# Sistemas Operativos

## Introducción

### Introducción al Entorno de Desarrollo

# Agenda

# Agenda

# Entorno de desarrollo

- En SO tomamos POSIX como modelo de referencia
  - Escogemos GNU Linux para la realización de prácticas
- Los estudiantes pueden usar sus propios equipos con:
  - Una instalación nativa de Linux (o un dualboot), o
  - Docker Desktop en Windows o Mac para desarrollar sobre contenedores Linux, o
    - Instrucciones en Entorno de desarrollo con Docker y Visual Studio Code
  - Una máquina virtual Linux (formato ova), o
    - Instrucciones de instalación en campus virtual video lección
- Se recomienda que en el laboratorio se familiaricen con la instalación nativa de Linux, ya que es el entorno que usarán para el examen.

# Interfaz de línea de comandos (CLI)

UNIX tiene un potente interfaz de línea de comandos:

- Accesible a través de consola o terminales virtuales
- Usan interpretes (shell), programas que leen ordenes del usuario
  - Las órdenes indican qué programas ejecutar y qué parámetros pasarles
  - Tienen órdenes internas que permiten programarlos
    - para automatizar tareas complejas
    - cuando se requiere la ejecución de varios programas
    - con decisiones en función del resultado de los programas o el valor escrito por su salida estándar
  - Interpretes: sh, bash, tcsh, fish, etc.
- Muchas utilidades ya programadas
  - Diseñadas para hacer una sóla cosa, pero suelen admitir muchas opciones
  - Fáciles de combinar pasando la salida estandar de una a la entrada estándar de otra (pipes)

También tiene un interfaz gráfico gestionado por programas de usuario

- Servidor X (xorg en Linux) + gestor de ventanas + xlib
- Servidor Wayland en Linux es la nueva arquitectura

# Bourne Again Shell (BASH)

- Intérprete que usan por defecto la mayor parte de las distribuciones
- Video lecciones (también disponibles a través del campus virtual):
  - Bash1: Concepto de CLI, introducción a bash.
  - Bash2: Utilidades básicas de CLI del sistema.
  - Bash3: Gestión de procesos, redirecciones, tuberías, etc.
  - Bash4: Lenguaje BASH, guiones shell, expansión de órdenes
  - Bash5: Variables de shell y variables de entorno

# Agenda

# ¿Por qué C?

Porque el API del sistema operativo es C:

- Es un lenguaje *pequeño y sencillo* (comparado con C++)
  - Tiene muy poca sobrecarga
  - Mayor rendimiento que otros con mayor nivel de abstracción
- Diseñado para la implementación de sistemas
  - acceso directo a memoria
  - gestión manual de la memoria
  - operaciones a nivel de bit

Pero es necesario asumir la responsabilidad de:

- La gestión de memoria
- La inicialización de variables
- El chequeo explícito de errores

# Herramientas básicas

Para el desarrollo de las prácticas usaremos las siguientes herramientas:

- Editor:
  - Nos permite escribir y/o modificar el código de nuestros programas
  - En el caso de vscode, nos servirá también como interfaz para interaccionar con otras herramientas
- Compilador:
  - Herramienta que genera un fichero ejecutable con instrucciones máquina a partir de nuestro programa escrito en C
  - En realidad se trata de un compendio de herramientas: preprocesador, compilador, ensamblador y enlazador.
- Herramienta make:
  - Permite automatizar la construcción/compilación de proyectos complejos
  - Varios objetivos, dependencias explicitas, etc
- Depurador:
  - Nos ayuda a identificar y corregir los errores de nuestros programas

El documento Entorno de desarrollo C para GNU/Linux ofrece una introducción detallada al uso de estas herramientas.

# Editor

Visual Studio Code (vscode)

- Software de Microsoft, multiplataforma, de descarga gratuita
- Editor extensible, gran cantidad de plugins
  - Tiene plugin de depuración con gdb (GNU debugger)
  - Dispone de plugin para integración de Docker
    - Consultar Entorno de desarrollo con Docker y Visual Studio Code
- Fácil integración de herramientas externas
- Configuración por proyecto a través de ficheros json
- Introducción básica y configuración de proyectos en video lección

Alternativas:

- Gráficos: gedit, nedit, kedit, kate, emacs, gvim, atom
- De terminal: vim, nano

# Compilador

GNU C Compiler: gcc

- Interfaz que permite usar todas las herramientas del toolchain: preprocesador, compilador, ensamblador y enlazador
- Interfaz de línea de comandos:
  - Preprocesado + compilación + ensamblado (compilación):

```
$ gcc -c A.c     Genera A.o
$ gcc -c B.c     Genera B.o
```

  - Enlazado:

```
$ gcc -o ejemplo A.o B.o     Genera ejecutable ejemplo
$
```

  - Pueden salvarse los ficheros intermedios

```
$ gcc --save-temps archi.c
$
```

- Habitualmente se usa la herramienta make para invocar al compilador

# Herramienta make

Herramienta para la construcción (build) de proyectos:

- Lee las instrucciones de un fichero de entrada makefile
  - Contiene definiciones de variables y reglas
  - Las reglas indican como construir un objetivo/fichero
- Cada regla indica el objetivo a construir, las dependencias y las instrucciones para construir el objetivo
  - Las dependencias pueden ser nombres de fichero construidos a partir de otras reglas
  - make construye un grafo de reglas usando las dependencias
  - La regla debe ejcutarse sólo si alguna de las dependencias tiene una modificación más reciente que el fichero objetivo
- Cuando se invoca make se puede elegir qué objetivo construir
  - make ejecuta sólo las reglas necesarias para construir ese objetivo
  - por defecto el primero

Más información en nuestro documento sobre make y la documentación oficial de gnu.

