



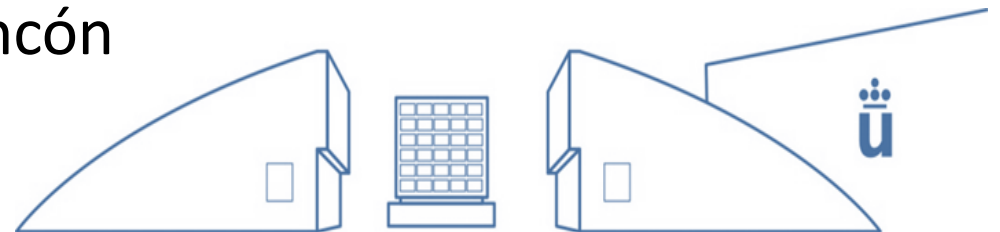
Tema 3

Introducción a la programación de Sistemas Operativos

Alberto Sánchez

Sofia Bayona

Luis Rincón





- **Introducción**
- Variables y operadores
- Punteros
- Funciones
- Entrada/Salida
- Aplicaciones modulares



- C es uno de los lenguajes de programación más extendidos hoy en día
- Historia de C:
 - 1970 Difícil de programar en PDP-11. Bell desarrolla un nuevo S.O. para PDP-11 -> UNICS -> UNIX.
 - 1972 *Dennis Ritchie* diseña el lenguaje de programación C
 - Fabricantes crean sus propios compiladores -> pérdida de portabilidad
 - 1980 Estandarización de C. ANSI C
 - 1983 surge C++, versión de C orientada a objetos (Bjarne Stroustrup)



- Estructurado: su esquema de programación es imperativo vs. Orientado a objetos (C++).
- Compilado. Las órdenes son transformadas a lenguaje máquina que se almacena en un archivo ejecutable.
- Potente. Aunque es un lenguaje de alto nivel, tiene capacidades de bajo nivel



```
#include <stdio.h>

int main()
{
    //Llamada a printf
    printf("Hola mundo\n");
    return 0;
}
```

- **#include** : es una directiva de preprocesador. Lo que hace es copiar todo el código del archivo `stdio.h` y pegarlo en esa línea.
- **int main()**: es la función que hace de punto de entrada del programa. Todo programa en C debe tener una función *main*.



- **Preprocesado:** Las directrices que comienzan por # son directrices del preprocesador. Son interpretadas por el preprocesador, normalmente con sustituciones.
- **Compilación:** El código fuente ya preprocesado se transforma en código máquina.
- **Enlazado:** se integra todo el código objeto de las distintas unidades compiladas en un solo programa ejecutable.



- El compilador que se utiliza habitualmente en entornos UNIX es *gcc* (GNU C Compiler)
- Sintaxis:
 - **`gcc archivo.c -c -o archivo.o`**: Preprocesa y compila *archivo.c* generando el fichero objeto *archivo.o*
 - **`gcc archivo1.o archivo2.o -o archivo`**: Enlaza ficheros *.o* y genera el ejecutable *archivo*
 - **`gcc archivo.c -o archivo`**: Preprocesa, compila y enlaza *archivo.c* generando el ejecutable *archivo*



- Muchas más opciones de gcc:
 - Estandar de C
 - Estándar de C de 1999: -std=c99
 - Estándar de C de 2011: -std=c11. Por defecto a partir de gcc-5
 - Optimización de rendimiento
 - Diferentes niveles -O2, -O3
 - Eficiencia de cache: -Os
 - Warnings
 - -Wall –Wextra
 - Durante realización de pruebas: -Werror
 - Para saber más, consultar el manual (man)
- Entornos de compilación adicionales: Geany, NetBeans, Eclipse, Dev C++ ...



- Introducción
- **Variables y operadores**
- Punteros
- Funciones
- Entrada/Salida
- Aplicaciones modulares



- Actualmente se recomienda incluir `<stdint.h>` y usar los tipos estándar
 - Enteros (`int`)
 - `int8_t`, `int16_t`, `int32_t`, `int_64t` : entero con signo
 - `uint8_t`, `uint16_t`, `uint32_t`, `uint_64t` : entero sin signo
 - `size_t` entero capaz de contener el mayor tamaño de memoria disponible
 - `float` : número en coma flotante, precisión simple 32 bits
 - `double` : número en coma flotante, doble precisión 64 bits
 - `char`: carácter. No se recomienda su uso para un único carácter, solo para secuencias
 - Para un carácter mejor usar `uint8_t`
 - `void` : tipo vacío



- Tipos enumerados

- Rango de valores pequeño
- Cada valor identificado por un nombre

`enum tMesesAnyo {enero=1, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre};`

- Definición de tipos con typedef

- Permite crear nuevos tipos de datos
- Mejora legibilidad del código

`typedef <tipo> <alias>;`



- Se declaran al principio de un bloque con **static**
- Son visibles dentro de ese bloque
- Al finalizar el bloque no se destruyen, conservando su valor entre distintas ejecuciones del bloque al que pertenecen
 - Una variable global estática solo es accesible desde el fichero donde está declarada.
 - Una variable local estática conserva el valor entre distintas llamadas a la función.



