Principios de Programación

Punteros

Direcciones de memoria en C para comprender `scanf`, paso por valor y funciones que actualizan datos existentes.

Motivación: actualizar un dato existente

INTENTO CON PASO POR VALOR

```c
void cambiar_edad(int edad) {
    edad = 30;
}

int main() {
    int edad = 19;
    cambiar_edad(edad);
    printf("%d\n", edad);
    return 0;
}
```

Resultado

La función modifica su parámetro local `edad`. La variable `edad` de `main()` conserva su valor.

SALIDA

```
19
```

Valor y dirección

Variable x

Nombre	Dirección	Valor
x	0x100	5

Valor

Es el dato guardado dentro de la variable.

En el ejemplo: `5`.

Dirección

Es la ubicación donde vive esa variable durante la ejecución.

En el ejemplo: `0x100`.

DIRECCIÓN DE UNA VARIABLE

```c
int x = 5;

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

&x produce la dirección

Nombre	Dirección	Valor
x	0x100	5

Regla

&x no produce el valor guardado en x. Produce la dirección de x.

Definición

Un puntero es una variable cuyo valor es una dirección de memoria. El tipo del puntero indica qué clase de dato se espera encontrar en esa dirección.

VARIABLE SIMPLE

```
1 int x = 5;
```

Lectura

x guarda un entero.

VARIABLE PUNTERO

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

Lectura

p guarda la dirección de un entero.

Declarar un puntero

FORMA GENERAL

```
tipo *nombre;
```

EJEMPLOS

```
int *p;
float *promedio;
char *letra;
```

Lectura

- `int *p` declara un puntero a `int`
- `float *promedio` declara un puntero a `float`
- El tipo indica qué clase de dato hay en la dirección apuntada

Inicialización

Declarar un puntero no alcanza. Antes de usarlo debe tener una dirección válida.

ACCESO AL DATO APUNTADO

```c
int x = 5;
int *p = &x;

printf("%d\n", *p);
*p = 8;
printf("%d\n", x);
```

Lectura

- p es la dirección guardada
- *p es el dato que está en esa dirección
- Asignar en *p modifica el dato apuntado

SALIDA

```
5
8
```

`&` y `*` juntos

DIRECCIÓN

```c
int x = 5;
int *p = &x;

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

p

Dirección guardada en el puntero.

DATO APUNTADO

```c
int x = 5;
int *p = &x;

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

***p**

Valor almacenado en la dirección apuntada.

Paso por valor

SIN PUNTEROS

```c
void cambiar(int n) {
    n = 8;
}

int main() {
    int x = 5;
    cambiar(x);
    printf("%d\n", x);
    return 0;
}
```

Lectura

n recibe una copia del valor de x. La asignación modifica esa copia, no la variable original.

SALIDA

```
5
```

Pasar una dirección

CON PUNTERO

```c
void cambiar(int *p) {
    *p = 8;
}

int main() {
    int x = 5;
    cambiar(&x);
    printf("%d\n", x);
    return 0;
}
```

Lectura

La función recibe una copia de la dirección de x. Con *p accede al dato original.

SALIDA

```
8
```

Patrón de modificación

1. Parámetro

La función declara un parámetro puntero.

```
int *p
```

2. Llamada

La llamada pasa una dirección.

```
cambiar(&x)
```

3. Uso

La función modifica el dato apuntado.

```
*p = 8
```

Regla

C sigue pasando argumentos por valor. En este caso, el valor copiado es una dirección.

LECTURA DE UN ENTERO

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

FUNCIÓN PROPIA CON EL MISMO PATRÓN

```c
void cargar_edad(int *edad) {
    scanf("%d", edad);
}

int main() {
    int edad;
    cargar_edad(&edad);
    return 0;
}
```

Lectura

`scanf` necesita una dirección porque debe escribir el dato leído dentro de una variable que ya existe.

NOMBRE DEL ARREGLO

