PUNTEROS EN C: CONCEPTOS BÁSICOS

1. Propósito

Los punteros son una de las características más importantes del lenguaje C. En este material se estudiarán únicamente los conceptos fundamentales necesarios para comprender:

- qué es una dirección de memoria;
- qué significa el operador `&`;
- qué es un puntero;
- qué significa `*p`;
- cómo se utiliza un puntero con una variable;
- diferencia entre paso por valor y paso mediante referencia;
- cómo una función puede modificar una variable original.

No se abordarán operaciones aritméticas con punteros, memoria dinámica ni estructuras avanzadas.

---

2. La memoria y las variables

Cuando un programa se ejecuta, las variables se almacenan en la memoria.

Por ejemplo:

```
int edad = 20;
```

Podemos imaginar que en memoria existe un espacio para almacenar el valor `20`.

La variable tiene dos características:

Valor

```
20
```

Dirección de memoria

```
0x7FF...
```

La dirección identifica el lugar donde está almacenada la variable.

Podemos representarlo así:



---

3. El operador `&`

El operador `&` permite obtener la **dirección de memoria de una variable**.

Ejemplo:

```c
int edad = 20;

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

La primera instrucción muestra el valor:

```
20
```

La segunda muestra la dirección de memoria donde está almacenada `edad`.

Por lo tanto:

```
edad
```

significa:

> Obtener el valor de la variable.

Mientras:

```
&edad
```

significa:

> Obtener la dirección de memoria de la variable.

---

4. ¿Qué es un puntero?

Un puntero es una variable capaz de almacenar una **dirección de memoria**.

Por ejemplo:

```
int edad = 20;

int *p;

p = &edad;
```

Aquí:

```
int *p;
```

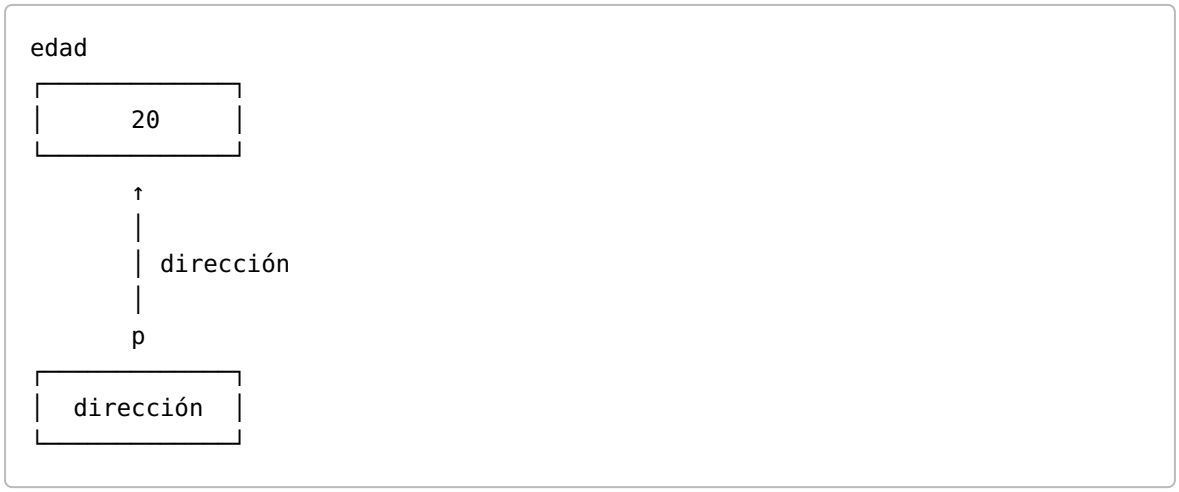
declara un puntero llamado `p`.

Después:

```
p = &edad;
```

hace que `p` almacene la dirección de `edad`.

Podemos imaginarlo así:



La idea principal es:

```
p  → contiene una dirección
```

---

5. El significado de `*p`

Si `p` es un puntero:

```
*p
```

permite acceder al valor almacenado en la dirección a la que apunta `p`.

Por ejemplo:

```
int edad = 20;

int *p = &edad;

printf("%d", *p);
```

El resultado es:

