MINICURSO DE C SOBRE GNU/LINUX

Basado en: Diapositivas de "Introducción a C sobre GNU/Linux" (GSYC, Universidad Rey Juan Carlos) y "Fundamentos de sistemas operativos: una aproximación práctica usando Linux"

PARTE I: INTRODUCCIÓN Y CONCEPTOS FUNDAMENTALES

1. CARACTERÍSTICAS DEL LENGUAJE C

C es un **lenguaje de programación imperativo estructurado** con características distintivas:

- **Programación imperativa estructurada:** Las instrucciones se ejecutan secuencialmente con control de flujo mediante sentencias condicionales y bucles.
- **Relativamente de "bajo nivel":** Dentro de los lenguajes de alto nivel, C es uno de los de menor abstracción, permitiendo acceso directo a memoria mediante punteros.
- **Lenguaje simple:** La funcionalidad principal está en las bibliotecas estándar, no en características del lenguaje.
- **Gestión de tipos fundamentales:** Principalmente maneja números (enteros de distintos tamaños), caracteres y direcciones de memoria.
- **Tipado débil:** El compilador no se queja si intentas asignar variables de distinto tamaño o signo.

2. COMPILACIÓN

La compilación en C se compone de **tres fases distintas**, que el programa `gcc` automatiza:

Preprocesado

Incluye ficheros de cabecera (`#include`), elimina comentarios, procesa directivas del preprocesador.

Compilación (compiling)

Genera código objeto (`.o`) a partir del código fuente, pasando internamente por código ensamblador.

Enlazado (linking)

A partir de uno o varios ficheros objeto y bibliotecas, genera el ejecutable final.

Comandos en la práctica:

```
# Preprocesado y compilación (genera .o)
gcc -g -c -Wall -Wshadow -Wvla programa.c

# Enlazado (genera el ejecutable)
gcc -g -o programa programa.o
```

Opciones importantes:

- `-g` : Incluye información de depuración
- `-c` : Solo compila, no enlaza
- `-Wall` : Muestra todos los warnings
- `-Wshadow` : Avisa de variables que ocultan otras
- `-Wvla` : Avisa de arrays de tamaño variable
- `-o fichero` : Especifica el nombre del ejecutable

Nota importante: Los warnings deben tratarse como errores durante el aprendizaje. Indican problemas potenciales en el código.

3. HOLA MUNDO: PRIMER PROGRAMA

El programa más simple que podemos escribir en C:

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

/* Comentario de una línea */
int
main(int argc, char *argv[])
{
    printf("hola mundo");
    exit(EXIT_SUCCESS);
}
```

Disección del programa:

- `#include <stdlib.h>` : Incluye la biblioteca estándar (funciones como `exit()`)
- `#include <stdio.h>` : Incluye funciones de entrada/salida (como `printf()`)
- `int main(...)` : Función principal, **punto de entrada** del programa
- `argc` : Número de argumentos pasados al programa
- `char *argv[]` : Array de cadenas con los argumentos
- `printf()` : Imprime texto por pantalla
- `exit(EXIT_SUCCESS)` : Termina el programa correctamente (status 0)

Cosas importantes:

- Los `#include` deben seguir un orden específico (consultar manuales)
- Los comentarios NO pueden estar anidados (`/* /* */ */` es un error)
- Todas las sentencias terminan con punto y coma (`;`)
- Un **bloque** es un grupo de sentencias entre llaves `{}` que se trata como una única sentencia

4. ACCESO A AYUDA

Las páginas del manual de Unix/Linux son imprescindibles:

```
# Acceder a manual
man [sección] asunto

# Ejemplos
man 1 gcc          # comando gcc
man 2 fork         # llamada al sistema fork()
man 3 printf       # función printf() de biblioteca

