiesto). Se calcula sumando los meses anteriores + el día, o bien con la diferencia de números de día contra el 1 de enero del mismo año.

> ***El `mod` negativo: la trampa N.º 1 de esta parte.*** *En C, `%` con un operando negativo puede dar negativo. Como los números de día* ***sí*** *pueden ser negativos (fechas antes de 1970), cualquier `% 7` sin blindar te puede devolver un día de semana inválido (como `-3`). Blindalo siempre con `((x % 7) + 7) % 7`.*

Práctica de la Parte 3

Ejercicio 19.6 — Sumar días (`date_add_days`)

Conceptos · composición `from(to(d) + n)` , cruce de fin de mes/año, `long`

Contexto · Calcular vencimientos, plazos y fechas relativas es sumar días. Simplificación: una función que suma (o resta, con `n` negativo) días a una fecha.

Consigna · Implementá `date_add_days(Date d, long n)` usando **solo** `date_to_day_number` y `date_from_day_number` (no reimplementes los saltos de mes). Comprobá los cruces difíciles: fin de febrero en bisiesto, cambio de año, y `n = 0` .

Entrada · Una `Date` y un `n` (puede ser negativo). **Salida** · La `Date` resultante.

Escenario de prueba

```
2024-02-28 + 2  → 2024-03-01      (2024 es bisiesto: el 29 existe)
2023-02-28 + 1  → 2023-03-01      (2023 no: salta directo a marzo)
2024-12-31 + 1  → 2025-01-01
2024-01-01 - 1  → 2023-12-31
2024-06-15 + 0  → 2024-06-15
2024-01-31 + 1  → 2024-02-01      (cruzar el fin de un mes de 31)
```

Identificadores sugeridos · `date_add_days` , `d` , `n` **Criterios de aceptación**

- ☐ Se implementa con `to_day_number` + `from_day_number` (no con bucles de mes propios).
- ☐ Los seis casos del escenario (incluidos los bisiestos) dan lo esperado.
- ☐ `n` negativo resta correctamente.
- ☐ El resultado siempre pasa `date_is_valid` .

Ejercicio 19.7 — Diferencia en días (`date_diff_days`)

Conceptos · resta de números de día, signo, `long`

Contexto · "¿Cuántos días faltan?", "¿cuánto tiempo estuvo abierto el ticket?" Simplificación: la distancia en días entre dos fechas.

Consigna · Implementá `date_diff_days(Date a, Date b)` que devuelva `b - a` **en días** (positivo si `b` es posterior a `a`, negativo si es anterior, `0` si son el mismo día). Documentá el signo.

Entrada · Dos `Date`. **Salida** · La diferencia en días (con signo).

Escenario de prueba

```
a = 2024-01-01, b = 2024-03-01  →  60
a = 2024-03-01, b = 2024-01-01  → -60
a = 2024-02-28, b = 2024-03-01  →   2      (2024 bisiesto)
a = 2024-01-01, b = 2024-01-01  →   0
a = 2000-01-01, b = 2024-01-01  → 8766
```

Identificadores sugeridos · `date_diff_days`, `a`, `b` **Criterios de aceptación**

- [ ] Es una simple resta de los números de día.
- [ ] El signo es el documentado (positivo cuando `b` es posterior).
- [ ] `diff(a, a) == 0`.
- [ ] Los cinco valores del escenario dan exactos.

Ejercicio 19.8 — Comparar (`date_compare`)

Conceptos · contrato `-1/0/1`, comparación de números, orden

Contexto · Ordenar una lista de turnos, facturas o eventos necesita comparar fechas. Simplificación: una función de comparación al estilo `strcmp`.

Consigna · Implementá `date_compare(Date a, Date b)`: devuelve `-1` si `a` es anterior a `b`, `0` si son iguales, y `1` si `a` es posterior. Basate en los números de día (no compares año/mes/día por separado; ya tenés el número). Después, usala para **ordenar** un pequeño arreglo de fechas (un ordenamiento por burbuja sencillo alcanza).

Entrada · Dos `Date` (y, para la demostración, un arreglo). **Salida** · El veredicto de la comparación y el arreglo ordenado.

Escenario de prueba

```
compare(2024-01-01, 2024-03-01)  → -1
compare(2024-03-01, 2024-01-01)  →  1
compare(2024-02-29, 2024-02-29)  →  0

arreglo:  [2024-03-01, 2020-12-31, 2024-01-01, 2023-06-15]
ordenado: [2020-12-31, 2023-06-15, 2024-01-01, 2024-03-01]
```

Identificadores sugeridos · `date_compare` , `a` , `b` , `dates` **Criterios de aceptación**

- [ ] Devuelve exactamente `-1` , `0` o `1` .
- [ ] Se apoya en el número de día (no en comparaciones encadenadas a mano).
- [ ] El arreglo queda ordenado de menor a mayor.
- [ ] Fechas del mismo año y mes comparan bien por día.

Ejercicio 19.9 — Día de la semana (`date_day_of_week`)

Conceptos · módulo, `mod` de negativos blindado, convención domingo = 0

Contexto · Saber en qué día de la semana cae una fecha es la base de cualquier calendario o agenda. Simplificación: devolver el día de la semana (0 = domingo … 6 = sábado), como `struct tm` .

Consigna · Implementá `date_day_of_week(Date d)` : usá el número de día y la constante que alinea `1970-01-01` (que fue **jueves**, o sea `4`) con la convención domingo = 0. **Blindá** el módulo para valores negativos (fechas antes de 1970). Mostrá el resultado con **nombre** (por ejemplo, un arreglo de nombres de días), no solo el número.

Entrada · Una `Date` . **Salida** · El nombre del día de la semana.

