

Comentarios en C: los comentarios sirven para explicar el código. No afectan a la ejecución

a) Comentario de una sola línea

```
c  
// Esto es un comentario de una sola línea  
int x = 5; // También puedes ponerlo al final de una línea
```

 Copy

b) Comentario de varias líneas

```
c  
/*  
 Esto es un comentario  
 que ocupa varias líneas  
*/  
int y = 10;
```

 Copy

Punto y coma: el punto y coma es obligatorio al final de cada sentencia. Marca el final de una instrucción. No se pone después de estructura pues de control como if, for, while(excepto cuando la sentencia dentro es una instrucción simple)

```
c  
int x = 5; // correcto  
x = x + 1; // correcto  
  
if (x > 0) { // NO va punto y coma aquí  
    printf("%d\n", x); // sí lleva punto y coma  
}
```

 Copy

main {
 ...
 j } punto y coma al final de cada
 sentencia

Librerías: se incluyen al inicio del programa con #include.
Permite usar funciones predefinidas.

#include <math.h>

#include <ctdlib.h> libreria

double (rand())/ RAND_MAX ;



genera un numero aleatorio entero

#include <type.h>

#include <stdio.h> incluir la librería stdio.h

stdio.h : standar input -output
.h head fichero cabecera

c

Copy

```
#include <stdio.h> // Para printf, scanf  
#include <math.h> // Para funciones matemáticas (opcional)
```

printf : impresión formateada

puedo utilizar printf porque he incluido la librería stdio.h

printf ("Primer programa\n") ;

\n

salto de linea

printf ("i1=%d\n i2=%d\n", i1, i2) ;

%d para entero

%o octagesimal

%x hexadecimal

Si quiero escribir por pantalla %d tengo que hacerlo así %.%d

Sintaxis básica

c

 Copy

```
printf("Texto a mostrar\n");
printf("Valor de x: %d\n", x);
printf("Valor de pi: %.2f\n", pi);
```

- \n → salto de línea.
- %d → entero
- %f → float o double
- %c → carácter
- %s → cadena de caracteres

`scanf("%.*f", &altura)`

[]

direccion de memoria de la variable donde
quiero que quede asignada

`% 28.14lf` [

- 28 espacios
- 14 decimales
- lf formato para float

alinea
`-15.10%5` [

- 15 ancho minimo del campo
- 10 indica maximo de
caracteres a imprimir.

`"%.2%,4`



lo sustituye: 4 en *

Tipos: definen que tipo de información puede almacenar una variable y como se interpreta en memoria.

Tipos int float double char
void no devuelve nada

La declaración de variable conlleva reservar un espacio de memoria para esa variable

int i1=7 El tipo indica el tamaño
 └ └
 tipo 7
 └
 i1

└ 7 └ 5 └ 12
i1 i2 i3

i3 += 1

i3 *= i1 + i2 → i3 * (i1 + i2)