# Buscar referencias a una palabra
apropos palabra
```

Secciones de interés:

- **1**: Comandos del sistema
- **2**: Llamadas al sistema
- **3**: Funciones de biblioteca C

PARTE II: TIPOS DE DATOS Y VARIABLES

5. TIPOS DE DATOS FUNDAMENTALES

En este curso usamos principalmente **tipos enteros**:

Tipo	Descripción	Tamaño	Ejemplo
`char`	Carácter con signo	1 byte	`'a'` , `12`
`int`	Entero con signo	4 bytes (x86_64)	`77` , `-11`
`unsigned char`	Carácter sin signo	1 byte	usado para bits
`unsigned int`	Entero sin signo	4 bytes (x86_64)	`77`
`long`	Entero largo con signo	variable	`431414341L`
`long long`	Entero más largo	8 bytes	`432423432423LL`
`void`	Vacío	N/A	para punteros genéricos

Comportamiento del tipado débil:

- C no se queja si asignas valores de distinto tamaño o signo
- Si asignas a una variable más pequeña, se **trunca**
- Los desbordes de variables **no se detectan automáticamente**
- El signo solo se tiene en cuenta en comparaciones

6. DECLARACIÓN E INICIALIZACIÓN DE VARIABLES

Variables globales (fuera de funciones):

- Se ven desde cualquier función del fichero
- Se localizan en el **segmento de datos**
- Si no se inicializan, se ponen a `0` automáticamente

Variables locales (dentro de funciones):

- Se ven solo dentro del bloque donde se declaran
- Se localizan en la **pila** (stack)

- Si no se inicializan, tienen valor **indeterminado**

Ejemplo:

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

int x = 1;         /* variable global, inicializada */
int k;             /* variable global, valor 0 */

int
main(int argc, char *argv[])
{
    int i, q = 1, u = 12;  /* variables locales */
    char c;                 /* variable local sin inicializar */
    char p = '0';           /* variable local inicializada */

    c = 'z';
    i = 13;

    exit(EXIT_SUCCESS);
}
```

Variables static:

Una variable `static` declarada dentro de una función conserva su valor entre invocaciones, porque se localiza en el segmento de datos, no en la pila:

```c
void
contador(void)
{
    static int llamadas = 0;
    llamadas++;
    printf("Llamada número %d\n", llamadas);
}
```

Literales (constantes):

```
777          /* Decimal */
0x777        /* Hexadecimal */
0777         /* Octal */
'a'          /* Carácter */
'\92'        /* Carácter con código ASCII octal */
431414341L   /* Long */
432423432423LL /* Long long */
```

7. CONSTANTES CON ENUM

Las constantes enteras se definen con **tipos enumerados** (`enum`), NO con `const` o `#define` :

```
enum {
    Lun,              /* valor 0 */
    Mar,              /* valor 1 */
    Mier,             /* valor 2 */
    Jue,              /* valor 3 */
    Vier,             /* valor 4 */
    Ndias,            /* valor 5 */
    SalarioBase = 2580,  /* valor asignado explícitamente */
};
```

Los valores se asignan consecutivamente si no se especifican.

PARTE III: OPERADORES

8. OPERADORES ARITMÉTICOS

Operador	Significado
+	Suma (enteros o reales)
-	Resta (enteros o reales)
*	Multiplicación (enteros o reales)

Operador	Significado
`/`	División (enteros o reales)
`%`	Módulo (solo enteros)

9. OPERADORES LÓGICOS

Las operaciones lógicas devuelven un entero: 0 = FALSE, cualquier otro valor = TRUE.

Operador	Significado
`a && b`	AND: 1 si ambos son distintos de 0
`a || b`	OR: 0 si ambos son iguales a 0
`!a`	NOT: 1 si `a` es 0, 0 en otro caso

10. OPERADORES DE RELACIÓN

Operador	Significado
`a < b`	¿a menor que b?
`a > b`	¿a mayor que b?
`a <= b`	¿a menor o igual que b?
`a >= b`	¿a mayor o igual que b?
`a == b`	¿a igual que b?
`a != b`	¿a distinto que b?

11. OPERADORES DE ASIGNACIÓN

Operador	Significado
`=`	Asignación simple
`+=`	Suma y asignación
`-=`	Resta y asignación
`*=`	Multiplicación y asignación
`/=`	División y asignación
`%=`	Módulo y asignación
`++`	Incremento (pre o post)
`--`	Decremento (pre o post)

12. OPERADORES DE BITS Y OTROS

Operador	Significado
`&` (unario)	Dirección-de: obtiene la dirección de una variable
`*` (unario)	Indirección (dereference): accede al valor apuntado
`~` (unario)	Complemento a uno (NOT de bits)
`&` (binario)	AND de bits
`^`	XOR de bits
`|`	OR de bits
`<<`	Desplazamiento binario a la izquierda
`>>`	Desplazamiento binario a la derecha
`sizeof`	Tamaño en bytes de un tipo o variable

Advertencia: Operaciones de bits sobre tipos con signo pueden dar resultados sorprendentes.