Escenario de prueba

```
1970-01-01  → jueves      (0)
2000-01-01  → sábado      (10957)
2024-01-01  → lunes       (19723)
2024-02-29  → jueves
1969-12-31  → miércoles   (número -1: probá que el módulo NO dé un índice inválido)
```

> ***Comprobación de la constante.*** `1970-01-01` *fue* ***jueves****. Con domingo = 0, jueves =* ***4****. Por eso la fórmula es* `(día + 4) mod 7`*. Si te da otro día, revisá el origen y la convención.*

Identificadores sugeridos · `date_day_of_week` , `names` , `w` **Criterios de aceptación**

- [ ] Usa el número de día + una constante (no cuenta días desde un punto fijo con un bucle gigante).
- [ ] El resultado siempre está en `0..6` , incluso con números de día negativos.
- [ ] Los cinco casos del escenario dan exactos.
- [ ] Se imprime el **nombre** del día, no solo el índice.

Ejercicio 19.10 — Día del año (`date_day_of_year`)

Conceptos · suma de meses anteriores, bisiesto, rango `1..366`

Contexto · Los calendarios y los reportes anuales numeran los días del año (el "día 100"). Es un dato que también sirve para medir distancias dentro del mismo año. Simplificación: el ordinal del día.

Consigna · Implementá `date_day_of_year(Date d)` : devuelve `1` para el 1 de enero y `365` / `366` para el 31 de diciembre (según bisiesto). Podés sumar los días de los meses anteriores + el día, o restar números de día contra el 1 de enero del mismo año. Comprobá el 1 de enero, un fin de año y el **29 de febrero**.

Entrada · Una `Date` . **Salida** · El día del año (`1..366`).

Escenario de prueba

```
2024-01-01  → 1
2024-02-29  → 60      (31 de enero + 29 de febrero)
2024-03-01  → 61
2023-03-01  → 60      (2023 no bisiesto: febrero tuvo 28)
2024-12-31  → 366
2023-12-31  → 365
```

Identificadores sugeridos · `date_day_of_year` , `sum` , `m` **Criterios de aceptación**

- [ ] El 1 de enero da `1` (no `0`).
- [ ] El fin de año da `365` o `366` según el bisiesto.
- [ ] `2024-03-01` da `61` y `2023-03-01` da `60` (la diferencia es el bisiesto).
- [ ] No se accede a meses fuera de `1..12`.

PARTE 4 — El reloj del sistema: `time_t`, `struct tm` y `mktime`

Hasta acá manejaste fechas **puras** (año, mes, día), sin horas ni husos horarios. Pero tarde o temprano vas a necesitar **el ahora** del sistema, o convertir tu fecha a un instante absoluto. Ahí entra `<time.h>`, y con él, una buena dosis de trampas.

4.1 El epoch y `time_t`: segundos desde el kilómetro 0

La computadora, por dentro, **no** guarda "2026-10-14". Guarda **un número**: la cantidad de **segundos** transcurridos desde un instante de referencia llamado **epoch**.

- `time_t`: el tipo que representa ese número. El estándar **no fija** su unidad ni su epoch, pero en la práctica (Linux, Windows, macOS) es **segundos** desde el **1970-01-01 00:00:00 UTC** (el "Unix epoch").
- `time(NULL)`: devuelve el `time_t` de **ahora**.

```
epoch: 1970-01-01 00:00:00 UTC  →  time_t = 0

2000-01-01 00:00:00 UTC   →  946684800
2024-01-01 00:00:00 UTC   →  1704067200
```

> ***Los segundos son UTC.*** *El epoch y* `time_t` ***no dependen*** *de tu huso horario: son un instante absoluto. Si hoy a las 12:00 en Buenos Aires y a las 12:00 en Tokio son instantes distintos, sus* `time_t` ***también*** *lo son.* `time_t` *es "un punto en la recta del tiempo universal"; el huso horario recién aparece cuando* ***mostrás*** *ese instante a un humano (4.3).*

4.2 `struct tm` : la fecha desarmada en piezas

Cuando querés "abrir" un `time_t` en año, mes, día, hora, etc., C te da una estructura: `struct tm` . Es como un **sobre con etiquetas**, una casilla por campo. Pero tiene **tres trampas históricas** que arruinan a todo el mundo la primera vez. Leelas dos veces.

Campo	Significado	Trampa
`tm_sec`	segundos `0–60`	El `60` es para los *leap seconds*.
`tm_min`	minutos `0–59`	—
`tm_hour`	horas `0–23`	No hay 24.
`tm_mday`	día del mes `1–31`	**1-based** (empieza en 1).
`tm_mon`	mes `0–11`	**¡Enero es 0!** (0-based)
`tm_year`	años **desde 1900**	**¡2026 se guarda como 126!**
`tm_wday`	día de semana `0–6`	Domingo = 0 (0-based)
`tm_yday`	día del año `0–365`	**0-based** (¡el 1 de enero es 0!)
`tm_isdst`	horario de verano	`>0` sí, `0` no, `<0` "no sé"

> ***El trío de trampas.*** *(1)* `tm_year` ***desde 1900****: para pasar de* `struct tm` *a año real,* ***sumás 1900****; para ir al revés,* ***restás 1900****. (2)* `tm_mon` ***0-based****: enero es* `0` *, diciembre es* `11` *; por eso casi todo el mundo escribe* `month - 1` *al armar una* `struct tm` *. (3)* `tm_yday` ***0-based****: el 1 de enero es* `0` *, no* `1` *. Cada una de estas te va a morder si te olvidás.*

> ***Compará con tu*** `Date` **.** *¿Ves por qué tu* `Date` *es 1-based y* `struct tm` *es 0-based? Vos elegiste lo* ***natural para un humano****; C eligió, por razones históricas, lo que era cómodo para* ***restar****. Tu librería te protege de eso: convertís a* `struct tm` ***solo*** *en la frontera (cuando hablás con el sistema) y volvés enseguida a tu mundo 1-based.*

4.3 `mktime` : de la fecha al instante (y la normalización)

`mktime` es la función que convierte una `struct tm` (fecha **local**) en un `time_t` :

```
struct tm (año/mes/día/hora LOCAL)  ── mktime ──►  time_t (segundos desde el epoch)
```

Y tiene un **superpoder escondido**: **normaliza**. Si le pasás un `tm_mday = 32` , no se queja: entiende "el día 32 del mes" y lo convierte al **1 del mes siguiente**. Lo mismo con `tm_mon = 12` (→ enero del año siguiente), o valores negativos. ¡Es una calculadora de fechas incorporada!

