# Capítulo 13 — Tipos más usados de C

> *Capítulo avanzado. Se apoya en tipos primitivos (`02`), operadores (`03`), arreglos (`06`), punteros (`07`) y `struct` (`08`).*
>
> *Recorre los tipos "con nombre" más usados de C — `size_t`, `wchar_t`, `NULL`, `bool`, los enteros de ancho exacto y los caracteres Unicode— y de qué cabecera sale cada uno. El objetivo es que sepas **cuándo NO alcanza con un** `int`.*

---

## 13.1 ¿Por qué no alcanza con un `int`?

Un `int` resuelve muchísimos casos, pero **no responde de forma portable** tres preguntas:

1. **¿Cuánto pesa esto?** → `sizeof` no devuelve un `int`, devuelve `size_t`.
2. **¿Cuántos bits ocupa este número?** → para garantizar ese tamaño están `intN_t` / `uintN_t`.
3. **¿Cuál es el carácter más grande que puedo guardar?** → un `char` (1 byte) no alcanza; hay `wchar_t` y `char32_t`.

> ***¿Qué es el "ancho" de un tipo?*** *Es **cuántos bits ocupa el número en memoria**. `int32_t` ocupa siempre 32 bits (4 bytes) en cualquier máquina; `int`, en cambio, mide 32 bits en la mayoría de los equipos de hoy, pero 16 en otros. Cuando el tamaño **tiene** que ser siempre el mismo —un archivo, un protocolo de red, hardware— no confíes en `int`: usá `int32_t` (con signo) o `uint32_t` (sin signo), donde ese número **es** el ancho en bits.*

La respuesta de C fueron **typedefs con nombre** que se comportan igual en cualquier plataforma. Casi ninguno es palabra del lenguaje: son **alias que aporta una cabecera**, y por eso importa saber **de dónde sale cada uno**.

> ***La idea central.*** *No elegís `size_t` ni `uint32_t` por capricho: son los tipos que la implementación **garantiza** que representan cierto valor en cualquier máquina. `int` y `long` **no** dan esa garantía: cambian de tamaño según la plataforma.*

## Los modelos de datos: por qué cambian los tamaños

El estándar **no fija** que `int` mida 4 bytes; solo exige un orden creciente ( `char ≤ short ≤ int ≤ long ≤ long long` ). Las decisiones reales de cada sistema son el **modelo de datos** (valores en bits):

| Tipo | LP32 | ILP32 | LLP64 | LP64 |
|---|---|---|---|---|
| `int` | 16 | **32** | **32** | **32** |
| `long` | 32 | 32 | 32 | **64** |
| puntero / `size_t` | 32 | 32 | **64** | **64** |

- **ILP32**: Windows y Unix de 32 bits. **LLP64**: **Windows 64** ( `long` sigue en 32, punteros 64).
- **LP64**: **Linux y macOS 64** ( `long` y punteros en 64). **LP32**: Windows de 16 bits (histórico).

> ***Lo que te tiene que quedar grabado:*** *en 64 bits, un* `int` *mide* ***32 bits*** *pero un puntero mide* ***64****. Por eso* `int` *no puede guardar una dirección, ni un tamaño mayor a 2 GiB. Ese es el motivo de que existan* `size_t` *y los tipos de ancho fijo.*

---

## 13.2 El mapa de los tipos

Casi ningún tipo "con nombre" es palabra del lenguaje: son **typedefs** de una cabecera. Estos son los que vas a usar todo el tiempo (tamaño típico en un sistema de 64 bits):

| Tipo | Bytes | Para qué |
|---|---|---|
| `char`, `int`, `long`, `float`, `double`, … | — | los primitivos de siempre |
| `size_t` | 8 | tamaños y conteos |
| `wchar_t` | 2 / 4 | carácter ancho |
| `NULL` | = puntero | puntero nulo |
| `int8_t` … `int64_t` / `uint8_t` … `uint64_t` | 1 / 2 / 4 / 8 | entero de ancho **exacto** |
| `bool` | 1 | verdadero / falso |
| `char16_t` / `char32_t` | 2 / 4 | Unicode UTF-16 / UTF-32 |
| `char8_t` | 1 | Unicode UTF-8 |
| `float16_t` … `float128_t` | 2 / 4 / 8 / 16 † | flotante de ancho **exacto** |