13. PRECEDENCIA Y ASOCIATIVIDAD

Precedencia (mayor a menor)	Operadores	Asociatividad
1	`() [] -> .`	izq. a der.
2	`! ++ -- * & ~ sizeof`	der. a izq.
3	`* / %`	izq. a der.
4	`+ -`	izq. a der.
5	`<< >>`	izq. a der.
6	`< <= > >=`	izq. a der.
7	`== !=`	izq. a der.
8	`&`	izq. a der.
9	`^`	izq. a der.
10	`|`	izq. a der.
11	`&&`	izq. a der.
12	`||`	izq. a der.
13	`?:`	der. a izq.
14	`= += -= *= /= %= &= ...`	der. a izq.
15	`,`	izq. a der.

PARTE IV: CONTROL DE FLUJO

14. SENTENCIA IF

La sentencia `if` ejecuta código condicionalmente basándose en si una expresión es verdadera (distinta de 0) o falsa (0):

```
if (expresión) {
    sentencias1...
} else if (otra_expresión) {
    sentencias2...
} else {
    sentencias3...
}
```

- Los paréntesis son **obligatorios**
- Si solo hay una sentencia, las llaves son opcionales
- La expresión se evalúa a un entero

Ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int edad = 25;

    if (edad >= 18) {
        printf("Eres mayor de edad\n");
    } else {
        printf("Eres menor de edad\n");
    }

    exit(EXIT_SUCCESS);
}
```

15. SENTENCIA SWITCH

La sentencia `switch` compara un valor contra múltiples casos:

```
switch (expresión) {
case valor1:
    sentencias1...
    break;
case valor2:
    sentencias2...
    break;
default:
    sentencias_default...
    break;
}
```

- El flujo **pasa al case que corresponde** con el valor
- Si no hay `break`, continúa al siguiente case (fall-through)
- Si no entra en ningún case, ejecuta `default` (si existe)
- `break` termina el switch
- `continue` pasa a la siguiente iteración (en bucles)

Ejemplo:

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

int
main(int argc, char *argv[])
{
    int opcion = 2;

    switch (opcion) {
    case 1:
        printf("Opción 1\n");
        break;
    case 2:
        printf("Opción 2\n");
        break;
    default:
        printf("Opción desconocida\n");
        break;
    }

    exit(EXIT_SUCCESS);
}
```

16. SENTENCIA WHILE

Ejecuta un bloque mientras la condición es verdadera:

```c
while (expresión) {
    sentencias...
}
```

- Se itera **hasta que la expresión evalúa a 0** (falso)
- `break` rompe el bucle
- `continue` salta a la siguiente iteración

Ejemplo:

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

int
main(int argc, char *argv[])
{
    int contador = 0;

    while (contador < 5) {
        printf("Iteración %d\n", contador);
        contador++;
    }

    exit(EXIT_SUCCESS);
}
```

17. SENTENCIA DO-WHILE

Similar a `while`, pero ejecuta el bloque **al menos una vez** antes de evaluar la condición:

```c
do {
    sentencias...
} while (expresión);
```

Ejemplo:

```c
int numero;
do {
    printf("Introduce un número entre 1 y 10: ");
    scanf("%d", &numero);
} while (numero < 1 || numero > 10);
```

18. SENTENCIA FOR

Ejecuta un bloque un número determinado de veces:

```
for (inicialización; condición; actualización) {
    sentencias...
}
```

- **Inicialización:** Se ejecuta una única vez, antes de la primera iteración
- **Condición:** Se evalúa antes de cada iteración; si es falsa, se sale del bucle
- **Actualización:** Se ejecuta al final de cada iteración
- `break` rompe el bucle
- `continue` salta a la siguiente iteración

Ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    /* Bucle básico */
    for (int i = 0; i < 5; i++) {
        printf("Valor: %d\n", i);
    }

    /* Bucle con inicialización vacía */
    int j = 10;
    for (; j > 0; j--) {
        printf("Cuenta atrás: %d\n", j);
    }

    exit(EXIT_SUCCESS);
}
```

PARTE V: FUNCIONES

19. DEFINICIÓN Y DECLARACIÓN DE FUNCIONES

Una **función** es un bloque de código reutilizable que puede recibir argumentos y retornar un valor.