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

int main(void) {
    /* Armamos una struct tm "desde cero". Ojo con las trampas: */
    struct tm t = { 0 };          /* todos los campos en 0 (buena práctica) */

    t.tm_year = 2000 - 1900;      /* ¡años desde 1900! → 100 */
    t.tm_mon  = 1 - 1;            /* ¡enero es 0!        → 0  */
    t.tm_mday = 1;                /* día 1                    */
    t.tm_isdst = -1;              /* "-1 = averiguá vos el horario de verano" */

    /* mktime interpreta la struct como hora LOCAL y devuelve el time_t.
       Con TZ=UTC, 2000-01-01 00:00:00 da exactamente 946684800. */
    time_t epoch = mktime(&t);
    printf("epoch = %ld\n", (long)epoch);
    return 0;
}
```

`tm_isdst = -1` : por qué. *El horario de verano (DST) cambia según la fecha y el país. Si no querés pelear, poné* `-1` *y dejá que* `mktime` *lo* ***averigüe*** *solo. Si lo pones mal, obtenés un* `time_t` *corrido una hora. La regla práctica:* ***`-1` y listo****.*

4.4 UTC vs local: dos relojes, y uno que a veces no existe

Acá está la fuente de **la mitad** de los bugs de fechas del mundo:

- **UTC** (Tiempo Universal Coordinado) es el reloj **absoluto**. No tiene husos ni verano. `time_t` vive en UTC.
- **Local** es el reloj **de tu pared**: UTC **más** el *offset* de tu huso horario (por ejemplo, Buenos Aires suele ser UTC−3) **más** el corrimiento de horario de verano, si aplica.

Dos funciones mueven entre ambos:

Función	Qué hace
`gmtime(&t)`	`time_t` → `struct tm` en **UTC** (Greenwich)
`localtime(&t)`	`time_t` → `struct tm` en **tu huso** (usa la variable `TZ`)

> ***El horario que no existe.*** *En los cambios de DST, hay una hora que* ***se repite*** *(cuando atrasan el reloj) y otra que* ***nunca ocurre*** *(cuando lo adelantan). Por eso* `local` *no es una función matemática: esa hora local puede ser* ***ambigua o inexistente****. La conclusión de oro:* ***guardá y calculá siempre en UTC*** *(* `time_t` *); convertí a local* ***solo al mostrar****.*

> ***Persistencia con UTC.*** *Si guardás fechas en un archivo (capítulo* `15` *) o las enviás por la red,* ***guardalas como*** `time_t` ***o como UTC****, nunca como "hora local sin huso". Si guardás "14:00" sin decir en qué reloj, otro programa en otro país las va a leer mal. Esto es un clásico de los sistemas distribuidos.*

Práctica de la Parte 4

Ejercicio 19.11 — De fecha a instante (`date_to_epoch`)

Conceptos · `struct tm` , resta de 1900, `mon - 1` , `mktime` , `tm_isdst`

Contexto · Para comparar con el reloj del sistema, o para guardar una fecha como número, hay que pasar de tu `Date` al `time_t` absoluto. Simplificación: convertir una `Date` + hora a epoch.

Consigna · Implementá `date_to_epoch(Date d, int hour, int min, int sec)` : armá una `struct tm` cuidando las **tres trampas** (`year - 1900` , `month - 1` , `tm_isdst = -1`), completá hora/min/seg, llamá a `mktime` y devolvé el `time_t` . Después comprobá la **ida y vuelta**: convertí ese `time_t` a `struct tm` con `gmtime` y verificá que los campos coincidan (con `TZ=UTC`).

Entrada · Una `Date` y hora/minuto/segundo. **Salida** · El `time_t` correspondiente.

Escenario de prueba

```
(con TZ = UTC)
date_to_epoch(1970-01-01, 0, 0, 0)  → 0
date_to_epoch(2000-01-01, 0, 0, 0)  → 946684800
date_to_epoch(2024-01-01, 0, 0, 0)  → 1704067200
```

> ***Si tus números difieren** por un múltiplo de la hora (por ejemplo, ±3 h), es tu **huso horario** metiéndose: `mktime` interpreta la `struct tm` como **local**. Para el escenario exacto, corré con `TZ=UTC` (`set TZ=UTC` en Windows, `TZ=UTC ./prog` en Linux/macOS). Es justamente el punto de 4.4.*

Identificadores sugeridos · `date_to_epoch` , `t` , `d` , `sec` **Criterios de aceptación**

- [ ] `tm_year = d.year - 1900` y `tm_mon = d.month - 1` (¡las trampas!).
- [ ] `tm_isdst = -1` .
- [ ] Con `TZ=UTC` , los tres valores del escenario dan exactos.
- [ ] Se rellena hora, minuto y segundo del parámetro.

Ejercicio 19.12 — El reloj de ahora (`date_today` + `date_now_local`)

Conceptos · `time(NULL)` , `gmtime` , `localtime` , `strftime` , diferencia UTC/local

Contexto · Un log, un turno o un mensaje de bienvenida necesitan "el ahora". Simplificación: mostrar la fecha y hora actual en **UTC** y en tu **huso local**, y calcular la diferencia.

Consigna · Implementá:

1. `date_today()` que devuelva **hoy** como `Date` (usá `time(NULL)` y `localtime`, cuidando las trampas de `struct tm`);
2. `date_now_local(struct tm *out)` que llene `out` con el "ahora" local.

En el `main`, mostrá el instante actual formateado en **UTC** y en **local** (con `strftime`), y la **diferencia** entre ambos (el *offset* de tu huso) en horas.

Entrada · Ninguna (lee el reloj). **Salida** · La fecha/hora UTC, la local, y el offset.

Escenario de prueba