```
20
```

¿Por qué?

Porque:

```
p   → dirección de edad
*p  → valor almacenado en esa dirección
```

Por tanto:

```
*p
```

representa el valor de la variable a la que apunta `p`.

---

6. Relación entre `&` y `*`

Esta relación es fundamental.

Supongamos:

```c
int numero = 50;
int *p = &numero;
```

Entonces:

```
numero
→ valor de la variable

&numero
→ dirección de la variable

p
→ dirección almacenada en el puntero

*p
→ valor ubicado en esa dirección
```

Podemos resumir:

```
              numero
                 |
                 |
               valor
                 |
                 ↓
                50


              &numero
                 |
                 ↓
             dirección
                 |
                 ↓
                 p
                 |
                 ↓
               *p
                 |
                 ↓
                50
```

Una regla sencilla para recordar:

```
&  → dirección

*  → valor al que apunta
```

---

7. Ejemplo básico

```c
#include <stdio.h>

int main()
{
    int numero = 10;

    int *p = &numero;

    printf("Valor de numero: %d\n", numero);
    printf("Direccion de numero: %p\n", (void *)&numero);
    printf("Direccion almacenada en p: %p\n", (void *)p);
    printf("Valor apuntado por p: %d\n", *p);

    return 0;
}
```

El estudiante debe observar que:

```
&numero
```

y:

```
p
```

representan la misma dirección.

Mientras:

```
numero
```

y:

```
*p
```

representan el mismo valor.

8. Modificar una variable utilizando `*p`

Una de las aplicaciones más importantes de los punteros es modificar una variable mediante su dirección.

Ejemplo:

```
int numero = 10;

int *p = &numero;

*p = 50;

printf("%d", numero);
```

Resultado:

```
50
```

¿Qué ocurrió?

Inicialmente:

```
numero = 10
```

El puntero `p` apunta a `numero`.

Cuando hacemos:

```
*p = 50;
```

estamos cambiando directamente el valor de `numero`.

Por eso:

```
antes:

numero = 10

después:

*p = 50
```

```
numero = 50
```

---

9. ¿Por qué son importantes los punteros en las funciones?

Una función puede recibir datos de diferentes maneras.

Por ejemplo:

```
void cambiar(int x)
{
    x = 100;
}
```

Aquí `x` recibe una **copia** del valor.

En cambio:

```
void cambiar(int *x)
{
    *x = 100;
}
```

la función recibe la **dirección** de la variable.

Esto permite modificar la variable original.

---

10. Paso por valor

El paso por valor significa que una función recibe una **copia del valor de una variable**.

Ejemplo:

```
#include <stdio.h>

void cambiar(int x)
{
    x = 100;
}
```

```c
int main()
{
    int numero = 10;

    cambiar(numero);

    printf("%d", numero);

    return 0;
}
```

Resultado:

```
10
```

Aunque dentro de la función hacemos:

```
x = 100;
```

la variable `numero` continúa siendo:

```
10
```

¿Por qué?

Porque la función recibió una copia.

Podemos representarlo:

```
numero = 10

        ↓
      copia

x = 10

        ↓
   x = 100

numero continúa en 10
```

---

11. Paso mediante referencia utilizando punteros

C no tiene un mecanismo de "paso por referencia" idéntico al de algunos otros lenguajes.

Sin embargo, podemos conseguir ese comportamiento pasando la **dirección de la variable mediante un puntero**.

Ejemplo:

```c
#include <stdio.h>

void cambiar(int *x)
{
    *x = 100;
}

int main()
{
    int numero = 10;

    cambiar(&numero);

    printf("%d", numero);

    return 0;
}
```

Resultado:

```
100
```

Aquí:

```
cambiar(&numero);
```

envía la dirección de `numero`.

La función recibe esa dirección:

```
int *x
```

y posteriormente:

```
*x = 100;
```

modifica el valor almacenado en esa dirección.

---

12. Comparación entre ambos casos

Paso por valor

```
void cambiar(int x)
{
    x = 100;
}
```

Llamada:

```
cambiar(numero);
```

Resultado:

```
La variable original NO cambia.
```

Paso mediante puntero

```
void cambiar(int *x)
{
    *x = 100;
}
```

Llamada:

```
cambiar(&numero);
```

Resultado:

```
La variable original SÍ puede cambiar.
```

---

13. Tabla comparativa

Característica	Paso por valor	Paso mediante puntero
Parámetro	`int x`	`int *x`
Llamada	`f(numero)`	`f(&numero)`
Se envía	Valor	Dirección
La función recibe	Una copia	Un puntero
Uso dentro de función	`x`	`*x`
Puede modificar la variable original	No	Sí

---

14. Un ejemplo muy sencillo: incrementar una variable

Paso por valor

```
void incrementar(int x)
{
    x = x + 1;
}
```

Programa:

```
int numero = 10;

incrementar(numero);

printf("%d", numero);
```

Resultado:

```
10
```

Paso mediante puntero

```
void incrementar(int *x)
{
```

```
    *x = *x + 1;
}
```

Programa:

```
int numero = 10;

incrementar(&numero);

printf("%d", numero);
```

Resultado:

```
11
```

La diferencia fundamental está aquí:

```
Paso por valor:

incrementar(numero);


Paso mediante puntero:

incrementar(&numero);
```

---

15. Otro ejemplo: modificar dos variables

Podemos modificar varias variables mediante punteros.

```
void modificar(int *a, int *b)
{
    *a = 100;
    *b = 200;
}
```

Programa:

```
int x = 10;
int y = 20;

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

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

Resultado:

```
100
200
```

La función recibió las direcciones:

```
&x
&y
```

y trabajó con los valores:

```
*x
*y
```

---

16. Ejemplo clásico: intercambiar dos variables

Este ejemplo es excelente para comprender el concepto.

Queremos transformar:

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

en:

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

La función será:

```
void intercambiar(int *a, int *b)
{
    int temporal;

    temporal = *a;
    *a = *b;
```

```
    *b = temporal;
}
```

Llamada:

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

Programa completo:

```
#include <stdio.h>

void intercambiar(int *a, int *b)
{
    int temporal;

    temporal = *a;
    *a = *b;
    *b = temporal;
}

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

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

    intercambiar(&x, &y);

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

    return 0;
}
```

Resultado:

```
Antes:
x = 10
y = 20

Despues:
```

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

---

17. Lectura de los símbolos

Para que el estudiante no confunda los símbolos:

```
int numero = 10;
```

`numero`

Representa:

```
el valor
```

`&numero`

Representa:

```
la dirección de numero
```

`int *p`

Declara:

```
un puntero a entero
```

`p`

Representa:

```
la dirección almacenada en p
```

`*p`

Representa:

```
el valor ubicado en esa dirección
```

---

18. Uso de `&` con `scanf`

El operador `&` también aparece frecuentemente cuando se utiliza `scanf`.

Por ejemplo:

```
int edad;

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

¿Por qué se utiliza `&`?

Porque `scanf` necesita saber **en qué posición de memoria debe almacenar el valor ingresado**.

Conceptualmente:

```
Usuario escribe 25
        ↓
      scanf
        ↓
     &edad
        ↓
memoria de edad
        ↓
       25
```

Por eso normalmente aparece:

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

y no:

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

---

19. Ejemplo con `float`

El concepto no cambia con otros tipos de datos.

```
float nota = 4.5;

float *p = &nota;
```

```
printf("%.2f", *p);
```

Resultado:

```
4.50
```

El puntero debe ser compatible con el tipo de dato:

```
int     → int *
float   → float *
double  → double *
char    → char *
```

---

20. Ejemplo con `char`

```
char letra = 'A';

char *p = &letra;

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

Resultado:

```
A
```

Si hacemos:

```
*p = 'B';
```

entonces:

```
printf("%c", letra);
```

muestra:

```
B
```

---

21. Función para modificar un `float`

```
void aumentar(float *valor)
{
    *valor = *valor + 1.0;
}
```

Uso:

```
float nota = 3.5;

aumentar(&nota);

printf("%.1f", nota);
```

Resultado:

```
4.5
```

---

22. Una función que devuelve dos resultados

Los punteros también permiten modificar varias variables desde una función.