Sintaxis:

```
/* Declaración (prototipo) - debe aparecer antes de usarla */
tipo_retorno nombre_funcion(tipo_arg1 arg1, tipo_arg2 arg2, ...);

/* Definición */
tipo_retorno
nombre_funcion(tipo_arg1 arg1, tipo_arg2 arg2, ...)
{
    /* cuerpo de la función */
    return valor;
}
```

Características importantes:

- La función debe estar **declarada antes de usarla**, pero puede estar **definida después**
- **Los argumentos siempre son por valor** (se pasa una copia)
- Para argumentos por referencia, usamos **punteros**
- Si no devuelve nada, el tipo es `void`
- Si no tiene argumentos, el prototipo es `tipo nombre(void)`

Ejemplo:

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

/* Prototipo (declaración) */
int suma(int a, int b);
void saludar(char *nombre);

int
main(int argc, char *argv[])
{
    int resultado = suma(10, 20);
    printf("10 + 20 = %d\n", resultado);

    saludar("Juan");

    exit(EXIT_SUCCESS);
}

/* Definición de suma */
int
suma(int a, int b)
{
    return a + b;
}

/* Definición de saludar */
void
saludar(char *nombre)
{
    printf("¡Hola, %s!\n", nombre);
}
```

PARTE VI: PUNTEROS Y MEMORIA

20. CONCEPTOS BÁSICOS DE PUNTEROS

Una **variable** ocupa una o varias direcciones contiguas de memoria con los bytes correspondientes.

Un **puntero** es una variable que **contiene una dirección de memoria**.

Sintaxis:

```
tipo *nombre;    /* declara un puntero a tipo */

int *ptr;        /* puntero a entero */
char *ptr_char;  /* puntero a carácter */
```

Operadores:

Operador	Significado
`&variable`	Obtiene la dirección de una variable (address-of)
`*puntero`	Accede al valor apuntado (dereference)

Ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int x = 10;
    int *ptr;              /* Declarar puntero */

    ptr = &x;              /* ptr apunta a x */

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

    *ptr = 20;             /* Cambiar x a través del puntero */
    printf("Nuevo valor de x: %d\n", x);

    exit(EXIT_SUCCESS);
}
```

Regla importante:

Un puntero que **no apunta a nada es peligroso**. Use `NULL` para punteros no inicializados:

```c
int *ptr = NULL;  /* seguro, no apunta a ningún lado */

if (ptr != NULL) {
    printf("%d\n", *ptr);  /* solo si ptr es válido */
}
```

21. ARITMÉTICA DE PUNTEROS

Los punteros pueden sumarse y restarse. Las operaciones se hacen en **múltiplos del tamaño** del tipo apuntado:

```c
char *cptr;   /* char ocupa 1 byte */
int *iptr;    /* int ocupa 4 bytes */

cptr = cptr + 4;   /* avanza 4 bytes */
iptr = iptr + 4;   /* avanza 16 bytes (4 * 4) */
```

Ejemplo:

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

int
main(int argc, char *argv[])
{
    int numeros[5] = {10, 20, 30, 40, 50};
    int *ptr = numeros;

    /* Acceder mediante aritmética de punteros */
    printf("Primer elemento: %d\n", *ptr);
    printf("Segundo elemento: %d\n", *(ptr + 1));
    printf("Tercero elemento: %d\n", *(ptr + 2));

    exit(EXIT_SUCCESS);
}
```

22. ARRAYS EN C

Un **array** es una secuencia contigua de elementos del mismo tipo. Los arrays en C son básicamente **azúcar sintáctico para punteros**:

```
int lista[N];      /* Reserva memoria para N enteros */
lista[NUM] = 3;    /* Escribe 3 en la posición NUM */
```

Características:

- Los índices van de **0 a N-1** (no se comprueban los límites)
- **No hay sobrecarga automática de límites** - es responsabilidad del programador
- El operador `sizeof` sobre un array devuelve el tamaño en bytes, **NO el número de elementos**

Inicialización de arrays:

```
/* Especificar tamaño e inicializar */
int lista1[5] = {1, 2, 3, 4, 5};

/* El compilador deduce el tamaño */
int lista2[] = {1, 2, 3, 4, 5};

/* Inicializar parcialmente */
int lista3[5] = {1, 2, 3};  /* los restantes son 0 */