> † *Solo desde **C23** (`<stdfloat.h>`) y **opcional**: flotantes de ancho exacto (16 / 32 / 64 / 128 bits) que en el C clásico (C99–C17) no existen.*

> ***Regla práctica.** Incluí **la cabecera del tipo que usás**: si usás `uint32_t`, incluí `<stdint.h>`; si usás `size_t`, `<stddef.h>`.*

---

## 13.3 `size_t` — los tamaños

**Cabecera:** `<stddef.h>`

Entero **sin signo**. Es el tipo del resultado de `sizeof`, el que devuelven `strlen` y `fread`, y el que reciben `malloc` y `memcpy`. Mide **4 bytes** en 32 bits y **8** en 64. Su máximo es `SIZE_MAX` (`<stdint.h>`).

**En vez de `int`.** Guardar un tamaño en un `int` **desborda** si pasa de ~2 GiB. Y comparar un `int` con un `size_t` es una trampa silenciosa: el `int` se convierte a sin signo.

```
int i = -1;
size_t n = 5;
if (i < n) { /* NO entra: -1 se volvió 18446744073709551615 (SIZE_MAX) */ }
```

**Cuidado.** `size_t` no tiene signo: `size_t x = 5; x -= 10;` da `SIZE_MAX - 4`, no `-5`. Nunca lo uses en un `for` inverso (`for (size_t i = n - 1; i >= 0; i--)` es **infinito**). **Se imprime con `%zu`.**

---

## 13.4 `wchar_t` — caracteres anchos

**Cabecera:** `<stddef.h>`

Un `char` ocupa 1 byte: alcanza para `A` o `z`, pero **no** para `ñ`, `€` ni 😀. Para poder guardar cualquier carácter existe un tipo **"ancho"**: `wchar_t`.

*Wide* significa "ancho": un carácter ancho ocupa **más de un byte** (2 o 4), así que entra completo en un solo `wchar_t`. Para escribir uno se antepone una `L`:

```
char     c  = 'A';        /* 1 byte */
wchar_t  wc = L'ñ';       /* carácter ancho: lleva el prefijo L */
wchar_t *ws = L"corazón"; /* cadena ancha: cada carácter, un wchar_t */
```

Y así como `strlen` recorre una cadena normal, `wcslen` recorre una cadena ancha (`wchar_t *`).

| Plataforma | `wchar_t` | Codificación |
|---|---|---|
| **Windows** | **2 bytes** | UTF-16 |
| **Linux / macOS** | **4 bytes** | UTF-32 |

> *El **mismo** `wchar_t` mide 2 bytes en Windows y 4 en Linux: código que asume uno u otro se rompe al portarlo. Para Unicode **portable** preferí `char32_t` ( `<uchar.h>` ), cuyo ancho no depende de la plataforma. Sus primos: `char16_t` (UTF-16) y `char8_t` (UTF-8, C23).*

**En vez de `int`.** Meter el carácter en un `int` tampoco arregla nada: `int` no es un tipo de carácter y confunde. Y `wchar_t` tampoco tiene ancho fijo (2 o 4 bytes, ver la tabla), así que la clave no es "agrandar" el tipo, sino **elegir el correcto**. **Se imprimen con `%lc` (carácter) y `%ls` (cadena), previo `setlocale`.**

---

## 13.5 `NULL` — el puntero nulo

**Cabecera:** `<stddef.h>`

`NULL` es una macro que expone una constante de puntero nulo (típicamente `((void*)0)` ). Representa "este puntero no apunta a nada": es lo que devuelve `malloc` al fallar y contra lo que se compara.

**En vez de `int` (y sin mezclar).** No confundas tres cosas distintas: `'\0'` (carácter nulo, para cadenas), `0` (número, para enteros) y `NULL` (puntero). `int x = NULL;` compila, pero es un error conceptual.

---

## 13.6 `bool` — verdadero o falso

**Cabecera:** `<stdbool.h>` (C99; en C23, palabra clave)

`bool` es un macro que expande a `_Bool` (C99), junto con `true` y `false` . Ocupa **1 byte**. En **C23** son palabras clave y ya no hace falta la cabecera.

**En vez de `int`.** `int flag = 1;` no dice qué es ese `1` ; `bool done = true;` sí. Además la conversión es distinta: `(bool)0.5` es **verdadero**, mientras que `(int)0.5` es `0` . **Cuándo NO usarlo:** si necesitás más de dos estados, un `enum` expresa mejor la intención.

