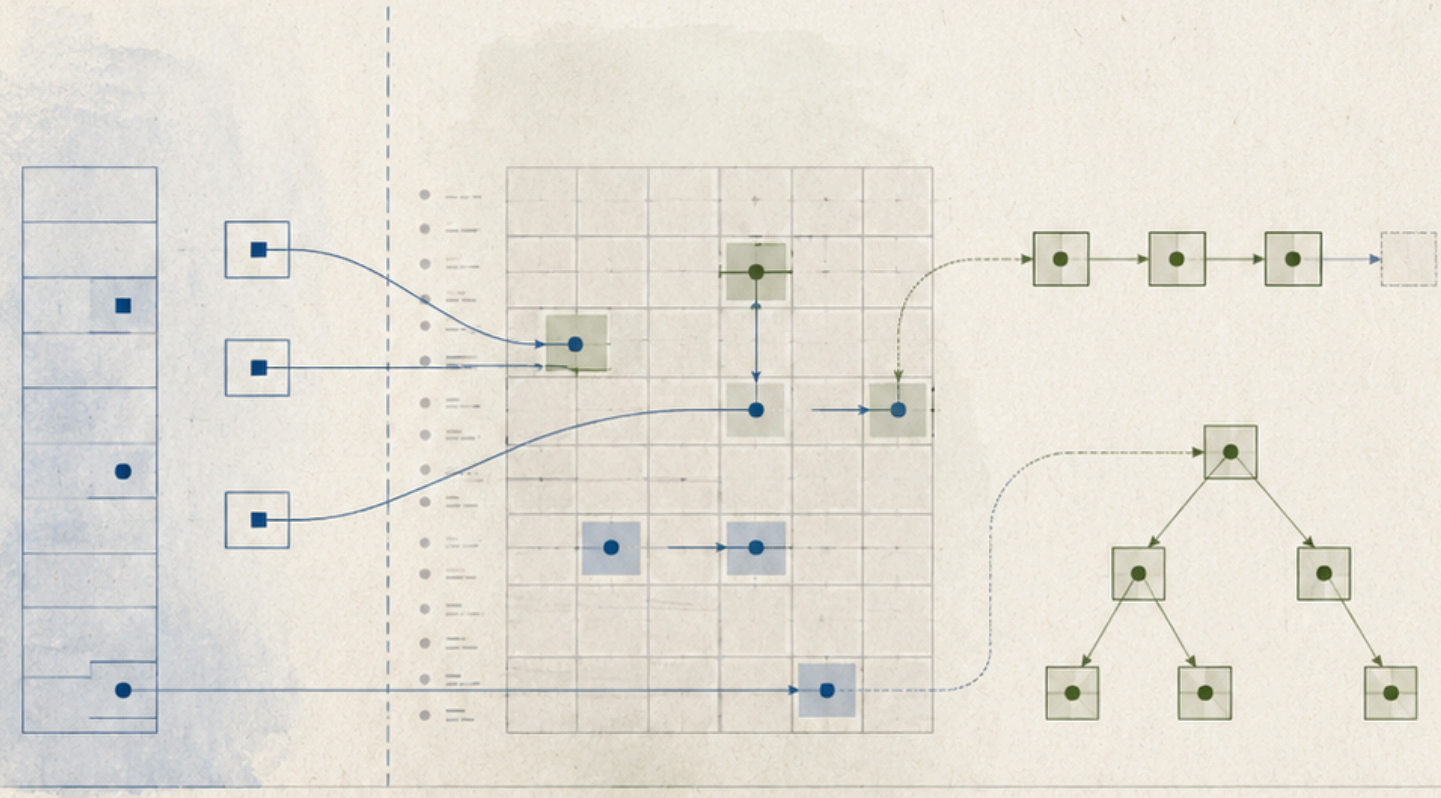
Punteros 101

Una introducción a punteros y memoria en C

- Principios de Programación
- Estructura de Datos y Algoritmos



Mariano Zunino

Nota del autor

Este libro no pretende ser una fuente de autoridad, ni mucho menos. Su propósito es transmitir, de forma ordenada, aquellos conceptos que suelen dificultar el pasaje de `Principios de Programación` a `Estructura de Datos y Algoritmos`.

La intención no es crear un «manual» que cubra todo el lenguaje C, ni mucho menos reemplazar a los textos de referencia. Más bien, este material puede entenderse como un puente entre ambos cursos, apoyado en ejemplos, diagramas de memoria y errores frecuentes que suelen aparecer al estudiar **punteros** por primera vez.

Los punteros tienen fama de ser un tema difícil, a menudo acompañado de cierto temor. Esa percepción es habitual, pero la verdad es que no se corresponde necesariamente con el concepto en sí. Lo que vuelve difícil al tema no es la idea de fondo, que es sencilla, sino el momento y la forma en que suele introducirse.

Muchas veces, los punteros aparecen por primera vez al estudiar memoria dinámica. Eso lleva a aprenderlos directamente en un contexto más complejo, donde además de entender qué es un puntero, también hay que comprender para qué sirve, cómo se usa correctamente y qué errores pueden aparecer. En ese escenario, no sorprende que el tema resulte abrumador.

Este libro propone otro orden. Se parte de un problema concreto que puede entenderse sin saber nada de punteros —querer modificar una variable desde otra función— y se introduce el puntero como la herramienta que naturalmente lo resuelve. Esa progresión ya está implícita en el modo en que se enseña `Principios de Programación`.

Si algo en el camino puede mejorarse, los comentarios son bienvenidos en mariano.zunino@fing.edu.uy.

Créditos

Este material toma como una de sus referencias principales el trabajo *A Tutorial on Pointers and Arrays in C*, de Ted Jensen.

El texto original de Ted Jensen declara el material como dominio público en su prefacio. El repositorio preservado que se consulta actualmente aparece publicado con licencia GPL-3.0. Por prudencia, cualquier fragmento traducido o adaptado directamente debe conservar atribución clara.

Referencias:

- Ted Jensen, *A Tutorial on Pointers and Arrays in C*.
- Repositorio preservado: https://github.com/jflaherty/ptrtut13

Licencia:

Comentarios y correcciones: mariano.zunino@fing.edu.uy.

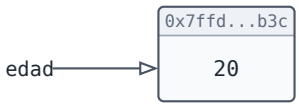
Indice

1 Introducción

En `Principios de Programación` se suele escribir código que usa variables, funciones, arreglos y estructuras. En muchos casos alcanza con pensar que una variable tiene un nombre y un valor:

```c
int edad = 20;
```

Esa descripción es suficiente para muchos programas iniciales. Sin embargo, en C también es importante reconocer que esa variable **ocupa un espacio concreto de memoria**. Dicho espacio está representado por una **dirección**:



La forma exacta de una dirección no es importante por ahora. Alcanza con retener que toda variable **ocupa** un lugar en **memoria** y que ese lugar **tiene una dirección** asociada.

A partir de esta noción puede introducirse la de puntero: un puntero es una variable que almacena la dirección de otra. En lugar de guardar un valor, guarda la ubicación donde ese valor está disponible.

La diferencia radica en distinguir dos cosas:

- una variable común guarda un valor
- un puntero guarda una dirección

Esa dirección abre una vía indirecta de acceso: a partir de ella, el programa puede llegar al valor guardado en otro espacio de memoria. Esta vía indirecta permite, entre otras cosas, escribir funciones que modifican variables externas y construir estructuras que crecen mientras el programa corre. Esas dos aplicaciones aparecen más adelante en el libro y son la razón principal por la cual los punteros importan al pasar a Estructura de Datos y Algoritmos.

Una analogía posible es la aplicación de contactos de un celular. Si pensamos la aplicación como una especie de «memoria», cada contacto guarda información que permite llegar a una persona. El contacto no es la persona en sí, sino una forma de acceder a ella: por ejemplo, mediante su número de teléfono.

De forma parecida, un puntero no es el dato al que queremos llegar. Es una variable que guarda una dirección, y esa dirección permite acceder al dato que se encuentra en otro lugar de memoria.

1.1 Un problema inicial

Un clasico problema es querer intercambiar dos variables.

Una primera versión podría escribirse así:

```c
#include <stdio.h>

void intercambiar(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(void) {
    int x = 10;
    int y = 20;

    intercambiar(x, y);

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```

Al ejecutar el programa, la salida es:

```
x = 10
y = 20
```

La razón es que los parámetros `a` y `b` son copias. Cuando se llama a `intercambiar(x, y)`, la función recibe el **valor** de `x` y el valor de `y`, pero no recibe las variables originales.

El intercambio ocurre dentro de la función, sobre esas copias. Al terminar la función, las copias dejan de existir. Las variables x e y de main no cambiaron.

1.2 La limitación del paso por valor

En C, los argumentos/parámetros de una función se pasan por valor, es decir la función recibe una copia del dato. Una alternativa sin punteros es hacer que la función retorne el valor modificado y dejar que el llamador/invocador lo reasigne:

```c
int duplicado(int n) {
    n = n * 2;
    return n;
}

x = duplicado(x);
```

Este patrón es útil y aparece con frecuencia, pero su límite se nota cuando una función debe modificar varias variables existentes a la vez, o cuando se quiere operar directamente sobre una estructura ya creada. En esos casos, retornar y reasignar deja de ser práctico.

Cuando se necesita modificar una variable existente desde otra función, no alcanza con pasar su valor. Hace falta pasar información sobre dónde está esa variable. Esa información es su dirección.

La herramienta que permite hacerlo es un puntero.

1.3 Una mejora con punteros

La siguiente versión de código presenta una solución para intercambiar dos variables utilizando punteros:

```c
#include <stdio.h>

void intercambiar(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    int x = 10;
    int y = 20;

    intercambiar(&x, &y);

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```

Ahora el programa imprime:

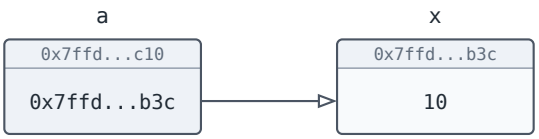
```
x = 20
y = 10
```

Todavía no hace falta entender en detalle cómo funcionan los punteros ni cada símbolo que aparece en el ejemplo. Ese desarrollo se retomará de forma sistemática en el capítulo 3, Punteros básicos. Por ahora alcanza con observar la idea general:

- **`&x`** obtiene la dirección de `x`
- **`&y`** obtiene la dirección de `y`
- **`int *a`** indica que `a` guarda la dirección de un `int`
- **`*a`** permite acceder al `int` apuntado por `a`

En vez de recibir copias de los valores, la función recibe direcciones. Mientras la función se ejecuta, sus parámetros `a` y `b` no son enteros comunes, sino punteros: variables que guardan las direcciones de `x` e `y`.

Dicho de otra forma, `a` permite acceder a `x` y `b` permite acceder a `y`. Por eso, cuando se modifica `*a` o `*b`, no se están modificando copias locales, sino los valores de las variables originales de `main`.



1.4 Qué se espera al terminar este libro

Al terminar este material, se espera que su lector pueda:

- explicar qué es una dirección de memoria
- declarar e inicializar punteros simples
- distinguir entre **`p`**, **`*p`** y **`&p`**
- usar punteros para que una función modifique una variable
- explicar la relación entre arreglos y punteros sin confundirlos
- reservar y liberar memoria dinámica
- leer la estructura básica de un nodo enlazado

La idea es poder razonar sobre qué datos existen, dónde están guardados y qué variables permiten acceder a ellos.

2 Variables, memoria y direcciones

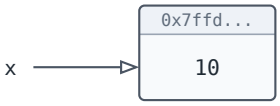
Una variable no es solamente un nombre que aparece en el código. Como anticipó el capítulo anterior, durante la ejecución del programa cada variable existe como un espacio concreto de memoria, y en ese espacio se guarda un valor. Este capítulo desarrolla esa idea y la asocia con a la noción de dirección.

Para razonar sobre punteros conviene separar cuatro ideas:

- el nombre de la variable
- el tipo de dato
- el valor guardado
- la dirección de memoria donde está guardado ese valor

Por ejemplo:

```
int x = 10;
```



En este caso:

- **`x`** es el nombre de la variable
- **`int`** es el tipo de dato
- **`10`** es el valor guardado
- Y, ¿ la dirección ?

La dirección no es algo que el programador suele decidir, sino que es el sistema operativo quien se encarga de esto. En el ejemplo anterior, la variable x podría estar ubicada en cualquier lugar de memoria, por ejemplo **`0x7ffd2c1a8b3c`**.

2.1 Valor y dirección

El valor de una variable y su dirección no son lo mismo.

El valor es el dato que se guardó:

```
printf("%d\n", x); // 10
```

La dirección es el lugar de memoria donde está ese dato:

```
printf("%p\n", &x); // 0x7ffd2c1a8b3c
```

De este manera se introduce el operador **`&`**, que se lee como **`direccion de`** (en inglés, **`address`**), el cual permite obtener la dirección de una variable.