# Ejemplo de makefile

```
CC = gcc
CFLAGS = -O0 -g -pthread
LDFLAGS = -pthread
LIBS =
TARGET = ejemplo
TARGET_SRC = A.c B.c
TARGET_OBJ = $(TARGET_SRC:%.c=%.o)

all: $(TARGET)

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

$(TARGET): $(TARGET_OBJ)
    $(CC) $(LDFLAGS) -o $@ $(TARGET_OBJ) $(LIBS)

.PHONY: clean

clean:
    -rm $(TARGET_OBJ) $(TARGET)
```

# Depurador

GNU Debugger, gdb

- Muy extendido y utilizado en multitud de ámbitos
- Depurador con interfaz de línea de comandos
  - Control de ejecución: next, step, until, finish, continue
  - Puntos de ruptura: break, commands, info break, delete
  - Mostrar valores: print
- También dispone de interfaz de texto con ncurses
  - Se activa con el comando `tui enable`
- vscode puede hacer de interfaz gráfico
  - incluido en extensión C/C++
  - tiene una consola integrada que permite mandar comandos a gdb si se preceden de -exec
- Hay otras aplicaciones que construyen un interfaz gráfico sobre gdb, como ddd, nemiver, etc.

El apartado `Depuración` del documento Entorno de desarrollo C para GNU/Linux ofrece una introducción básica al uso de gdb.

# Agenda

# Sintaxis básica: control de flujo

- `if ( ) { } else { }`
- `while ( ) { }`
- `do { } while ( )`
- `for (i=1; i <= 100; i++) { }`
- `switch ( ) {case 1: ...}`
- `continue; break;`

# Sintaxis básica: tipos de datos simples

| Tipo | Bytes<br>min - hab | Rango | Formato |
|---|---|---|---|
| char | 1 - 1 | $[-128, 127]$ o $[0, 255]$ | %c |
| unsigned char | 1 - 1 | $[0, 255]$ | %uc |
| short (int) | 2 - 2 | $[-(2^{15}-1), (2^{15}-1)]$ | %hd |
| int | 2 - 2 | $[-(2^{15}-1), (2^{15}-1)]$ | %d |
| | 2 - 4 | $[-(2^{31}-1), (2^{31}-1)]$ | |
| long (int) | 4 - 8 | $[-(2^{31}-1), (2^{31}-1)]$ | %ld |
| long long (int) | 8 - 8 | $[-(2^{63}-1), (2^{63}-1)]$ | %lld |
| float | 4 | $\pm$[1.2E-38, 3.4E+38] | %f |
| double | 8 | $\pm$[2.3E-308, 1.7E+308] | %lf |
| long double | 10 | $\pm$[3.4E-4932, 1.1E+4932] | %Lf |

Usar siempre `sizeof(<tipo_de_dato>)` y definiciones de `limits.h` (p.ej. `INT_MAX`)

# Datos en memoria

```
int   x = 5,    y = 10;
float f = 12.5, g = 9.8;
char  c = 'c',  d = 'd';
```

| Tag | x | y | f | g | c | d |
|---|---|---|---|---|---|---|
| | 5 | 10 | 12.5 | 9.8 | 'c' | 'd' |
| Addr | 0x4300 | 0x4304 | 0x4308 | 0x430c | 0x4310 | 0x4311 |

# Sintaxis básica: Operadores

- Aritméticos: `+ - / * % ++ --`
  - `i = i+1; i++; i--; i *= 2; i = i%3;`
- Operadores de bits: `& | ^ << >> ~`
  - `i = i&0x0f; i |= 0x1; i ^= i; i = i<<2;`
- Operadores relacionales: `< > <= >= == !=`
- y operadores lógicos: `&& || !`
  - `if (((i<100) && (i!= 4)) || !finish) {...}`

# Biblioteca Estándar de C (Stdlib)

El lenguaje C no incluye operaciones de E/S, cadenas de carecteres, etc.
Todas esta funcionalidad se proporciona a través de su librería estándar:

- Operaciones básicas sobre ficheros
  - `fopen, fclose, fread, fwrite, fflush, ...`
- Operaciones de entrada/salida (E/S) estándar:
  - `printf, sprintf, snprintf, fprintf, scanf, fscanf, ...`
  - `getc, fgetc, fgets, getchar, ungetc, ...`
- Operaciones de procesado de cadenas de caracteres
  - `strlen, strcat, strcmp, strcpy, ...`
- Operaciones de gestión de memoria:
  - `malloc, realloc, calloc, free, memcpy, memset, ...`
- Operacione matemáticas
  - `sin, cos, tan, sqrt, rand, ...`

# Hello world en C

```c
#include <stdio.h>

void main(void)
{
    printf("Hello World. \n \t and you! \n");
    /* print out a message */
    return;
}
```

Salida por terminal:

```
$ ./example1
Hello World.
        and you !
$
```

# Hello world en C

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

- Directiva include del preprocesador
- Inserta el contenido del fichero de cabecera `stdio.h`
- No es necesario ';' al final
- Sólo letras minúsculas (C distingue entre mayúsculas y minúsculas)

```
void main(void){ ... }
```

- Función de entrada al programa, código a ejecutar

```
printf(" /* mensaje */ ");
```