---

## 13.7 Enteros de ancho exacto

**Cabecera:** `<stdint.h>`

Estos resuelven el "¿ `int` o `long` ?" de una vez: su **número de bits es exacto** y **no cambia** entre plataformas. Son los preferidos para protocolos, archivos binarios y hardware.

| Tipo | Bytes | Mínimo | Máximo |
|---|---|---|---|
| `int8_t` | 1 | -128 | **127** |
| `uint8_t` | 1 | 0 | **255** |
| `int16_t` | 2 | -32 768 | **32 767** |
| `uint16_t` | 2 | 0 | **65 535** |
| `int32_t` | 4 | -2 147 483 648 | **2 147 483 647** |
| `uint32_t` | 4 | 0 | **4 294 967 295** |
| `int64_t` | 8 | -9 223 372 036 854 775 808 | **9 223 372 036 854 775 807** |
| `uint64_t` | 8 | 0 | **18 446 744 073 709 551 615** |

> *Los tipos **sin signo** arrancan en `0` y llegan hasta `2ⁿ − 1`. Los **con signo** reparten esos bits: la mitad para negativos, por eso `int8_t` llega a `127` (no a `128`). Cada máximo tiene su nombre en `<stdint.h>`: `INT8_MAX`, `UINT8_MAX`, `INT64_MAX`, `UINT64_MAX`, …*

**En vez de `int`.** Un `int` mide 4 bytes en 64 bits, pero **2** en LP32: no garantiza 32 bits. `int32_t` sí. Para **imprimir** uno de estos tipos, convertí explícitamente a un tipo estándar de igual o mayor ancho, así el formato coincide:

```c
#include <stdint.h>
#include <stdio.h>
uint64_t big = 123456789012345ULL;
printf("%llu\n", (unsigned long long)big);
```

> ***Ojo:** `uint8_t` y `unsigned char` casi siempre miden 1 byte, pero no son lo mismo. `uint8_t` es para **bytes de datos**; `unsigned char`, para **memoria cruda**. Y `intN_t` es **opcional** en el estándar: existe solo si la plataforma tiene ese ancho exacto.*

> *¿Y los `float`? En el C clásico (C99–C17) **no** hay `intN_t` para la coma flotante: `float`, `double` y `long double` miden 4, 8 y 8 o 16 bytes según la máquina, sin garantía del estándar. Solo desde **C23** existe `<stdfloat.h>` con `float16_t`, `float32_t`, `float64_t` y `float128_t` (los `_FloatN`): el equivalente de `intN_t`. Son **nuevos y opcionales**, mucho menos portables.*

---

## 13.8 ¿Tengo que dejar de usar `int`?

**No.** Estos tipos **no reemplazan** a `int`: lo **complementan** para casos puntuales. El `int` sigue siendo el tipo por defecto para números enteros comunes.

Seguí usando `int` salvo que tengas un motivo concreto: contadores, cuentas, índices chicos, resultados de operaciones, o valores que le pasás a funciones que ya devuelven o reciben `int` (`getchar`, `strcmp`, `printf`).

Pasá a un **tipo preciso** solo cuando **exista un motivo**:

- `size_t` → tamaños y conteos: `sizeof`, `strlen`, índices de arreglos, `malloc`.
- `intN_t` / `uintN_t` → el ancho **tiene** que ser exacto: archivos binarios, protocolos de red, registros de hardware, o datos que deben ocupar lo mismo en cualquier máquina.
- `bool` → una respuesta sí/no.
- `wchar_t` / `char32_t` → un carácter que no entra en un `char`.

> ***Y ojo con el revés:** usar un tipo chico "por las dudas" también trae problemas. `int8_t` llega hasta 127 (ver la tabla de 13.7): si lo usás para contar, se desborda enseguida. Y encima la aritmética se hace en `int` igual.*

```c
/* MAL: contar hasta 1000 con un int8_t (máximo 127) */
int8_t i = 0;
for (i = 1; i <= 1000; i++) { /* i nunca pasa de 127: el bucle se rompe */ }

/* BIEN: un int común alcanza de sobra */
for (int i = 1; i <= 1000; i++) { /* ... */ }
```

```c
#include <stdint.h>
int8_t a = 100, b = 100;
int c = a + b;   /* 200: la suma se calcula en int, y al achicar se desborda */
```

| Situación | Tipo |
|---|---|
| Un número entero común (contar, sumar, cuentas) | `int` |
| Tamaño o conteo ( `sizeof` , `strlen` , `malloc` ) | `size_t` |
| Posición en un arreglo | `size_t` |
| Dato de ancho exacto (archivo, protocolo, hardware) | `intN_t` / `uintN_t` |
| Una condición (sí / no) | `bool` |
| Un carácter fuera de `char` | `char32_t` / `wchar_t` |