- Declaración:
 - Usando la directiva del preprocesador `#define`: `#define MAX_SIZE 64`
 - Usando `const`: `const int MAX_SIZE = 64;`
- Constantes carácter: encerrado entre apóstrofes
 - Uso de `\` para caracteres especiales

Representación	Descripción	Carácter	Descripción ASCII
<code>'\n'</code>	Nueva línea	LF	10
<code>'\t'</code>	Tabulación	HT	9
<code>'\b'</code>	Espacio	BS	8
<code>'\\'</code>	Barra invertida	<code>\</code>	92
<code>'\"'</code>	Comilla simple	<code>'</code>	39
<code>'\"'</code>	Doble comilla	<code>' '</code>	34
<code>'\0'</code>	Carácter nulo		0



- Aritméticos: +, -, *, /, % (módulo)
- Lógicos: no existe un tipo de datos *boolean*. En C se utilizar enteros.
 - 0 es falso y distinto de 0 es verdadero.
 - A partir de C99 se puede usar `<stdbool.h>` que define `true` como 1 y `false` como 0.
 - &&, ||, ! : and, or y not.
- Relacionales (devuelven un entero que indica si se cumple o no la condición) :
 - >, >=, <, <=, !=, == (Obs: comparador de igualdad).



- **Asignación:**
 - `a = b` : asigna a a el valor de b
 - `a++`, `++a`, `a--`, `--a` : postincremento, preincremento, postdecremento, predecremento
 - `+=`, `-=`, `*=`, `/=` : suma, resta, multiplicación y división con asignación.
 - Ejemplo: `a += b`; es equivalente a: `a = a + b`;
- **Operadores de bit:**
 - `&`, `|`, `^`, `~` : and, or, xor y not binarios
- **Otros:**
 - `<<`, `>>` : desplazamiento de bits a izquierda y a derecha
 - `sizeof(variable o tipo)`: tamaño en memoria de la variable o tipo



- Conversión de tipos:
 - Implícita:
 - En expresiones el tipo mas bajo promociona al mas alto
 - En asignaciones el valor del lado derecho se convierte al tipo de la variable de la izquierda
 - Explicita (*casting*)
 - (tipo) expresion



- Precedencia
 - Prioridad de unos operadores frente a otros
 - Puede modificarse con paréntesis
- Asociatividad
 - Define el orden de ejecución para operadores con idéntica precedencia

Operadores	Asociatividad
()	de izq. a der.
! ++ -- -(unario)	de der. a izq.
* / %	de izq. a der.
+ - (binario)	de izq. a der.
< <= > >=	de izq. a der.
== !=	de izq. a der.
&&	de izq. a der.
	de izq. a der.



- Introducción
- Variables y operadores
- **Punteros**
- Funciones
- Entrada/Salida
- Aplicaciones modulares

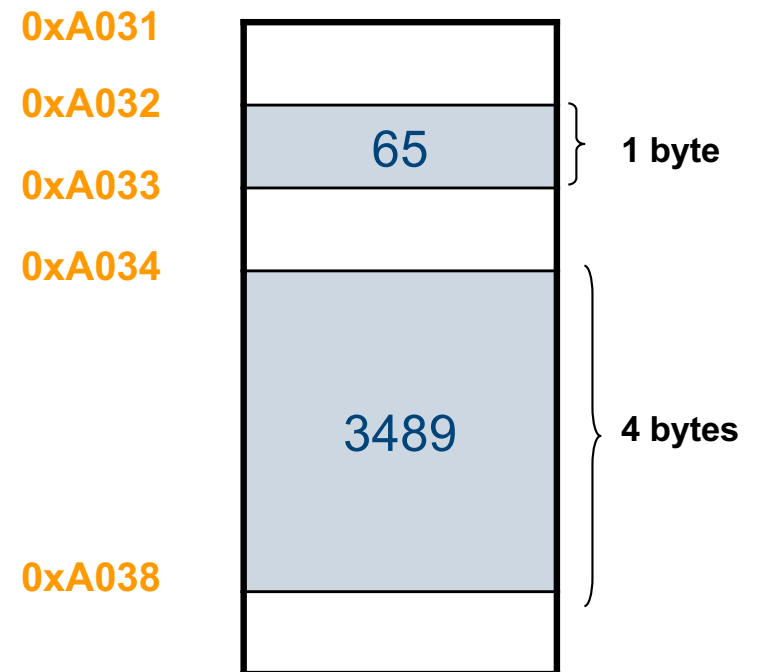


- Concepto de variable

- Región en memoria reservada para almacenar un dato de un tipo determinado
 - Posición de memoria inicial
 - Tamaño en bytes

```
char letra = 'A';
```

```
unsigned int entero = 3489;
```





■ Punteros:

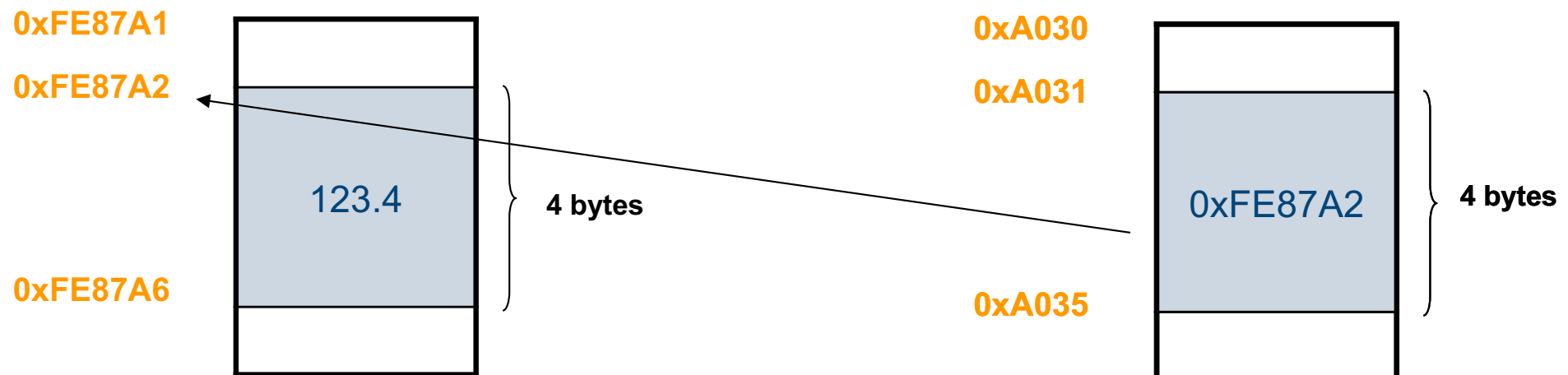
- Variable que representa una posición de memoria que contiene un dato.
- Se declaran anteponiendo un * al nombre de la variable.

