MEMORIA, APUNTADORES Y REFERENCIAS EN C

Conceptos, operaciones, vectores y funciones

Guía de estudio y apoyo para programación en lenguaje C

Idea central: Una variable tiene un valor y una dirección de memoria. El operador & obtiene la dirección y un apuntador almacena esa dirección. El operador * permite acceder al valor ubicado en ella.

1. La memoria y las variables

Cuando un programa se ejecuta, las variables se almacenan en posiciones de la memoria RAM. Cada variable puede analizarse desde dos perspectivas: su valor y la dirección de memoria en la que ese valor está guardado.

```
int edad = 20;

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

El primer printf muestra el valor de la variable. El segundo muestra su dirección. La dirección concreta cambia entre ejecuciones y entre equipos, por lo que no debe asumirse un número fijo.

2. El operador & : obtener la dirección

El operador & se utiliza para obtener la dirección de memoria de una variable. Esto es especialmente importante cuando una función necesita recibir la ubicación de una variable para modificarla, o cuando scanf necesita saber dónde almacenar un dato.

```
int numero = 10;
printf("Valor: %d\n", numero);
printf("Direccion: %p\n", (void*)&numero);
```

Regla: numero significa el valor almacenado; &numero significa la dirección donde está almacenado.

3. ¿Qué es un apuntador?

Un apuntador o pointer es una variable cuyo contenido representa una dirección de memoria. El tipo del apuntador indica el tipo de dato que se espera encontrar en esa dirección.

```
int edad = 20;
int *p;
p = &edad;
```

Después de ejecutar p = &edad, p contiene la dirección de edad. Por tanto, p representa la dirección almacenada, mientras que *p permite acceder al valor que se encuentra en esa dirección.

4. El operador * y la desreferenciación

El símbolo * tiene dos funciones frecuentes. En una declaración, identifica un apuntador. En una expresión, realiza la desreferenciación: accede al valor almacenado en la dirección a la que apunta el apuntador.

```
int edad = 20;
int *p = &edad;

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

El programa imprime 20. Conceptualmente: p -> dirección; *p -> valor contenido en esa dirección.

5. Relación fundamental entre &, * y una variable

Expresión	Significado
edad	Valor de la variable
&edad	Dirección de edad
p	Dirección almacenada en p
*p	Valor ubicado en esa dirección

Equivalencias: Si p = &edad, entonces p == &edad y *p == edad en el sentido del valor almacenado. Estas equivalencias son la base para comprender el resto del tema.

6. Modificar una variable mediante un apuntador

Una de las principales ventajas de los apuntadores es que permiten modificar directamente la variable original a través de su dirección.

```
int edad = 20;
int *p = &edad;

*p = 25;

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

El resultado es 25 porque *p representa la misma posición de memoria donde se encuentra edad.

7. Apuntadores y diferentes tipos de datos

El tipo del apuntador debe corresponder al tipo de dato al que apunta. Esto permite al compilador interpretar correctamente el contenido de memoria y aplicar la aritmética de apuntadores de forma adecuada.

Dato	Apuntador	Ejemplo
int	int *	int *p;
float	float *	float *p;
double	double *	double *p;
char	char *	char *p;

8. Direcciones con printf y %p

Para imprimir direcciones se utiliza %p. Es recomendable convertir la dirección a void* para la impresión con printf.

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

printf("Valor: %d\n", numero);
printf("Direccion: %p\n", (void*)&numero);
printf("Apuntador: %p\n", (void*)p);
printf("Valor apuntado: %d\n", *p);
```

9. & y scanf

scanf necesita la dirección de memoria donde debe almacenar el dato leído. Por eso, al leer una variable escalar, normalmente se escribe &variable.

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

La idea es: el usuario introduce un dato -> scanf recibe la dirección -> scanf escribe el valor en esa posición de memoria.

Precaución: Para un vector, normalmente se pasa el nombre del vector sin &: scanf("%d", &v[i]) usa la dirección de un elemento específico; scanf("%d", v) sería incorrecto para un entero si v es un vector de int y no se acompaña de un índice.

10. Paso por valor

Cuando una variable se pasa a una función por su nombre, la función recibe una copia del valor. Cambiar el parámetro no modifica la variable original.

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

```
}

int main()
{
    int numero = 10;
    cambiar(numero);
    printf("%d", numero);
    return 0;
}
```

El programa imprime 10. x es una copia independiente del valor original.

11. Paso mediante apuntador: comportamiento de referencia

C no posee el paso por referencia con una sintaxis propia como algunos otros lenguajes. Sin embargo, se consigue un comportamiento equivalente pasando la dirección de la variable y recibiendo un apuntador.

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

int main()
{
    int numero = 10;
    cambiar(&numero);
    printf("%d", numero);
    return 0;
}
```

El resultado es 100 porque la función recibe la dirección de numero y *x permite modificar el contenido de esa posición.

12. Comparación: paso por valor vs. mediante apuntador

Aspecto	Paso por valor	Mediante apuntador
Llamada	f(x)	f(&x)
Parámetro	int x	int *x
Contenido recibido	Copia del valor	Dirección
Modifica el original	No, normalmente	Sí, usando *x
Uso típico	Cálculos sobre el dato	Modificar datos y devolver resultados

13. Ejemplo clásico: intercambiar dos variables

Un ejemplo que muestra claramente la utilidad de los apuntadores es el intercambio de dos valores. Si los parámetros se pasan por valor, solo se intercambian las copias. Para modificar los originales deben pasarse sus direcciones.

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

int x = 10;
int y = 20;
intercambiar(&x, &y);
```

Después de la llamada: x = 20 y y = 10.