- Escritura en salida estándar
- Cadena con formato
- Caracteres precedidos de '\' son caracteres especiales
  - '\n' = salto de línea
  - '\t' = tabulador

# Preprocesador: directivas y macros

```c
#include <stdio.h>
#define DANGERLEVEL 5  /*Constant C Preprocessor macro*/

void main(void)
{
  float level=1;

  if (level <= DANGERLEVEL) {  /*replaced by 5*/
    printf("Low on gas!\n");
  } else {
    printf("Good driver !\n");
  }
  return;
}
```

# Salida estándar

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

`format` admite marcas para la inserción de valores de variables:

- %d: para enteros con signo
- %u: para enteros sin signo
  - con modificadores: `%llu`: para `long long int`
- %s: para cadenas de caracteres
- %p: para punteros
- Consultar la página de manual

Ejemplo de uso:

```
#include <stdio.h>

int main(void)
{
  int nstudents;
  ...
  printf("Cornell has %d students.\n", nstudents);
  return 0;
}
```

# Entrada estándar

```
int scanf(const char *format, ...);
```

- `format` admite las mismas marcas que printf para la entrada de valores de variables.
- por cada marcador hay que pasar la dirección de un buffer para almacenar el valor

Ejemplo de uso:

```
#include <stdio.h>
void main(void)
{
  int nstudents = 0;            /*Init. required */
  printf("How many students does Cornell have ?:");
  scanf ("%d", &nstudents);     /* Read input */
  printf("Cornell has %d students.\n", nstudents);
  return ;
}
```

Salida por terminal:

```
$ ./example2
How many students does Cornell have ?: 20000 (enter)
Cornell has 20000 students.
$
```

# Estructuras compuestas: Arrays

Array: colección homogenea de elementos, almacenados en memoria en posiciones consectivas.

- En C el nombre del array es un símbolo con el valor de la dirección del primer elemento del array.
- El array no almacena el tamaño del mismo, y no tiene valor inicial
- Accedemos al i-ésimo elemento del array `A` con: `A[i]`

Ejemplo:

```
#include <stdio.h>
void main(void)
{
    int number[12];    /*12 cells, one cell per student*/
    int i, sum = 0;
    /* Always initialize array before use */
    for (i = 0; i < 12; i++) {
        number[i] = i;
    }
    /* now, number[i]=i; will cause error:why ?*/
    for (i = 0; i < 12; i = i + 1) {
        sum += number[i];  /* sum array elements */
    }
    return;
}
```

# Arrays multidimensionales

Se pueden añadir más dimensiones al array:

- La dimensión de la derecha es la más interna
  - los elemenos consecutivos en esta dimensión están almacenados en posiciones consecutivas de memoria
- En las otras los elementos tienen una separación igual al número de bytes ocupados por las dimensiones más internas a ésta

Ejemplo:

```
int A[3][4];    /* NOT A[3,4] */
A [1][3] = 12;
printf("%d", A[1][3]);
printf("%p %p %p\n", &A[0][0], &A[0][1], &A[1][0]);
```

Salida por terminal:

```
$ ./array2d
12
0x7ffec823dae0 0x7ffec823dae4 0x7ffec823daf0
```

# Estructuras compuestas: structs

Colección de campos heterogéneos

- El tamaño de la estructura puede ser mayor que la suma del tamaño de los campos.
  - Usar el operando `sizeof` para obtener el número de bytes que ocupa una variable struct
- El indexado es como en C++:
  - A partir de una variable struct: `nombrevar.nombre_campo`
  - A partir de un puntero a un struct: `puntero->nombre_campo`
- Ejemplo de declaración y uso de un struct:

```
struct alumno {
  int id;
  char nombre[100];
};
struct alumno mialumno;
mialumno.id = 100;
strncpy(&mialumno.nombre, "Juan Perez Luque", 100);
```

# Estructuras compuestas: union

Buffer que puede contener uno de los campos miembro de la unión

- El tamaño de la unión es igual al tamaño del mayor de sus campos
- Se accede al buffer accediendo a uno de sus campos con la misma sitaxis que con los structs
- Ejemplo de declaración y uso de una unión:

```
union address {
    unsigned char ipv4[4];
    unsigned char ipv6[16];
};
union address midir;
midir.ipv4[0] = 192;
midir.ipv4[1] = 168;
midir.ipv4[2] = 1;
midir.ipv4[3] = 2;
```

# Definición de tipos

Operando `typedef`:

```
typedef tipo nombre;
```

- `tipo`: un tipo de datos ya definido o una definición de tipo compueto
- `nombre`: nombre alternativo que se le da

Ejemplo de uso:

```
typedef struct {
  int id;
  char nombre[100];
} alumno_t;
alumno_t mialumno;
mialumno.id = 100;
strncpy(&mialumno.nombre, "Juan Perez Luque", 100);
```