tipo * identificador;

- Se puede acceder a una posición de memoria mediante &variable

float datoReal =123.4;

float * pDatoReal = &datoReal;





■ Operaciones:

- El operador & devuelve la dirección de memoria donde se encuentra una variable. Es decir se obtiene un puntero a la variable.
- El operador * *dereferencia* el puntero. Es decir, se accede al valor apuntado.

```
int a = 0;  
int *b;
```

```
b = &a; // b apunta a la dirección donde está a  
*b = 127; // se guarda 127 en la dir. apuntada por b  
//En este punto la variable a tiene valor 127
```

- Operaciones:
 - Los punteros se pueden sumar, restar, etc
 - Esas operaciones se hacen en múltiplos del tamaño del dato apuntado

```
int16_t *a = . . . //a apunta a algún lugar  
a = a + 1; //apunta a sizeof(int16_t) más adelante  
a = a + 4; //apunta a 4*sizeof(int16_t) más adelante
```



- Cuando se declara una variable de tipo puntero, no apunta a ningún sitio “válido”, y no hay memoria reservada para esa variable.
- Para manejar memoria de forma dinámica, se utilizan las llamadas:
 - *void * malloc(size_t size)*: devuelve un puntero a una zona de memoria de *size* bytes. Como devuelve un puntero *void** hay que convertirlo al tipo de puntero que se quiere usar.
 - *void * calloc(size_t nmemb, size_t size)*: además de reservar memoria, inicializa a 0 la memoria reservada.
 - *void * realloc(void *ptr, size_t size)*: redimensiona el espacio asignado de forma dinámica anteriormente a un puntero.
 - *void free(void *ptr)*: libera la zona de memoria apuntada por *ptr* y obtenida previamente con *malloc*



```
float *a;
int *b;

//se reserva espacio en memoria para 24 floats
a = (float *) malloc (24*sizeof(float));
//se reserva espacio en memoria para 16 ceros
b = (int *) calloc (16, sizeof(int));

...
// utilización de a y b
float *tmp_a = realloc(a, 24*sizeof(float));
a = tmp_a;

...
//se libera los espacios de memoria reservados
free(a);
free(b);
```

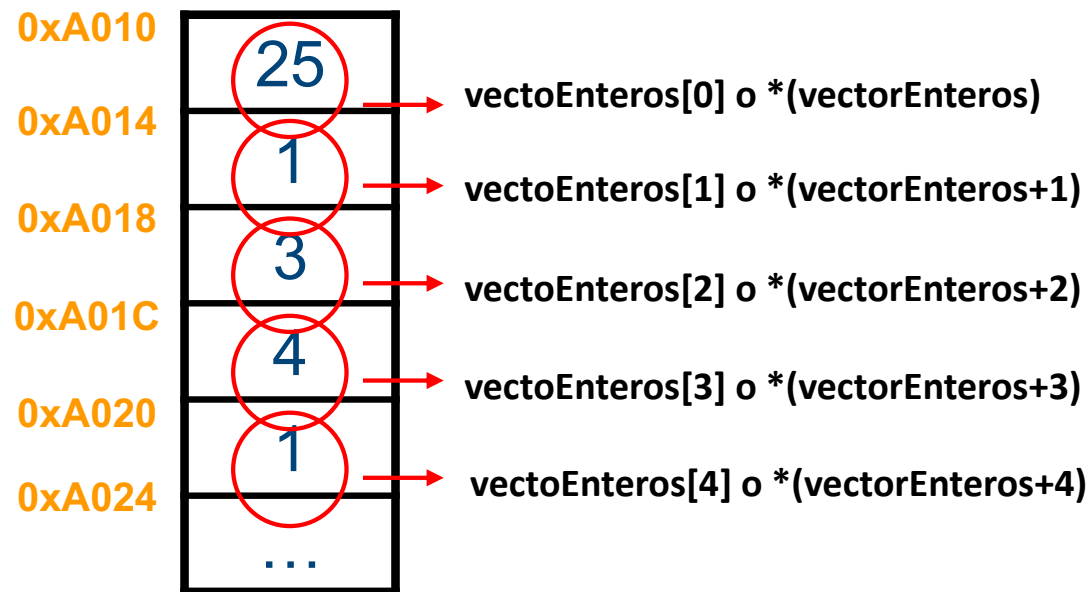

Relación entre punteros y arrays



S I S T E M A S O P E R A T I V O S

- Los arrays se pueden comportar como punteros, y los punteros como arrays.
- Una variable de tipo array es un puntero al primer elemento del array.
- Un puntero al primer elemento de una colección puede usarse como un array.

```
int vectorEnteros[]={25,1,3,4,1};
```





- Diferencias:
 - Un array tiene un tamaño asociado, un puntero no.
 - **Un array debe definir su tamaño en tiempo de compilación**
 - A partir de c99 se permite arrays de longitud variable siempre que sean pequeños
 - Un array es un puntero estático, no se puede cambiar el lugar al que apunta.

```
size_t a[128]; //array de 128 elementos
int s1 = sizeof(a); //s1 valdrá 128*sizeof(size_t)

size_t *b = (size_t *) malloc (128*sizeof(size_t);
//b es un puntero a una colección de 128 enteros
int s2 = sizeof(b);
//s2 valdrá 4 u 8, que es el tamaño de una dirección
```