char C1 = '7' ^① comillas simples

char { carácter
 y
 el código ASCII que representa

No hay booleano en C

C3 variable carácter

C3 = C2 + C1 sumar códigos

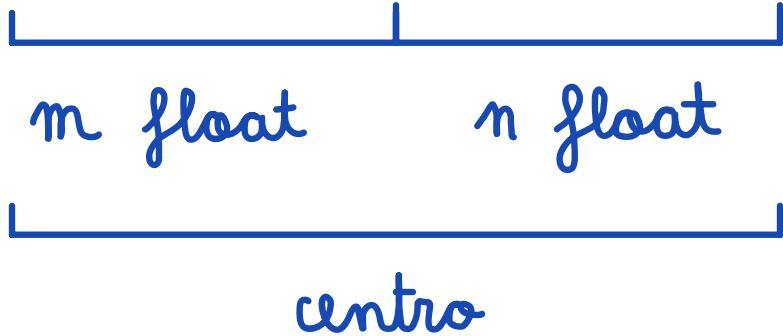
'7'
C3

65
C3

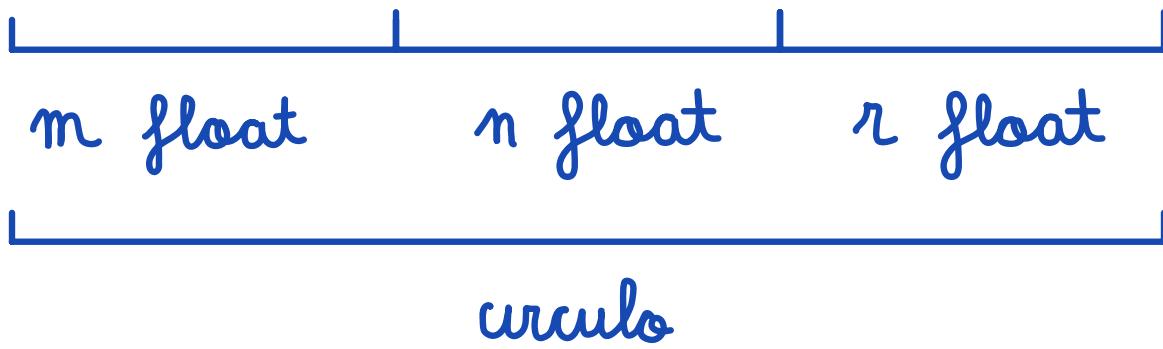
codigo ascii

char w[] array de caracteres

Tipos derivados:



centro



circulo

struct circulo C j

const float pi = 3.1415

⑧ C.cen.m

direccion de memoria

#define: para definir constantes . No tiene tipo, el tipo se deduce según como se use. Reemplaza todas las apariciones. No ocupa memoria.

define radio 0.5

Sustituye 0.5 cuando aparezca radio en el programa

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

#define PI 3.14159 // Define constante PI

int main() {
    float radio = 0.5;
    float area = PI * radio * radio; // Usamos PI definido con #define

    printf("El área del círculo es: %.5f\n", area);
    return 0;
}
```



Const: para declara una constante. La constante no puede cambiar después de asignarla

const float pi = 3.1415 reserva espacio

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

int main() {
    const float PI = 3.14159; // Constante
    // PI = 3.14; // Esto daría error, no se puede modificar
    printf("PI es: %.5f\n", PI);
    return 0;
}
```

Enum: permite crear valores simbólicos enteros.

```
#include <stdio.h>

// Definición de un enum
enum Colores { ROJO, VERDE, AZUL };

int main() {
    enum Colores color_favorito;

    color_favorito = VERDE; // Asignamos valor simbólico
    printf("Color favorito tiene valor: %d\n", color_favorito); // Imprime 1
    return 0;
}
```



- Por defecto, ROJO = 0, VERDE = 1, AZUL = 2
- Puedes cambiar los valores explícitamente:

c

Copy

```
enum Colores { ROJO = 10, VERDE = 20, AZUL = 30 };
```

enum dias { lunes , martes } j
0 1
enum estudios { mates = 1, fisica } j
2

& i1 dirección de memoria para la variable i1

No hay ** de exponentación, en su defecto se usa la función pow de la librería de mates.

Las operaciones se hacen de izquierda a derecha

No es recomendable hacer una comparativa de igualdad para float o double

$$c = (5 \underline{.} / 9)$$

para obtener la división decimal

iter += 1 es lo mismo que iter++

getchar() leer cada carácter

islower(ch) isupper(ch)

toupper(ch) tolower(ch)

Sentencias de control

1. Sentencias condicionales



c

Copy

```
if (condicion) {  
    // código si la condición es verdadera  
}
```

para cierto
para falso

Para if: ($h > 0$) ? h : 0

condición



if...else

c

Copy

```
if (condicion) {  
    // se ejecuta si es verdadera  
} else {  
    // se ejecuta si es falsa  
}
```



if...else if...else

c

Copy

```
if (condicion1) {  
    // ...  
} else if (condicion2) {  
    // ...  
} else {  
    // ...  
}
```