/* PELIGRO: esto compila pero es un error */
int lista4[4] = {1, 2, 3, 4, 5};  /* ¡overflow! */
```

Ejemplo completo:

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

int
main(int argc, char *argv[])
{
    int numeros[5] = {10, 20, 30, 40, 50};

    /* Acceso mediante índices */
    for (int i = 0; i < 5; i++) {
        printf("numeros[%d] = %d\n", i, numeros[i]);
    }

    /* Acceso mediante punteros */
    int *ptr = numeros;
    for (int i = 0; i < 5; i++) {
        printf("*(ptr + %d) = %d\n", i, *(ptr + i));
    }

    exit(EXIT_SUCCESS);
}
```

23. PASAR DIRECCIONES COMO ARGUMENTOS

Para modificar una variable desde una función, pasamos su dirección (un puntero):

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

void
cambiar_valor(int *ptr)
{
    *ptr = 100;    /* Modifica el valor apuntado */
}

int
main(int argc, char *argv[])
{
    int x = 10;

    printf("Antes: x = %d\n", x);
    cambiar_valor(&x);  /* Pasar dirección */
    printf("Después: x = %d\n", x);

    exit(EXIT_SUCCESS);
}
```

24. CADENAS DE CARACTERES (STRINGS)

Una **cadena de caracteres** es un array de caracteres terminado con el carácter nulo `'\0'` :

```c
char str[] = "hola";        /* Equivalente a: */
char str2[] = {'h','o','l','a','\0'};
```

Características:

- Si no termina en `'\0'` , **NO es una string válida**
- El carácter nulo marca el final
- Las funciones de string esperan el nulo terminal

Funciones importantes de string:

Función	Significado
`strlen(s)`	Devuelve el tamaño sin contar nulo

Función	Significado
`strcmp(s1, s2)`	Compara dos strings (0 si son iguales)
`strcat(dest, src)`	Concatena `src` al final de `dest`
`strcpy(dest, src)`	Copia `src` a `dest`
`snprintf(dest, n, fmt, ...)`	Similar a `printf`, pero escribe en `dest`

Ejemplo:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    char nombre[50];
    char saludo[100];

    /* Leer una string segura */
    snprintf(nombre, 50, "Juan");

    /* Usar snprintf para construir string */
    snprintf(saludo, 100, "Hola, %s", nombre);
    printf("%s\n", saludo);

    /* Usar strlen */
    printf("Longitud: %zu\n", strlen(nombre));

    exit(EXIT_SUCCESS);
}
```

PARTE VII: ESTRUCTURAS (REGISTROS)

25. DEFINICIÓN Y USO DE ESTRUCTURAS

Una **estructura** es un tipo de dato compuesto que agrupa variables de distintos tipos:

```c
struct Coordenada {
    int x;
    int y;
};

struct Coordenada c = {13, 33};  /* inicialización */
```

Características:

- El tamaño en memoria **no es necesariamente la suma** de sus campos (hay relleno)
- Solo se pueden hacer 3 operaciones: copiar/asignar, obtener dirección (`&`), acceder a campos
- Usar `typedef` para crear un tipo más cómodo

Ejemplo con typedef:

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

typedef struct {
    int x;
    int y;
    char nombre[50];
} Punto;

int
main(int argc, char *argv[])
{
    Punto p1 = {10, 20, "origen"};
    Punto p2;

    p2.x = 30;
    p2.y = 40;

    printf("p1: (%d, %d) - %s\n", p1.x, p1.y, p1.nombre);
    printf("p2: (%d, %d)\n", p2.x, p2.y);

    exit(EXIT_SUCCESS);
}
```

Acceso a campos mediante punteros:

El operador `->` accede a campos de estructuras a través de punteros:

```
Punto *ptr = &p1;

ptr->x = 50;           /* Equivalente a: (*ptr).x = 50; */
printf("%d\n", ptr->y);
```

PARTE VIII: MEMORIA DINÁMICA

26. MALLOC Y FREE

Para reservar memoria en tiempo de ejecución, usamos **malloc** y **free**:

malloc:

```
void *malloc(size_t bytes);
```

- Reserva `bytes` bytes en el **heap**
- Devuelve un puntero a la memoria reservada
- Si no hay memoria, devuelve `NULL`
- La memoria puede contener cualquier valor (basura)

free:

