



GNU-Linux y C para principiantes

¿Qué debería saber?



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).



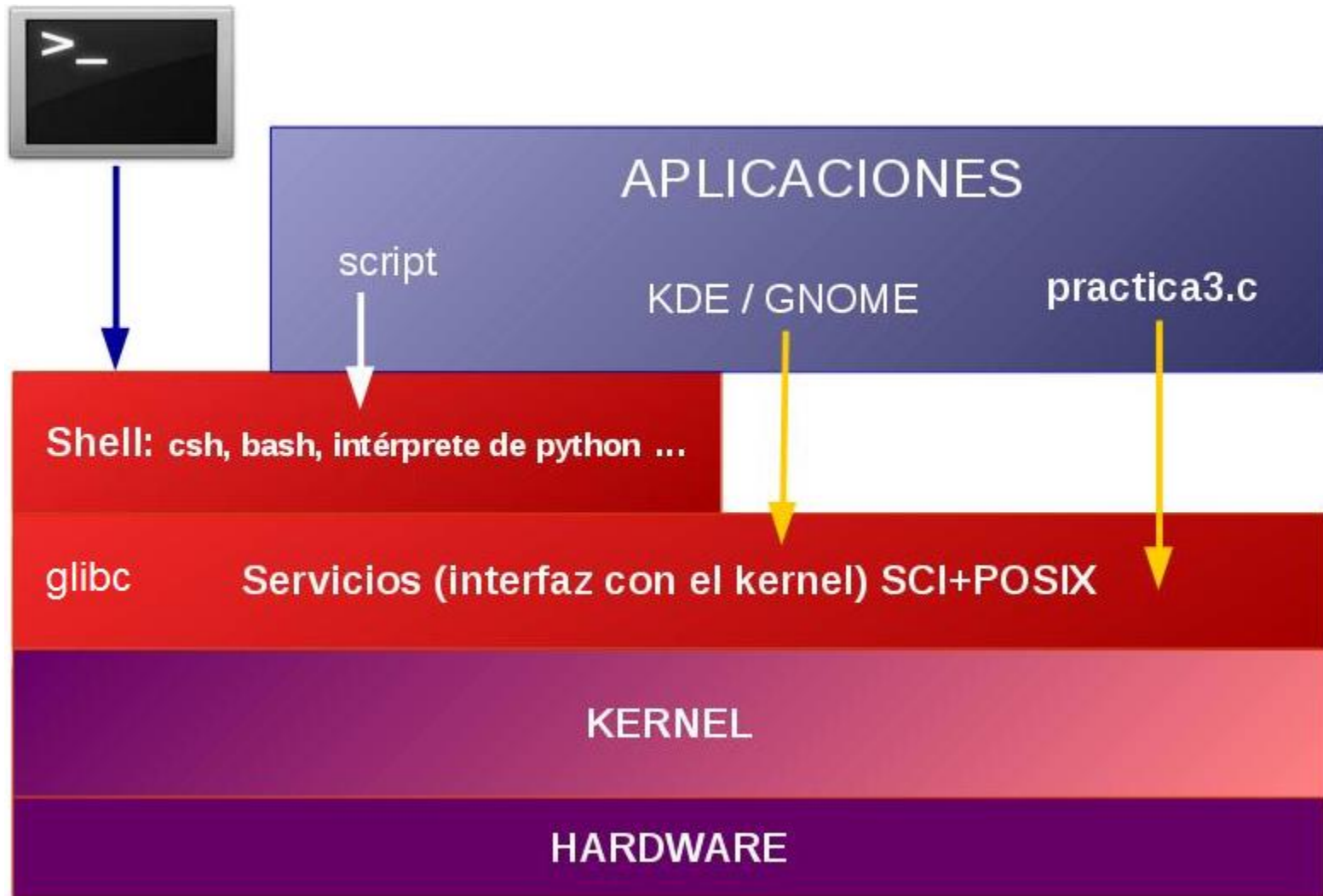
GNU-Linux

- Todo empezó con UNIX
 - Sistema operativo desarrollado en los Laboratorios Bell
 - En sus comienzos era código abierto
- GNU is not Unix
 - Stallman y la gente de la FSF modificaron, mejoraron el UNIX original
- Linus Torvalds
 - Desarrolla el núcleo del sistema operativo, la interacción con el usuario o con el hardware esta desarrollada por el proyecto GNU

GNU-Linux: la interacción con el usuario o con el hardware esta desarrollada por el proyecto GNU y el kernel es de Linus



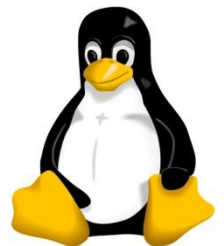
GNU-Linux





GNU-Linux

- Programar
 - La mayoría de los programas están hechos en C, pero no es el único lenguaje posible
- Ejecutar un programa
 - ¿Cómo sabemos que un fichero es un programa?
 - **PREGUNTA MAL FORMULADA**
 - ¿Cómo sabemos que un fichero se puede ejecutar?
 - Windows: *.exe
 - GNU-Linux: Mirando los atributos del fichero