> ***Regla corta.*** `int` *por defecto. Cambiá de tipo* ***solo cuando tengas un motivo*** *—ancho exacto, tamaño, signo o un carácter—. Si no hay motivo, quedate con* `int` .

---

## 13.9 ¿Los tipos chicos ahorran memoria?

**Depende de dónde.** Guardar **muchos** valores en tipos chicos sí ahorra memoria; una variable suelta, no.

- **Una variable sola:** no ahorra. El compilador la guarda en un registro o le reserva un espacio alineado, así que `int8_t i;` e `int i;` terminan ocupando lo mismo. Y encima la aritmética se hace en `int` .
- **Un arreglo grande:** sí, mucho. Un millón de elementos:

```c
#include <stdint.h>
int8_t small[1000000];   /* 1 000 000 bytes  (≈ 1 MB) */
int    large[1000000];   /* 4 000 000 bytes  (≈ 4 MB) */
```

- **Como campos de una `struct`:** puede achicar la estructura (el tipo chico ocupa menos), aunque el padding puede comerse parte del ahorro. Conviene medir con `sizeof`.

```c
#include <stdint.h>
struct Color1 { uint8_t r, g, b; };   /* 3 bytes  */
struct Color2 { int r, g, b; };       /* 12 bytes */
```

> ***Cuándo vale la pena un tipo chico.*** *Cuando guardás* ***muchos*** *valores y cada byte cuenta: imágenes, audio, sensores, buffers leídos de un archivo o de la red, tablas grandes. Ahí* `uint8_t` *o* `int16_t` *hacen una diferencia real.*

> ***Lo que NO te da:*** *velocidad. La CPU trabaja con palabras (* `int` *), así que operar con tipos chicos suele ser igual o más lento (hay que extender el signo). El motivo para usarlos es* ***memoria*** *(y el ancho exacto), no rendimiento.*

---

## 13.10 Errores comunes

- Comparar un `int` negativo con `sizeof` o `strlen` (el `-1` se vuelve `SIZE_MAX`).
- Guardar un tamaño (`sizeof`, tamaño de un archivo) en un `int`.
- `for (size_t i = n - 1; i >= 0; i--)`: nunca termina.
- Asumir `sizeof(wchar_t)`, o que `int` == `int32_t`.
- Imprimir `size_t` con `%d` en vez de `%zu`.
- Confundir `'\0'`, `0` y `NULL`.

---

## 13.11 Ejercicios

> *De **noob** a **pro**, con dificultad creciente. Cada ejercicio declara sus conceptos objetivo; resolvelos **usando esos conceptos**. Trae un **escenario de prueba** (entrada/salida esperada, nunca el algoritmo) para que compares lo que imprimís. Compilá con un estándar moderno (C11 o C23) y avisos activos (`-Wall -Wextra`).*

### Ejercicio 13.1 — Radiografía de tipos

**Conceptos** · `sizeof`, `size_t`, `printf` (`%zu`), tipos de `<stddef.h>` y `<stdint.h>`

**Contexto** · Vas a portar un programa a otra máquina y querés saber, antes de confiar en él, qué tamaño tiene cada tipo. Simplificación: solo imprimís tamaños.

**Consigna** · Imprimí una tabla con el tamaño en bytes de: `char`, `int`, `long`, `void *`, `size_t`, `wchar_t`, `bool`, `int32_t` y `uint8_t`. Usá `%zu` para el tamaño e incluí la cabecera de cada tipo.