```
void free(void *ptr);
```

- Libera la memoria previamente reservada con `malloc`
- **No se puede liberar memoria que no vino de malloc**
- Debe liberarse cuando ya no se necesita

Ejemplo:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    /* Reservar memoria para 10 enteros */
    int *numeros = malloc(10 * sizeof(int));

    if (numeros == NULL) {
        fprintf(stderr, "No hay memoria\n");
        exit(EXIT_FAILURE);
    }

    /* Usar la memoria */
    for (int i = 0; i < 10; i++) {
        numeros[i] = i * 10;
    }

    /* Liberar la memoria */
    free(numeros);
    numeros = NULL;  /* Buena práctica: poner NULL después */

    exit(EXIT_SUCCESS);
}
```

PARTE IX: ARGUMENTOS DE MAIN

27. ARGC Y ARGV

La función `main` recibe argumentos del sistema:

```c
int main(int argc, char *argv[])
```

- **`argc`** : Número de argumentos (incluyendo el nombre del programa)
- **`argv`** : Array de strings con los argumentos
  - `argv[0]` : Nombre del programa

- `argv[1]` : Primer argumento
- `argv[n]` : Último argumento
- Siempre termina con `NULL`

Ejemplo:

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

int
main(int argc, char *argv[])
{
    printf("Número de argumentos: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    /* Ejemplo de uso útil */
    if (argc < 2) {
        fprintf(stderr, "Uso: %s <nombre>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    printf("Hola, %s\n", argv[1]);

    exit(EXIT_SUCCESS);
}
```

Para compilar y ejecutar:

```
gcc -g -c -Wall -Wshadow -Wvla programa.c
gcc -g -o programa programa.o
./programa Juan María Pedro
```

PARTE X: BIBLIOTECAS ESTÁNDAR

28. BIBLIOTECAS MÁS COMUNES

Las bibliotecas proporcionan funciones predefinidas:

Biblioteca	Propósito	Ejemplos
`<stdio.h>`	Entrada/Salida estándar	`printf()`, `scanf()`, `fprintf()`
`<stdlib.h>`	Librería estándar de C	`malloc()`, `free()`, `exit()`, `atoi()`
`<string.h>`	Operaciones con strings	`strlen()`, `strcat()`, `strcmp()`, `strcpy()`
`<math.h>`	Funciones matemáticas	`sqrt()`, `sin()`, `cos()`, `pow()`
`<ctype.h>`	Clasificación de caracteres	`isdigit()`, `isalpha()`, `toupper()`
`<unistd.h>`	Llamadas al sistema Unix	`fork()`, `read()`, `write()`, `close()`
`<fcntl.h>`	Control de ficheros	`open()`, `creat()`

29. PRINTF - SALIDA FORMATEADA

Función para imprimir con formato:

```c
int printf(const char *format, ...);
```

El parámetro `format` contiene texto con especificadores que se reemplazan con los argumentos:

Especificador	Tipo	Ejemplo
`%d`	Entero decimal	`printf("%d", 42)` → `42`
`%x`	Hexadecimal sin signo	`printf("%x", 255)` → `ff`
`%o`	Octal sin signo	`printf("%o", 8)` → `10`
`%c`	Carácter	`printf("%c", 'A')` → `A`
`%s`	String	`printf("%s", "hola")` → `hola`

Especificador	Tipo	Ejemplo
`%p`	Puntero	`printf("%p", ptr)` → `0x7fff...`
`%f`	Punto flotante	`printf("%f", 3.14)` → `3.140000`
`%%`	Carácter `%`	`printf("%%")` → `%`

Ejemplo:

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

int
main(int argc, char *argv[])
{
    int edad = 25;
    char nombre[50] = "Juan";
    float altura = 1.75;

    printf("Nombre: %s\n", nombre);
    printf("Edad: %d años\n", edad);
    printf("Altura: %.2f metros\n", altura);
    printf("Número en hex: %x\n", 255);

    exit(EXIT_SUCCESS);
}
```

PARTE XI: DEPURACIÓN Y ANÁLISIS

30. GDB - DEPURADOR

`gdb` permite inspeccionar y depurar programas:

Uso básico:

```
gdb programa
```

Comandos principales:

```
run [args]          # ejecutar el programa
break función       # poner punto de ruptura
continue            # continuar ejecución
step                # ejecutar siguiente línea (entra en funciones)
next                # siguiente línea (salta funciones)
print variable      # mostrar valor de variable
backtrace (bt)      # mostrar pila de llamadas
frame N             # seleccionar marco N
info locals         # variables locales
info args           # argumentos de función
whatis variable     # tipo de variable
quit                # salir
```