```
UTC   : 2026-10-14 12:00:00
local : 2026-10-14 09:00:00      (UTE-3, por ejemplo)
offset: -3 horas
```

> ***No es determinista*** *(depende de cuándo lo corras). Lo que* ***sí*** *es verificable: la diferencia entre UTC y local debe ser un número* ***entero de horas*** *(o media hora en algunos países), y el* `time_t` *que usás para las dos vistas debe ser* ***el mismo****.*

Identificadores sugeridos · `date_today`, `date_now_local`, `now`, `utc`, `local`
Criterios de aceptación

- [ ] Ambas vistas parten del **mismo** `time_t` (`time(NULL)` una sola vez).
- [ ] `localtime` se usa para el huso; `gmtime` para UTC.
- [ ] Las trampas de `struct tm` se manejan al convertir a `Date`.
- [ ] La diferencia UTC–local queda expresada en horas.

PARTE 5 — Formatear y parsear: del número al texto y de vuelta

Última pieza: **mostrar** la fecha (para un humano) y **leerla** (de un archivo o del teclado). Es la frontera entre tu mundo interno (números, estructuras) y el mundo externo (texto).

5.1 `strftime`: la plantilla de formato

`strftime` toma una `struct tm` y una **plantilla** (una cadena de formato, como en `printf`) y escribe el texto resultante en un búfer. Cada `%` y una letra significan un campo:

Formato	Qué produce	Ejemplo
`%Y`	año, 4 dígitos	`2024`
`%m`	mes, `01–12`	`02`
`%d`	día del mes, `01–31`	`29`
`%H`	hora, `00–23`	`13`
`%M`	minuto, `00–59`	`45`
`%S`	segundo, `00–60`	`07`
`%j`	día del año, `001–366`	`060`
`%w`	día de semana, `0–6` (dom = 0)	`4`
`%A`	nombre del día	`Thursday`
`%B`	nombre del mes	`February`
`%F`	atajo de `%Y-%m-%d`	`2024-02-29`
`%T`	atajo de `%H:%M:%S`	`13:45:07`
`%%`	un `%` literal	`%`

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

int main(void) {
    /* Armamos una struct tm fija (OJO: año desde 1900, mes 0-based). */
    struct tm t = { 0 };
    t.tm_year = 2024 - 1900;
    t.tm_mon  = 2 - 1;
    t.tm_mday = 29;

    /* mktime completa tm_wday y tm_yday a partir del año/mes/día.
       Sin esto, %A y %j no tendrían de dónde salir. */
    mktime(&t);

    char buf[64];
    /* strftime(búfer, tamaño, plantilla, &tm): escribe el texto formateado.
```

```
       Devuelve cuántos bytes escribió (sin contar el '\0'). */
    strftime(buf, sizeof buf, "%Y-%m-%d (%A)", &t);
    printf("%s\n", buf);   /* 2024-02-29 (Thursday) */
    return 0;
}
```

El detalle de `%A` y `%B` : dependen del locale. *Con la configuración por defecto (el locale `"C"`), los nombres salen en **inglés**: `Thursday` , `February` . Para que salgan en español (`jueves` , `febrero`) hay que **configurar el locale** con `setlocale(LC_TIME, "es_ES.UTF-8")` (o el que exista en la máquina). Y ojo: **no todos los sistemas tienen el mismo locale instalado**, así que los nombres traducidos pueden no aparecer. Para texto **estable**, usá números (`%Y-%m-%d`) y vos armás los nombres si los necesitás.*

5.2 Devolver el texto: el `const char *` prestado

Escribir en un búfer ajeno (como hace `date_format`) está bien, pero es **verboso**: declarás el arreglo, le pasás el tamaño, te acordás de que existe. A veces querés lo cómodo: **que la función te devuelva el texto y listo**, para meterlo directo en un `printf` .

```
printf("Hoy es %s\n", date_to_string(today, "%d/%m/%Y"));
```

Para que eso funcione, `date_to_string` tiene que devolver un `const char *` . Y acá hay una pregunta que en C **no se puede esquivar**: ¿**dónde vive** ese texto? La función retorna, su marco de pila se destruye… y un puntero a una variable local quedaría apuntando a **tierra arrasada** (el *dangling pointer* del capítulo `07`).