- En C no existe un tipo de datos específico String para representar las cadenas de caracteres.
 - Se utilizan arrays de caracteres (`char *`, `char[]`) acabados en el caracter `'\0'`
- Una cadena de caracteres entre comillas dobles representa un *string*.

```
char string1[] = "Hola";  
char *string2 = "Hola";  
char *string3 = {'H', 'o', 'l', 'a', '\0'}
```



- Para trabajar con *strings* se utilizan diferentes funciones de biblioteca:
 - `strlen`: devuelve la longitud del string.
 - `strcat`: concatena dos strings.
 - `strcpy`: copia un string sobre otro.
 - `strdup`: duplica un string.
 - `strchr`: busca un carácter en el string.
- Se encuentran en el fichero `<string.h>`

- Son tipos compuestos formados por elementos heterogéneos.
- Dentro del registro cada elemento tiene un identificador y un tipo.

```
//Declaración del registro  
typedef struct Complejo {  
    float real;  
    float imag;  
} Tcomplejo;
```

```
//Declaración de variables del tipo registro  
Tcomplejo uncomplejo, otrocomplejo;
```



- Para acceder a los elementos de un registro estructura se utiliza el operador punto (.)

```
uncomplejo.real = 1.3;  
uncomplejo.imag = 2.7;
```

- Si se trata de un puntero a registro se puede atajar con el operador flecha (->)

```
Tcomplejo *a =  
(Tcomplejo *) malloc (sizeof(Tcomplejo));  
  
a->real = 2.4; //lo mismo que (*a).real = 2.4;  
a->imag = 3.14; //lo mismo que (*a).imag = 3.14;
```



- Introducción
- Variables y operadores
- Punteros
- **Funciones**
- Entrada/Salida
- Aplicaciones modulares



- Las funciones se pueden “declarar”, para que en el resto del código se sepa que existen y como es su interfaz (prototipo).
 - Luego, hay que implementar su funcionalidad (cuerpo).
 - Orden: antes de llamar a una función es necesario conocer su cabecera por lo menos.

```
//Declaración de la función
int media(int a, int b);

--- Código que utiliza media(a,b)

int media(int a, int b)
{
    int result;
    result = (a + b)/2;
    return result;
}
```


- Por defecto, los argumentos de las funciones se pasan por **valor**.
- Si se quiere pasar un argumento por **referencia** se deben usar punteros de forma explícita.

```
//Ejemplo de paso de argumentos por referencia
void intercambia(void *a, void *b) {
    size_t aux;
    aux = (size_t) *a;
    (size_t) *a = (size_t) *b;
    *b = aux;
}

//Llamada a la función
size_t x = 10, y = 20;
. . .
intercambia(&x, &y);
```



- La función `main` es el punto de entrada del programa.
 - Solo puede haber una función `main` por ejecutable (aunque este formado por varios archivos)
- Puede recibir argumentos que serán los argumentos que se le pasen por línea de mandatos a la hora de ejecutar el programa.

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

- `argc`: es un entero que indica el número de argumentos recibido.
- `*argv[]`: es un puntero a un *array* de *strings*. Cada entrada del *array* es uno de los argumentos.
- `*envp[]`: es un puntero a un *array* de *strings*. Cada entrada del *array* es una variable de entorno heredada del proceso padre.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // Imprime número de argumentos
    printf("Se han recibido %d argumentos\n",argc);
    int i;
    for(i = 0; i < argc; i++)
    {
        // Imprime argumento a argumento
        printf("Argumento %d = %s\n",i,argv[i]);
    }
    return 0;
}
```



- Introducción
- Variables y operadores
- Punteros
- Funciones
- **Entrada/Salida**
- Aplicaciones modulares



- Definidas en `<stdio.h>`
- Funciones disponibles para entrada/salida:
 - `printf`: permite escribir en la pantalla
 - `scanf`: para leer del teclado con formato.
 - `getchar/putchar`: leer/escribir un caracter.
 - `gets`: para leer una línea del teclado.
 - `puts`: para escribir una línea en la pantalla.



```
printf ("format", ...);
```

- Format: es el string que se imprimirá en pantalla. Admite una serie de caracteres de sustitución que serán reemplazados en orden por el resto de argumentos e la función.

```
char dia[] = "Domingo";  
int8_t hora = 12;  
int8_t min = 30;  
  
printf("Hoy es %s y son las %d:%d\n", dia, hora, min);
```



- Se debe especificar el formato de cada dato a utilizar

`[flags] [ancho][.precisión] [longitud] especificador_formato`

Especificador de formato		Descripción
%c		carácter individual
%d	%i	entero decimal con signo
%zu		size_t
%o	%ou	entero octal con y sin signo
%x	%X	entero hexadecimal con signo
%f	%F	double con notación decimal-punto
%e	%E	double con notación exponencial
%p		valor de un puntero hexadecimal. Recomendable hacer cast a (void *)
%s		cadena de caracteres



- Lectura por teclado de los datos de entrada

```
scanf (cadena_de_control, &var1, &var2, ...);
```

- `cadena_de_control`
 - contiene especificadores de formato
 - establecen la forma de leer los datos
 - son los mismos que los utilizados en `printf`
 - tantos especificadores como datos a leer