Ejemplo de sesión:

```
gdb ./mi_programa
(gdb) break main
Breakpoint 1 at 0x400456
(gdb) run argumento1
Starting program: ./mi_programa argumento1
Breakpoint 1, main () at programa.c:10
10    int x = 10;
(gdb) next
11    printf("x = %d\n", x);
(gdb) print x
$1 = 10
(gdb) continue
x = 10
Program exited normally
(gdb) quit
```

31. VALGRIND - ANÁLISIS DE MEMORIA

Herramienta para detectar errores de memoria automáticamente:

```
# Detectar memory leaks
valgrind --leak-check=full ./programa

# Opciones útiles
valgrind --leak-check=full --show-leak-kinds=all ./programa
valgrind --track-origins=yes ./programa
```

Ejemplo de salida:

```
==1234== HEAP SUMMARY:
==1234==     in use at exit: 0 bytes in 0 blocks
==1234==   total heap usage: 1 allocs, 1 frees, 100 bytes allocated
==1234==
==1234== LEAK SUMMARY:
==1234==     definitely lost: 0 bytes in 0 blocks
==1234==     indirectly lost: 0 bytes in 0 blocks
==1234==       possibly lost: 0 bytes in 0 blocks
==1234==     still reachable: 0 bytes in 0 blocks
==1234==     suppressed: 0 bytes in 0 blocks
==1234==
==1234== For counts of detected and suppressed errors, rerun with: -s
==1234== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

PARTE XII: PROGRAMAS CON VARIOS FICHEROS

32. ORGANIZACIÓN CON MÚLTIPLES FICHEROS

Para proyectos más grandes, se divide el código en varios archivos:

- **Ficheros** `.c` : Implementación
- **Ficheros** `.h` : Declaraciones (cabeceras)

Reglas importantes:

- Una variable debe estar **definida en un único fichero** `.c`
- Variables globales usadas en otros ficheros se declaran con `extern`
- Funciones y variables `static` no son visibles desde otros ficheros
- Incluir ficheros locales con comillas: `#include "fichero.h"`

- No incluir dos veces el mismo fichero (usar guardias)

Ejemplo: fichero.h

```c
#ifndef FICHERO_H
#define FICHERO_H

int suma(int a, int b);
void saludar(char *nombre);

#endif /* FICHERO_H */
```

Ejemplo: fichero.c

```c
#include "fichero.h"
#include <stdio.h>

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

void
saludar(char *nombre)
{
    printf("Hola, %s\n", nombre);
}
```

Ejemplo: main.c

```c
#include "fichero.h"
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int resultado = suma(10, 20);
    saludar("Juan");
    exit(EXIT_SUCCESS);
}
```

Compilación:

```
gcc -g -c -Wall -Wshadow -Wvla fichero.c
gcc -g -c -Wall -Wshadow -Wvla main.c
gcc -g -o programa fichero.o main.o
```

RESUMEN RÁPIDO DE CONSTRUCCIONES

```c
/* Tipos de datos */
int, char, long, unsigned int, float, double

/* Declaración de variables */
int x = 10;           /* global */
static int y = 20;    /* estática */

/* Arrays */
int lista[5] = {1, 2, 3, 4, 5};
char str[] = "hola";

/* Punteros */
int *ptr = &x;
printf("%d\n", *ptr);

/* Estructuras */
struct Punto { int x; int y; };
typedef struct Punto Punto;

/* Funciones */
int suma(int a, int b) { return a + b; }
void mostrar(void) { printf("hola\n"); }

/* Control de flujo */
if (condición) { ... }
else { ... }

switch (valor) {
case 1: ... break;
default: ... break;
}

while (condición) { ... }
do { ... } while (condición);
for (ini; cond; act) { ... }

/* Memoria dinámica */
int *ptr = malloc(10 * sizeof(int));
if (ptr != NULL) { ... }
free(ptr);
```

```c
ptr = NULL;

/* Entrada/salida */
printf("formato", argumentos);
scanf("formato", &variable);

/* Strings */
strlen(s), strcmp(s1, s2), strcpy(dest, src), strcat(s1, s2)

/* Compilación */
gcc -g -c -Wall -Wshadow -Wvla archivo.c
gcc -g -o programa archivo.o
```