La respuesta del C clásico es un **búfer estático interno**: una variable local declarada `static`, que **no** vive en la pila sino que se crea una sola vez y **sobrevive** a todas las llamadas.

```c
#include <stdio.h>

/* Ilustrativo: el patrón del "búfer prestado". No es la solución de un ejercicio. */
const char *greeting(void) {
    static char buf[32];      /* NO está en la pila: vive TODO el programa */
    snprintf(buf, sizeof buf, "hola");
    return buf;               /* válido: 'buf' sigue vivo al volver */
}
```

> ***Pensalo como el pizarrón del aula.*** *El aula (programa) tiene* ***un solo*** *pizarrón (`static char buf`). Cualquiera lo usa y se va; el pizarrón queda. Cuando otra función (`ctime`, `localtime`, `date_to_string`) escribe en* ***su*** *pizarrón,* ***borra*** *lo que había antes. Si vos "te guardaste" el contenido anterior* ***copiándolo a tu carpeta*** *(tu propio arreglo), perfecto; si solo* ***apuntaste al pizarrón****, vas a leer lo nuevo, no lo viejo.*

Ese es el **contrato de "puntero prestado"**: la función te da una dirección válida, pero el contenido es **de ella** y puede cambiarlo cuando quiera. Consecuencias, todas clásicas:

```c
p1 = date_to_string(d1, "%Y");   /* p1 apunta al búfer interno → "2024" */
p2 = date_to_string(d2, "%Y");   /* la MISMA dirección; el búfer ahora dice "2025" */
printf("%s %s\n", p1, p2);       /* sale: "2025 2025"  ← ¡p1 también cambió! */
```

- **No es *thread-safe*.** Dos hilos compartirían el mismo búfer y se pisarían entre sí.
- **No es reentrante.** No la podés llamar desde dentro de un `printf` que ya la está usando… bueno, técnicamente sí compila, pero el resultado es impredecible.
- **El resultado dura hasta la próxima llamada.** Si lo necesitás más tiempo, **copialo**.

> ***No es una rareza: es exactamente lo que hace*** `<time.h>`***.*** *`ctime`, `asctime`, `localtime` y `gmtime` devuelven punteros a búferes internos (a veces* ***compartidos*** *entre varias de ellas). Por eso "guardar el `char *` de `localtime` y después llamarla de nuevo" es el bug clásico del horario que "se cambia solo". Tu `date_to_string` hereda el mismo contrato:* ***el texto es prestado****.*

> ***¿Cuándo usar cada uno?*** *Regla práctica:* ***prestado*** *(`date_to_string`) para usar y tirar — un `printf`, una comparación al vuelo—;* ***tu búfer*** *(`date_format`) cuando necesités guardar el texto o tener varios vivos a la vez;* ***arena*** *(capítulo `12`) cuando sean muchos o de tamaño variable. Y si de verdad necesitás varios "prestados" simultáneos en un programa concurrente, existe el almacenamiento thread-local (`_Thread_local`), que es tema para más adelante.*

> ***No te olvides del tope.*** *El búfer static tiene un tamaño* ***fijo*** *(como el de `greeting`). Si el texto formateado no entra, se* ***trunca*** *(o `strftime` devuelve `0` y no escribió nada). Un `%B` con un nombre de mes largo, más un `%A`, más hora, pueden pasarse. Elegí el tope con margen y* ***tratá el caso "no entró"****.*

5.3 Parsear: del texto a la fecha

Parsear es lo inverso de formatear. Tenés el texto `"2024-02-29"` y querés una `Date`. Dos caminos:

1. **`sscanf`** (capítulo `14`): el estándar, seguro y **portable**. Le indicás el formato con `%d` y separadores literales:

   ```
   int y, m, d;
   if (sscanf("2024-02-29", "%d-%d-%d", &y, &m, &d) == 3) {
       /* los tres se leyeron bien */
   }
   ```

2. **`strptime`** (POSIX): inversa de `strftime`, pero **no estándar** de ISO C. Existe en Linux y macOS; en Windows suele requerir un reemplazo (`strptime` como función propia o `sscanf`).

```
"2024-02-29"  ──parse(" %d-%d-%d")──►  { 2024, 2, 29 }  ──date_is_valid──►  OK
"2023-02-29"  ──parse(" %d-%d-%d")──►  { 2023, 2, 29 }  ──date_is_valid──►  INVÁLIDA
```

> ***Parsear NO es validar.*** `sscanf` *te da tres números;* ***no*** *sabe si el* `2-29` *existe. Podés parsear* `"2023-02-29"` *sin problema: los tres enteros entran. La* ***validación*** *(* `date_is_valid`*, Parte 1) es un paso* ***aparte*** *y* ***obligatorio****. Confundirlos es meter una fecha imposible en tu sistema.*

> ***El infierno de las fechas ambiguas.*** `"01/02/2024"` *puede ser el* ***1 de febrero*** *o el* ***2 de enero****, según el país. Nunca adivines:* ***imponé un formato*** *(por ejemplo, el ISO* `AAAA-MM-DD`*, que ordena alfabéticamente y no se presta a confusión) y rechazá lo que no lo cumpla.*

Práctica de la Parte 5

Ejercicio 19.13 — Formatear en tu búfer (`date_format`)

Conceptos · `strftime`, búfer acotado, armar un `struct tm` desde `Date`, valor de retorno

Contexto · Mostrar fechas en reportes, logs y carteles. Simplificación: formatear una `Date` con una plantilla.

Consigna · Implementá `date_format(Date d, const char *fmt, char *out, size_t cap)`: convertí `d` a un `struct tm` (cuidando las trampas; podés llamar a `mktime` para que se completen `tm_wday` / `tm_yday`), llamá a `strftime` con la plantilla, y devolvé cuántos bytes se escribieron (`0` si no entró o hubo error). Comprobá varias plantillas sobre la misma fecha.

Entrada · Una `Date`, una plantilla y un búfer. **Salida** · El texto formateado.