```
size_t limite;
```

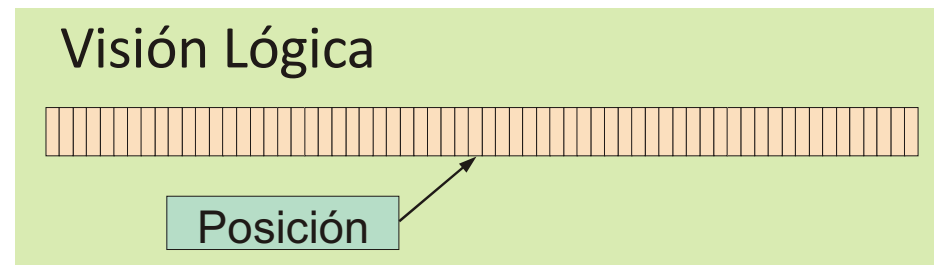
```
printf("Introduzca el limite superior de la serie: ");  
scanf("%d", &limite);
```




- Introducción
- Variables y operadores
- Punteros
- Entrada/Salida
 - **Llamadas al sistema**
 - Funciones de biblioteca
 - Control de errores
- Aplicaciones modulares



- Un fichero es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de información relacionada entre sí bajo un mismo nombre.
- Visión lógica de un fichero: conjunto de datos en secuencia con un puntero de posición que apunta al último dato accedido.
- Visión física de un fichero: conjunto de bloques almacenados en el disco.

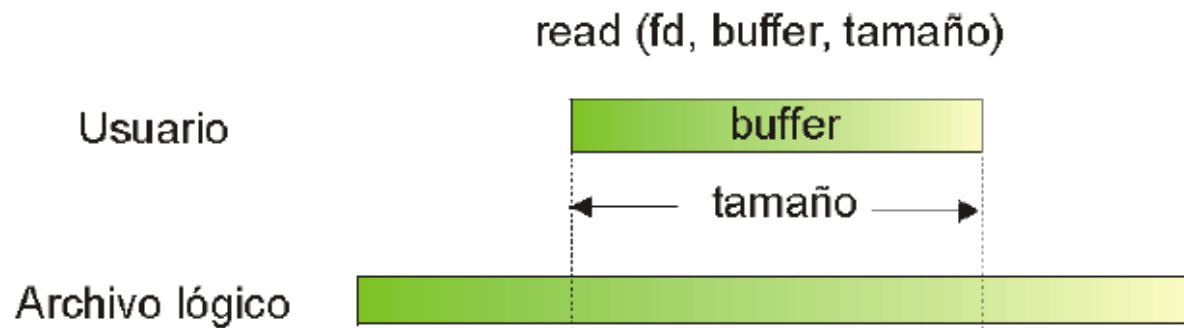


Archivo A
Bloques: 13
20
1
8
3
16
19
10
29





- **Ficheros binarios:** contienen información binaria “en crudo”, tal como se almacena en memoria principal.
 - Pensados para ser manipulados por programas y no por personas.
 - Se accede a ellos por bloques de bytes de un cierto tamaño.



- **Ficheros de texto:** contienen caracteres ASCII.
 - Pensados para ser manipulados directamente por personas.
 - Son accesibles por caracteres, por cadenas o por líneas.



- Entero no negativo que identifica el índice de la tabla de descriptores de ficheros, única por cada proceso. Cada entrada de la tabla:
 - Representa un fichero abierto.
 - Lleva asociado un offset o puntero indicando la posición por la cual esta leyendo/escribiendo.
- Algunos descriptores están predefinidos:
 - 0: entrada estándar
 - 1: salida estándar
 - 2: salida de error
- Se asignan en orden
 - Todas las llamadas al sistema que devuelven un descriptor de fichero, devuelven el más bajo disponible (por proceso), salvo *dup2*.

Descriptores de ficheros



S I S T E M A S O P E R A T I V O S

```
fd1 = open("datos.txt", O_RDONLY);
```

```
...
```

```
fd2 = open("datos.txt", O_RDONLY); fd
```

Tabla de ficheros:
(una por cada programa)

0	15
1	8
2	7
fd1 → 3	2
4	9
5	18
fd2 → 6	3

Tabla intermedia:
(perteneciente al SO)

I-nodo	Posición	Nº duplicados
12	14	1
12	0	1

Tabla de i-nodos:

nopens	
12	1 2



- open: apertura de un fichero
- creat: creación de un fichero
- read: lectura de un fichero
- write: escritura en un fichero
- lseek: posicionamiento del puntero
- close: cierre de un fichero
- dup, dup2: duplicar un descriptor de fichero



```
int open (char* name, int flags [, mode_t mode]);
```

- Argumentos:

- name: cadena de caracteres con el nombre del fichero

- flags:

- O_RDONLY: El fichero se abre de sólo lectura

- O_WRONLY: El fichero se abre de sólo escritura

- O_RDWR: El fichero se abre de lectura y escritura

- O_APPEND: Se escribe a partir del final del fichero

- O_CREAT: Si no existe el fichero, se crea y no da error

- O_TRUNC: Se trunca el fichero

- Se pueden aplicar varios a la vez separándolos con |

- mode: permisos del fichero (sólo con flag O_CREAT)

- Valor devuelto: Descriptor de fichero ó -1 en caso de error



```
int creat (char* name, mode_t mode);
```

■ Argumentos:

- name: cadena de caracteres con el nombre del fichero
- mode: bits de permisos del fichero:
 - S_IRUSR, S_IWUSR, S_IXUSR: R, W, X (user)
 - S_IRGRP, S_IWGRP, S_IXGRP: R, W, X (group)
 - S_IROTH, S_IWOTH, S_IXOTH: R, W, X (others)

■ Valor devuelto:

- Descriptor de fichero ó -1 en caso de error

■ Funcionamiento:

- Crea un fichero si no existe, o lo trunca si ya existía
- Similar a `open` con `O_WRONLY` | `O_CREAT` | `O_TRUNC`