# Punteros

Un puntero es una variable que almacena una dirección de memoria

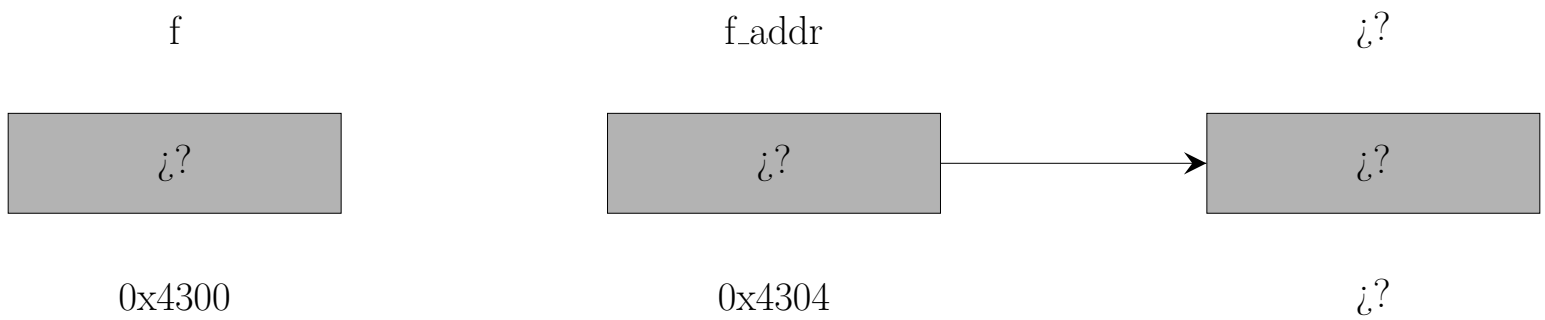
- Se le debe asignar una dirección válida antes de usarlo
  - la dirección de una variable o
  - una dirección devuelta por el sistema (por ejemplo por `malloc`)

Tiene dos operadores propios:

- Desreferencia `*`: acceso a la dirección almacenada
- Dirección de `&`: obtiene dirección de variable, para ser asignada a un puntero

Ejemplo:

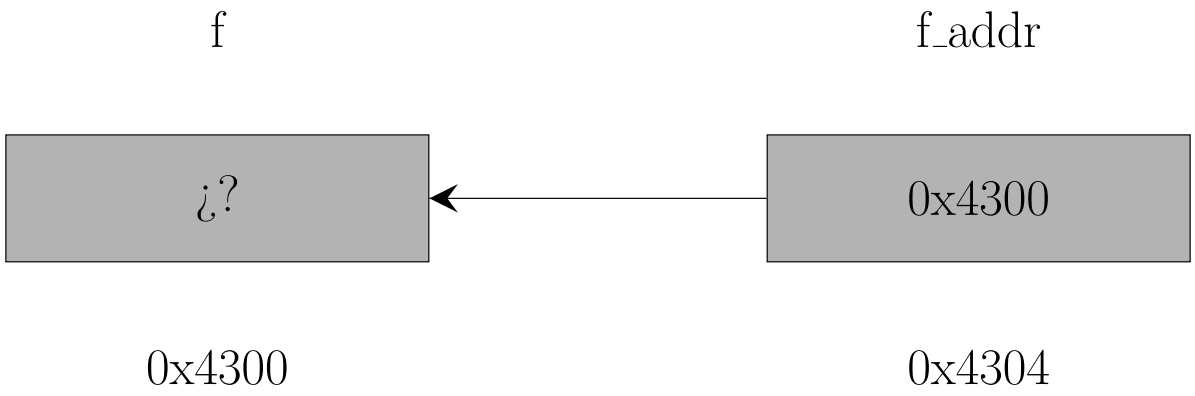
```
float f;          /* declaración de variable tipo float */
float *f_addr;    /* declaración de variable tipo puntero */
```



# Punteros

Asignando la dirección de una variable el puntero se dice que apunta a la variable

f_addr = &f;




Podemos modificar el valor de la variable desreferenciando el puntero:

*f_addr = 5.4;

# Punteros: ejemplos de uso

```
int month[12];
/* month is a pointer to base address 0x430*/

int *ptr = month + 2;
/* ptr points to month[2], => ptr is now (0x430+2*4)=0x438 */

month[3] = 7;
/* month address + 3 * sizeof(int) => int at (0x430+3*4) is 7 */

ptr[5] = 12;
/* int at (0x438+5*4) is now 12. Thus, month[7]=12 */

ptr++;
/* ptr <- 438 + 1 * sizeo(int) = 43C */

(ptr + 4)[2] = 12;
/* accessing ptr[6] i.e., month[9] */
```

# Cadenas de caractéres

Se representan como una secuencia de enteros con signo de tamaño byte (`char`) finalizados en un byte a 0 (byte null ó `'\0'`):

- Pueden declararse como un array:

```
char message[6] = {'H','E','L','L','O','\0'};
printf("%s", message);     /*print until '\0'*/
```

- Pueden declararse como un literal de cadena con ""

```
printf("%s", "hello");     /*print until '\0'*/
```

- que puede usarse para inicializar una array:

```
char message[] = "hello";
printf("%s", message);     /*print until '\0'*/
```

- o asignar su dirección a un puntero:

```
char *message = "hello";
printf("%s", message);     /*print until '\0'*/
```

# Punteros y cadenas de caracteres

```
#include <stdio.h>
void main(void) {

  char msg[10];              /* array of 10 chars */
  char *p;                   /* pointer to a char */
  char msg2[]="Hello";       /* msg2 = 'H''e''l''l''o''\0' */

  msg = "Bonjour";    /* ERROR. msg has a const address.*/
  p   = "Bonjour";    /* address of "Bonjour" goes into p */
  msg = p;            /* ERROR. Msg has a const. address */

  p = msg;            /* OK */

  p[0]='H', p[1]='i',p[2]='\0'; /* msg and *p are now "Hi" */

}
```