Por ejemplo:

```
void calcular(int a, int b, int *suma, int *producto)
{
    *suma = a + b;
    *producto = a * b;
}
```

Uso:

```
int suma;
int producto;

calcular(5, 3, &suma, &producto);

printf("Suma = %d\n", suma);
printf("Producto = %d\n", producto);
```

Resultado:

```
Suma = 8
Producto = 15
```

La función recibió:

```
a       → valor
b       → valor
suma    → dirección
producto → dirección
```

---

23. Regla práctica para analizar una función

Cuando aparezca:

```
void funcion(int x)
```

pregúntese:

> ¿Está recibiendo un valor o una dirección?

Respuesta:

```
Un valor.
```

Cuando aparezca:

```
void funcion(int *x)
```

pregúntese:

> ¿Está recibiendo un valor o una dirección?

Respuesta:

```
Una dirección.
```

Y cuando vea:

```
*x
```

pregúntese:

> ¿Qué significa?

Respuesta:

```
El valor almacenado en la dirección que contiene x.
```

---

24. Regla fundamental de las llamadas

Si la función recibe un dato:

```
void f(int x)
```

normalmente llamamos:

```
f(numero);
```

Si la función recibe un puntero:

```
void f(int *x)
```

normalmente llamamos:

```
f(&numero);
```

Esta relación debe aprenderse:

```
int x       → f(numero)

int *x      → f(&numero)
```

---

25. Errores frecuentes

Error 1: confundir `&` con `*`

Incorrecto pensar:

```
& = valor
* = dirección
```

La relación correcta es:

```
& → dirección

* → valor al que apunta
```

Error 2: pasar incorrectamente un argumento

Si tenemos:

```
void cambiar(int *x)
```

la llamada correcta es:

```
cambiar(&numero);
```

No:

```
cambiar(numero);
```

Error 3: olvidar `*` dentro de la función

Por ejemplo:

```
void cambiar(int *x)
{
    x = 100;
}
```

Esto no representa la modificación del valor apuntado.

Debe utilizarse:

```
void cambiar(int *x)
{
```

```
    *x = 100;
}
```

---

26. La idea central del tema

Todo el material puede resumirse en esta cadena:

```
VARIABLE
   ↓
tiene un VALOR
   ↓
también tiene una DIRECCIÓN
   ↓
& obtiene la dirección
   ↓
la dirección puede almacenarse en un PUNTERO
   ↓
*p permite acceder al VALOR
   ↓
el puntero puede enviarse a una FUNCIÓN
   ↓
la función puede modificar la variable original
```

---

27. Resumen para el estudiante

Debe recordar solamente estas ideas:

```
1. Una variable tiene un valor y una dirección.

2. &variable obtiene la dirección de la variable.

3. Un puntero almacena una dirección.

4. Si p es un puntero:
      p  → dirección
      *p → valor apuntado

5. Paso por valor:
      la función recibe una copia.

6. Paso mediante puntero:
      la función recibe una dirección.

7. Para enviar la dirección:
```

```
      funcion(&variable);

8. Para modificar el valor desde la función:
      *puntero = nuevo_valor;
```

---

28. Ejemplo final integrador

```c
#include <stdio.h>

void cambiar(int *x)
{
    *x = 500;
}

int main()
{
    int numero = 100;

    printf("Antes: %d\n", numero);

    cambiar(&numero);

    printf("Despues: %d\n", numero);

    return 0;
}
```

Análisis:

```
numero = 100

&numero
   ↓
direccion de numero

cambiar(&numero)
   ↓
x recibe la direccion

*x
   ↓
accede al valor de numero

*x = 500
```

```
   ↓
numero cambia a 500
```

Resultado:

```
Antes: 100
Despues: 500
```

CONCEPTO CLAVE

La frase que el estudiante debe poder explicar al finalizar es:

> **`&` permite obtener la dirección de una variable y `*p` permite acceder al valor almacenado en la dirección que contiene el puntero. Esto permite que una función modifique directamente una variable original cuando recibe su dirección.**