UTN · FRMdP

MATERIAL DE CÁTEDRA · PROGRAMACIÓN I

Funciones en C: Parámetros por referencia y punteros

Universidad Tecnológica Nacional · Facultad Regional Mar del Plata

PROGRAMACIÓN I | LENGUAJE C | 2026

01 · Recordando lo básico

En la presentación anterior vimos que los parámetros se pasan **por valor**: la función recibe una copia, por lo que no puede modificar las variables originales. Para obtener un resultado, usábamos `return`.

Pero ¿y si queremos que una función modifique directamente una variable (por ejemplo, intercambiar dos valores, o leer datos como hace `scanf`)? Necesitamos una nueva herramienta: **los punteros**.

02 · ¿Qué es un puntero?

Un puntero es una variable que guarda una **dirección de memoria**. En lugar de contener un valor como 5 o 3.14, contiene la ubicación donde está almacenado otro dato.

Imaginemos la memoria como un gran edificio de departamentos. Cada departamento tiene una dirección única. Una variable normal vive en un departamento y tiene un contenido. Un puntero es como un papel donde anotamos la dirección de ese departamento.

03 · Operadores básicos con punteros

- **& (operador dirección):** nos da la dirección de memoria de una variable. Por ejemplo: &x devuelve la dirección donde está guardada x.
- *** (operador de indirección o desreferencia):** dado un puntero, accede al valor almacenado en la dirección que contiene. Por ejemplo: si p es un puntero a x, entonces *p nos da el valor de x.

```c
#include <stdio.h>
int main() {
    int x = 10;
    int *p;               // Declaramos un puntero a entero
    p = &x;               // p guarda la dirección de x

    printf("Valor de x: %d\n", x);
    printf("Dirección de x: %p\n", &x);
    printf("Valor de p (dirección): %p\n", p);
    printf("Valor apuntado por p: %d\n", *p);

    *p = 20;              // Cambiamos el valor de x a través del puntero
    printf("Nuevo valor de x: %d\n", x);
    return 0;
}
```

Salida (las direcciones variarán):

```
Valor de x: 10
Dirección de x: 0x7ffd5a3e4a4c
Valor de p (dirección): 0x7ffd5a3e4a4c
Valor apuntado por p: 10
Nuevo valor de x: 20
```

Observemos que al modificar *p estamos modificando directamente la variable x.

04 · Inicialización de punteros: NULL

Al igual que las variables normales, los punteros también deben ser inicializados antes de usarlos. Si no tenemos una dirección válida para asignar, podemos inicializarlos a `NULL`.

`NULL` es una constante que representa una dirección nula, es decir, un puntero que no apunta a ninguna parte. Es una buena práctica inicializar los punteros a `NULL` para evitar que contengan direcciones basura que podrían causar errores difíciles de detectar.

```
int *p = NULL;   // puntero inicializado a NULL
```

Antes de usar un puntero, debemos asegurarnos de que no sea `NULL` y de que apunte a una dirección válida.

05 · Punteros como parámetros de funciones

Si en lugar de pasar un valor, pasamos un **puntero** (la dirección de una variable), la función puede acceder a esa variable original y modificarla.

Esto se llama **paso por referencia** (aunque en C estrictamente seguimos pasando el puntero por valor, pero lo usamos para referenciar la variable original).

Ejemplo: función que intercambia dos valores (swap)

```
#include <stdio.h>

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

int main() {
    int x = 5, y = 10;
    printf("Antes: x = %d, y = %d\n", x, y);
    intercambiar(&x, &y);
    printf("Después: x = %d, y = %d\n", x, y);
    return 0;
}
```

Salida:

```
Antes: x = 5, y = 10
Después: x = 10, y = 5
```

¿Qué ocurre aquí?

1. En `main`, `&x` y `&y` son las direcciones de las variables.
2. La función `intercambiar` recibe esas direcciones en los parámetros formales `a` y `b` (que son punteros).
3. Dentro de la función, `*a` accede al valor de `x` y `*b` al valor de `y`.
4. Al modificar `*a` y `*b`, estamos modificando directamente `x` e `y`.