✓ switch

Útil para múltiples condiciones sobre un mismo valor.

```
c                                     ⌂ Copy

switch (opcion) {
    case 1:
        // código
        break;
    case 2:
        // código
        break;
    default:
        // otro caso
}
```

sino pongo break; sigue evaluando el siguiente break

2. Bucles

Bucles for

for (i = 0; i < 10; i++) {
 inicialización
 i variable entera o float
}

Ejercicio: contar todos los números de tres cifras cuya sumas al cubo es igual al número.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(){
5     int i, j, contador;
6     int x, y, z;
7     contador = 0;
8     for (i = 100; i<=999; i++){
9         int aux = i ;
10        for (j = 1; j <4; j++){
11            if(j==1){
12                x = aux %10;
13            }
14            else if (j==2){
15                y = aux %10;
16            }
17            else{
18                z = aux %10;
19            };
20            aux = aux /10;
21        };
22        if (i == (pow(x,3) + pow(y,3) + pow(z,3))){
23            contador++;
24        };
25    };
26    printf("Contador = %d\n", contador);
27    return 0;
28 }
```

Fichero: Contar.c

Bucle while

```
while( condicion ) {  
    ...  
} ;
```

puede que nunca se execute

```
do {  
    ...  
} while( condicion ) ;
```

se ejecuta al menos una vez

```
while( 1 ) {  
    ...  
} ;
```

bucle ∞ que paramos con un break

Ejercicio: calcular el número de ceros de n!

Programar $n \frac{15}{5}$

$n_1 \frac{15}{5}$

n_2

$n_1 + n_2 + \dots + n_k$

$n_k \frac{15}{0}$

```
1 #include <stdio.h>
2
3 int main() {
4     int n;
5
6     // Sigue al usuario que introduzca un número
7     printf("Introduce un número entero positivo: ");
8     scanf("%d", &n);
9
10    // Validación simple (opcional)
11    if (n < 0) {
12        printf("El número debe ser positivo.\n");
13        return 1;
14    }
15
16    int cociente = n;
17    int contar = 0;
18
19    // Cuenta los ceros finales de n!
20    while (cociente > 0) {
21        cociente /= 5;
22        contar += cociente;
23    }
24
25    printf("Ceros finales en %d! = %d\n", n, contar);
26    return 0;
27 }
```

3. Control de flujo dentro del bucle

✓ **break**

Sale del bucle o del switch.

c

 Copy

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) break;  
}
```

✓ **continue**

Salta a la siguiente iteración.

c

 Copy

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) continue;  
    printf("%d\n", i);  
}
```

✓ **goto (rara vez recomendado)**

c

 Copy

```
goto etiqueta;  
  
etiqueta:  
    printf("Saltaste aquí\n");
```

Macros

Las macros tienen la capacidad de sustituir constantes simbólicas por valores o expresiones

Preprocesamiento : al principio del fichero previo al programa

Las macros no son variables , ni funciones de C y no utilizan memoria del ordenador

(buscar → sustituir)