Un programa completo:

```c
#include <stdio.h>

int main(void) {
    int x = 10;

    printf("valor: %d\n", x); // imprimir el valor de X (10)
    printf("direccion: %p\n", &x); // imprimir la dirección de X (0x7ffd2c1a8b3c)

    return 0;
}
```

Una salida posible:

```
valor: 10
direccion: 0x7ffd2c1a8b3c
```

Para imprimir una dirección con `printf`, se usa el formato **`%p`**:

```c
printf("%p\n", &x);
```

No se usa **`%d`** para imprimir direcciones. Una dirección no se trata como un entero regular, tiene su propio tipo, llamado **puntero**, que se introduce en el próximo capítulo.

Por eso se usa `%p`, que habitualmente la muestra en formato hexadecimal.

Notar que la dirección de una variable **puede cambiar entre ejecuciones**. Un error frecuente es pensar que una dirección concreta siempre sera la misma. Es por eso que usamos el operador `&`, para asegurar la obtenibilidad de la dirección concreta.

2.2 Direcciones distintas

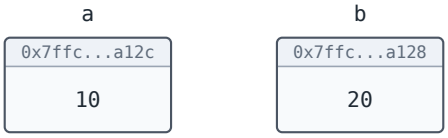
Dos variables distintas ocupan espacios de memoria distintos.

```c
#include <stdio.h>

int main(void) {
    int a = 10;
    int b = 20;

    printf("a: valor %d, direccion %p\n", a, &a);
    printf("b: valor %d, direccion %p\n", b, &b);

    return 0;
}
```

Una salida posible:

```
a: valor 10, direccion 0x7ffc8f01a12c
b: valor 20, direccion 0x7ffc8f01a128
```

Visualmente, las dos variables ocupan cajas separadas en memoria:



Los valores son distintos y las direcciones también. No importa cuánto difieren las direcciones ni en qué orden quedan: lo único relevante es que sean distintas, porque **cada variable ocupa su propio espacio**.

Incluso si dos variables guardan el mismo valor, siguen siendo variables distintas.

```
int a = 10;
int b = 10;
```

En ese caso, `a` y `b` guardan el mismo valor, pero `&a` y `&b` son direcciones diferentes.

2.3 Direcciones y funciones

La dirección de una variable permite identificar dónde está guardada esa variable. Por eso, en el capítulo anterior, la versión de `intercambiar` que sí lograba modificar las variables del llamador no recibía `x` e `y`, sino **`&x`** y **`&y`**.

Pasar `x` significa pasar el valor.

Pasar `&x` significa pasar la dirección.

El mismo mecanismo está detrás de algo que ya aparece en `Principios de Programación`: el `&` usado en `scanf("%d", &n)`. Al pasarle a `scanf` la dirección de `n`, la función puede llegar a esa variable y escribir allí el valor leído.

Sin el `&`, `scanf` recibiría una copia del valor de `n` y no tendría forma de modificar la variable original.

Esa diferencia es la base para entender cómo una función puede modificar una variable declarada en otro lugar. El operador `*` apareció informalmente en el ejemplo de `intercambiar` del capítulo anterior; su desarrollo se cubre en el capítulo 3.

2.4 Errores frecuentes

Un error frecuente es confundir el valor con la dirección.

```
int x = 10;

printf("%d\n", x);        // imprime el valor
printf("%p\n", &x);       // imprime la direccion
```

Otro error es pensar que una dirección concreta siempre será la misma. En general no lo será. El sistema puede ubicar las variables en direcciones distintas cada vez que se ejecuta el programa.

Otro error frecuente es confundir `&x` con `%p`. El operador `&x` produce una dirección como valor de una expresión; se puede pasar a una función, guardar en un puntero o imprimir. `%p` es solo la forma de imprimirla. Una cosa es la dirección en sí; otra, el formato con que se decide mostrarla.

3 Punteros básicos

Un puntero es una variable que guarda una dirección de memoria, generalmente la de otra variable del programa.

3.1 Declaración

Para declarar un puntero se usa un asterisco (*) entre el tipo y el nombre:

```
int *p;
```

Esto declara una variable `p` que puede guardar la dirección de un `int`. El tipo completo de `p` se lee como «puntero a `int`» y se escribe `int *`. Ese es el tipo que el capítulo anterior dejó pendiente al hablar del operador `&` y de `%p`.

El tipo apuntado importa: marca qué clase de dato se espera encontrar en la dirección guardada. Un `int *` apunta a un `int`; un `char *` apuntaría a un `char`. La consecuencia práctica de esa distinción aparece más adelante, al trabajar con arreglos.

En la sintaxis de C, el * se asocia al nombre que sigue, no al tipo. Esto no es una preferencia de estilo, sino una regla del lenguaje: el compilador siempre liga el * al nombre, sin importar los espacios. Por eso conviene escribirlo junto al nombre:

```
int *p;
```

Las formas `int* p;` o `int * p;` son equivalentes y solo difieren visualmente. En este libro se usa preferentemente `int *p`, porque hace evidente la regla del lenguaje, especialmente al declarar varias variables en la misma línea:

```
int *p, q;
```

En este caso, `p` queda como puntero a `int`, pero `q` queda como `int` común. Para declarar dos punteros al mismo tipo, el asterisco se repite delante de cada nombre:

```
int *p, *q;
```

3.2 Inicialización

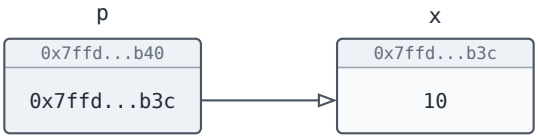
Un puntero recién declarado no apunta automáticamente a algo útil. Antes de usarlo, conviene asignarle una dirección concreta.

Una forma habitual es asignarle la dirección de una variable existente:

```
int x = 10;
int *p = &x;
```

Esa línea hace dos cosas. Declara `p` como puntero a `int` y le asigna una dirección inicial, en este caso la de `x`. Un puntero, como cualquier variable, guarda un valor; lo distintivo es que ese valor es siempre una dirección. A partir de ese momento, se puede decir que `p` apunta a `x`.

La situación puede representarse así:



En cada caja, la franja superior muestra la dirección donde está guardada la variable; la parte inferior, el valor que contiene. La caja de `p` guarda como valor la dirección de `x`. La caja de `x` guarda el entero. La flecha indica la relación «`p` apunta a `x`».

3.3 Desreferenciación

Cuando un puntero apunta a una variable, el programa puede leer y modificar esa variable a través del puntero. Para eso se usa el operador * aplicado al puntero:

```c
int x = 10;
int *p = &x;

printf("%d\n", *p);
*p = 20;
printf("%d\n", x);
```

El operador * aplicado a un puntero se llama **desreferenciación**. La idea es que el puntero refiere al dato de manera indirecta —guarda su dirección, no el dato en sí— y desreferenciar es deshacer esa indirección para llegar al valor concreto. Su efecto es ir a la dirección guardada en el puntero y trabajar con el valor que está allí.

La primera llamada a `printf` muestra el valor de `*p`, es decir, el dato guardado en la dirección apuntada por `p`. Como `p` apunta a `x`, ese valor es `10`. La asignación `*p = 20` cambia ese mismo dato. Como la dirección apuntada es la de `x`, el contenido de `x` pasa a ser `20`, lo que confirma la última llamada a `printf`.

Esta es la misma idea que está detrás del `&` en `scanf("%d", &n);`: al pasarle a `scanf` la dirección de `n`, la función puede modificar esa variable a través de su dirección, igual que `*p = 20` modifica a `x`. El mecanismo se desarrolla en el capítulo siguiente.

3.4 Cómo leer la sintaxis de un puntero

La sintaxis de los punteros se presta a confusión por dos razones complementarias. Por un lado, el mismo símbolo * aparece en contextos distintos con significados diferentes, e incluso fuera de los punteros funciona como operador de multiplicación. Por otro, dado un puntero `p`, hay tres formas que se refieren a cosas distintas.

El primer punto aparece al comparar una **declaración** con una **expresión**:

Como declaración:

```c
int *p;
```

El * pertenece al tipo: declara `p` como puntero a `int`. No se desreferencia nada todavía, porque el puntero recién se está creando.

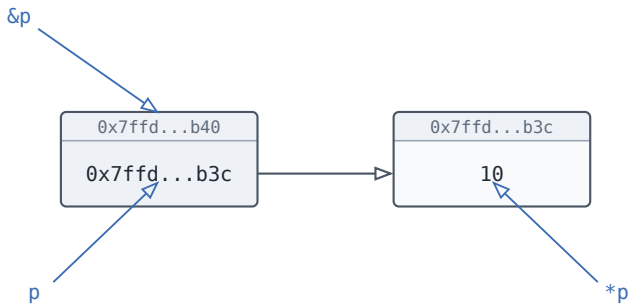
Como expresión:

```c
*p = 10;
printf("%d\n", *p);
```

El * desreferencia `p` y permite trabajar con el `int` guardado en la dirección apuntada: la primera línea le asigna `10`, y la segunda lee ese mismo valor y lo imprime.

Cuando aparece el símbolo *, conviene leerlo según el contexto: junto al tipo en una declaración, indica que la variable es un puntero; en una expresión sobre un puntero existente, accede al valor apuntado.

El segundo punto aparece dentro de una expresión, una vez que el puntero existe. Hay tres formas que se refieren a cosas distintas. Visualmente se ubican así sobre el mismo diagrama:



Cada expresión apunta a algo distinto:

- `p` es el puntero. Su valor es una dirección, la dirección de la variable apuntada.
- `*p` es el valor guardado en esa dirección, es decir, el de la variable apuntada.
- `&p` es la dirección de `p` en sí. Como `p` es una variable, también ocupa memoria y tiene su propia dirección; el operador `&` se aplica a `p` igual que a cualquier otra variable.

El siguiente programa muestra las tres lecturas:

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *p = &x;

    printf("p  = %p\n", p);
    printf("*p = %d\n", *p);
    printf("&p = %p\n", &p);

    return 0;
}
```

Una salida posible:

```
p  = 0x7ffd2c1a8b3c
*p = 10
&p = 0x7ffd2c1a8b40
```

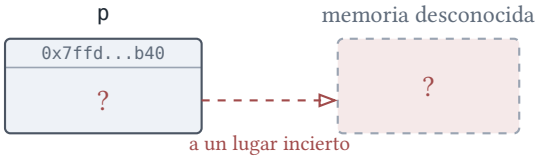
En este caso, `p` contiene la dirección de `x`, `*p` permite acceder al valor de `x` y `&p` es la dirección donde está guardada la variable `p`. Esa dirección es distinta de la dirección de `x`, porque `p` y `x` son dos variables diferentes y cada una ocupa su propio espacio de memoria. Conviene notar también que `%p` sirve para imprimir cualquier dirección: en el capítulo anterior se aplicaba a `&x`; aquí se aplica también a `p` y a `&p`.

3.5 Punteros sin inicializar

Cuando se declara un puntero sin asignarle una dirección, su contenido queda indeterminado. Puede contener cualquier patrón de bits que haya quedado en esa zona de memoria (basura).

```c
int *p;
*p = 10;
```

Visualmente, la situación se parece a esto:



Desreferenciar un puntero sin inicializar es un error grave. El programa puede escribir en una dirección arbitraria. A veces parece funcionar, otras veces termina con una falla de segmentación, y en otros casos corrompe datos sin aviso.

La regla práctica es inicializar el puntero antes de usarlo. Si todavía no se sabe a dónde debe apuntar, se le puede asignar `NULL`, como se presenta en el apartado siguiente.

3.6 El valor `NULL`

`NULL` es una constante definida en encabezados estándar como `<stdio.h>` y `<stdlib.h>`. Representa un puntero que no apunta a ninguna dirección válida.