**Entrada** · Ninguna. **Salida** · Una fila por tipo, con nombre y tamaño.

**Escenario de prueba** ·

```
LP64 (Linux/macOS 64-bit)          LLP64 (Windows 64-bit)
  int .......... 4                   int .......... 4
  long ......... 8   <--              long ......... 4   <-- ¡ojo!
  void* ........ 8                   void* ........ 8
  size_t ....... 8                   size_t ....... 8
  wchar_t ...... 4   <--             wchar_t ...... 2   <-- ¡ojo!
  bool ......... 1                   bool ......... 1
  int32_t ...... 4   (siempre)       int32_t ...... 4   (siempre)
  uint8_t ...... 1   (siempre)       uint8_t ...... 1   (siempre)
```

**Identificadores sugeridos** · `sizeOfTypes`, `printSize` **Criterios de aceptación** ·

- Cada tamaño sale de un `size_t` y se imprime con `%zu`.
- Compila **sin** avisos de conversión.
- Se explica en un comentario por qué `long` y `wchar_t` pueden diferir entre plataformas.

## Ejercicio 13.2 — Sumar con `size_t`

**Conceptos** · `size_t`, arreglos, `for`, casos borde

**Contexto** · Un carrito suma precios que ya vienen en un arreglo. Simplificación: no hay entrada interactiva.

**Consigna** · Escribí una función que reciba un arreglo de `int` y su **cantidad** (`size_t`) y devuelva la suma. Recorré con un índice `size_t`. Con cantidad `0`, devolvé `0` sin leer fuera.

**Entrada** · Una cantidad y esa cantidad de valores. **Salida** · La suma.

**Escenario de prueba** ·

```
count = 3 ; [10, 20, 30] -> 60
count = 0 ; (sin valores) -> 0     (no se accede a memoria)
count = 1 ; [-5]          -> -5
```

**Identificadores sugeridos** · `sumPrices`, `count`, `index` **Criterios de aceptación** ·

- El índice y la cantidad son `size_t` (no `int`).
- `count == 0` devuelve `0` sin desreferenciar nada.
- No hay conversiones de `size_t` a `int`.

## Ejercicio 13.3 — El índice que se convierte en gigante

**Conceptos** · conversiones con/sin signo, `size_t`, comparaciones

**Contexto** · Revisión de código: un bucle compara un `int` contra una cantidad `size_t` y, cuando el índice llega a `-1`, "funciona" de forma inesperada. Simplificación: se reproduce a mano.

**Consigna** · Compará un `int` que valga `-1` contra un `size_t` que valga `5` y mostrá que `-1 < 5` da **falso**. Luego escribí la versión correcta con `size_t` y explicá en un comentario por qué.

**Entrada** · `i = -1`, `n = 5`. **Salida** · El resultado de la comparación y la explicación.

**Escenario de prueba** ·

```
(int -1) < (size_t 5)  -> falso
explicación: -1 convertido a size_t = SIZE_MAX (18446744073709551615), y SIZE_MAX < 5 es
falso.
```

**Identificadores sugeridos** · `index` , `count` **Criterios de aceptación** ·

- Se reproduce el comportamiento sorprendente.
- La explicación menciona la conversión a `SIZE_MAX` .
- La versión corregida no depende de conversiones implícitas.

---

## Ejercicio 13.4 — Texto que no entra en un byte

**Conceptos** · `wchar_t` , `L""` , `%lc` , `%ls` , `<uchar.h>`

**Contexto** · Un cartel muestra en consola el nombre de una ciudad con acentos. Simplificación: guardás y mostrás caracteres anchos.

**Consigna** · Guardá un carácter no ASCII (por ejemplo `ñ` o `€` ) en un `wchar_t` y una cadena con acentos en un `wchar_t *` . Imprimilos con los especificadores correctos. Además, imprimí `sizeof(wchar_t)` y explicá por qué para Unicode **portable** conviene `char32_t` en lugar de `wchar_t` .

**Entrada** · Uno o más caracteres anchos. **Salida** · El carácter y la cadena, más el tamaño.

**Escenario de prueba** ·

```
setlocale(LC_ALL, "");
wchar_t c = L'ñ';  wchar_t *s = L"corazón";
  c = ñ ; s = corazón
  sizeof(wchar_t) = 2 en Windows (UTF-16) / 4 en Linux (UTF-32)
contraste: guardar 'ñ' en un char y usar %c da basura o un byte suelto.
```

**Identificadores sugeridos** · `wideChar` , `wideString` , `wideSize` **Criterios de aceptación** ·

- Se usa `wchar_t` para el carácter y la cadena, con `%lc` y `%ls`.
- Se configura el locale antes de imprimir.
- Se explica por qué un `char` no alcanza y por qué `char32_t` es más portable.