Escenario de prueba

```
fecha = 2024-02-29
  fmt "%Y-%m-%d"      → "2024-02-29"
  fmt "%d/%m/%Y"      → "29/02/2024"
  fmt "%j"            → "060"
  fmt "%Y-%m-%d (%A)" → "2024-02-29 (Thursday)"   (locale "C": en inglés)
  fmt "%F %T"         → "2024-02-29 00:00:00"
```

> ***Sobre `%A`:*** *sale `Thursday` porque el locale por defecto es `"C"`. Si configurás `setlocale`, puede salir en español; si tu máquina no tiene el locale, no aparece. Por eso el escenario usa el inglés.*

Identificadores sugeridos · `date_format`, `t`, `written` **Criterios de aceptación**

- [ ] Se arma el `struct tm` con `year - 1900` y `month - 1`.
- [ ] `strftime` recibe el **tamaño** del búfer (`cap`), no un número inventado.
- [ ] Devuelve cuántos bytes se escribieron (`0` si no entró).
- [ ] Las cinco plantillas del escenario dan lo esperado.

Ejercicio 19.14 — Devolver el texto (`date_to_string`)

Conceptos · `static` (duración de almacenamiento), puntero prestado, `strftime`, `const char *`

Contexto · Para imprimir una fecha formateada al vuelo querés escribir `printf("%s\n", date_to_string(d, "%d/%m/%Y"))` sin declarar un arreglo. Simplificación: una variante cómoda de `date_format` que devuelve un `const char *` en vez de escribir en un búfer tuyo.

Consigna · Implementá `date_to_string(Date d, const char *fmt)`: convertí `d` a un `struct tm` (cuidando las trampas), formatealo con `strftime` en un **búfer `static` interno** de tamaño fijo, y devolvé el puntero a ese búfer. Si el texto **no entra** o hay error, devolvé un puntero a cadena vacía (`""`), nunca `NULL`. Después, **demostrá en un mismo `main`** las consecuencias del contrato prestado:

1. dos llamadas **seguidas** y tu `printf` mostrando que el primer puntero "cambió" (apunta al mismo búfer que el segundo);
2. la salida de `date_format` **no** sufre ese efecto (dos búferes tuyos, dos resultados distintos).

Entrada · Una `Date` y una plantilla. **Salida** · Un `const char *` al texto formateado (válido hasta la próxima llamada).

Escenario de prueba

```
date_to_string(2024-02-29, "%Y-%m-%d")  → "2024-02-29"
date_to_string(2024-02-29, "%d/%m/%Y")  → "29/02/2024"

demostración del "se pisa":
  const char *a = date_to_string(f1, "%Y");   /* a apunta al búfer interno */
  const char *b = date_to_string(f2, "%Y");   /* la MISMA dirección */
  printf("%s | %s\n", a, b);                  → "2025 | 2025"   (¡a "cambió"!)
  /* con date_format y dos búferes propios: */
  → "2024 | 2025"   (bien)
```

> ***Ojo con el tope del búfer.*** *Elegí un tamaño fijo con margen (por ejemplo* `64` *) y* ***tratá*** *el caso "no entró":* `strftime` *devuelve* `0` *y no escribió. Este es el precio del búfer static: un tamaño que no podés adaptar a la plantilla.*

Identificadores sugeridos · `date_to_string` , `buf` , `t` , `n` Restricciones / pistas

- El búfer debe ser `static` **dentro** de la función; si fuera un arreglo local común, devolverías un *dangling pointer* (capítulo `07`).
- No uses `malloc` (capítulo `09`): acá el contrato es "prestado", sin reserva dinámica.
- Documentá en un comentario, junto a la función, que el puntero **se pisa en la próxima llamada**.
- Diferenciá bien: `date_format` = **tu** búfer; `date_to_string` = **búfer interno prestado**.

Criterios de aceptación

- ☐ El búfer es `static char` dentro de la función (no un local común).
- ☐ Se arma el `struct tm` con `year - 1900` y `month - 1` .
- ☐ Si `strftime` no entra (`0`), devuelve `""` (nunca `NULL` ni basura).
- ☐ Las dos primeras conversiones del escenario dan lo esperado.
- ☐ La demostración prueba que dos llamadas seguidas comparten el mismo búfer (el primero "se pisa").
- ☐ La comparación con `date_format` (dos búferes propios) da resultados **distintos** y correctos.

Ejercicio 19.15 — Parsear con validación (`date_parse`)

Conceptos · `sscanf` , valor de retorno, `date_is_valid` , formatos ambiguos

Contexto · Leer fechas de un archivo (`fgets` , capítulo `15`) o de `argv` (capítulo `13`) y convertirlas a `Date` , rechazando lo mal formado **y** lo inexistente. Simplificación: parser con formato fijo ISO + validación.

Consigna · Implementá `date_parse(const char *s, const char *fmt, Date *out)` : usá `sscanf` con el formato, verificá que haya leído **los tres** valores (`== 3`), armá la `Date` , y **validala** con `date_is_valid` . Devolvé `1` si todo salió bien (parseo **y** validación) y `0` si no. **No** modifiques `*out` si falla.

Entrada · Un texto, un formato y un puntero a `Date` . **Salida** · `1` / `0` , y la `Date` llena si correspondía.

Escenario de prueba

```
parse("2024-02-29", "%d-%d-%d")  → 1,  {2024, 2, 29}
parse("2023-02-29", "%d-%d-%d")  → 0   (parsea, pero 2023-02-29 NO existe)
parse("2024-13-01", "%d-%d-%d")  → 0   (mes 13)
parse("29/02/2024", "%d/%m/%Y")  → 1,  {2024, 2, 29}
parse("no es una fecha","%d-%d-%d") → 0
parse("2024-02", "%d-%d-%d")     → 0   (leyó solo dos)
```

> ***El caso estrella:*** *`"2023-02-29"` **parsea bien** (tres enteros válidos) pero **valida mal**. Ese es exactamente el bug que este ejercicio previene: parsear no es validar.*

Identificadores sugeridos · `date_parse` , `y` , `m` , `d` , `tmp` , `n` **Criterios de aceptación**

- [ ] Se comprueba que `sscanf` leyó **tres** valores.
- [ ] Después de parsear, se llama a `date_is_valid` (parsear no alcanza).
- [ ] `2023-02-29` da `0` (parsea pero se rechaza), y `2024-02-29` da `1` .
- [ ] Si falla, no se toca la `Date` de salida.