```
size_t read (int fd, void* buf, size_t n_bytes);
```

- Argumentos:

- `fd`: descriptor de fichero del fichero que se va a leer
- `buf`: buffer donde se van a almacenar los Bytes leídos
- `n_bytes`: número de Bytes que se quiere leer

- Valor devuelto:

- Número de Bytes leídos ó -1 en caso de error

- Funcionamiento:

- Lee `n_bytes` como máximo (menos si el fichero es menor)
- Se incrementa el puntero del fichero por cada Byte leído



```
size_t write (int fd, void* buf, size_t n_bytes);
```

■ Argumentos:

- `fd`: descriptor de fichero del fichero que se va a leer
- `buf`: buffer que contiene los Bytes a escribir en el fichero
- `n_bytes`: número de Bytes que se quiere escribir

■ Valor devuelto:

- Número de Bytes escritos ó -1 en caso de error

■ Funcionamiento:

- Escribe `n_bytes` (si no se interrumpe por una señal)
 - Podría escribir menos. Es necesario comprobar el número de bytes escritos
- Se incrementa el puntero del fichero por cada byte escrito



```
off_t lseek (int fd, off_t offset, int whence);
```

■ Argumentos:

- `fd`: descriptor de fichero del fichero cuyo puntero se cambia
- `offset`: desplazamiento, positivo o negativo, del puntero
- `whence`: tipo de desplazamiento del puntero:
 - `SEEK_SET`: posición del puntero = `offset`
 - `SEEK_CUR`: posición del puntero = posición actual + `offset`
 - `SEEK_END`: posición del puntero = tamaño fichero + `offset`

■ Valor devuelto:

- La nueva posición del puntero, ó -1 en caso de error

■ Funcionamiento:

- Desplaza el puntero de acceso del fichero asociado a `fd`



```
int close (int fd);
```

- Argumentos:
 - `fd`: descriptor de fichero del fichero que se quiere cerrar
- Valor devuelto:
 - 0 si todo ha ido bien, ó -1 en caso de error

Ejemplo: mycp_bin.c



S I S T E M A S O P E R A T I V O S

```
#include ...
#define LONBUFFER 2561024
int main (void) {
    int fde, fds, nr;
    char *entrada = "entrada";
    char *salida = "salida";
    char buffer[LONBUFFER];
    if ((fde=open(entrada,O_RDONLY)) < 0) {
        printf("Error al abrir el fichero %s.\n%s.\n", entrada,strerror(errno));
        return 1;
    }
    else if ((fds=open(salida,O_WRONLY|O_TRUNC|O_CREAT,S_IRUSR|S_IWUSR|S_IRGRP,S_IROTH)) < 0) {
        printf("Error al abrir el fichero %s.\n%s.\n", salida,strerror(errno));
        close(fde);
        return 1;
    }
    while ((nr = read(fde,buffer,LONBUFFER)) > 0)
        write(fds,buffer,nr);
    close(fde);
    close(fds);
    return 0;
}
```



- Crea una copia del descriptor de fichero
- Se comparten *locks*, indicadores de posición de fichero y *flags*.
 - Cerrojos (*locks*): permiten regular el acceso a los ficheros.
 - Alcance: fichero completo o parte de él.
 - Acceso compartido o exclusivo.
 - Funciones: flock, fstat.
 - *Flags*: modo de apertura del fichero (lectura, escritura, concatenación, etc).
- Formas de llamada
 - *int dup(int oldfd)*: usa el descriptor libre con menor numeración posible como nuevo descriptor.
 - *int dup2(int oldfd, int newfd)*: hace que el nuevo descriptor sea la copia del viejo, cerrando primero el nuevo si es necesario.



```
int dup (int fd);
```

- Argumentos:
 - `fd`: descriptor de fichero que se quiere duplicar
- Valor devuelto:
 - Un descriptor de fichero, ó -1 en caso de error
- Funcionamiento:
 - Crea una copia del descriptor de fichero `fd`
 - Devuelve el descriptor libre más bajo posible



```
int dup2 (int oldfd, int newfd);
```

- Argumentos:

- `oldfd`: descriptor de fichero que se quiere duplicar
- `newfd`: nuevo descriptor de fichero

- Valor devuelto:

- El nuevo descriptor de fichero, ó -1 en caso de error

- Funcionamiento:

- Hace que `newfd` sea una copia de `oldfd`, cerrando `newfd` si es necesario

Duplicar un descriptor de fichero



S I S T E M A S O P E R A T I V O S

```
fd1 = open("datos.txt", O_RDONLY);
```

```
...
```

```
dup2(fd1, 1);
```

Tabla de ficheros:
(una por cada programa)

fd
0
1
2
3
4
5

15
8 2
7
2
9
18

Tabla intermedia:
(perteneciente al SO)

I-nodo	Posición	Nº duplicados
12	14	1 2

Tabla de i-nodos:

nopens	
12	
	1



- mkdir: creación de un directorio
- rmdir: borrado de un directorio
- getcwd: obtención del directorio actual
- chdir: cambio de directorio



```
int mkdir (char* name, mode_t mode);
```

■ Argumentos:

- name: cadena de caracteres con el nombre del directorio
- mode: bits de protección del directorio:
 - S_IRUSR, S_IWUSR, S_IXUSR: R, W, X (user)
 - S_IRGRP, S_IWGRP, S_IXGRP: R, W, X (group)
 - S_IROTH, S_IWOTH, S_IXOTH: R, W, X (others)

■ Valor devuelto:

- 0 si todo ha ido bien, ó -1 en caso de error



```
char* getcwd (char* buf, size_t size);
```

- Argumentos:
 - `buf`: buffer donde se va a almacenar el directorio actual
 - `size`: tamaño en Bytes del buffer
- Valor devuelto:
 - Puntero a `buf`, ó `NULL` en caso de error
- Funcionamiento:
 - Obtiene en `buf` el nombre del directorio actual