---

### Ejercicio 13.5 — El entero de ancho exacto (pro)

**Conceptos** · `<stdint.h>`, tipos de ancho exacto, portabilidad

**Contexto** · Leés un header binario de red cuyos campos tienen anchos **exactos** (8, 16, 32, 64 bits). Un `int` te haría depender de la plataforma. Simplificación: escribís y leés los valores, sin red real.

**Consigna** · Definí un pequeño "paquete" con campos `uint8_t`, `uint16_t`, `uint32_t` y `uint64_t`. Asignales valores, imprimilos convirtiendo explícitamente a un tipo estándar de ancho suficiente (por ejemplo `unsigned long long` para el de 64 bits) y comprobá que sobreviven. Explicá por qué `int32_t` en lugar de `int`.

**Entrada** · Valores de distinto ancho. **Salida** · Cada campo impreso, sin avisos.

**Escenario de prueba** ·

```
kind   = 0xAB          (uint8_t)  -> 171
length = 4900          (uint16_t) -> 4900
seq    = 3000000000U   (uint32_t) -> 3000000000
total  = 123456789012345ULL (uint64_t) -> 123456789012345
```

**Identificadores sugeridos** · `Packet`, `kind`, `length`, `seq`, `total` **Criterios de aceptación** ·

- Los tipos son de ancho exacto (`uint8_t` … `uint64_t`).
- Cada valor se imprime con un formato que coincide con su tipo (sin warnings).
- Se explica por qué `int32_t` es mejor que `int` para datos de ancho fijo.

---

## 13.12 Cierre

Los tipos de C no son un adorno: son la forma de decir **qué querés representar** y **de qué tamaño**. `size_t` para tamaños, `uintN_t` / `intN_t` para ancho exacto, `bool` para verdades, `wchar_t` / `char32_t` para texto ancho y `NULL` para el puntero vacío. Cada uno vive en **su cabecera**, y elegir el tipo (y la cabecera) correctos es la mitad del camino hacia código portable.

Regla mental: cada vez que escribas `int`, preguntate **"¿este valor podría superar `INT_MAX`, venir de un `sizeof`, o necesitar un ancho exacto?"**. Si la respuesta es sí, hay un tipo mejor que `int` — y ahora sabés cuál es y dónde vive.

**Resumen: cada tipo y su cabecera.**

| Tipo | Cabecera |
|---|---|
| `size_t`, `wchar_t`, `NULL` | `<stddef.h>` |
| `intN_t` / `uintN_t` | `<stdint.h>` |
| `bool` | `<stdbool.h>` |
| `char16_t`, `char32_t`, `char8_t` | `<uchar.h>` |
| `int`, `long`, ..., `INT_MAX` | `<limits.h>` |
| `float`, `double`, ..., `FLT_MAX` | `<float.h>` |
| `float16_t` ... `float128_t` † | `<stdfloat.h>` |

† `float16_t` ... `float128_t` *son de* ***C23*** *(* `<stdfloat.h>` *) y* ***opcionales****: no están en el C clásico ni los trae cualquier compilador.*

***¿Alguno de estos viene solo, sin incluirlo?*** *A veces sí, de yapa. Como* `<stdio.h>` ***usa*** `size_t` *(por ejemplo, en* `fread` *) y devuelve* `NULL` *cuando algo falla (por ejemplo,* `fopen` *), esa cabecera está* ***obligada a traer*** `size_t` *y* `NULL`*. Así que si ya incluiste* `<stdio.h>`*,* `<stdlib.h>` *o* `<string.h>`*, esos dos ya están disponibles.*

***Pero no te acostumbres a la yapa.*** *El estándar solo te garantiza un tipo si incluís* ***su*** *cabecera; cuáles trae de arrastre cada header cambia de compilador en compilador. Regla de oro:* ***incluí siempre la cabecera del tipo que usás.*** *Si tu programa compila "de casualidad" porque otro header lo trajo, se rompe el día que cambies de compilador.*

Así se ve la buena costumbre:

```c
#include <stdio.h>   /* para printf             */
#include <stdint.h>  /* para uint32_t: su cabecera */

int main(void) {
    uint32_t n = 42;
    printf("%u\n", (unsigned)n);
    return 0;
}
```