14. Apuntadores y vectores

En muchas expresiones, el nombre de un vector representa la dirección de su primer elemento. Por ello, para un vector int v[5], se cumple conceptualmente v == &v[0].

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

printf("%d", numeros[0]);
printf("%d", *numeros);
```

Ambas expresiones acceden al primer elemento y producen 10.

15. Aritmética de apuntadores

Los apuntadores permiten desplazarse por posiciones consecutivas de un mismo tipo. Si p es un int*, p + 1 apunta al siguiente int, no necesariamente al siguiente byte.

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

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

El primer valor es 10 y después de p++ el valor apuntado es 20. El incremento se ajusta automáticamente al tamaño del tipo apuntado.

16. Índices de vectores y apuntadores

Existe una relación fundamental entre la notación de índices y la aritmética de apuntadores:

```
numeros[0]  == *(numeros + 0)
numeros[1]  == *(numeros + 1)
numeros[2]  == *(numeros + 2)
numeros[i]  == *(numeros + i)
```

Esta equivalencia explica por qué los vectores y los apuntadores están tan estrechamente relacionados en C.

17. Vectores como argumentos de funciones

Una función puede recibir un vector utilizando la sintaxis con corchetes o la sintaxis con apuntador. En este contexto, ambas representan el acceso a los elementos del vector.

```
void mostrar(int v[], int n)
{
    for(int i = 0; i < n; i++)
        printf("%d ", v[i]);
}

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

```
void mostrar(int *v, int n)
{
    for(int i = 0; i < n; i++)
        printf("%d ", v[i]);
}
```

En ambos casos se trabaja con la dirección del primer elemento y la función necesita conocer el tamaño del vector. C no transmite automáticamente la cantidad de elementos junto con el vector.

18. Modificar elementos de un vector desde una función

```
void modificar(int *v)
{
    v[0] = 100;
}

int numeros[3] = {10,20,30};
modificar(numeros);
printf("%d", numeros[0]);
```

El resultado es 100. La función trabaja sobre las posiciones originales del vector.

19. Apuntadores para devolver varios resultados

Los apuntadores también permiten que una función produzca varios resultados a través de parámetros de salida.

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

int suma, producto;
operaciones(5, 3, &suma, &producto);
```

Después de la llamada: suma = 8 y producto = 15. Esta técnica es muy útil cuando una función necesita modificar varias variables del programa que la llamó.

20. Apuntadores y matrices

Los conceptos se pueden extender a matrices, aunque su tratamiento requiere entender cómo se organiza en memoria un arreglo de dos dimensiones.

```
int matriz[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};

printf("%d", matriz[1][2]);
```

El acceso por índices es el punto de partida. Después pueden estudiarse las expresiones con apuntadores a filas y el paso de matrices a funciones.

21. Apuntadores y const

Cuando una función solo necesita leer un vector y no debe modificarlo, puede declararlo como apuntador a datos constantes.

```
void mostrar(const int *v, int n)
{
    for(int i = 0; i < n; i++)
        printf("%d ", v[i]);
}
```

La declaración comunica la intención: la función puede consultar los datos, pero no debería modificarlos a través de ese apuntador.

22. Errores frecuentes

Problema	**Explicación / ejemplo**
Desreferenciar un apuntador no inicializado	int *p; *p = 10;
Confundir p con *p	p es una dirección; *p es el valor apuntado.
Confundir &x con *x	&x obtiene una dirección de x; *x desreferencia un apuntador.
Usar el tipo de apuntador equivocado	float * debe apuntar a float, int * a int, etc.
Esperar que el paso por valor modifique el original	La función recibe una copia.
Olvidar el tamaño del vector	Al pasar un vector, normalmente también se pasa n.

23. Resumen conceptual

Una variable representa un valor almacenado en memoria. El operador & obtiene su dirección. Un apuntador almacena esa dirección y el operador * permite acceder al valor que se encuentra en ella. En una frase: valor -> dirección -> apuntador -> desreferenciación.

24. ¿Por qué son importantes los apuntadores en C?

Los apuntadores son una parte central del lenguaje C porque permiten trabajar de manera explícita con direcciones de memoria. Gracias a ellos es posible modificar variables desde funciones, recorrer vectores, gestionar datos de forma eficiente y construir estructuras de datos dinámicas.

En aplicaciones más avanzadas se utilizan para memoria dinámica, cadenas, estructuras enlazadas, pilas, colas, árboles, interacción con bibliotecas y recursos de bajo nivel, entre otros escenarios.

25. Ruta recomendada de aprendizaje

1. Memoria y variables.
2. Dirección de memoria con &.
3. Declaración y uso de apuntadores.
4. Desreferenciación con *.
5. Modificación mediante apuntadores.
6. Paso por valor.
7. Paso mediante apuntadores para modificar originales.
8. Vectores y relación con direcciones.
9. Aritmética de apuntadores.
10. Apuntadores como parámetros de funciones.
11. Matrices y apuntadores.
12. const y, posteriormente, memoria dinámica.

26. Ejercicios sugeridos para el aula

- Crear una variable entera, imprimir su valor, su dirección y el valor obtenido mediante un apuntador.
- Modificar una variable utilizando únicamente un apuntador.
- Comparar una función por valor y otra mediante apuntador.
- Implementar una función que intercambie dos números.
- Recorrer un vector utilizando índices y luego utilizando p++.
- Escribir una función que calcule el máximo y el mínimo de un vector usando parámetros de salida.
- Explicar qué representan &x, p y *p en cada ejercicio.

Clave didáctica: El estudiante debe distinguir siempre tres ideas: el dato, la dirección del dato y el valor al que se accede a través de una dirección. La confusión entre estas tres nociones explica la mayoría de los errores iniciales con apuntadores.