PARTE 6 — El catálogo completo: instantes, duraciones y el jefe

Para cerrar, tres herramientas más (`difftime` , la duración entre instantes, y el manejo de la famosa "bomba" de 2038) y un desafío final que integra **todo**.

6.1 `difftime` : la resta de instantes

`date_diff_days` te da **días** entre fechas. Cuando tenés **instantes** (`time_t`), la resta te da **segundos**, y ahí entra `difftime` :

```
difftime(t2, t1)  →  la diferencia (t2 − t1) en segundos, como double
```

Es la forma **portable** de restar dos `time_t` (porque `time_t` puede no ser "segundos" en todas las plataformas). Dividiendo, obtenés horas, días, etc.

6.2 El fin de un `int` de 32 bits: el efecto 2038

Como en muchas plataformas `time_t` es un entero **con signo de 32 bits**, el máximo es `2147483647` segundos, que corresponde a:

```
2038-01-19 03:14:07 UTC   →  el último instante que entra en un time_t de 32 bits
```

Un segundo después, se **desborda** y (si sigue con signo) da un número **negativo**: la fecha salta a **1901**. Ese es el **efecto 2038** (el "Y2K de Unix"). En las máquinas de 64 bits de hoy no es problema (el rango se va a la estratósfera), pero en sistemas embebidos con `time_t` de 32 bits sigue vivo.

> ***Cómo te protegés.*** *Usá* `time_t` *(deja que la plataforma elija su ancho) y* ***no*** *guardes el epoch en un* `int` *. Y cuando guardes en archivo, guardá* `time_t` */64 bits, no un* `int32` *.*

6.3 El ejercicio jefe: el calendario

Ahora juntá **todo**: años bisiestos, largos de mes, día de la semana y formato, para imprimir el calendario de un mes como la vieja herramienta `cal`.

```
(ejemplo: febrero de 2024)

     febrero 2024
do lu ma mi ju vi sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29
```

Práctica de la Parte 6

Ejercicio 19.16 — El calendario de un mes (jefe)

Conceptos · todo: `date_is_leap`, `date_days_in_month`, `date_day_of_week`, alineación con `printf`

Contexto · `cal` es una de las herramientas más viejas de Unix y ejercita toda tu librería de una sola pasada. Simplificación: imprimir el calendario de **un** mes de un año dado.

Consigna · Escribí un programa que reciba un mes y un año (por `argv`, capítulo `13`, o fijos en el `main`) e imprima el calendario de ese mes:

1. el encabezado con el nombre del mes y el año;
2. la fila de días de la semana (por ejemplo `do lu ma mi ju vi sa`, con **domingo primero**, para coincidir con `date_day_of_week`);
3. las semanas alineadas: la primera fila deja tantos huecos como indique el día de la semana del día 1; cada número ocupa **dos columnas** (usá `%2d`), y las columnas se separan con un espacio;
4. un `\n` al final de cada semana;
5. validá el mes/año con tu librería antes de imprimir.

Después, subí la apuesta: un `main` que imprima **los doce meses** del año en columnas (tres por fila), como `cal 2024`.

Entrada · Un mes (`1–12`) y un año. **Salida** · El calendario del mes, alineado.

Escenario de prueba

```
entrada: mes = 2, año = 2024

        febrero 2024
   do lu ma mi ju vi sa
                 1  2  3
    4  5  6  7  8  9 10
   11 12 13 14 15 16 17
   18 19 20 21 22 23 24
   25 26 27 28 29
```

```
entrada: mes = 9, año = 2026     (1 de septiembre de 2026 cae martes)

        septiembre 2026
   do lu ma mi ju vi sa
    1  2  3  4  5  6  7
    8  9 10 11 12 13 14
   15 16 17 18 19 20 21
   22 23 24 25 26 27 28
   29 30
```

> ***Cómo comprobar la alineación:*** *el día 1 de febrero de 2024 fue* ***jueves****. Con domingo = 0, jueves = 4, así que la primera fila lleva* ***4*** *huecos antes del* `1` *. Si tu calendario arranca el* `1` *en otra columna, revisá* `date_day_of_week` *.*

Identificadores sugeridos · `print_month` , `print_year` , `year` , `month` , `dow` , `day` **Restricciones/pistas**

- Reusá **toda** la librería: `date_is_valid`, `date_days_in_month`, `date_day_of_week`, y `date_from...` si te sirve para armar cada día.
- La alineación sale del `date_day_of_week(Date{year, month, 1})`: esa cantidad de **huecos** va antes del primer número.
- Usá `%2d` para cada día (dos columnas) y un separador de **un** espacio entre columnas.
- Nombres de meses y días: un arreglo de cadenas en español (no dependas del locale, que puede no existir).
- Validá mes (`1..12`) y año razonable; si no, avisá por `stderr` y no imprimas basura.

Criterios de aceptación

- [ ] El 1 del mes cae en la columna correcta (según `date_day_of_week`).
- [ ] Febrero de un bisiesto muestra **29** días; de un no bisiesto, **28**.
- [ ] Cada número ocupa **dos** columnas y las semanas se cierran con `\n`.
- [ ] Los dos escenarios (feb 2024 y sep 2026) salen **idénticos** al ejemplo.
- [ ] Reusa las funciones de la librería (no repite la regla del bisiesto ni la del día de semana).
- [ ] Un mes o año inválido se reporta sin imprimir un calendario roto.
- [ ] (Extra) El año completo imprime los doce meses en columnas.