```
# define POT2 (X) X*X
```

m = 7

POT2 (m) |
 |
 sustitucion

$$\text{POT2}(m+1) \quad | \quad m+1 * m+1 = m + m+1$$

sustitucion

$$\text{POT2}((m+1)) \quad | \quad (m+1)*(m+1)$$

sustitucion

intercambiar con macros

Fichero: Macros-4.c

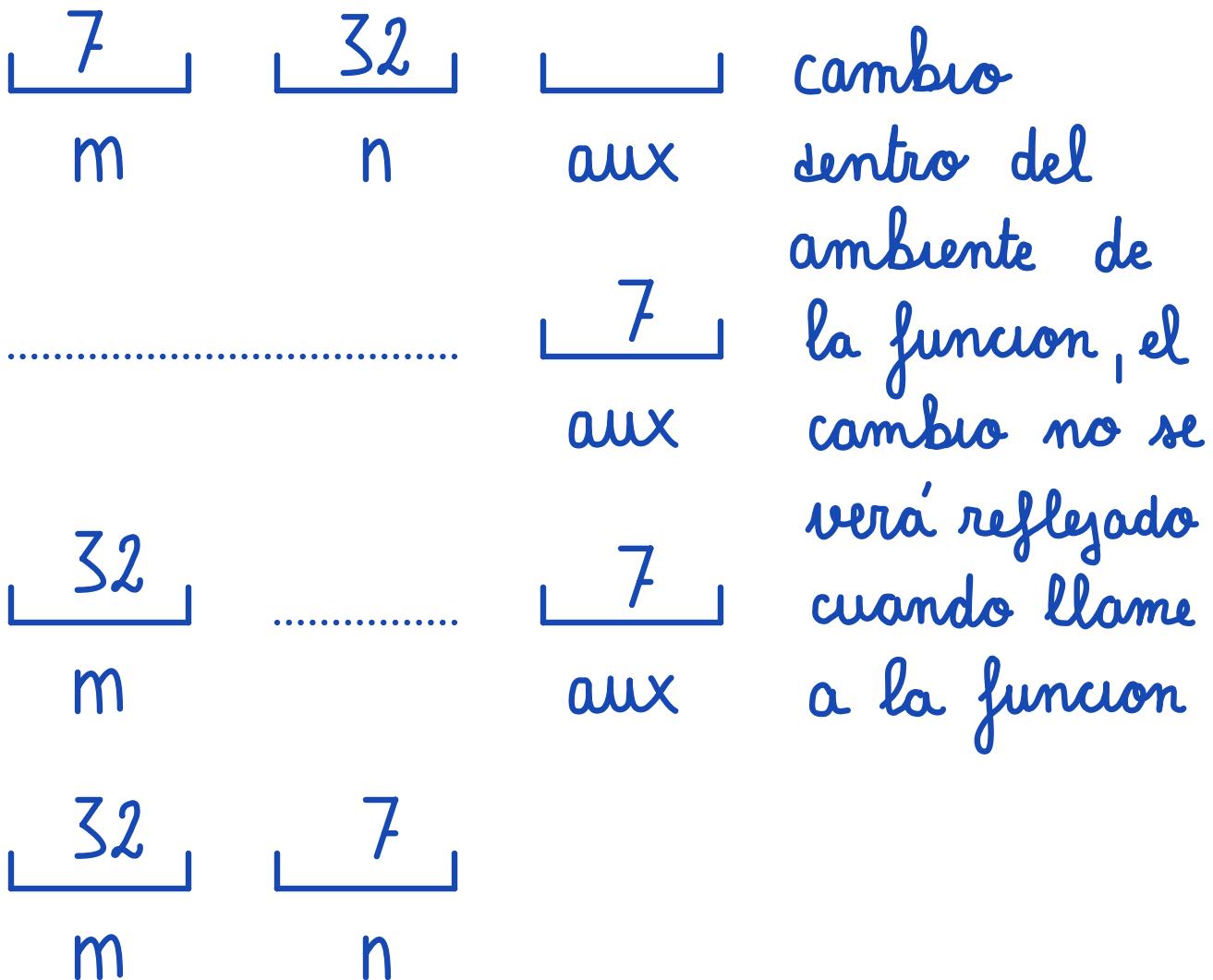
```
1  /* Expansion de macros. Reemplazo de argumentos. Operadores #, ## */
2
3  #include <stdio.h>
4  #define unestr(x,y) #x #y           /* #x genera "x" las cadenas se unen */
5  #define une(x,y) x ## y           /* concantena los tokens x y; crea xy */
6
7  #define iescribe(i) printf (#i " = %i\n", i)
8  #define fescribe(f) printf (#f " = %f\n", f)
9  #define gscribe(g) printf (#g " = %g\n", g)
10
11 main () {
12     int n1n2=7;
13     float x1=1.1;
14
15     printf ("%s\n", unestr(uno, dos)); /* el espacio solo separa argumentos*/
16     printf ("%s\n", unestr(n1,n2));
17
18     printf ("\n");
19     n1n2 += une(n1,n2); /*preguntar que genera esto*/
20     iescribe (n1n2);    /* printf ("n1n2" " = %i\n", n1n2); */
21
22     printf ("\n");
23     x1 += une(x,1);
24     fescribe (x1);
25     gscribe (x1);
26 }
```

Funciones

Paso por valor vs Paso por referencia

Fichero: Referencia_y_valor.c