```
int chdir (char* name);
```

- Argumentos:
 - name: cadena de caracteres con el nombre del directorio

- Valor devuelto:
 - 0 si todo ha ido bien, ó -1 en caso de error

- Funcionamiento:
 - Cambia el directorio actual al directorio name



- Introducción
- Variables y operadores
- Punteros
- Entrada/Salida
 - Llamadas al sistema
 - **Funciones de biblioteca**
 - Control de errores
- Aplicaciones modulares



- Construidas sobre las llamadas al sistema
- Permiten realizar operaciones de E/S como si se realizaran sobre un *stream* o flujo de datos.
 - Un *stream* nos permite establecer una conexión (entre un origen y un destino) a través de la cual circula información:
- En vez de utilizar descriptores de ficheros se utilizan tipos de datos creados expresamente:
 - FILE: tipo de datos que representa un fichero abierto. Existen varias variables predefinidas
 - *stdin* = descriptor 0
 - *stdout* = descriptor 1
 - *stderr* = descriptor 2
 - DIR: tipo de datos que representa un directorio abierto



- Funciones de biblioteca para manejo de ficheros:
 - fopen: abre un fichero.
 - fclose: cierra un fichero.
 - fread: leer datos de un fichero.
 - fwrite: escribir datos a un fichero.
 - fprintf/fscanf: igual que printf y scanf pero con ficheros en lugar de *stdout* y *stdin*.
 - fputs/fgets: escribe/lee una línea completa de un fichero.
 - ftell/fseek: leer/modificar el puntero de posición (int).
 - fgetpos/fsetpos: leer/modificar el puntero de posición (fpos_t).
 - rewind: poner el puntero al comienzo del FILE *.
 - rename: renombrar fichero.
 - remove: borrar fichero.



- Apertura de un archivo

```
FILE* fopen(const char* nombreArchivo, const char* modo);
```

Modo	Descripción
r	Solo lectura.
w	Trunca el fichero para escritura.
a	Escritura al final del fichero.
r+	Lectura y escritura.
w+	Lectura y escritura. Fichero creado si no existe.
a+	Lectura desde comienzo y escritura desde el final.

- Devuelve un puntero a `NULL` si no se puede abrir el archivo



■ Cierre de archivos

```
int fclose( FILE* pf);
```

- Importancia de cerrar un fichero
 - Los flujos `FILE` llevan asociado un buffer intermedio
 - Exceptuando *stderr*
 - **Este buffer sólo se vuelca al fichero cuando esta lleno**
 - De esta forma se consigue reducir el número de accesos a un fichero durante la ejecución de un programa
 - Sino se cierra un archivo puede que no se guarden los últimos cambios hechos.
- Devuelve un cero si el archivo se cerró con éxito



- Lee / escribe líneas de texto

```
char* fgets(char* cad, int n, FILE *pf );
```

```
int fputs (const char *cad, FILE *pf);
```

- Escribe / lee con formato específico

```
int fprintf (FILE* PF, const char* cadControl,...);
```

```
int fscanf(FILE* pf, const char* cadControl, ...);
```

- Lee / escribe un carácter

```
int fgetc(FILE* pf);
```

```
int fputc(int c, FILE* pf);
```

Ejemplo: mycp_ascii.c



S I S T E M A S O P E R A T I V O S

```
#include ...
#define LONBUFFER 1024
int main (void) {
    FILE *fe, *fs;
    int nr;
    char *entrada = "entrada";
    char *salida = "salida";
    char buffer[LONBUFFER];
    if ((fe=fopen(entrada,"r")) == NULL) {
        fprintf(stderr,"Error al abrir el fichero %s.\n%s.\n",entrada,strerror(errno));
        return 1;
    }
    else if ((fs=fopen(salida,"w")) == NULL) {
        fprintf(stderr,"Error al abrir el fichero %s.\n%s.\n",salida,strerror(errno));
        fclose(fe);
        return 1;
    }
    while (fgets(buffer,LONBUFFER,fe) != NULL) /* Lee una línea */
        fputs(buffer,fs);
    fclose(fe);
    fclose(fs);
    return 0;
}
```



- Lee / escribe bloques de datos en binario

```
size_t fread(void* ptr, size_t tam, size_t n, FILE* pf);
```

```
size_t fwrite(const void* ptr, size_t tam, size_t n, FILE* pf);
```

- En los ficheros binarios la información se escribe tal cual se almacena en memoria
 - Optimiza el tamaño ocupado por un archivo



- Funciones de E/S para reposicionar el puntero
 - Leer / posicionar el puntero

```
int ftell (FILE *stream);
```

```
int fseek (FILE *stream, long offset, int whence);
```

```
int fgetpos (FILE *stream, fpos_t *pos);
```

```
int fsetpos (FILE *stream, fpos_t *pos);
```

- Rebobinar el fichero (poner el puntero al principio)

```
void rewind (FILE *stream);
```



- Obtener el FILE * de un fichero ya abierto a partir de su descriptor
 - El fichero se abrió o creó con anterioridad mediante open, creat, etc.

```
FILE * fdopen (int fd, char *mode);
```

- Obtener descriptor de un fichero ya abierto a partir de su FILE *
 - El fichero se abrió o creó con anterioridad mediante fopen.