19.17 Errores comunes

- **Olvidar el bisiesto de los 400.** `1900` **no** es bisiesto, `2000` **sí**. "Cada 4 años" es incorrecto.
- **La regla del 100 al revés.** Un año divisible por 100 **no** es bisiesto, **salvo** que sea divisible por 400. No lo resumas a "divisible por 4".
- `tm_year` **sin el** `-1900` / `+1900`**.** `struct tm` guarda **años desde 1900**: `2026` es `126`.
- `tm_mon` **o** `tm_yday` **0-based.** Enero es `0` (no `1`); el 1 de enero es `tm_yday == 0` (no `1`).
- **Módulo de negativos sin blindar.** `-1 % 7` da `-1` en C, no `6`. Usá `((x % 7) + 7) % 7` para el día de la semana.
- **Creer que** `time_t` **es siempre "segundos".** El estándar **no** lo garantiza; en la práctica sí, pero usá `difftime` para restar, no una resta cruda.
- **Mezclar local con UTC.** Guardá y calculá en **UTC**; convertí a local **solo al mostrar**. Guardar "hora local sin huso" en un archivo es un bug esperando a mudarse de país.
- **Ignorar el DST.** La hora local puede **repetirse** o **no existir** el día del cambio de horario. `tm_isdst = -1` y dejá que el sistema decida.
- **Parsear sin validar.** `"2023-02-29"` se parsea bien y **no** existe. Validá siempre después.
- **Confiar en los nombres de mes/día del locale.** Sin configurar `setlocale`, salen en inglés; y no todos los sistemas tienen el locale en español. Para texto estable, usá números.
- **Asumir un ancho de** `time_t`**.** No guardes epoch en un `int` (efecto 2038). Dejá que `time_t` mande.
- **Escribir a** `data[-1]` **al armar días.** Al iterar meses/días, cuidá los límites (mes `1..12`, día `1..n`): no leas "un día antes del 1".
- **Devolver un puntero a un local.** Si `date_to_string` usara un arreglo local **común** en vez de `static`, devolverías un *dangling pointer* (capítulo `07`): memoria que ya no es tuya.
- **Creer que el** `const char *` **prestado es tuyo.** El de `date_to_string` (y el de `ctime`, `asctime`, `localtime`) se **pisa** en la próxima llamada. Guardalo (copialo) si lo necesitás después; no lo liberes con `free`.

19.18 Cierre

Si tuvieras que quedarte con **una sola idea**, que sea esta: **una fecha es una etiqueta irregular; convertila a un número de día y todo se vuelve sumas y restas**.

- Dejaste de pelear con meses y bisiestos al hacer aritmética: **linealizás** la fecha a un número de día, operás con enteros, y **la volvés** a fecha. Sumar, restar, comparar y ubicar el día de la semana salen de ahí.
- Aprendiste a **validar** antes de calcular: no toda etiqueta existe (`2023-02-29` no), y para saberlo necesitás la regla **completa** del bisiesto (4 / 100 / 400).
- Separaste **dos mundos**: tu `Date` 1-based, sin husos, para la lógica de calendario; y `time_t` / `struct tm` para hablar con el reloj del sistema, cuidando sus **tres trampas**.
- Entendiste que el tiempo se **guarda en UTC** y se **muestra en local**, y por qué eso evita la mitad de los bugs de fechas del mundo.
- Distinguiste los **dos contratos de formateo**: `date_format` escribe en tu búfer (seguro); el `const char *` de `date_to_string` es **prestado** (cómodo, pero se pisa) — el mismo trato de `ctime` y `localtime` .
- Escribiste una **librería completa**: validación, linealización, aritmética, epoch, formato, parseo y un impresor de calendarios.

Resumen de la caja de herramientas.

Quiero…	Función	Idea en una frase
¿Año bisiesto?	`date_is_leap`	Divisible por 4, no por 100, salvo por 400.
Largo del mes	`date_days_in_month`	Tabla + febrero según bisiesto.
¿Fecha real?	`date_is_valid`	Mes `1..12` y día `1..días_del_mes`.
Linealizar	`date_to_day_number`	Fecha → número de día (origen 1970 = 0).
Vuelta atrás	`date_from_day_number`	Número de día → fecha (inversa de la anterior).
Sumar/restar días	`date_add_days`	`from(to(fecha) + n)`.
Días entre fechas	`date_diff_days`	Resta de números de día.
Comparar	`date_compare`	`-1`, `0`, `1`.
Día de la semana	`date_day_of_week`	`((día + 4) % 7 + 7) % 7` (dom = 0).
Día del año	`date_day_of_year`	Meses anteriores + día (`1..366`).
Instante absoluto	`date_to_epoch`	`struct tm` (local) → `mktime` → `time_t`.
Ahora	`date_today` / `date_now_local`	`time(NULL)` + `localtime`.
Mostrar (tu búfer)	`date_format`	`strftime` con plantilla, escribís en tu arreglo.
Mostrar (prestado)	`date_to_string`	`strftime` en búfer `static`; se pisa en la próxima llamada.
Leer	`date_parse`	`sscanf` + `date_is_valid`.
Duración entre instantes	`difftime`	Resta de `time_t` en segundos.

> ***El puente a lo que sigue.*** *Ahora las fechas dejaron de ser un dolor. Con lo mismo (linealizar, operar, volver) podés resolver horarios, agendas, edades, plazos y reportes por rango de fechas. Y si algún día guardás miles de eventos, la combinación de este capítulo con el* `18` *(cadenas) y el* `12` *(arenas) te deja construir un sistema de turnos completo* ***sin reservar memoria de más*** *y sin sorpresas de huso horario. El calendario, ese edificio de cuartos irregulares, ahora lo recorrés vos.*