```
/* ----- */  
/* Paso por valor (copias de los argumentos) */  
  
void cambio_valor (int m, int n) {  
    int aux;  
  
    printf ("\nValor del primer numero = %d", m);  
    printf ("\nValor del segundo numero = %d", n);  
  
    aux = m ; m = n ; n = aux;  
}
```



```

/*
 * Paso por referencia (direcciones de los argumentos)
 */

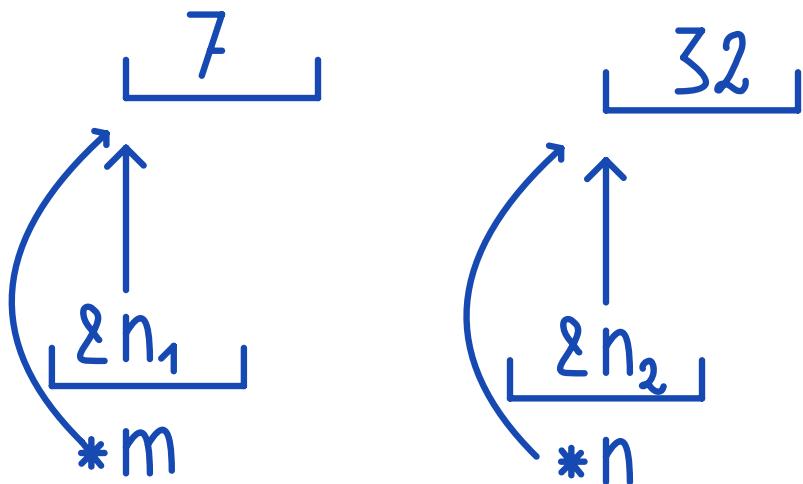
void cambio_referencia (int *m, int *n) {
    int aux;
    puntero a un entero
    printf ("\nPrimer numero = %d", *m);
    printf ("\nSegundo numero = %d", *n);

    printf ("\nDireccion del primer argumento = %d", m);
    printf ("\nDireccion del segundo argumento = %d", n);

    aux = *m ; *m = *n ; *n = aux;
}

```

puntero : una dirección
de memoria



Al llamar a la función le tengo que pasar una dirección de memoria

cambio_referencia ($\&n_1, \&n_2$)

$\text{aux} = \boxed{*m}$

accede al contenido de la dirección
de memoria

$*m = \boxed{*n}$

contenido de M

contenido de N

$*n = \text{aux}$

Fichero C: Referencia_Valor.c

```
1 #include <stdio.h>
2
3 void main(){
4     int var1, var2;
5     int *punt1;
6
7     var1 = 2;
8     var2 = 11;
9     punt1 = &var1;
10
11    printf("\nVariable var1 = %d", var1);
12    printf("\nVariable punt1 = %d\n", *punt1);
13    printf("\nVariable var2 = %d\n", var2);
14
15    *punt1 = 40;
16    printf("\nVariable var1 = %d", var1);
17    printf("\nVariable punt1 = %d", *punt1);
18    printf("\nVariable var2 = %d\n", var2);
19
20    punt1 = &var2;
21    *punt1 = 40;
22    printf("\nVariable var1 = %d", var1);
23    printf("\nVariable punt1 = %d", *punt1);
24    printf("\nVariable var2 = %d \n", var2);
25
26 }
```

punt1 = & var1



direccion de memoria

modificar var1 utilizando el puntero :

*punt1 = 40

Resultado

```
Variable var1 = 2  
Variable punt1 = 2
```

```
Variable var2 = 11
```

```
Variable var1 = 40  
Variable punt1 = 40  
Variable var2 = 11
```

```
Variable var1 = 40  
Variable punt1 = 40  
Variable var2 = 40
```

Ejercicio: sea $f(x) = \cos(x)-x$ en $[0,4]$. Implementar el método de la bisección

Tol = 10^{-6}

ep = 10^{-7}

$$|a - b| < tol \text{ ó } |f(x)| < ep$$