# Operaciones sobre cadenas de caracteres

Implementadas por la biblioteca estándar de C

- Requiere la inclusión del fichero de cabecera (`strings.h`)
- Copia: strcpy, strncpy
- Concatenado: strcat, strncat
- Comparación: strcmp, strncmp
- Longitud: strlen
- Duplicado: strdup
- Creación con formato: sprintf, snprintf

```
char message[100];
char msg_hello[] = "Hello ";
char msg_world[] = "World!";
strncpy(message, msg_hello, 100);
strncat(message, msg_world, 100 - strlen(msg_hello));
printf("%s\n", message);
```

Salida por terminal:

```
$ ./ejemplo
Hello World!
```

# Versiones seguras (con n)

Ejemplo: `strcat` y `strncat`

```
#include <string.h>
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ambas funciones concatenan dos cadenas de caracteres, copiando la cadena apuntada por `src` a continuación de `dest`.

- `strcat`: copia los bytes hasta encontrar el fin de línea de `src`.
- `strncat`: copia como **máximo n** bytes.

Si `src` es un dato de entrada, nunca podemos estar seguros de cuál será su tamaño, debemos limitar la escritura al tamaño del buffer apuntado por `dest`

# Asignación dinámica de memoria

La biblioteca estándar de C proporciona funciones para la gestión de la memoria dinámica (heap):

- `malloc`: reserva de memoria dinámica
- `calloc`: reserva de memoria e inicialización a 0
- `realloc`: modificación del tamaño reservado
- `free`: liberación de memoria reservada con `malloc`

```
#include <stdio.h>
void my_function(void) {
  char c;
  int *ptr;
  /* allocate space to hold an int */
  ptr = malloc(sizeof(int));
  /* do stuff with the space */
  *ptr = 4;
  /* free up the allocated space */
  free(ptr);
}
```

Consultar las páginas de manual de estas funciones.

# Funciones

Recurso principal para organizar el código y permitir su reuso.

- C sólo admite paso de parámetros por valor

```c
#include <stdio.h>
/* function prototype at start of file */
int sum(int a, int b);

void main(void)
{
    int total = sum(4,5);   /* call to the function */
    printf("The sum of 4 and 5 is %d\n", total);
}

int sum(int a, int b) /* arguments passed by value*/
{
    return (a+b);           /* return by value */
}
```

# Funciones

El paso por referencia lo conseguimos pasando explícitamente la dirección de una variable a un argumento tipo puntero

```
#include <stdio.h>
void swap(int *, int *);

void main(void)
{
    int num1 = 5, num2 = 10;
    swap(&num1, &num2); /* num1 and num2 passed by reference */
    printf("num1 = %d and num2 = %d\n", num1, num2);
}

void swap(int *n1, int *n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

# Parámetros tipo puntero

Cuando leemos un prototipo de una función como:

```
void dosomething(int *ptr);
```

lo que nos dice es que la función espera como argumento una **dirección válida** de un buffer en el que pueda almacenarse un entero.

- **Nunca** debemos pasar un puntero sin inicializar
- **Siempre** debemos pasar una dirección válida, de una variable o devuelta el sistema (p.e. por malloc).

# ¿Por qué es incorrecto este código?

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void)
{
    int *p;
    dosomething(p)
    printf("%d\n", *p);      /* will this work ? */
}
/* passed and returned by reference */
void dosomething(int *ptr)
{
    int temp=32+12;
    *ptr = temp;
}
```

posible error durante la ejecución

# Posible solución: dirección de variable

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void) {
    int a;
    dosomething(&a)
    printf("%d\n", a);
}
/* passed and returned by reference */
void dosomething(int *ptr)
{
    int temp=32+12;
    *ptr = temp;
}
```

# Posible solución: memoria dinámica

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void)
{
    int *p = malloc(sizeof(int));
    dosomething(p)
    printf("%d", *p);
    free(p);
}
/* passed and returned by reference */
void dosomething(int *ptr)
{
    int temp=32+12;
    *ptr = temp;
}
```

# Arrays como argumento: siempre por referencia

Se copia el valor del símbolo, la dirección de comienzo del array

- **No se puede pasar un array por copia**
- El array no tiene un tamaño definido, se pasa el tamaño como un argumento adicional

```
#include <stdio.h>
/* Size of the array is passed as an aditional argument */
void init_array(int array[], int size) ;

void main(void) {
    int i,list[5];
    init_array(list, 5);
    for (i = 0; i < 5; i++)
        printf("next: %d", list[i]);
}

void init_array(int array[], int size)
{ /* arrays ALWAYS passed by reference */
    int i;
    for (i = 0; i < size; i++)
        array[i] = 0;
}
```

# Estructuras como argumentos

- Pueden pasarse por copia, pero es ineficiente
- Suele preferirse el paso por referencia, usando un puntero a la estructura como argumento en la función
- Pueden devolverse por valor, por el mismo motivo no suele hacerse

```
/* pass struct by value - inefficient: why ? */
void display_year_1(struct birthday mybday) {
    printf("I was born in %d\n", mybday.year);
}

/* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
    printf("I was born in %d\n", pmybday->year);
    /* warning ! '->', not '.', after a struct pointer*/
}

/* return struct by value */
struct birthday get_bday(void){
    struct birthday newbday;
    newbday.year = 1971;  /* '.' after a struct */
    return newbday;
}
```

# Punteros a función

Almacenan la dircción de una función y puden ser usados para invocarla con el operador ()

- Ofrecen mucha flexibilidad en el código
- Permiten pasar una función como argumento,
- O almacenar la dirección de una función a la que invocar en el futuro (*callback*)

```
/* function returning integer */
int func(void);

/* function returning pointer to integer */
int *func(int a);