06 · Comparación: paso por valor vs paso por referencia (con punteros)

Paso por valor

Se pasa una copia del dato.
La función no puede modificar la variable original.
Útil cuando solo necesitamos el valor de entrada.
Ejemplo: `int suma(int a, int b)`

Paso por referencia (con punteros)

Se pasa la dirección del dato (copia de la dirección).
La función puede modificar la variable original mediante el puntero.
Útil cuando necesitamos devolver múltiples resultados o modificar variables.
Ejemplo: `void intercambiar(int *a, int *b)`

07 · ¿Por qué scanf necesita &?

Ahora entendemos por qué `scanf` usa el &: necesita modificar las variables que le pasamos. Por ejemplo:

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

`scanf` recibe la dirección de `edad` y, mediante un puntero, asigna el valor leído desde el teclado directamente a esa variable. Si no usáramos &, estaríamos pasando el valor de `edad` (que además podría no estar inicializado), y `scanf` no podría modificar la original.

> Cuando la función no va a modificar el dato, a veces también se pasa un puntero por eficiencia (evitar copiar grandes estructuras), pero eso lo veremos más adelante.

Otro ejemplo: función que duplica el valor de una variable

```
#include <stdio.h>
void duplicar(int *n) {
    *n = *n * 2;
}
int main() {
    int valor = 7;
    printf("Antes: %d\n", valor);
    duplicar(&valor);
    printf("Después: %d\n", valor);
    return 0;
}
```

Salida:

```
Antes: 7
Después: 14
```

Observemos que aquí no necesitamos `return`, porque la modificación es directa sobre la variable original.

08 · Precauciones con punteros

- **Siempre inicializar los punteros:** un puntero sin inicializar apunta a una dirección desconocida y usarlo puede causar errores graves (crash del programa). Inicializar a `NULL` es una buena práctica.
- **No confundir * en la declaración con * en el uso:**
  - `int *p;` declara que p es un puntero a entero.
  - `*p = 10;` asigna 10 a la variable apuntada por p.
- **Asegurarse de que el puntero apunte a una variable válida** antes de usarlo (por ejemplo, que no sea `NULL`).

09 · Paso de arrays a funciones (anticipo)

Cuando pasamos un array a una función, lo que realmente se pasa es la **dirección del primer elemento**. Por eso las funciones pueden modificar los elementos del array original. Esto lo veremos en detalle más adelante, pero es importante saber que los arrays y los punteros están relacionados.

10 · Ejemplo integrador: función que calcula suma y promedio a la vez

Con paso por valor solo podríamos devolver un valor con `return`. Con punteros podemos devolver varios resultados.

```c
#include <stdio.h>

void calcular(int a, int b, int *suma, float *promedio) {
    *suma = a + b;
    *promedio = (a + b) / 2.0;
}

int main() {
    int x = 8, y = 6;
    int s;
    float p;
    calcular(x, y, &s, &p);
    printf("Suma: %d\n", s);
    printf("Promedio: %.2f\n", p);
    return 0;
}
```

Aquí la función recibe dos enteros por valor (solo los necesita para leer) y dos punteros para devolver los resultados.

11 · Resumen

- Un **puntero** es una variable que almacena direcciones de memoria.
- `&` obtiene la dirección de una variable.
- `*` accede al valor apuntado por un puntero.
- Es buena práctica inicializar los punteros a `NULL` cuando no tienen una dirección válida.
- Pasando punteros como parámetros, las funciones pueden modificar las variables originales (paso por referencia simulado).
- Esto es útil para devolver múltiples valores o modificar variables.
- `scanf` usa punteros para poder guardar el valor leído en nuestras variables.
- Hay que tener cuidado con inicializar punteros y no usarlos sin apuntar a algo válido.

12 · Ejercicios para practicar

(Opcionales, para afianzar conceptos)

1. Escribir una función `void elevar_al_cuadrado(int *n)` que modifique el valor apuntado para que sea su cuadrado.
2. Escribir un programa que lea dos números enteros y use una función `void ordenar(int *a, int *b)` que los deje en orden ascendente.
3. Investigar qué ocurre si intentamos usar un puntero sin inicializar.
4. Implementar una función que genere IDs autoincrementales usando `static` y probarla. (Relacionado con ámbito, no con punteros, pero útil para practicar).

Con los punteros hemos abierto la puerta a un control más fino de la memoria y a la posibilidad de crear funciones más flexibles. En el futuro los usaremos con arrays, cadenas y estructuras.