```
int *p = NULL;
```

Un puntero igual a `NULL` se entiende como «no apunta todavía a nada útil». Es una marca explícita de ausencia.

A diferencia de los diagramas anteriores, aquí no hay caja del lado derecho: el puntero no apunta a ninguna variable. La barra al final de la línea expresa que la flecha no llega a ningún destino.

Conviene escribir `NULL` y no `0` en estos contextos: deja en evidencia que se trata de un puntero y no de un entero común.

`NULL` aparece principalmente en dos situaciones:

- inicializar un puntero cuando todavía no hay una dirección válida para asignarle
- indicar que una función no pudo producir un puntero válido, por ejemplo cuando falla `malloc` (**EDA**)

Un puntero que empieza en `NULL` puede pasar más tarde a apuntar a una variable concreta:

```
int *p = NULL;

int x = 42;
p = &x;
```

Después de la asignación `p = &x;`, el puntero deja de estar en `NULL` y queda apuntando a `x`. Esta secuencia es habitual: el puntero se declara como `NULL` para dejar claro que todavía no apunta a nada, y se le asigna una dirección cuando ya hay algo concreto a lo que apuntar.

Antes de desreferenciar un puntero que puede ser `NULL`, conviene comprobarlo:

```
if (p != NULL) {
    printf("%d\n", *p);
}
```

Desreferenciar `NULL` también es un error y normalmente termina con una falla del programa.

3.7 Errores frecuentes

Un error frecuente es declarar un puntero y desreferenciarlo sin asignarle antes una dirección:

```
int *p;
*p = 5;
```

Otro error es asignarle al puntero un entero arbitrario como si fuera una dirección:

```
int *p = 10;
```

En los ejemplos previos del capítulo, `10` aparecía como el valor entero guardado en `x` (`int x = 10;`). Aquí, en cambio, el `10` se intenta tomar como si fuera una dirección de memoria. Esa dirección no corresponde a ninguna variable del programa. No se asignan números arbitrarios a un puntero.

3.8 Regla práctica

Cada vez que aparece un puntero, conviene separar tres preguntas:

- qué dirección guarda el puntero
- qué valor hay en esa dirección
- dónde está guardado el puntero en sí

Mientras esas tres ideas se mantengan separadas, los punteros se vuelven previsibles. Cuando se mezclan, suelen aparecer los errores típicos descritos arriba.

4 Funciones que modifican datos

En C, al código que invoca a una función se lo llama **llamador/invocador**. Una función no recibe directamente las variables del invocador, sino copias de los valores pasados como argumentos. Esa regla no cambia al usar punteros.

La diferencia es qué valor se copia. Si se copia un entero, la función solo puede cambiar su copia local. Si se copia una dirección, la función también recibe una copia, pero esa copia le permite llegar a una variable existente y modificarla.

Este capítulo desarrolla la idea sobre una misma operación, duplicar un entero, escrita con tres variantes de propósito distinto. Dos son nuevas en este capítulo: `duplicar` recibe un entero por valor y solo modifica su copia local, mientras que `duplicar_en_lugar` recibe una dirección y modifica la variable original a través de un puntero. La tercera, `duplicado`, retoma la versión del capítulo 1 que retorna el resultado y deja al invocador la responsabilidad de reasignar.

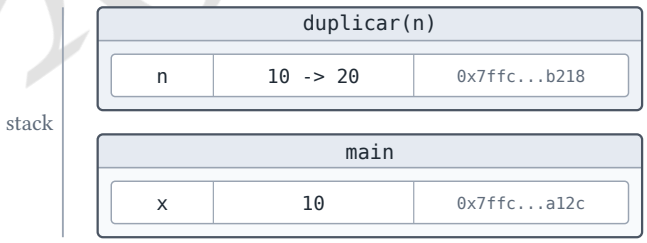
4.1 Parámetros como copias

La primera variante recibe un entero por valor:

```c
void duplicar(int n) {
    n = n * 2;
}
```

Al llamarla con `int x = 10; duplicar(x);`, el valor de `x` se copia en el parámetro `n`. Dentro de la función, `n` empieza valiendo `10`, pero es otra variable. Al asignar `n = n * 2`, cambia la copia. La variable `x` del invocador no cambia.

Visualmente, mientras la función se ejecuta, puede pensarse como dos **stack frames**. Un **stack frame** es el espacio asociado a una llamada de función, donde se ubican sus parámetros y variables locales. En el dibujo, cada fila muestra el nombre de la variable, el valor guardado y una dirección ficticia. La notación `10 -> 20` resume dos momentos de la ejecución en una sola caja: antes y después de la asignación.



El orden vertical de los frames es una convención del diagrama. Lo importante no es si `duplicar` aparece arriba o abajo de `main`, sino que cada llamada tiene su propio espacio. El hecho de que ambas variables tengan el mismo valor no las convierte en la misma variable. `x` y `n` ocupan espacios distintos de memoria. Por eso modificar `n` no modifica `x`.

4.2 Pasar una dirección

Para que una función modifique una variable existente, se le puede pasar la dirección de esa variable.

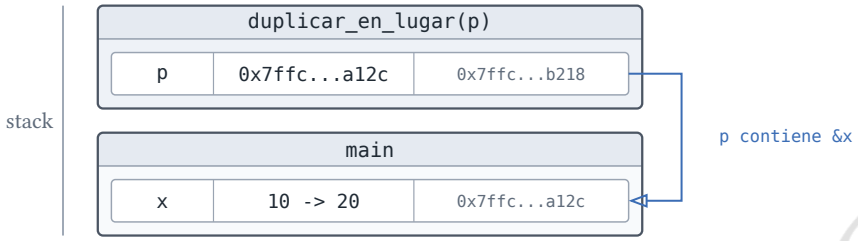
```c
void duplicar_en_lugar(int *p) {
    *p = *p * 2;
}
```

La llamada usa &x:

```c
int x = 10;
duplicar_en_lugar(&x);
```

Lo que se copia en `p` no es el valor entero `10`, sino la dirección de `x`. Como `p` contiene esa dirección, la función puede desreferenciarlo y modificar el valor guardado allí.

La situación dentro de la función puede representarse así:



La variable `p` sigue siendo local de la función. Sin embargo, su valor es la dirección de `x`. Por eso la flecha sale del valor guardado en `p`, no de la dirección donde vive `p`. Al desreferenciar, `*p` no modifica a `p`: modifica el entero guardado en la dirección apuntada por `p`, que en este caso es `x`.

Un programa completo:

```c
#include <stdio.h>

void duplicar_en_lugar(int *p) {
    *p = *p * 2;
}

int main(void) {
    int x = 10;

    duplicar_en_lugar(&x);
    printf("x = %d\n", x);

    return 0;
}
```

La salida es:

```
x = 20
```

La regla práctica es distinguir qué se copia. En `duplicar(x)`, se copia el valor de `x`. En `duplicar_en_lugar(&x)`, se copia la dirección de `x`.

4.3 Modificar mediante desreferenciación

La línea central de la función anterior es:

```c
*p = *p * 2;
```

La expresión de la derecha, `*p * 2`, lee el valor apuntado por `p` y lo multiplica por dos. Como `p` apunta a `x`, lee el valor de `x`.

La expresión de la izquierda, `*p`, indica dónde se guardará el resultado. Como `p` apunta a `x`, la asignación escribe en la variable `x`.

No se está cambiando la dirección guardada en `p`. Se está cambiando el valor ubicado en esa dirección. Esa diferencia es la misma que se trabajó en el capítulo anterior al separar `p` de `*p`.

4.4 El caso de `scanf`

El uso de `&` en `scanf` responde al mismo mecanismo.

```
int n;
scanf("%d", &n);
```

`scanf` necesita escribir el número leído dentro de la variable `n`. Para hacerlo, no alcanza con recibir el valor actual de `n`, que además puede no estar inicializado. Necesita recibir la dirección donde debe guardar el dato. Por eso se le pasa `&n`.

Una forma simple de pensarlo es esta:

- `printf` recibe valores para mostrarlos
- `scanf` recibe direcciones donde escribir los valores leídos

El mismo razonamiento explica un detalle que suele generar dudas: cuando una variable ya es un puntero, no se agrega otro `&` para pasarla a una función que espera una dirección.

Por ejemplo:

```
void leer_entero(int *destino) {
    scanf("%d", destino);
}
```

Una llamada posible es:

```
int n;
leer_entero(&n);
```

Al entrar en `leer_entero`, el parámetro `destino` recibe una copia de `&n`. Por lo tanto, `destino` ya contiene la dirección donde `scanf` debe escribir. La llamada usa `destino`, no `&destino`.

Si se escribiera `&destino`, se estaría tomando la dirección del parámetro local `destino`, es decir, la dirección donde está guardado el puntero dentro de la función. Ese no es el lugar donde se quiere guardar el entero leído.

En términos de tipos, `destino` es una dirección de `int`. En cambio, `&destino` es la dirección de un puntero a `int`: agrega otro nivel de indirección. Ese caso se conoce como **puntero doble** (puntero a puntero) y queda fuera del alcance de este libro. Para leer un `int` con `scanf`, la dirección correcta es la que ya está guardada en `destino`.

4.5 Intercambiar dos variables

El ejemplo clásico de esta idea es intercambiar dos variables.

```
void intercambiar(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

La función recibe dos punteros. Por eso, la llamada debe pasar dos direcciones:

```
intercambiar(&x, &y);
```

El cuerpo puede leerse paso a paso:

- `int tmp = *a` guarda una copia del valor apuntado por `a`
- `*a = *b` copia el valor apuntado por `b` en el lugar apuntado por `a`
- `*b = tmp` guarda el valor original de `*a` en el lugar apuntado por `b`

Un programa completo queda así:

```
#include <stdio.h>

void intercambiar(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    int x = 10;
    int y = 20;

    intercambiar(&x, &y);

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```

La salida es:

```
x = 20
y = 10
```

La función no retorna los dos valores intercambiados. Los escribe directamente en las variables del invocador usando las direcciones recibidas.

4.6 Retornar un valor o modificar mediante punteros

No toda función que calcula algo necesita punteros. Si una función produce un único resultado nuevo, suele ser más claro retornarlo:

```
int duplicado(int n) {
    return n * 2;
}
```

Luego, el invocador decide dónde guardar ese resultado:

```
x = duplicado(x);
```

Los punteros son convenientes cuando la función necesita modificar datos existentes:

- cambiar más de una variable del invocador
- escribir sobre una estructura ya creada
- cargar un dato leído desde la entrada

- preparar operaciones sobre arreglos y, más adelante, sobre estructuras que crecen mientras el programa corre, como listas

Este criterio aparece mucho en EDA. Una función que modifica una lista, una pila o una cola necesita operar sobre una estructura existente, no solo calcular un valor aislado y devolverlo.

4.7 Errores frecuentes

Un error frecuente es olvidar el & al llamar a una función que espera un puntero:

```
int x = 10;
duplicar_en_lugar(x);  // incorrecto
```

La función espera una dirección (`int *`), pero la llamada le entrega un entero (`int`). La llamada correcta es:

```
duplicar_en_lugar(&x);
```

Otro error es cambiar el puntero cuando se quería cambiar el valor apuntado. La asignación importante lleva *:

```
*p = *p * 2;
```

Sin el *, se trabaja con el puntero mismo, no con el entero apuntado. Por ejemplo:

```
int y = 99;
p = &y;   // cambia hacia dónde apunta p
```

Esa asignación puede ser válida, pero no modifica el valor de `x` ni duplica ningún entero. Solo cambia la dirección guardada en la copia local del puntero.

Como `p` es un parámetro, ese cambio se pierde al salir de la función. Para modificar el puntero original del invocador haría falta pasar también la dirección de ese puntero, lo que introduce otro nivel de indirección. Ese tema queda fuera de los ejemplos centrales del libro.

4.8 Regla práctica

Al leer una llamada a función con punteros, conviene separar dos preguntas:

- qué valor recibe cada parámetro
- qué variable puede modificarse al desreferenciar ese parámetro

En C, el parámetro sigue siendo una copia. La diferencia está en que una copia de una dirección permite llegar a la variable original.

5 Punteros, arreglos y cadenas

Los arreglos ya aparecen en PP como una forma de guardar varios valores del mismo tipo bajo un mismo nombre. Al estudiar punteros, conviene mirar el mismo concepto desde la memoria.

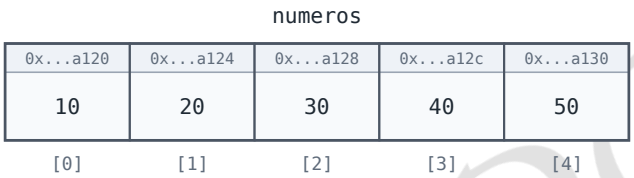
Un arreglo ocupa un bloque contiguo: sus elementos están guardados uno a continuación del otro. Esa propiedad explica por qué los punteros se relacionan tan estrechamente con los arreglos en C.

5.1 Arreglos como bloques contiguos

Conviene comenzar con un arreglo chico:

```c
int numeros[5] = {10, 20, 30, 40, 50};
```

En memoria, puede representarse así:



Cada celda corresponde a un elemento del arreglo. La parte inferior muestra el índice usado por el programa: `numeros[0]`, `numeros[1]`, `numeros[2]`, y así sucesivamente. La franja superior muestra direcciones ficticias.

No importa memorizar esos números. Lo importante es reconocer dos ideas:

- el arreglo ocupa un bloque contiguo
- cada elemento tiene su propia dirección

Por ejemplo, `&numeros[0]` es la dirección del primer elemento, `&numeros[1]` es la dirección del segundo y `&numeros[2]` es la dirección del tercero.

La distancia entre esas direcciones depende del tamaño de cada elemento. En un arreglo de `int`, las direcciones suelen avanzar de a varios bytes. En un arreglo de `char`, avanzan de a un byte. Esa diferencia no cambia la idea central: el siguiente elemento está a continuación del anterior.

5.2 El nombre del arreglo

En muchas expresiones, el nombre de un arreglo se convierte en un puntero al primer elemento. Esa conversión no ocurre siempre. En este capítulo conviene reconocer dos excepciones importantes: cuando se usa `&numeros`, porque se toma la dirección del arreglo completo, y cuando se usa `sizeof numeros` en el mismo lugar donde el arreglo fue declarado, porque allí `sizeof` mide el bloque completo.

Por eso, en un programa como este:

```c
#include <stdio.h>

int main(void) {
    int numeros[5] = {10, 20, 30, 40, 50};

    printf("%p\n", numeros);
    printf("%p\n", &numeros[0]);
    printf("%d\n", *numeros);

    return 0;
}
```

las dos primeras líneas suelen imprimir la misma dirección. La primera usa `numeros`, que en esa expresión se convierte en un puntero al primer elemento. La segunda escribe esa misma dirección de forma explícita: `&numeros[0]`.

La tercera línea desreferencia ese puntero. Como `numeros` apunta al primer elemento, `*numeros` es lo mismo que `numeros[0]`, es decir, `10`.

Esto no significa que el arreglo sea un puntero. Significa que, en muchas expresiones, C convierte el nombre del arreglo en una dirección útil para acceder a sus elementos. El arreglo completo sigue siendo el bloque de cinco `int`.

5.3 `numeros`, `&numeros[0]` y `&numeros`

Hay tres expresiones parecidas que conviene distinguir:

```c
numeros
&numeros[0]
&numeros
```

Las dos primeras apuntan al primer `int` del arreglo cuando se usan en expresiones comunes. Por eso, para recorrer elementos, se comportan como direcciones equivalentes.

La tercera, `&numeros`, toma la dirección del arreglo completo. Esa dirección puede tener el mismo número impreso que la dirección del primer elemento, porque el bloque empieza en el mismo lugar. Sin embargo, su significado no es el mismo: no apunta a un `int`, sino al arreglo entero.

La flecha de `numeros` y `&numeros[0]` llega al primer elemento. La marca de `&numeros`, en cambio, abarca el bloque completo. Por eso el número de dirección puede coincidir, pero el tipo de la expresión es distinto: `numeros` se usa como puntero a `int`; `&numeros` es un puntero a un arreglo de cinco `int`.

En una primera lectura, alcanza con esta regla práctica: para obtener la dirección del primer elemento, se usa `numeros` o `&numeros[0]`. La expresión `&numeros` existe, pero no es la forma habitual de recorrer el arreglo elemento por elemento.

5.4 Arreglos como parámetros de función

Cuando un arreglo se pasa a una función, no se copia todo el bloque. La función recibe una dirección al primer elemento.

Por eso estas dos declaraciones de parámetro son equivalentes para el compilador:

```
int sumar(int arr[], int largo);
int sumar(int *arr, int largo);
```

En ambos casos, `arr` se comporta dentro de la función como un puntero al primer elemento. La forma `int arr[]` suele ser más clara cuando la función trabaja conceptualmente con un arreglo. La forma `int *arr` deja más visible que la función recibe una dirección.

Un ejemplo completo:

```
#include <stdio.h>

int sumar(int arr[], int largo) {
    int suma = 0;

    for (int i = 0; i < largo; i++) {
        suma += arr[i];
    }

    return suma;
}

int main(void) {
    int numeros[5] = {10, 20, 30, 40, 50};

    printf("suma = %d\n", sumar(numeros, 5));

    return 0;
}
```

La función recibe la dirección del primer elemento, pero no recibe automáticamente la cantidad de elementos. Por eso se pasa también `largo`.

Esto es una diferencia importante con la variable local `numeros` en `main`. Allí el compilador conoce que `numeros` es un arreglo de cinco `int`. Dentro de `sumar`, en cambio, `arr` es un parámetro que se comporta como puntero, y la función no puede deducir por sí sola dónde termina el arreglo.

5.5 Recorrer con índices

La forma más común de recorrer un arreglo sigue siendo la de PP:

```
for (int i = 0; i < largo; i++) {
    printf("%d\n", arr[i]);
}
```

Aunque se escriba con índices, la operación usa la misma base: partir de la dirección del primer elemento y avanzar hasta el elemento indicado.

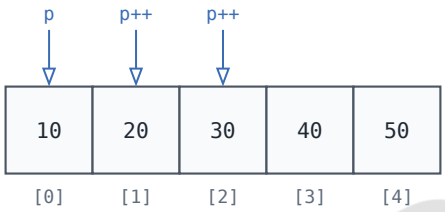
La expresión `arr[i]` puede leerse como «el elemento que está `i` posiciones después del comienzo del arreglo». C sabe cuánto debe avanzar porque conoce el tipo de los elementos. Si `arr` apunta a `int`, avanzar una posición significa avanzar hasta el siguiente `int`, no hasta el siguiente byte cualquiera.

5.6 Recorrer con punteros

También se puede recorrer un arreglo moviendo un puntero:

```
int *p = numeros;

for (int i = 0; i < 5; i++) {
    printf("%d\n", *p);
    p++;
}
```

Al principio, `p` apunta a `numeros[0]`. Después de `p++`, apunta a `numeros[1]`. En la siguiente vuelta apunta a `numeros[2]`, y así sucesivamente.



La suma sobre punteros se interpreta en unidades del tipo apuntado. Si `p` es `int *`, entonces `p + 1` apunta al siguiente `int`. No hace falta calcular manualmente cuántos bytes ocupa cada elemento.

La relación entre índices y punteros puede resumirse así:

```
arr[i]      /* elemento i */
*(arr + i) /* el mismo elemento */
```

La primera forma es la más clara para la mayoría de los recorridos. La segunda muestra lo que C puede hacer gracias a que el arreglo está en memoria contigua.

Un límite importante: solo tiene sentido mover punteros dentro del mismo arreglo, o hasta la posición inmediatamente posterior al último elemento para comparar límites. Desreferenciar fuera del arreglo es un error, aunque el programa compile.

Esa posición posterior sirve como marca de final en recorridos con punteros: se puede comparar si un puntero llegó allí, pero no se debe leer `*p` en esa posición porque no contiene un elemento del arreglo.

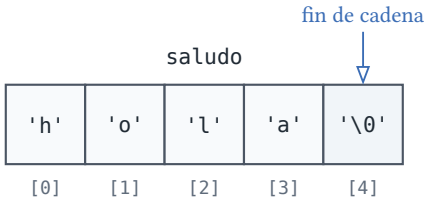
El nombre del arreglo tampoco puede moverse con `++`. Una variable puntero como `p` puede cambiar y pasar a apuntar al elemento siguiente. El nombre `numeros`, en cambio, representa el arreglo declarado y no puede reasignarse ni incrementarse.

5.7 Cadenas como arreglos de `char`

Una cadena de caracteres en C es un arreglo de `char` terminado por el carácter especial `'\0'`.

```
char saludo[] = "hola";
```

El arreglo contiene cinco caracteres, no cuatro:



Los primeros cuatro son las letras visibles. El quinto es `'\0'`, que marca el final de la cadena. Las funciones que trabajan con cadenas, como `printf` con `%s`, avanzan carácter por carácter hasta encontrar esa marca.

Por eso, en esta llamada:

```
printf("%s\n", saludo);
```

no se pasa el largo de la cadena. `printf` recibe la dirección del primer `char` y sigue leyendo hasta `'\0'`.

La misma idea permite escribir funciones que recorren cadenas:

```
int longitud(char texto[]) {
    int i = 0;

    while (texto[i] != '\0') {
        i++;
    }

    return i;
}
```

Aquí `texto` se comporta como puntero al primer carácter. La función no necesita recibir el largo porque la propia cadena contiene una marca de final.

Si falta el `'\0'`, las funciones de cadenas no tienen una señal clara para detenerse. En ese caso pueden seguir leyendo memoria más allá del arreglo hasta encontrar un byte cero por casualidad, o pueden fallar. Por eso una cadena en C no es solo un arreglo de `char`: además debe estar correctamente terminada.

5.8 Errores frecuentes

Un error frecuente es resumir la relación diciendo «los arreglos son punteros». Esa frase oculta una diferencia importante. El arreglo es el bloque completo de elementos. En muchas expresiones, su nombre se convierte en un puntero al primer elemento.

Otro error es creer que una función puede conocer el largo de cualquier arreglo recibido como parámetro:

```
int contar(int arr[]) {
    return sizeof(arr) / sizeof(arr[0]);  // incorrecto
}
```

Dentro de la función, `arr` se comporta como puntero. Por eso `sizeof(arr)` no mide el arreglo original completo. Para trabajar correctamente, la función debe recibir el largo como parámetro, o usar alguna marca de final cuando el formato lo permita, como ocurre con `'\0'` en las cadenas.

También es un error escribir fuera de los límites del arreglo:

```
int numeros[5] = {10, 20, 30, 40, 50};
numeros[5] = 60;  // incorrecto
```

El último índice válido es `4`. La posición `5` está inmediatamente después del arreglo. Escribir allí modifica una zona que no pertenece al arreglo y produce comportamiento indefinido.

5.9 Regla práctica

Al leer código con arreglos y punteros, conviene separar tres ideas:

- el arreglo como bloque completo de elementos
- la dirección del primer elemento
- el largo del arreglo

El nombre del arreglo muchas veces permite obtener la segunda idea, pero no reemplaza a las otras dos. Para recorrer con seguridad, además de una dirección, hace falta conocer el límite.

6 Punteros a structs

Los `struct` permiten agrupar varios datos bajo un mismo tipo. En PP suelen aparecer como una forma de representar entidades del problema: un alumno, una fecha, un producto, una posición.

Al incorporar punteros, se puede trabajar con la dirección de un `struct` completo. Esto permite cambiar una estructura existente desde una función y prepara una idea central de EDA: construir nodos que contienen datos y direcciones hacia otros nodos.

6.1 Un struct en memoria

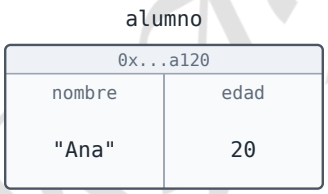
Conviene comenzar con un `struct` simple:

```
struct Alumno {
    char nombre[40];
    int edad;
};
```

Una variable de ese tipo agrupa sus campos en un mismo objeto:

```
struct Alumno alumno = {"Ana", 20};
```

En memoria puede pensarse así:



La variable `alumno` ocupa un espacio concreto. Dentro de ese espacio están sus campos: `nombre` y `edad`. La atención está puesta en la dirección del `struct` completo, porque esa es la que se guarda en un puntero a `struct`.

Para acceder a un campo cuando se tiene la variable directamente, se usa el punto:

```
printf("%s\n", alumno.nombre);
printf("%d\n", alumno.edad);
```

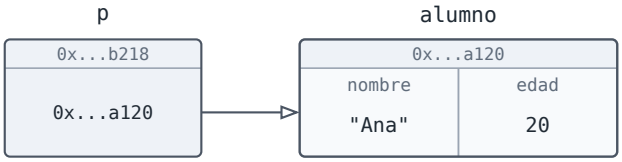
El operador `.` dice: tomar este `struct` y acceder a uno de sus campos.

6.2 Un puntero a struct

También se puede guardar la dirección de un `struct` en un puntero:

```
struct Alumno *p = &alumno;
```

El tipo `struct Alumno *` se lee como «puntero a `struct Alumno`». El valor guardado en `p` es la dirección de la variable `alumno`.



La situación es la misma que con `int *p`, pero el objeto apuntado no es un entero. Es una estructura completa, con campos internos.

Por eso, si se desreferencia `p`, se obtiene el `struct` apuntado:

```
*p
```

Esa expresión permite acceder al `struct Alumno` guardado en la dirección contenida en `p`. Desreferenciar `p` no crea una copia nueva por sí solo: permite trabajar con el objeto apuntado. Si luego se asignara ese resultado a otra variable de tipo `struct Alumno`, recién ahí se copiaría el `struct`.

6.3 Acceder a campos desde un puntero

Si `p` apunta a un `struct`, una forma de acceder a un campo es desreferenciar primero y luego usar el punto:

```
printf("%d\n", (*p).edad);
```

Los paréntesis son importantes. La expresión `*p` obtiene el `struct` apuntado. Luego `.edad` accede al campo `edad` de ese `struct`.

Sin los paréntesis, la expresión no significa lo mismo:

```
*p.edad   // incorrecto
```

El operador `.` se aplica antes que `*`. No hace falta memorizar ahora una tabla completa de prioridades; alcanza con reconocer este caso. Sin paréntesis, la expresión se agrupa como `*(p.edad)`, es decir, primero `p.edad` y después la desreferenciación. Pero `p.edad` no tiene sentido: `p` no es un `struct`, es un puntero a `struct`, y el operador `.` solo se aplica a `struct`. Por eso la forma correcta con desreferenciación explícita es:

```
(*p).edad
```

La lectura puede hacerse en dos pasos:

- `p` contiene la dirección de `alumno`
- `*p` es el `struct` apuntado
- `(*p).edad` es el campo `edad` de ese `struct`

Como `(*p).edad` accede al `struct` original, una asignación sobre ese campo también modifica el original:

```
(*p).edad = 21;
```

Como acceder a campos mediante punteros es muy común, C ofrece una escritura más directa:

```
p->edad
```

Esta expresión significa lo mismo que:

```
(*p).edad
```

El operador `->` combina dos acciones: seguir el puntero y acceder al campo. Por eso se usa cuando se tiene un puntero a `struct`.

Un programa completo:

```c
#include <stdio.h>

struct Alumno {
    char nombre[40];
    int edad;
};

int main(void) {
    struct Alumno alumno = {"Ana", 20};
    struct Alumno *p = &alumno;

    printf("%s: %d\n", p->nombre, p->edad);

    return 0;
}
```

La salida es:

```
Ana: 20
```

En `p->nombre`, primero se sigue el puntero `p` para llegar al `struct` y luego se accede al campo `nombre`, que es un arreglo de `char`. Recién allí se aplica la misma idea del capítulo anterior: al pasarlo a `printf` con `%s`, el nombre del arreglo se usa como dirección del primer carácter.

6.4 Modificar un struct desde una función

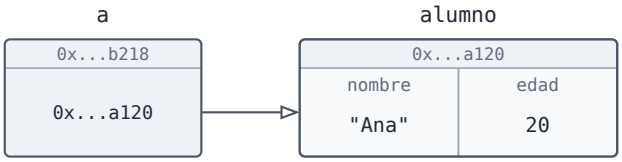
Una función puede recibir un puntero a `struct` para modificar una estructura existente:

```c
void cumplir_anios(struct Alumno *a) {
    a->edad++;
}
```

La llamada pasa la dirección del `struct`:

```c
cumplir_anios(&alumno);
```

El parámetro `a` recibe una copia de la dirección de `alumno`. Al escribir `a->edad++`, la función no modifica una copia local del `struct`: modifica el campo `edad` del `struct` apuntado.



La caja de `a` corresponde al parámetro local de la función. Su valor es una dirección. Esa dirección apunta al mismo `alumno` declarado en `main`.

Un programa completo:

```c
#include <stdio.h>

struct Alumno {
    char nombre[40];
    int edad;
};

void cumplir_anios(struct Alumno *a) {
    a->edad++;
}

int main(void) {
    struct Alumno alumno = {"Ana", 20};

    cumplir_anios(&alumno);
    printf("%d\n", alumno.edad);

    return 0;
}
```

La salida es:

```
21
```

La idea es la misma que en las funciones que modificaban enteros mediante punteros. La diferencia es que ahora el valor apuntado tiene campos, y se elige cuál campo modificar.

6.5 Retornar una copia o modificar el original

También se puede escribir una función que reciba un `struct` por valor y retorne una copia modificada:

```c
struct Alumno con_un_anio_mas(struct Alumno a) {
    a.edad++;
    return a;
}
```

En ese caso, el parámetro `a` es una copia completa del `struct` recibido. La función modifica la copia y la retorna. Para que el cambio quede en la variable original, el llamador debe guardar el resultado:

```c
alumno = con_un_anio_mas(alumno);
```

Esta forma puede ser clara cuando la intención es producir un nuevo valor y dejar que el llamador decida dónde guardarlo. En cambio, el puntero es útil cuando se quiere modificar una estructura existente en su lugar, cuando evitar la copia importa, o cuando la estructura forma parte de algo mayor.

En EDA esto aparece constantemente. Por ahora, alcanza con pensar un nodo como un `struct` que guarda un dato y una dirección hacia otro nodo. Una función que modifica un nodo no trabaja con una copia aislada: necesita operar sobre el nodo existente, porque ese nodo puede estar enlazado con otros.

6.6 Errores frecuentes

Un error frecuente es usar `.` cuando la variable disponible es un puntero:

```
struct Alumno *p = &alumno;
p.edad = 21;  // incorrecto
```

`p` no es un `struct Alumno`; es un puntero a `struct Alumno`. Para acceder al campo se usa:

```
p->edad = 21;
```

Otro error es olvidar el `&` al llamar a una función que espera un puntero:

```
cumplir_anios(alumno);  // incorrecto
```

La función espera una dirección de `struct Alumno`. La llamada correcta es:

```
cumplir_anios(&alumno);
```

También es un error creer que `->` cambia el puntero. La expresión `a->edad++` no modifica la dirección guardada en `a`. Modifica el campo `edad` del `struct` apuntado por `a`.

Por último, `->` no vuelve seguro a un puntero inválido. Si `p` vale `NULL`, entonces `p->edad` intenta acceder a un `struct` que no existe:

```
struct Alumno *p = NULL;
p->edad = 21;  // incorrecto
```

Antes de usar `->`, el puntero debe apuntar a un `struct` válido.

6.7 Regla práctica

Al leer código con structs y punteros, conviene distinguir qué se tiene a mano:

- si se tiene un `struct`, se accede con `.`
- si se tiene un puntero a `struct`, se accede con `->`
- si una función debe modificar un `struct` existente, se le pasa su dirección

El operador `->` no introduce una idea nueva de memoria. Es una abreviatura para desreferenciar un puntero a `struct` y acceder a uno de sus campos.

7 Memoria dinámica

Hasta ahora, las variables y arreglos usados en los ejemplos se declaraban dentro de una función, existían mientras esa función estaba ejecutándose y dejaban de existir cuando la función terminaba. Ese espacio suele llamarse `stack`. También se lo asocia con memoria automática, porque el programa no pide ni libera cada variable local a mano: ese manejo ocurre al entrar y salir de la función.

Sin embargo, no siempre alcanza con declarar todo de antemano. A veces el programa necesita decidir cuánta memoria usar mientras se ejecuta. También puede necesitar que un dato siga existiendo aunque termine la función que lo creó. Para esos casos se usa memoria dinámica: memoria pedida explícitamente durante la ejecución y liberada explícitamente cuando deja de hacer falta.

La zona de memoria donde se reserva este espacio suele llamarse `heap`.

7.1 El problema del tamaño fijo

Un arreglo declarado así tiene un tamaño fijo:

```
int numeros[5];
```

Ese arreglo contiene cinco enteros. No se puede hacer que el mismo arreglo pase a tener diez elementos más adelante. Si el programa necesita trabajar con una cantidad que se conoce recién durante la ejecución, hace falta otra herramienta.

La idea de la memoria dinámica es pedir un bloque del tamaño necesario:

```
int *numeros = malloc(n * sizeof(int));
```

Todavía conviene leer esa línea por partes:

- `malloc` pide un bloque de memoria
- `n * sizeof(int)` indica cuántos bytes se necesitan para guardar `n` enteros
- el resultado se guarda en un puntero a `int`

`malloc` recibe una cantidad de bytes porque es una herramienta general. No sabe si el bloque se usará para enteros, caracteres, structs o nodos. Esa interpretación aparece después, cuando la dirección se guarda en un puntero de cierto tipo.

Como el bloque no tiene nombre propio, la única forma de llegar a él es conservar la dirección devuelta por `malloc`.

7.2 Pedir memoria con `malloc`

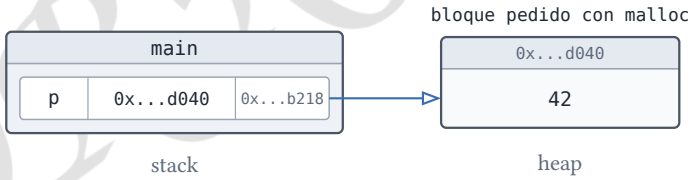
Para usar `malloc` hace falta incluir `stdlib.h`:

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

Un primer ejemplo puede pedir espacio para un solo entero:

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

int main(void) {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        printf("No se pudo reservar memoria\n");
        return 1;
    }

    *p = 42;
    printf("%d\n", *p);

    free(p);
    p = NULL;

    return 0;
}
```

La llamada `malloc(sizeof(int))` pide memoria suficiente para guardar un `int`. Si puede reservarla, devuelve la dirección del bloque. Esa dirección se guarda en `p`.



El puntero `p` es una variable local de `main`. Esa variable está en el `stack` y deja de existir cuando termina `main`. El entero reservado con `malloc`, en cambio, está en el `heap` y seguirá reservado hasta que se llame a `free`. La flecha no significa que el bloque tenga un nombre: significa que `p` guarda la dirección que permite encontrarlo.

Después de comprobar que `p` no es `NULL`, se puede desreferenciar:

```c
*p = 42;
printf("%d\n", *p);
```

La regla es la misma que en capítulos anteriores. Si `p` apunta a un `int` válido, entonces `*p` es el entero apuntado.

`malloc` puede fallar. Por ejemplo, puede no haber memoria suficiente para satisfacer el pedido. En ese caso devuelve `NULL`.

Por eso, después de llamar a `malloc`, se debe comprobar el resultado antes de desreferenciar el puntero:

```
int *p = malloc(sizeof(int));
if (p == NULL) {
    printf("No se pudo reservar memoria\n");
    return 1;
}
```

No alcanza con escribir la llamada y asumir que funcionó. Si `p` vale `NULL`, una expresión como `*p = 42` intenta escribir donde no hay un entero válido.

Esta comprobación tiene la misma forma conceptual que los casos anteriores con `NULL`: primero se verifica que el puntero apunte a algo válido, después se usa.

7.3 Reservar en una función

La diferencia de vida útil se ve mejor cuando una función reserva memoria y devuelve la dirección:

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

int *crear_entero(int valor) {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        return NULL;
    }

    *p = valor;
    return p;
}

int main(void) {
    int *n = crear_entero(42);
    if (n == NULL) {
        printf("No se pudo reservar memoria\n");
        return 1;
    }

    printf("%d\n", *n);
    free(n);
    return 0;
}
```

El puntero local `p` deja de existir cuando termina `crear_entero`. El bloque reservado no. Lo que vuelve al llamador es una copia de la dirección. Por eso `main` puede seguir usando el entero mediante `n`.

En este ejemplo, `main` queda como dueño del bloque: recibió la dirección y tiene la responsabilidad de llamar a `free` cuando ya no la necesita. En programas más grandes, esta regla debe quedar clara entre funciones. Quien recibe un puntero a memoria dinámica debe saber si solo lo usa o si también debe liberarlo.

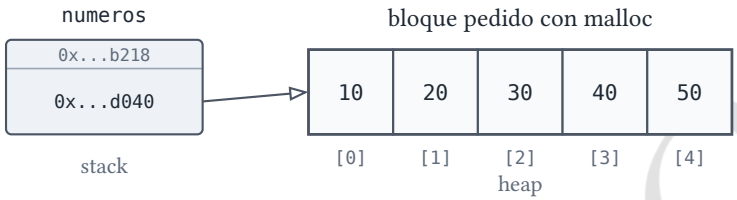
7.4 Pedir un arreglo dinámico

También se puede pedir un bloque para varios elementos:

```
int n = 5;
int *numeros = malloc(n * sizeof(int));
```

El resultado es un puntero al primer `int` del bloque. A partir de ahí, se puede usar notación de arreglo:

```
for (int i = 0; i < n; i++) {
    numeros[i] = (i + 1) * 10;
}
```



Aunque se escriba `numeros[i]`, no hay un arreglo declarado con nombre `numeros` como en `int numeros[5]`. Lo que hay es un puntero que guarda la dirección del primer elemento de un bloque dinámico.

Un programa completo:

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

int main(void) {
    int n = 5;
    int *numeros = malloc(n * sizeof(int));
    if (numeros == NULL) {
        printf("No se pudo reservar memoria\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        numeros[i] = (i + 1) * 10;
    }

    for (int i = 0; i < n; i++) {
        printf("%d\n", numeros[i]);
    }

    free(numeros);
    numeros = NULL;

    return 0;
}
```

La salida es:

```
10
20
30
40
50
```

En el cálculo del tamaño se escribió `sizeof(int)` porque deja visible qué tipo de elementos se quieren guardar. En código C también es común escribir:

```
int *numeros = malloc(n * sizeof(*numeros));
```

Conviene observar que `sizeof(*numeros)` no llega a desreferenciar `numeros` en tiempo de ejecución: `sizeof` solo mira el tipo de la expresión y devuelve el tamaño correspondiente. Por eso la expresión es segura aun cuando `numeros` todavía no apunte a un objeto válido, como ocurre cuando se la usa para llamar a `malloc`.

Esa forma tiene una ventaja: si más adelante cambia el tipo de `numeros`, el tamaño sigue calculándose a partir de lo que el puntero apunta. En una primera lectura, `sizeof(int)` suele ser más directo.

7.5 Liberar memoria con `free`

La memoria pedida con `malloc` no se libera sola cuando deja de usarse. El programa debe devolverla explícitamente:

```
free(numeros);
```

Después de `free`, el bloque deja de pertenecer al programa. La dirección que estaba guardada en el puntero ya no debe usarse para leer ni escribir.

Esto significa que el bloque fue devuelto al administrador de memoria. El número de dirección puede seguir guardado en el puntero, pero el programa ya no tiene derecho a tratar esa zona como si todavía contuviera su dato. Leer o escribir mediante esa dirección produce comportamiento indefinido: puede parecer funcionar, imprimir basura, fallar, o dañar otro dato.