/* pointer to function returning integer */
int (*func)(void);

/* pointer to func returning ptr to int */
int *(*func)(int);
```

# Punteros a función: Ejemplo

```
#include <stdio.h>
void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
  myproc(10);              /*call myproc with parameter 10*/
  mycaller(myproc,10);     /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
  (*f)(param);             /* call function *f with param */
}

void myproc (int d){
  . . .                    /* do something with d */
}
```

# Ficheros de cabecera y módulos

Un módulo implementa determinada funcionalidad de un programa

- El programa se compone de varios módulos
- Los módulos pueden usar la funcionalidad de otros módulos invocando las funciónes de su interfaz
- El interfaz se define en un fichero de cabecera

Los ficheros de cabecera sólo deben contener:

- Declaraciones de tipos de datos
- Declaraciones adelantadas de funciones
- Declaraciones **externas** de variables definidas en el fichero de implementación del módulo (`.c`)

Para usar la funcionalidad de un módulo en otro debemos incluir su fichero de cabecera.

# Ejemplo de módulos C

my_pgm.h:

```
void myproc(void);
extern int mydata;
```

my_pgm.c:

```
#include <stdio.h>
#include "mypgm.h"

int mydata=0;

void myproc(void){
  mydata=2;
  /* some code */
}
```

main.c:

```
#include <stdio.h>
#include "mypgm.h"

void main(void){
  printf("%d", mydata);
  myproc();
}
```

# Parámetros de la linea de comandos

La función main recibe un array de cadenas de caracteres y su tamaño

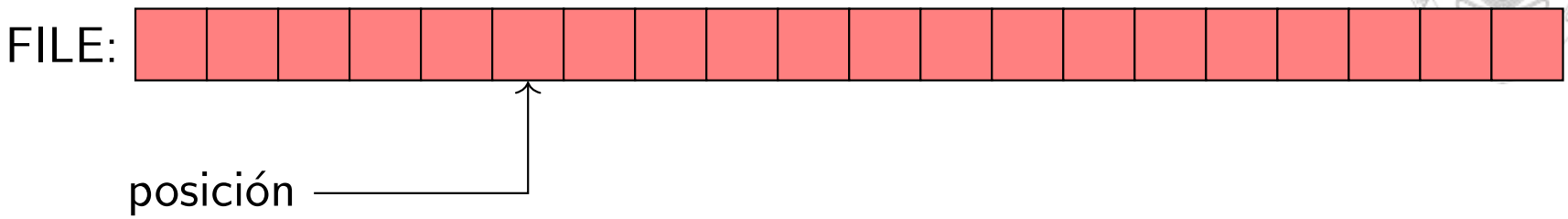
```c
#include <stdio.h>
/* program called with cmd line parameters */
void main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc; i++)
        printf("Argument #%d->|%s|\n", i, argv[i]);
    /* ex., argv[0] == the name of the program */
}
```

Salida por terminal:

```
$ ./example param1 param2 3 4 5 'Hello world'
Argument #0->|./example|
Argument #1->|param1|
Argument #2->|param2|
Argument #3->|3|
Argument #4->|4|
Argument #5->|5|
Argument #6->|Hello world|
```

# Agenda

# Modelo




En C un fichero se modela como:

- una secuencia ordenada de bytes
  - byte **n**: saltar **n** bytes desde el comienzo
- un marcador o puntero de posición en dicha secuencia, que indica la posición a partir de la cual se hará la siguiente operación de lectura o escritura.
  - no confundir con una variable tipo puntero
  - una lectura o escritura hacen avanzar este puntero
- una estructura FILE, cuyo contenido es opaco al programador
- unas operaciones básicas sobre el fichero

# API stdlib para ficheros

```
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *pathname, const char *mode,
              FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
              FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
int fflush(FILE *stream);
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

# Ejemplo: lectura de un string

Queremos una función que:

- Reciba un `stream` (`FILE *`), la dirección de un buffer de salida y el tamaño de dicho buffer.
- leea una cadena de caracteres C válida del stream y la escriba en el buffer de salida
- Devuelva el número de bytes escritos en el buffer.
  - Si la cadena es más larga que el tamaño del buffer la función debe interrumpir la lectura cuando se llene dicho buffer.

La cadena devuelta debe ser siempre una cadena C bien formada, terminada en carácter null (`'0'`)

# Ejemplo: lectura de un string

```
int find_str(char *buf, size_t size, FILE *stream)
{
    int bread, n;

    if (size <= 0)
        return 0;

    bread = 0;
    do {
        if ((n = fread(buf, 1, 1, stream)) == 1){
            bread++;
            buff++;
        }
    } while ((bread < size) && (n > 0) &&
            (*(buf-1) != '\0'));

    *(buf-1) = '\0';
    return bread;
}
```