```c
int notas[4] = {10, 20, 30, 40};
char nombre[20];

scanf("%19s", nombre);
printf("%p\n", notas);
```

Base del arreglo

En muchas expresiones, el nombre de un arreglo representa la dirección de su primer elemento.

Relación con `scanf`

Por eso `scanf("%19s", nombre)` no lleva `&`: `nombre` ya aporta una dirección inicial.

DECLARACIÓN Y ASIGNACIÓN

```c
struct Alumno {
    char nombre[20];
    int edad;
    float promedio;
};

struct Alumno a1 = {"Ana", 19, 8.7};
struct Alumno *p = &a1;
```

Lectura

`p` guarda la dirección de una variable de tipo `struct Alumno`.

Tipo

`struct Alumno *` significa puntero a una estructura `Alumno`.

Acceso a campos desde un puntero

CON * Y .

```c
struct Alumno *p = &a1;

(*p).edad = 30;
printf("%d\n", (*p).edad);
```

CON ->

```c
struct Alumno *p = &a1;

p->edad = 30;
printf("%d\n", p->edad);
```

Equivalencia

`p->edad` es una forma más cómoda de escribir `(*p).edad`.

PARÁMETRO PUNTERO

```c
void cambiar_edad(struct Alumno *a) {
    a->edad = 30;
}

int main() {
    struct Alumno a1 = {"Ana", 19, 8.7};
    cambiar_edad(&a1);
    printf("%d\n", a1.edad);
    return 0;
}
```

Lectura

La función recibe la dirección de `a1`. Al usar `a->edad`, modifica el campo de la estructura original.

SALIDA

```
30
```

FUNCIÓN CON PUNTERO A `struct`

```c
void cargar_alumno(struct Alumno *a) {
    printf("Nombre: ");
    scanf("%19s", a->nombre);

    printf("Edad: ");
    scanf("%d", &a->edad);

    printf("Promedio: ");
    scanf("%f", &a->promedio);
}

int main() {
    struct Alumno a1;
    cargar_alumno(&a1);
    return 0;
}
```

Lectura de `cargar_alumno`

`a->nombre`

`nombre` es un arreglo de `char`. Para `%s`, se pasa el arreglo.

`&a->edad`

`edad` es un campo simple. Para `scanf`, se pasa su dirección.

`a`

`a` ya es una dirección: apunta a una estructura.

`a->campo`

Accede a un campo dentro de la estructura apuntada.

Usar un puntero no inicializado

CÓDIGO CON ERROR

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

CORRECCIÓN

El puntero debe apuntar a una variable válida antes de usar *p.

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

Confundir dirección y dato

MENSAJE / PROBLEMA

```
int x = 5;
int *p = &x;
p = 8;
```

CORRECCIÓN

Para modificar el dato apuntado:

```
*p = 8;
```

Falta & en la llamada

MENSAJE / PROBLEMA

```
void cambiar(int *p);
int x = 5;
cambiar(x);
```

CORRECCIÓN

La función espera una dirección:

```
cambiar(&x);
```

Error frecuente con estructuras

INCORRECTO

```c
void cambiar_edad(struct Alumno *a) {
    a.edad = 30;
}
```

CORRECTO

```c
void cambiar_edad(struct Alumno *a) {
    a->edad = 30;
}
```

Lectura

`a` no es una estructura; es un puntero a estructura. Para acceder a sus campos se usa `->`.

`&x`

Obtiene la dirección de una variable.

`int *p`

Declara un puntero a `int`.

`*p`

Accede al dato almacenado en la dirección guardada por `p`.

`a->campo`

Accede a un campo dentro de la estructura apuntada.

Idea central

Los punteros permiten que una función trabaje sobre un dato existente sin devolver una copia modificada.

Ejercicio 1

Escribir una función `void duplicar(int *x)` que multiplique por 2 el valor original recibido.

Ejercicio 2

Definir `struct Producto` con nombre, precio y stock. Escribir `void cargar_producto(struct Producto *p)`.

Ejercicio 3

Corregir una función que recibe una estructura por valor e intenta modificar uno de sus campos.