```
int fileno (FILE *stream);
```



- Funciones de biblioteca para el manejo de directorios:
 - opendir: abre un fichero.
 - closedir: cierra un fichero.
 - readdir: leer una entrada de directorio.
 - rewinddir: rebobinar un directorio.



- Creación de un directorio:

`DIR* opendir (char* dirname) ;`

- Argumentos:

- `dirname`: nombre del directorio que se quiere abrir

- Valor devuelto:

- Un puntero a una estructura `DIR`, o `NULL` en caso de error

- Funcionamiento:

- Abre un directorio y se coloca en su primer elemento



- Lectura de un directorio:

```
struct dirent* readdir (DIR* dirp);
```

- Argumentos:

- `dirp`: puntero a la estructura devuelta por `opendir`

- Valor devuelto:

- Un puntero que representa una entrada de un directorio, o `NULL` en caso de error

- Funcionamiento:

- Devuelve la siguiente entrada del directorio que representa `dirp` y avanza el puntero a la siguiente



- Creación de un directorio:

```
int closedir (DIR* dirp);
```

- Argumentos:

- `dirp`: puntero a la estructura devuelta por `opendir`

- Valor devuelto:

- 0 si todo ha ido bien, ó -1 en caso de error



- Introducción
- Variables y operadores
- Punteros
- Entrada/Salida
 - Llamadas al sistema
 - Funciones de biblioteca
 - **Control de errores**
- Aplicaciones modulares



- Muchas de las funciones de biblioteca y llamadas al sistema proporcionan un código de error si no han conseguido realizar su propósito.
 - 0 suele significar funcionamiento correcto
 - Cualquier otro numero significa un código de error
- La biblioteca estándar proporciona un mecanismo para controlar estos errores y convertirlos en algo “legible”.
 - Las funciones devuelven el código de error en una variable global denominado *errno*.
 - La función *strerror(int errno)* convierte el código de error a una cadena de texto más explicativa.

■ Ejemplo:

```
#include <stdio.h>
#include <errno.h> //Para errno
#include <string.h> //Para strerror

int main()
{
FILE *f1;
f1 = fopen("file.txt", "r+");
if(f1 == NULL)
    fprintf(stderr, "Error al abrir el fichero. %s\n",strerror(errno));
else
    fprintf(f1,"Hola mundo\n");
}
```



- Introducción
- Variables y operadores
- Punteros
- Funciones
- Entrada/Salida
- **Aplicaciones modulares**



- La definición de un módulo en C viene dada por:
 - Un archivo de cabecera (.h)
 - Define la interfaz del módulo, es decir, qué ofrece el módulo definido
 - Debería contener: definiciones de tipos y declaraciones de funciones (prototipos)
 - Para evitar que se incluya en fichero mas de un vez
#pragma once
 - Un archivo fuente (.c)
 - Define los detalles de implementación del módulo
 - Inclusión de su correspondiente fichero de cabecera
#include "file.h"
 - Implementación de funciones declaradas y no declaradas en el fichero de cabecera



- La posibilidad de definir módulos en C permite:
 - Definición de nuevos tipos de datos junto con sus operaciones (TADs)
 - Colección de valores legales de datos y operaciones primitivas que se pueden realizar sobre esos valores
 - La definición de un TAD implica la definición de:
 - Un interfaz público
 - Representación privada o implementación
 - La creación de bibliotecas o librerías de funciones

- **Librería estática (`libnombre.a` (Linux) / librería.`lib` (Windows))**
 - “Se copia” el código dentro del ejecutable cuando lo compilamos.
 - Sólo se copia aquella parte de la librería que se necesite.
- **Librería dinámica (`libnombre.so` (Linux) / librería.`dll` (Windows)):**
 - No se copia en nuestro programa al compilarlo.
 - Cuando se ejecuta un programa que la requiere, cada vez que el código necesite algo de la librería, irá a buscarlo a ésta.



■ Creación:

- Obtener los ficheros objeto (.o) de todos nuestros fuentes (.c).

gcc -c fuente.c -o fuente.o

- La opción **-c** le dice al compilador que no cree un ejecutable, sino sólo un fichero objeto

- Crear la librería (.a).

ar -rv libnombre.a fuente1.o fuente2.o ...

- La opción **-r** le dice al comando *ar* que tiene que insertar (o reemplazar si ya están) los ficheros objeto en la librería.

■ Compilación:

gcc -o miprograma miprograma.c -l<path1> -l<path2> ... -L<path1> -L<path2> ... -llibreria1 -llibreria2

- **-llibreria** indica que se debe coger esa librería. El prefijo **lib** y la extensión **.a** lo pone automáticamente el compilador.
- Es necesario ponerlas por orden si una depende de otra.



■ Creación:

- Compilar los fuentes, igual que antes, para obtener los objetos.
- Crear la librería (.so)

```
ld -o liblibreria.so objeto1.o objeto2.o ... -shared
```

- La opción **-o** liblibreria.so indica el nombre que queremos dar a la librería.
- La opción **-shared** indica que debe crear una librería y no un ejecutable (opción por defecto).

■ Compilación:

```
gcc -o miprograma miprograma.c -I<path1> -I<path2> ...  
-L<path1> -L<path2> ... -Bdynamic -llibreria1 -llibreria2
```

- Se debe definir la variable de entorno **LD_LIBRARY_PATH** con todos los directorios donde haya librerías dinámicas de interés.

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path1>:<path2>:<path3>  
$ export LD_LIBRARY_PATH
```

- Comprobar los ficheros de librería utilizados: mandato **ldd**

- Para simplificar el proceso de compilación lo habitual es hacer un fichero **Makefile** en el mismo directorio donde estén los fuentes
 - Ejecución desde la shell: `make objetivo`
- Posible definición de variables:

```
CFLAGS=-I<path1> -I<path2> ...  
CC=gcc
```
- Ordenados en forma de reglas:

```
objetivo: dependencias  
comandos
```
- Reglas virtuales: sin dependencias

```
clean :  
rm *.o *~
```
- Reglas implícitas:

```
programa : programa.o  
programa.o : programa.c
```
- Reglas patrón:

```
%.o : %.c  
$(CC) $(CFLAGS) $< -o $@
```