# Agenda

# Escritura en ascii vs binaria

Si tenemos el siguiente programa:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a = 5;
  fwrite(&a, sizeof(a), 1, stdout);
  return 0;
}
```

y lo ejecutamos desde un terminal...

- ¿se verá algo en el terminal?
- ¿y si volcamos la salida en un fichero y lo abrimos?
- ¿y si le hacemos un cat?

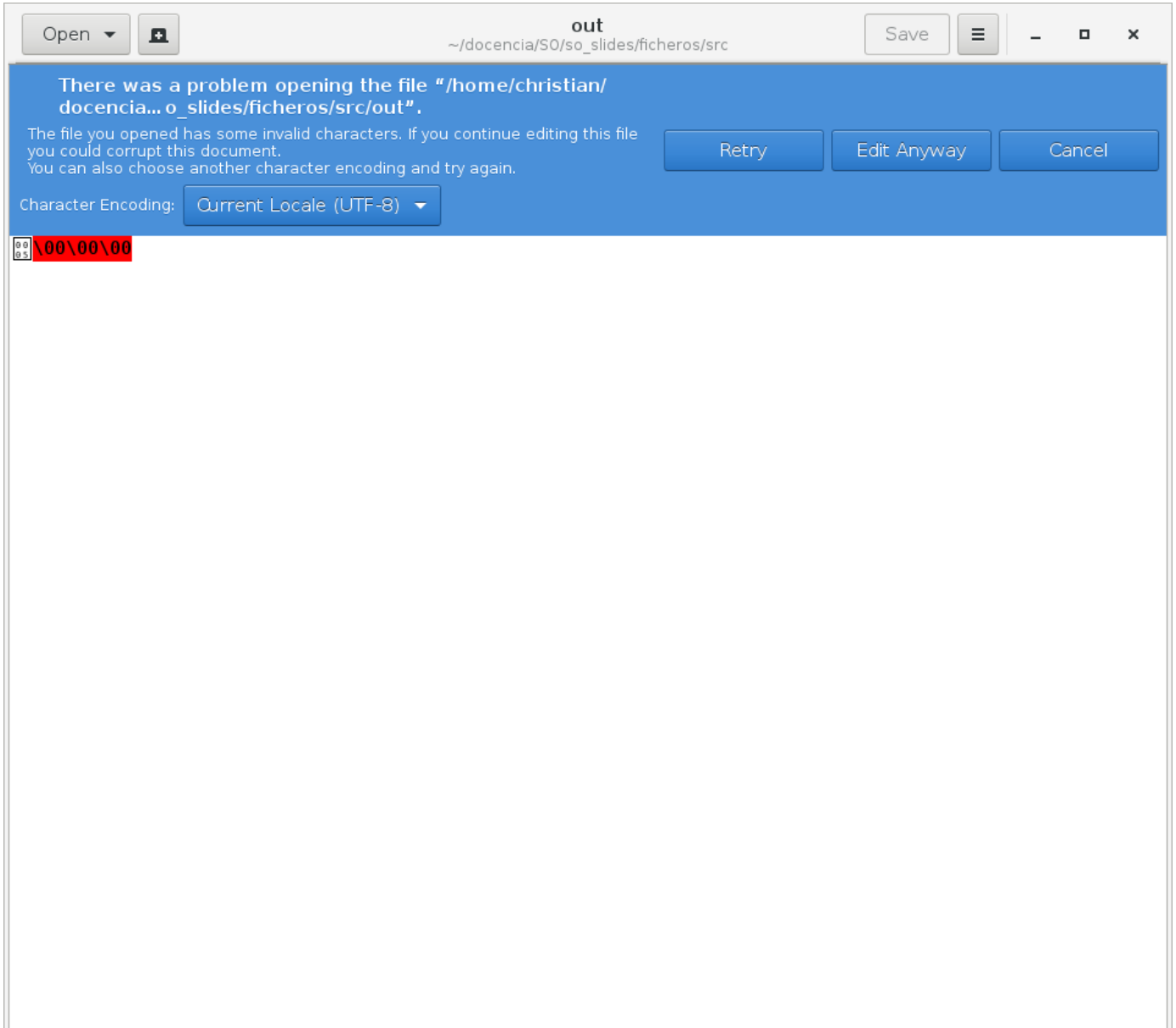
# Escritura en ascii vs binaria

**Escritura binaria**

```
$ ./binwrite
$ ./binwrite | tee
$ ./binwrite > out
$ cat out
$ hexdump -C out
00000000  05 00 00 00                                       |....|
00000004
```

# Escritura en ascii vs binaria

Si lo editamos con un editor:



# Escritura en ascii vs binaria

Repitamos el ejercicio con el siguiente programa:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a = 5;
    fprintf(stdout, "%d", a);
    return 0;
}
```