Por eso, cuando el puntero podría volver a usarse por error, conviene asignarle `NULL`:

```
free(numeros);
numeros = NULL;
```

Esto no libera dos veces. La liberación ocurre con `free`. La asignación posterior solo cambia el valor del puntero para que ya no conserve una dirección vieja.

Si el puntero deja de existir inmediatamente, por ejemplo porque la función termina justo después de `free`, no siempre hace falta asignar `NULL`. La regla práctica es usar `NULL` cuando ayuda a evitar un uso accidental posterior.

También es válido llamar a `free(NULL)`: no hace nada. Esto permite escribir código que libera un puntero sin tratar `NULL` como un caso especial. De todos modos, no convierte en seguro liberar dos veces una dirección vieja que no fue reemplazada por `NULL`.

Además de `malloc`, C ofrece otras funciones para memoria dinámica. `calloc` se usa cuando se quiere pedir memoria para varios elementos y comenzar con todos los bytes en cero. `realloc` aparece cuando se quiere intentar agrandar o achicar un bloque reservado previamente, por ejemplo para una colección que crece.

En esta introducción alcanza con reconocer que existen. La herramienta principal para entender memoria dinámica es `malloc`, junto con la regla de liberar con `free` cada bloque que se haya reservado correctamente.

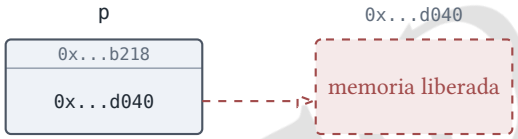
7.6 Punteros colgantes y fugas de memoria

Un puntero colgante es un puntero que conserva la dirección de memoria que ya fue liberada o que ya no es válida.

Por ejemplo:

```
int *p = malloc(sizeof(int));
if (p == NULL) {
    return 1;
}

*p = 42;
free(p);

printf("%d\n", *p);  // incorrecto
```

Después de `free(p)`, el bloque apuntado por `p` ya fue devuelto. El puntero todavía contiene una dirección, pero esa dirección ya no habilita a usar el bloque.



El error es tentador porque `p` no cambió de valor. Sin embargo, la validez de una dirección no depende solo del número guardado en el puntero. También depende de si ese espacio de memoria sigue disponible para ese uso.

Asignar `NULL` después de liberar ayuda a que un uso posterior sea más fácil de detectar:

```
free(p);
p = NULL;
```

Esto no reemplaza al cuidado de diseño, pero evita conservar una dirección vieja. Si después aparece un intento de desreferenciar `p`, el error será el mismo caso ya conocido de desreferenciar `NULL`, en lugar de usar una dirección que parece válida pero ya no lo es.

Una fuga de memoria ocurre cuando el programa pierde la posibilidad de liberar un bloque que había pedido con `malloc`.

Un ejemplo pequeño:

```
int *p = malloc(sizeof(int));
if (p == NULL) {
    return 1;
}

p = NULL;  // incorrecto: se perdió la dirección del bloque
```

La asignación `p = NULL` borra la única dirección que permitía llegar al bloque. Como no se llamó a `free` antes, ese bloque queda reservado hasta que el programa termine.

Si el programa termina enseguida, el sistema recuperará la memoria del proceso. Aun así, dentro de un programa que sigue corriendo, una fuga repetida puede hacer que cada vez quede menos memoria disponible. Por eso conviene incorporar la regla desde los ejemplos pequeños.

La fuga también puede ocurrir al sobrescribir un puntero con otra dirección:

```
int *p = malloc(sizeof(int));
if (p == NULL) {
    return 1;
}

p = malloc(sizeof(int));  // incorrecto si no se liberó el primer bloque
```

Antes de perder una dirección devuelta por `malloc`, debe estar claro quién va a liberarla.

7.7 Reservar un `struct`

Los ejemplos anteriores pidieron memoria para enteros y para arreglos de enteros. La misma idea aplica a un `struct`. En lugar de declarar la variable directamente en el `stack`:

```
struct Alumno alumno;
```

se puede pedir el bloque a `malloc` y guardar la dirección en un puntero a `struct`:

```
struct Alumno *p = malloc(sizeof(struct Alumno));
```

La sintaxis para acceder a los campos ya quedó vista en el capítulo 6: como `p` es un puntero a `struct`, se usa `p->nombre` y `p->edad`. Lo que cambia respecto de aquel capítulo es la procedencia de la dirección: ya no apunta a una variable local del `stack`, sino a un bloque del `heap` que sigue vivo hasta que se llame a `free(p)`.

Esa diferencia importa cuando el `struct` debe existir más allá de la función que lo creó, o cuando forma parte de una estructura mayor cuya cantidad de elementos se conoce recién durante la ejecución. El próximo capítulo retoma esta idea para construir el primer nodo dinámico.

7.8 Regla práctica

Al trabajar con memoria dinámica, conviene seguir siempre la misma secuencia:

- pedir memoria con `malloc`
- comprobar si el resultado es `NULL`
- usar el bloque solo mientras siga siendo válido
- liberar el bloque con `free`
- evitar conservar direcciones viejas cuando puedan usarse por error

La memoria dinámica permite crear datos cuyo tamaño o vida útil no queda fijado por una declaración local. Esa flexibilidad exige una responsabilidad adicional: el programa debe saber quién es dueño de cada bloque y quién lo libera.

8 Primer nodo dinámico

Los capítulos anteriores presentaron las piezas por separado: punteros, `struct`, memoria dinámica y liberación con `free`. Este capítulo las une en una construcción pequeña, pero central para EDA: un nodo creado durante la ejecución.

Un nodo es un `struct` que guarda un dato y, además, una dirección hacia otro nodo. Esa dirección permite encadenar varios nodos sin que estén dentro de un mismo arreglo.

La idea no es implementar todavía una lista completa. El objetivo es reconocer por qué un nodo necesita punteros y por qué suele crearse con memoria dinámica.

8.1 Un struct que se refiere a su mismo tipo

Un nodo para guardar enteros puede definirse así:

```
struct Nodo {
    int dato;
    struct Nodo *sig;
};

typedef struct Nodo Nodo;
```

El campo `dato` guarda el valor del nodo. El campo `sig` guarda la dirección del siguiente nodo. Si no hay siguiente nodo, `sig` vale `NULL`.

La definición tiene una forma particular:

```
struct Nodo *sig;
```

No se escribe `Nodo *sig` dentro del `struct` porque el alias `Nodo` se crea recién después, con el `typedef`. Mientras se está definiendo la estructura, el nombre disponible es `struct Nodo`.

La estructura no contiene otro `struct Nodo` completo dentro de sí misma. Contiene un puntero a otro nodo. Esa diferencia es necesaria: si un nodo contuviera otro nodo completo, ese otro nodo contendría otro nodo completo, y así sin terminar. En cambio, un puntero tiene tamaño fijo: solo guarda una dirección.

La lectura del tipo puede hacerse en dos pasos:

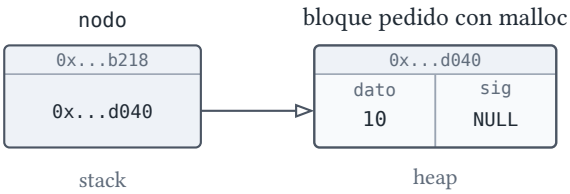
- `struct Nodo` nombra el tipo de estructura que se está definiendo
- `struct Nodo *sig` declara un puntero a otro objeto de ese tipo

8.2 Crear un nodo

Un nodo puede pedirse en el `heap` con `malloc`:

```
Nodo *nodo = malloc(sizeof(Nodo));
if (nodo == NULL) {
    printf("No se pudo reservar memoria\n");
    return 1;
}

nodo->dato = 10;
nodo->sig = NULL;
```

Después de `malloc`, el puntero `nodo` guarda la dirección del bloque reservado. Como ese bloque se interpreta como un `Nodo`, se puede acceder a sus campos con `->`.



En el diagrama, la variable `nodo` está en el `stack`. El bloque pedido con `malloc` está en el `heap`. Dentro de ese bloque están los campos `dato` y `sig`.

El campo `sig` vale `NULL` porque todavía no hay otro nodo después. Esta inicialización es importante: deja explícito que el nodo no apunta a un siguiente nodo.

Un programa mínimo:

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

struct Nodo {
    int dato;
    struct Nodo *sig;
};

typedef struct Nodo Nodo;

int main(void) {
    Nodo *nodo = malloc(sizeof(Nodo));
    if (nodo == NULL) {
        printf("No se pudo reservar memoria\n");
        return 1;
    }

    nodo->dato = 10;
    nodo->sig = NULL;

    printf("%d\n", nodo->dato);
    free(nodo);
    return 0;
}
```

La salida es:

```
10
```

8.3 Enlazar dos nodos

El campo `sig` se vuelve útil cuando existe otro nodo. Como hay dos reservas, cada una debe comprobarse:

```
Nodo *primero = malloc(sizeof(Nodo));
if (primero == NULL) {
    printf("No se pudo reservar memoria\n");
    return 1;
}

Nodo *segundo = malloc(sizeof(Nodo));
if (segundo == NULL) {
    printf("No se pudo reservar memoria\n");
    free(primero);
    return 1;
}
```

Si falla la segunda reserva, se libera el primer nodo antes de terminar. El programa ya había recibido ese bloque, por lo que también queda con la responsabilidad de devolverlo.

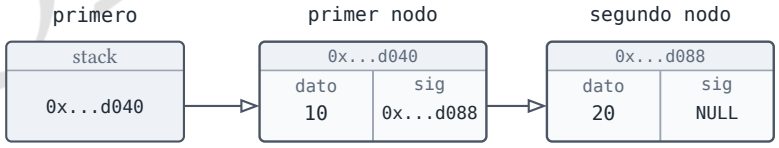
Después de esas comprobaciones, se pueden cargar los campos:

```
primero->dato = 10;
primero->sig = segundo;

segundo->dato = 20;
segundo->sig = NULL;
```

La asignación clave es:

```
primero->sig = segundo;
```

`segundo` es un puntero. Su valor es la dirección del segundo nodo. Al copiar ese valor dentro de `primero->sig`, el primer nodo queda enlazado con el segundo.



El campo `sig` no contiene el nodo completo. Solo contiene la dirección que permite llegar a él. Por eso dos nodos pueden estar en lugares separados del `heap` y, aun así, quedar conectados.

El último nodo tiene `sig == NULL`. Esa marca indica que el recorrido termina allí. Es la misma idea anticipada en la introducción, ahora construida con `struct`, punteros y memoria dinámica.

Conviene inicializar `sig` con `NULL` apenas se crea un nodo, incluso si después se lo va a enlazar con otro. Así el nodo siempre queda en un estado reconocible: o apunta a un siguiente nodo válido, o marca explícitamente que no tiene siguiente.

8.4 Recorrer con un puntero auxiliar

Para recorrer nodos enlazados se usa un puntero auxiliar. Ese puntero va tomando la dirección del nodo actual:

```
Nodo *actual = primero;

while (actual != NULL) {
    printf("%d\n", actual->dato);
    actual = actual->sig;
}
```

La condición `actual != NULL` evita intentar acceder a campos cuando ya no hay nodo. Dentro del bucle, `actual->dato` lee el dato del nodo actual. Luego:

```
actual = actual->sig;
```

copia en `actual` la dirección del siguiente nodo. Si el nodo actual era el último, su campo `sig` vale `NULL`, y el bucle termina.

Conviene observar que `actual` no es un nodo nuevo. Es una variable puntero que se mueve por la cadena de nodos. Cambiar `actual` no destruye ni copia los nodos. Solo cambia qué dirección se guarda en ese puntero auxiliar.

En el ejemplo anterior, los valores de `actual` avanzan así:

- al comienzo guarda la misma dirección que `primero`
- después de la primera vuelta guarda la dirección del segundo nodo
- después de la segunda vuelta guarda `NULL`

Si se conserva el puntero `primero`, todavía se puede volver al inicio. Si en cambio se usara `primero` como puntero de recorrido y se escribiera `primero = primero->sig`, se perdería la dirección del primer nodo salvo que estuviera guardada en otro lugar.

8.5 Liberar los nodos

Cada nodo pedido con `malloc` debe liberarse con `free`. En el caso de dos nodos conocidos por separado, se puede escribir:

```
free(segundo);
free(primero);
```

En este caso se conservan los dos punteros, `primero` y `segundo`. Por eso `segundo` se puede liberar directamente: no hace falta leer `primero->sig` para encontrarlo.

La situación cambia cuando solo se conserva la dirección del primer nodo. En ese caso, antes de liberar el nodo actual, hay que guardar la dirección del siguiente:

```
Nodo *actual = primero;
while (actual != NULL) {
    Nodo *siguiente = actual->sig;
    free(actual);
    actual = siguiente;
}
```

La línea `Nodo *siguiente = actual->sig;` se ejecuta antes de `free(actual)`. Después de liberar el nodo actual, ya no sería válido leer sus campos. Por eso el enlace hacia el siguiente nodo debe guardarse antes.

Esta es solo la forma básica de liberar una cadena de nodos. El trabajo sistemático con listas, incluyendo inserciones y borrados en distintas posiciones, corresponde a EDA.

8.6 Errores frecuentes

Un error frecuente es asignar `dato` y olvidar asignar `sig`:

```
nodo->dato = 10;
```

En ese caso, `sig` queda con un valor indeterminado. Si más adelante se intenta recorrer desde ese nodo, el programa podría interpretar basura como si fuera la dirección de otro nodo. Por eso conviene asignar también `nodo->sig = NULL;` cuando todavía no hay un siguiente.

Otro error es reservar memoria para el puntero, pero no para el nodo:

```
Nodo *nodo;
nodo->dato = 10;  // incorrecto
```

La variable `nodo` existe, pero no apunta a un `Nodo` válido. Antes de usar `nodo->dato`, el puntero debe recibir la dirección de un nodo existente, por ejemplo mediante `malloc`.

También es un error liberar el primer nodo y luego intentar seguir el enlace:

```
free(primero);
primero = primero->sig;  // incorrecto
```

Después de `free(primero)`, ya no se debe acceder a `primero->sig`. Si se necesita esa dirección para continuar, debe guardarse antes de liberar el nodo.

8.7 Regla práctica

Un nodo dinámico combina tres ideas:

- un `struct` para agrupar el dato y el enlace
- un puntero dentro del `struct` para guardar la dirección del siguiente nodo
- memoria dinámica para crear nodos mientras el programa se ejecuta

Esta combinación es la base de las listas enlazadas. El paso siguiente no consiste en aprender una sintaxis nueva, sino en organizar estas operaciones: crear nodos, enlazarlos, recorrerlos y liberarlos sin perder direcciones.

9 Errores comunes

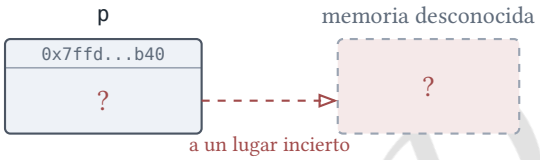
Los errores con punteros suelen parecer distintos entre sí, pero muchos responden a la misma pregunta: ¿el puntero guarda la dirección de un objeto válido para el uso que se quiere hacer?

Este capítulo reúne fallos frecuentes ya anticipados en los capítulos anteriores. La idea es presentar cada patrón en su forma más corta, junto a una versión segura, y remitir al capítulo donde se desarrolló por primera vez para repasar los detalles.

9.1 Usar un puntero sin inicializar

Un puntero local declarado sin asignación contiene un valor indeterminado, según se introdujo en el capítulo 3. Desreferenciarlo es un error:

```
int *p;
*p = 10;  // incorrecto
```



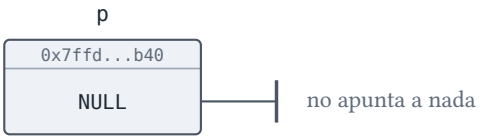
El patrón reconocible es la ausencia de asignación entre la declaración del puntero y su uso. La forma segura le da una dirección antes de desreferenciar:

```
int x = 0;
int *p = &x;
*p = 10;
```

9.2 Desreferenciar `NULL`

El valor `NULL` indica que el puntero no apunta a ningún objeto. Su definición y el diagrama aparecen en el capítulo 3. Con memoria dinámica, la comprobación contra `NULL` se vuelve sistemática después de cada `malloc`, según el patrón fijado en el capítulo 7. El error es desreferenciar sin chequear:

```
int *p = NULL;
printf("%d\n", *p);  // incorrecto
```



La versión segura comprueba antes de usar:

```
if (p != NULL) {
    printf("%d\n", *p);
}
```

9.3 Olvidar `free`

Toda memoria pedida con `malloc` debe liberarse con `free`. La regla completa, junto a sus matices —que `free(NULL)` es válido, que el sistema recupera la memoria al terminar el proceso pero un programa que sigue corriendo puede acumular fugas— está desarrollada en el capítulo 7. El error es perder la dirección antes de liberar el bloque:

```
int *p = malloc(sizeof(int));
if (p == NULL) {
    return 1;
}

*p = 42;
return 0;  // incorrecto: falta free(p)
```

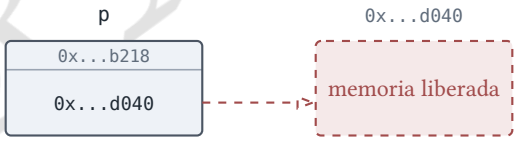
La versión correcta conserva la dirección hasta devolver el bloque:

```
*p = 42;
free(p);
return 0;
```

9.4 Usar memoria después de `free`

Después de `free`, la dirección guardada en el puntero ya no habilita a usar el bloque. El capítulo 7 introdujo esta situación como puntero colgante: el número de dirección puede no cambiar, pero el bloque ya fue devuelto.

```
free(p);
printf("%d\n", *p);  // incorrecto
```



Cuando el puntero podría volver a usarse por error, anularlo deja una marca reconocible:

```
free(p);
p = NULL;
```

Esto no vuelve seguro desreferenciarlo, pero transforma una dirección vieja en un caso más fácil de detectar.

9.5 Retornar la dirección de una variable local

Una función no debe devolver la dirección de una variable local automática:

```
int *crear(void) {
    int x = 42;
    return &x;  // incorrecto
}
```

La variable `x` ocupa un espacio reservado solo mientras se ejecuta `crear`. Cuando la función termina, ese espacio se libera y puede ser reutilizado por la siguiente llamada a función o por otra variable local. La dirección devuelta sigue siendo un número, pero ya no apunta a un `int` que pertenezca al programa.

Si el dato debe seguir existiendo después de que la función termina, una opción es reservar memoria dinámica:

```
int *crear(void) {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        return NULL;
    }

    *p = 42;
    return p;
}
```

En ese caso, quien recibe el puntero queda como dueño del bloque y tiene la responsabilidad de llamar a `free`. Esta convención debe quedar registrada en el código que rodea a la función: si una función reserva memoria y la devuelve, su nombre o un comentario breve deben dejar claro que el llamador debe liberarla.

9.6 Escribir fuera de los límites de un arreglo

Los punteros también permiten recorrer arreglos, según se vio en el capítulo 5. Eso no elimina los límites del arreglo:

```
int numeros[3] = {10, 20, 30};
int *p = numeros;

p[3] = 40;  // incorrecto
```

Los índices válidos son `0`, `1` y `2`. La expresión `p[3]` intenta escribir después del último elemento. El mismo error puede aparecer al mover un puntero: la dirección `numeros + 3` sirve como tope para comparar en un recorrido, pero no para desreferenciar.

Un uso seguro de esa dirección aparece como límite del recorrido:

```
int *fin = numeros + 3;
for (int *p = numeros; p != fin; p++) {
    printf("%d\n", *p);
}
```

El bucle se detiene cuando `p` alcanza la posición `fin`, sin desreferenciarla.

9.7 Confundir `sizeof(p)` con `sizeof(*p)`

Cuando `p` es un puntero, `sizeof(p)` mide el tamaño del puntero, no el tamaño del objeto apuntado:

```
int *p = malloc(sizeof(p));  // incorrecto
```

La intención era pedir espacio para un `int`, pero `sizeof(p)` responde otra pregunta: cuántos bytes ocupa la variable puntero. Las formas correctas, ya presentadas en el capítulo 7, son `malloc(sizeof(int))` o `malloc(sizeof(*p))`. La primera deja visible el tipo; la segunda lo calcula a partir de lo que `p` apunta. En ambos casos, después de `malloc` corresponde comprobar si el resultado es `NULL`.

9.8 Confundir p, *p y &p

La distinción entre las tres expresiones se desarrolló en el capítulo 3: `p` es la dirección guardada en el puntero, `*p` es el objeto apuntado y `&p` es la dirección de la variable puntero. Un error frecuente es modificar el puntero cuando se quería modificar el dato apuntado:

```
int x = 10;
int *p = &x;

p = NULL;  // cambia el puntero, no el entero
*p = 20;   // cambia x, el entero apuntado
```

También puede aparecer la confusión inversa entre pasar `p` y pasar `&p` a una función. El criterio práctico es preguntarse qué tiene que modificar la función. Si solo necesita leer o escribir el objeto apuntado, alcanza con pasar `p`. Si además necesita cambiar a qué objeto apunta el puntero del llamador, debe recibir `&p`, porque para modificar la variable puntero hace falta su dirección. Ese segundo caso introduce otro nivel de indirección y corresponde al **puntero doble** presentado en el capítulo 4.

9.9 Regla práctica

Antes de usar un puntero, conviene responder cuatro preguntas:

- ¿guarda la dirección de un objeto válido?
- ¿ese objeto sigue existiendo?
- ¿el tipo del puntero coincide con el tipo del objeto apuntado?
- ¿se quiere usar la dirección, el objeto apuntado o la dirección del puntero?

Si alguna respuesta no está clara, el programa probablemente está cerca de uno de los errores de este capítulo. La mayoría de los fallos con punteros se diagnostican volviendo a separar esas cuatro ideas: dirección, vida útil del objeto, tipo apuntado y nivel de indirección.

10 Puente a EDA

Los capítulos anteriores presentaron las piezas básicas: dirección, puntero, paso por dirección, relación entre arreglos y punteros, punteros a `struct`, memoria dinámica y un primer nodo enlazado. Estructura de Datos y Algoritmos (EDA) toma esas piezas y las usa para construir estructuras que cambian de tamaño durante la ejecución y para razonar sobre las operaciones que las manipulan.

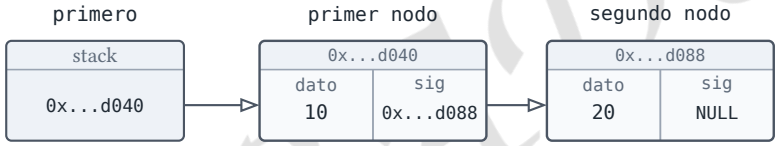
Este capítulo no introduce conceptos técnicos nuevos. Su objetivo es señalar dónde reaparecen las ideas anteriores y qué preguntas conviene tener en mente al empezar EDA.

10.1 Listas enlazadas

El capítulo 8 mostró cómo crear y enlazar dos nodos con `malloc` y un campo `sig` de tipo puntero. Una lista enlazada es la generalización de esa idea: una secuencia de nodos donde cada nodo guarda un dato y la dirección del siguiente.

En EDA, esta estructura aparece en varias formas:

- listas simplemente enlazadas, con un puntero al primer nodo
- listas que conservan además un puntero al último nodo
- listas doblemente enlazadas, donde cada nodo guarda también la dirección del nodo anterior
- listas circulares, donde el último nodo apunta al primero



El esqueleto sigue siendo el mismo: un puntero al primer nodo y, dentro de cada nodo, uno o más punteros que llevan a otros nodos. Cada operación —insertar, eliminar, buscar, recorrer— se describe en términos de cómo cambian esos enlaces.

A diferencia del primer nodo creado a mano, una lista en EDA se maneja con funciones que reciben un puntero a la lista y la modifican. Esa forma de trabajar permite implementar las operaciones de una lista una sola vez y usarla desde muchos programas distintos.

10.2 Pilas y colas

Una pila es una lista en la que solo se inserta y se elimina por uno de sus extremos. La operación que agrega un elemento suele llamarse **apilar**; la que quita el último elemento agregado, **desapilar**. El extremo donde ocurren ambas se llama **tope**.

Una cola, en cambio, inserta por un extremo y elimina por el otro. El extremo donde se elimina es el **frente**; el extremo donde se inserta es el **final**.

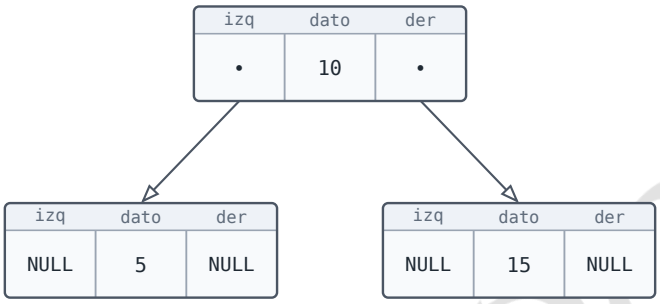
Estos tipos no exigen una estructura nueva. Pueden implementarse con nodos enlazados y reglas más restringidas sobre dónde ocurren las operaciones. La pila suele apoyarse en una lista simple, donde el tope es el primer nodo. La cola se apoya en una lista con un puntero al primer nodo (donde se elimina) y otro al último (donde se inserta), para que ambas operaciones sean rápidas.

Por eso EDA dedica primero tiempo a las listas. Una vez entendido cómo recorrer y modificar una lista enlazada, las pilas y las colas son sobre todo decisiones de diseño sobre qué operaciones quedan disponibles desde afuera.

10.3 Árboles

Un árbol agrega una idea: cada nodo puede tener más de un siguiente. Un árbol binario, por ejemplo, contiene en cada nodo un dato y dos punteros, uno al hijo izquierdo y otro al hijo derecho.