```
1 #include <stdio.h>
2 #include <math.h>
3
4 // Prototipo
5 void biseccion_con_tabla(float a, float b, float tol, float ep, int maxiter, FILE* salida);
6
7 int main() {
8     FILE *entrada, *salida;
9     int maxiter;
10    float a, b, tol, ep;
11
12    // Abrimos archivo de entrada
13    entrada = fopen("entrada.dat", "r");
14    if (entrada == NULL) {
15        printf("Error al abrir archivo de entrada.\n");
16        return 1;
17    }
18
19    // Leemos datos: maxiter a b tol ep
20    fscanf(entrada, "%d %f %f %f %f", &maxiter, &a, &b, &tol, &ep);
21    fclose(entrada);
22
23    // Abrimos archivo de salida
24    salida = fopen("salida.txt", "w");
25    if (salida == NULL) {
26        printf("Error al crear archivo de salida.\n");
27        return 1;
28    }
29
30    // Cabecera
31    fprintf(salida, "Iter\t\t\t a\t\t\t b\t\t\t fa\t\t\t fb\t\t\t m\n");
32    fprintf(salida, "-----\n");
33
34    // Ejecutar bisección y guardar iteraciones
35    biseccion_con_tabla(a, b, tol, ep, maxiter, salida);
36
37    fclose(salida);
38    printf("✓ Resultados guardados en 'salida.txt'\n");
39    return 0;
40 }
```

```
42 void biseccion_con_tabla(float a, float b, float tol, float ep, int maxiter, FILE* salida) {
43     float fa, fb, fm, m;
44     float tola, epa;
45     int i = 1;
46
47     fa = cos(a) - a;
48     fb = cos(b) - b;
49
50     do {
51         m = (a + b) / 2.0;
52         fm = cos(m) - m;
53
54         // Guardamos esta iteración en el fichero
55         fprintf(salida, "%d\t%10.6f\t%10.6f\t%10.6f\t%10.6f\t%10.6f\n", i, a, b, fa, fb, m);
56
57         tola = fabs(b - a);
58         epa = fabs(fm);
59
60         if (fa * fm < 0) {
61             b = m;
62             fb = fm;
63         } else {
64             a = m;
65             fa = fm;
66         }
67         i++;
68     } while (tola > tol && epa > ep && i <= maxiter);
69 }
```

Fichero entrada : entrada.dat

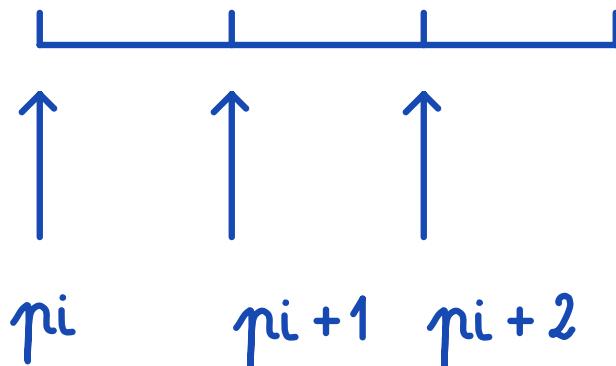
Fichero código C: Biseccion.c

Fichero salida: salida.txt

Aritmética de punteros

recorrer el array sumando
 $\pi_i + 1$,
 $\pi_i + 2$ una unidad
al puntero

Imprimir un puntero



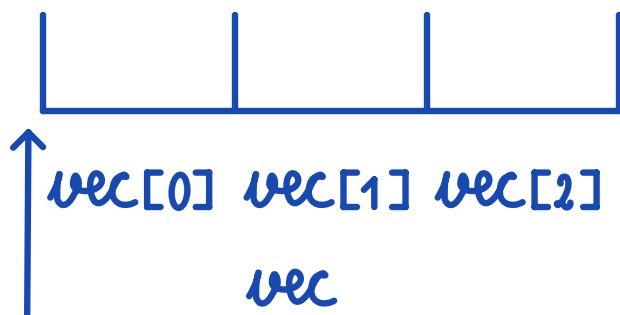
$$* \pi_i = 88$$

$$* (\pi_i + 2) = 29$$

$\pi_i + 2$ dirección de memoria

$* (\pi_i + 2)$ el contenido

vec es un array



vec[0] = vec = direccion de memoria
vec + 1 = direccion de memoria

Matrices

[rear matrices en tiempo de ejecución

** A definir matrices

- calloc
- malloc
- free

a =

(double **)calloc (m, sizeof (double *))

a[0] =

(double *)calloc (m * n, sizeof (double))

sizeof (____) devuelve el tamaño en memoria de la variable

Ficheros

Fichero es un puntero

FILE *f1 puntero de tipo file
 puntero a un fichero

'r' read el fichero tiene que existir, si no
 existe no funciona.

'w' write Si no existe lo crea y si ya
 existe lo machaca y escribe.

'a' append

fclose → importante cerrar el fichero

`fprintf (puntero al fichero, _____)`

`fscanf (f1, %.f %.f %n, &s11, &s12)`



leer en funcion de este formato

⌚ RESUMEN: TIPOS DE PUNTEROS EN C

📌 ¿Qué es un puntero?

Un **puntero** es una variable que almacena la **dirección de memoria** de otra variable.