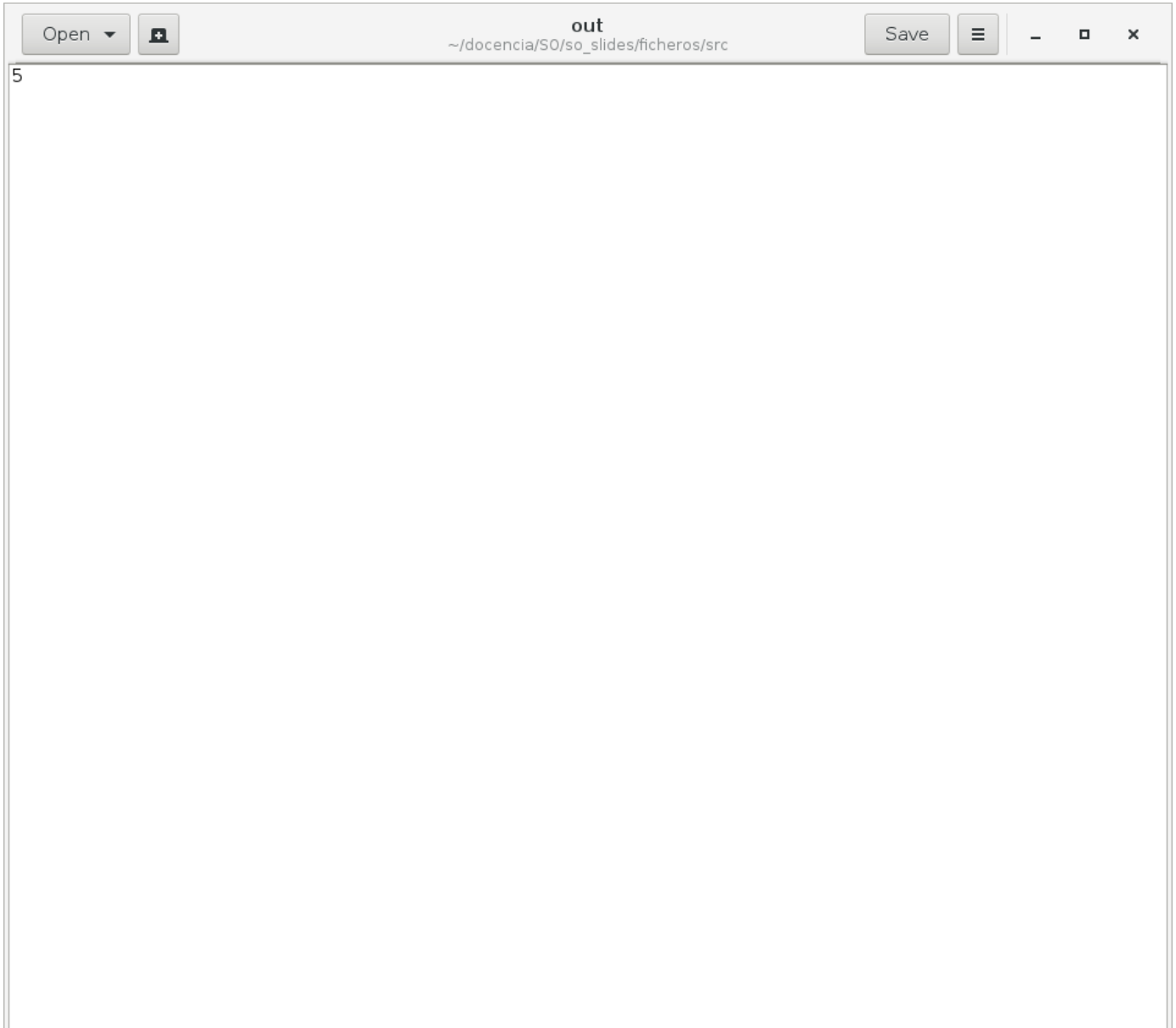
¿Qué pasa ahora?

**Escritura ascii**

```
$ ./asciiwrite
5$ ./asciiwrite | tee
5$ ./asciiwrite > out
$ cat out
5$
```

# Escritura en ascii vs binaria

Si lo editamos con un editor:



# Agenda

# Errores en un sistema POSIX

El lenguaje C no soporta excepciones

- Las llamadas al sistema y muchas funciones de biblioteca devuelven un valor entero que indica si se ha producido un error

Semántica de llamadas al sistema:

- Un valor -1 indica un error
  - Se esribe un código de error en la variable global `errno`
- Un valor distinto no es un error
- Hay algunas excepciones
  - Debemos consultar la página de manual, en la sección de valor de retorno

Semántica de funciones de biblioteca:

- Muchas usan la misma semántica que las llamadas al sistema
- Otras codifican el valor de error en el valor de retorno, como un valor negativo.
  - Debemos consultar la página de manual

# Identificación de errores

- El error producido se codifica numericamente en la variable errno
- El sistema proporciona una serie de funciones de biblioteca que nos permiten informar del error producido:
  - `strerror`: devuelve una cadena de caracteres que describe el error
  - `perror`: imprime por la salida estándar una cadena seguida de la descripción del error que se ha producido

```
int fd;

fd = open("datos.txt", O_RDONLY);
if (fd == -1) {
    perror("open datos.txt");
    exit(EXIT_FAILURE);
}
```

Salida por terminal:

```
$ ./errores1
open datos.txt: No such file or directory
```

# Identificación de errores

Otra alternativa es el uso de las funciones:

```
void err(int eval, const char *fmt, ...);
void warn(const char *fmt, ...);
```

que muestran por la salida estándar una cadena de caracteres de la forma:

```
nombre_programa: cadena_fmt: cadena_strerror
```

o las funciones

```
void errx(int eval, const char *fmt, ...);
void warnx(const char *fmt, ...);
```

que muestran por la salida estándar una cadena de caracteres de la forma:

```
nombre_programa: cadena_fmt
```

Las funciones `err` y `errx` terminan el programa devolviendo al sistema el valor `eval`

# Códigos de error

Son dependientes de la arquitectura

- En un sistema dado puede obtenerse una lista ejecutando el comando `errno -l`:

Salida por terminal:

```
$ errno -l
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
ESRCH 3 No such process
EINTR 4 Interrupted system call
EIO 5 Input/output error
ENXIO 6 No such device or address
E2BIG 7 Argument list too long
ENOEXEC 8 Exec format error
EBADF 9 Bad file descriptor
ECHILD 10 No child processes
EAGAIN 11 Resource temporarily unavailable
...
```

# Semántica de ejecución de programas

Los programas devuelven un valor entero al sistema

- Un valor 0 indica que el programa ha ejecutado correctamente
- Un valor distinto de 0 indica que se ha producido algún error
- Es frecuente devolver el valor de errno o usar las macros `EXIT_SUCCESS` y `EXIT_FAILURE`

Este valor puede consultarse desde el shell, que almacena el valor devuelto en la variable `$?`