```
struct Nodo {
    int dato;
    struct Nodo *izq;
    struct Nodo *der;
};
```



En el diagrama, el nodo raíz contiene un dato y dos enlaces hacia otros nodos. Cuando un hijo no existe, el puntero correspondiente vale `NULL`. Los nodos hoja son aquellos cuyos dos punteros valen `NULL`.

Los recorridos sobre un árbol son naturalmente recursivos: la función que procesa un nodo se llama a sí misma sobre cada uno de sus hijos. Cada nodo se trata como raíz de un subárbol, y el mismo procedimiento se aplica a cada hijo. La idea es la misma que en una lista —saltar de un nodo a otro siguiendo una dirección guardada en un campo— pero con dos direcciones disponibles en cada paso.

Al programar con árboles aparecen las mismas dudas que con listas, repetidas en cada rama. Si un hijo es `NULL`, no hay subárbol que recorrer. Si un hijo apunta a otro nodo, hay que tratarlo como una raíz más pequeña.

10.4 TADs y encapsulamiento

Un Tipo Abstracto de Datos (TAD) describe un conjunto de operaciones sin fijar la representación interna. Una pila, por ejemplo, ofrece operaciones para apilar y desapilar, además de consultar el tope sin modificarlo. Quien la usa no necesita saber si por dentro hay un arreglo o una lista enlazada.

En C, un TAD suele aparecer repartido en dos archivos:

- un archivo de cabecera con extensión `.h`, donde se declaran las operaciones disponibles
- un archivo de implementación con extensión `.c`, donde esas operaciones están desarrolladas con código

Otro programa que quiera usar el TAD agrega un `#include` del archivo `.h`, igual que se hace con `stdio.h` o `stdlib.h`. A partir de ahí, ese programa solo invoca las funciones declaradas en la cabecera. Para identificar la pila o lista con la que está trabajando, mantiene un puntero al `struct` que la representa y se lo pasa a esas funciones. Los detalles internos —si la pila es una arreglo dinámico, una lista enlazada o algo más— quedan ocultos dentro del archivo `.c`.

Esa separación entre interfaz y representación es lo que hace que, una vez resuelta una pila como TAD, pueda usarse desde muchos lugares sin volver a pensar en sus nodos. También permite cambiar la implementación sin tocar el código que la usa.

10.5 Invariantes y dueño de la memoria

Cuando hay memoria dinámica y punteros, conviene mantener tres preguntas a la vista:

- ¿quién es dueño de cada bloque pedido con `malloc`?
- ¿quién está obligado a llamar a `free`?
- ¿qué punteros pueden valer `NULL` y en qué casos?

Las respuestas a estas preguntas forman lo que se llaman **invariantes** del tipo: condiciones que se mantienen verdaderas a lo largo de toda la vida del dato y que cada operación debe respetar. Por ejemplo, una lista enlazada puede tener este invariante: cada nodo de la lista fue pedido con `malloc` y es responsabilidad de la operación que destruye la lista liberarlos a todos. Si una función agrega un nodo, debe pedirlo con `malloc`. Si una función elimina un nodo, debe llamar a `free`.

Buena parte de los errores típicos —fugas, liberar dos veces el mismo bloque, usar memoria después de `free`, punteros colgantes, desreferenciar `NULL`— aparece cuando esos invariantes no se respetan o cuando no son claros para quien escribe la función.

En programas más grandes, los invariantes deben quedar registrados de algún modo: en el nombre de la función, en un comentario breve o en la documentación del TAD. Esa información es tan importante como el código que la implementa.

10.6 Memoria dinámica en obligatorios

En las primeras semanas de EDA suele aparecer un obligatorio que pide implementar una estructura dinámica desde cero. Las dificultades habituales no son sintácticas. Suelen ser de razonamiento sobre punteros:

- olvidar inicializar un campo de tipo puntero
- no comprobar el resultado de `malloc`
- liberar dos veces el mismo bloque
- recorrer una lista avanzando el mismo puntero que guardaba la dirección del primer nodo, y quedarse sin forma de volver al inicio
- perder la dirección de un bloque al sobrescribir el único puntero que la conservaba, por ejemplo con un nuevo `malloc` antes de liberar el bloque anterior

Todos estos errores aparecieron en los capítulos anteriores. La diferencia es que en una estructura grande pueden propagarse y volverse más difíciles de aislar. La estrategia más útil sigue siendo dibujar el estado de memoria antes y después de cada operación, especialmente cuando algo no se comporta como se espera.

10.7 Para cerrar

La lista del capítulo 1 fijó objetivos. Esta lista funciona como autoevaluación operativa: para cada pregunta, conviene poder dar una respuesta breve sin volver al texto.

- dado `int *p;` sin inicializar, ¿qué problema introduce `*p = 10;`? ¿qué hace falta antes de usar el puntero?
- sobre el diagrama de `int x = 10; int *p = &x;`, ¿a qué se refiere cada una de `p`, `*p` y `&p`?
- ¿cómo se escribe una función que duplique el valor de una variable del llamador? ¿qué cambia respecto de una que retorna el doble?
- frente a la frase «los arreglos son punteros», ¿qué diferencia oculta? ¿cómo se reformula con precisión?
- al recorrer un arreglo con punteros, ¿qué papel cumple la dirección inmediatamente posterior al último elemento?
- si `p` es un `struct Alumno *`, ¿qué significan `(*p).edad` y `p->edad`? ¿en qué se diferencian?
- ¿qué pasos componen el ciclo de vida de un bloque dinámico, desde `malloc` hasta `free`? ¿qué se comprueba después de `malloc`?
- en `struct Nodo { int dato; struct Nodo *sig; };`, ¿por qué `sig` no puede ser `struct Nodo` directamente?
- ¿en qué se diferencia un puntero colgante de una fuga de memoria? ¿cómo se origina cada uno?

Si alguna respuesta no aparece con claridad, conviene volver al capítulo donde se desarrolló esa idea antes de avanzar: cada pregunta apunta a un tema ya trabajado en el libro.

Si en cambio esas respuestas salen sin esfuerzo, las estructuras dinámicas de EDA serán fundamentalmente una organización de operaciones sobre nodos: insertar, recorrer, eliminar, destruir. Lo realmente nuevo ya no estará en la sintaxis de los punteros, sino en cómo combinarlos para construir y mantener estructuras correctas.

Glosario español-inglés

Este glosario reúne los términos técnicos usados en el libro y sus equivalentes habituales en inglés, tal como aparecen en bibliografía de C y de estructuras de datos. Algunos términos —`stack`, `heap`, `NULL`, `malloc`, `free`— se mantienen en inglés a lo largo del texto y se incluyen aquí con una definición compacta.

- **alcance** (`scope`). Región del programa donde un nombre declarado es visible y utilizable. Una variable local tiene alcance limitado a la función que la declara.
- **apilar** (`push`). Operación que agrega un elemento al tope de una pila.
- **apuntar a** (`point to`). Relación entre un puntero y la dirección que guarda. Un puntero apunta al objeto que vive en esa dirección.
- **árbol** (`tree`). Estructura donde cada nodo puede tener varios sucesores, organizados de modo que no formen ciclos.
- **árbol binario** (`binary tree`). Árbol en el que cada nodo tiene a lo sumo dos hijos, llamados habitualmente izquierdo y derecho.
- **aritmética de punteros** (`pointer arithmetic`). Operaciones de suma y resta sobre punteros. Avanzar `1` en un puntero a `int` desplaza la dirección la cantidad de bytes que ocupa un `int`. Solo tiene sentido dentro de un arreglo o de un bloque reservado.
- **arreglo** (`array`). Bloque contiguo de elementos del mismo tipo, accesibles por índice. En muchas expresiones, el nombre del arreglo se convierte en un puntero al primer elemento.
- **bucle** (`loop`). Estructura de control que repite un fragmento de código, como `while` o `for`.
- **cabecera** (`header`). Archivo con extensión `.h` que declara funciones y tipos para que otros archivos los usen mediante `#include`.
- **cadena de caracteres** (`string`). Arreglo de `char` terminado por el carácter `'\0'`. Las funciones de `string.h` esperan ese terminador.
- **cola** (`queue`). Estructura donde se inserta por un extremo, llamado **final**, y se elimina por el otro, llamado **frente**.
- **comportamiento indefinido** (`undefined behavior`). Situación que el estándar de C no define. Suele aparecer al desreferenciar un puntero inválido, escribir fuera de los límites de un arreglo o usar memoria después de `free`.
- **desapilar** (`pop`). Operación que quita el elemento que está en el tope de una pila.
- **desreferenciar** (`dereference`). Acceder al valor guardado en la dirección a la que apunta un puntero. Se escribe con el operador `*`.
- **dirección** (`address`). Número que identifica una ubicación en la memoria del programa. Se obtiene con el operador `&` aplicado a una variable.
- **dueño de la memoria** (`owner`). Convención que indica qué parte del código es responsable de llamar a `free` sobre un bloque pedido con `malloc`.
- **encapsulamiento** (`encapsulation`). Práctica de ocultar la representación interna de un tipo detrás de un conjunto fijo de operaciones que actúan como única vía de acceso.
- **interfaz** (`interface`). En el contexto de TADs, conjunto de operaciones públicas que una implementación pone a disposición del usuario, sin exponer los detalles internos de cómo están representados los datos.

- **falla de segmentación** (`segmentation fault`). Error de ejecución que el sistema emite cuando un programa accede a memoria que no le corresponde. Acceder a través de un puntero inválido suele provocarla.
- **final** (`back`, `rear`). Extremo de una cola donde se insertan elementos.
- **frente** (`front`). Extremo de una cola donde se eliminan elementos.
- **fuga de memoria** (`memory leak`). Situación en la que un bloque pedido con `malloc` queda sin punteros que lo alcancen y sin haberse liberado. El bloque permanece reservado hasta que termina el programa.
- **heap**. Región de la memoria del programa de donde se obtienen los bloques pedidos con `malloc`. En el libro se mantiene en inglés porque no hay una traducción de uso unánime.
- **hoja** (`leaf`). En un árbol, nodo cuyos hijos son todos `NULL`.
- **indirección** (`indirection`). Acceso a un valor a través de la dirección que lo identifica. Cada `*` en una expresión introduce un nivel de indirección.
- **invariante** (`invariant`). Condición que un tipo o estructura mantiene siempre verdadera, y que cada operación se compromete a respetar.
- **liberar** (`free`). Devolver al sistema un bloque de memoria dinámica que ya no se usa, mediante una llamada a la función `free`.
- **lista enlazada** (`linked list`). Secuencia de nodos donde cada nodo guarda un dato y la dirección del siguiente.
- **llamador** (`caller`). Función o fragmento de código que invoca a otra función.
- **memoria dinámica** (`dynamic memory`). Memoria pedida durante la ejecución del programa con `malloc`, `calloc` o `realloc`, y liberada explícitamente con `free`.
- **NULL**. Valor especial de puntero que indica que no apunta a ningún objeto. Desreferenciar un puntero igual a `NULL` es comportamiento indefinido.
- **nodo** (`node`). En el contexto de listas y árboles, `struct` que guarda un dato y al menos un puntero hacia otro nodo.
- **paso por dirección** (`pass by address`, `pass by reference`). Mecanismo por el cual una función recibe la dirección de una variable existente y, mediante desreferenciación, puede modificar su valor.
- **paso por valor** (`pass by value`). Mecanismo por el cual una función recibe una copia del valor del argumento. Las modificaciones sobre el parámetro no afectan a la variable original.
- **pila** (`stack` como TAD). Estructura donde solo se inserta y se elimina por un extremo, llamado **tope**. En el libro, **pila** se reserva para el TAD; la región de memoria del programa con el mismo nombre en inglés aparece sin traducir como `stack`, para evitar la confusión.
- **puntero** (`pointer`). Variable que guarda una dirección de memoria.
- **puntero colgante** (`dangling pointer`). Puntero cuyo valor sigue siendo una dirección, pero el bloque al que apuntaba ya fue liberado.
- **raíz** (`root`). En un árbol, nodo desde el cual parten todos los recorridos.
- **reservar** (`allocate`). Pedir un bloque de memoria al sistema, normalmente con `malloc`.
- **retorno** (`return`). Valor que una función entrega al llamador mediante la instrucción `return`.
- **stack**. Región de la memoria del programa donde viven las variables locales y los parámetros de cada función. Su vida útil queda atada a la ejecución de la función que las define. En el libro se mantiene en inglés para evitar la confusión con el TAD **pila**.

- **subárbol** (`subtree`). Árbol formado por un nodo y todos sus descendientes.
- **TAD** (`ADT`, `abstract data type`). Tipo Abstracto de Datos. Conjunto de operaciones disponibles sobre un tipo, sin fijar la representación interna.
- **tope** (`top`). En una pila, extremo donde se inserta y se elimina.
- **valor indeterminado** (`indeterminate value`). Valor que toma una variable que no fue inicializada. Leerlo es comportamiento indefinido.