```
int a = 10;  
int *p = &a; // p apunta a la dirección de a
```

◆ 1. Punteros Básicos

```
int *p;      // puntero a entero  
char *c;     // puntero a carácter  
float *f;    // puntero a float
```

- Se usan para acceder y modificar el valor original de la variable.
- Se usa el operador `*` (desreferenciación) y `&` (dirección).

```
int a = 5;  
int *p = &a;  
printf("%d\n", *p); // imprime 5
```

◆ 2. Punteros Nulos (`NULL`)

```
int *ptr = NULL;
```

- Apuntan a **ningún lugar válido de memoria**.
- Útiles para inicializar punteros o comprobar si apuntan a algo.

```
if (ptr == NULL) {  
    // el puntero no apunta a ninguna dirección válida  
}
```

◆ 3. Punteros Dobles (Puntero a puntero)

```
int **pp; // puntero a puntero a entero
```

- Se usan para manejar arreglos dinámicos, matrices, funciones que modifican punteros, etc.

```
int a = 5;  
int *p = &a;  
int **pp = &p;  
printf("%d\n", **pp); // imprime 5
```

◆ 4. Punteros a Funciones

```
int suma(int a, int b) {
    return a + b;
}

int (*fptr)(int, int) = suma;
printf("%d\n", fptr(2, 3)); // imprime 5
```

- Permiten pasar funciones como argumentos a otras funciones.
 - Se usan en callbacks y estructuras de funciones.
-

◆ 5. Punteros a Arreglos

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;
```

- El nombre del arreglo (`arr`) se comporta como un puntero al primer elemento.
- Puedes recorrer un array usando punteros.

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

◆ 6. Punteros a void (punteros genéricos)

```
void *ptr;
```

- Pueden apuntar a cualquier tipo de dato.
- Se deben **convertir explícitamente** antes de usarlos.

```
int a = 10;
void *p = &a;
printf("%d\n", *(int *)p); // casteo necesario
```

◆ 7. Punteros Constantes

A. const en los datos apuntados:

```
const int *p;
// El valor al que apunta p NO se puede modificar.
int x = 10;
const int *p = &x;
*p = 20; // ❌ error
```

B. const en el puntero:

```
int *const p = &x;
// No puedes hacer que p apunte a otro sitio, pero sí modificar *p
```

C. const en ambos:

```
const int *const p = &x;  
// No puedes cambiar ni *p ni la dirección
```

◆ 8. Punteros a Structs

```
struct Persona {  
    int edad;  
};  
  
struct Persona p = {25};  
struct Persona *ptr = &p;  
  
printf("%d", ptr->edad); // acceso con ->
```

◆ 9. Punteros como Parámetros de Función

Los punteros permiten que las funciones **modifiquen directamente** los valores de las variables.

```
void modificar(int *p) {  
    *p = 42;  
}  
  
int a = 5;  
modificar(&a);  
printf("%d", a); // imprime 42
```

◆ 10. Punteros a Matrices (2D arrays)

```
int m[2][2] = {{1,2}, {3,4}};  
int (*ptr)[2] = m;  
  
printf("%d\n", ptr[1][1]); // imprime 4
```