GNU-Linux

- Entonces una *aplicación* se representa mediante:
 - Un fichero ejecutable
 - Un script
- GNU-Linux no los diferencia
- Dónde suelen almacenarse
 - `/bin`: binarios y programas usados para iniciar el sistema
 - `/usr/bin`: Binarios del usuario y programas estándar típicos
 - `/usr/local/bin`: Binarios locales y programas específicos para una instalación





GNU-Linux

■ Comandos básicos (I):

\$ cat prueba.txt

- Su misión **concatenar** archivos aunque puede utilizarse también para visualizar el contenido de un archivo de texto

\$ ls /home/directorio

- Permite **listar** el contenido de un directorio o fichero
- El comando ls tiene varias opciones que permiten organizar la salida
 - **-a** para mostrar los archivos ocultos
 - **-l** para mostrar los usuarios, permisos y la fecha de los archivos

\$ cd practicas

- **Cambiar de directorio**, en el ejemplo pasarías de /home a /home/practicas
- También puedes dar la ruta completa `cd /home/practicas`
- Para subir un nivel `cd ..`

\$ mkdir /home/practicas

- **Make directory** crea un directorio nuevo tomando en cuenta la ubicación actual
- **-p** es una opción bastante útil que permite crear un árbol de directorios completo que no existe.

\$ mkdir -p /home/practicas/prac1/parteA



GNU-Linux



■ Comandos básicos (II)

```
$ cp /home/solucion.txt /home/practicas/solucion.txt
```

- Copia un archivo o directorio origen (siempre primero) a un archivo o directorio destino, origen y destino pueden tener nombres diferentes.
- El comando también cuenta con la opción `-r` que copia no sólo el directorio especificado sino todos sus directorios internos

```
$ mv /home/alumno1/solucion.txt /home/alumno2/misolucion.txt
```

- Mueve un archivo a una ruta específica, y a diferencia de `cp`, lo elimina del origen finalizada la operación. Si indicamos un nombre de destino diferente, `mv` moverá el archivo o directorio con el nuevo nombre (pero siempre borrará el original).

```
$ rm /home/solucion.txt
```

- Borrar un archivo o directorio (remove)
- La opción `-r` borra todos los archivos y directorios.
- Por otra parte, `-f` borra todo sin pedir confirmación.

```
$ touch /home/practica1/solucion.txt
```

- Para crear un archivo vacío

```
$ pwd
```

- Se utiliza para saber dónde estamos (print working directory) imprime nuestra ruta o ubicación al momento de ejecutarlo, así evitamos perdernos si estamos trabajando con múltiples directorios y carpetas

```
$ clear
```

- Si hemos trabajado mucho y nuestro terminal está lleno de basura podemos limpiarlo (`clear`)





GNU-Linux

- **iiii Tengo que utilizar una orden de GNU-Linux y no sé lo que hace !!!!**

\$ man orden

- Siempre que tengas dudas sobre alguna orden en particular puedes consultar la ayuda utilizando esta orden (MANual)
- Acostúmbrate porque te solucionará muchos problemas



POSIX



- Portable Operating Sistem Interface X
 - Es un estandar para mantener la compatibilidad entre sistemas operativos
 - Para poder trasladar de manera sencilla aplicaciones de una versión a otra del sistema operativo
 - La X es por UNIX
 - Si vas a trabajar con este Sistema Operativo necesitas conocerlo



Programación en C



C

- Programar en C no puede ser difícil cuando sólo tiene 32 palabras claves

```
// Declaraciones globales: variables y funciones
// El programa principal SIEMPRE se llama main
int main()
{
    // variable locales
    // secuencia de instrucciones
    return 0;
}
void funcion1(int a, int b)
{
    // variable locales
    // secuencia de instrucciones
}
...
```





C

- Qué debemos saber de C para Sistemas Operativos
 - Además de las instrucciones básicas (for, if-else ...)
 - Diferencia entre variables locales y globales
 - Uso de bibliotecas
 - Paso por parámetros y por referencias (punteros)
 - Gestión de ficheros



C: Variables locales vs globales

■ Programando: Variable globales

```
#include <stdio.h>
void funcion(void)
int numero;
int main()
{
    numero = 5;
    funcion();
    return 0;
}
void funcion()
{
    printf("El numero es %d", numero);
}
```

Es global y puede utilizarse por cualquier programa, tendrá el último valor que se le haya dado

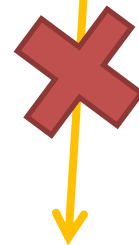


C : Variables locales vs globales

■ Programando: Variable locales

```
#include <stdio.h>
void funcion(int valor)
int main()
{
    int numero= 5;
    funcion(numero);
    return 0;
}
void funcion(int valor)
{
    printf("El numero es %d", numero);
}
```

Es local sólo puede utilizarse por el programa que la define



Da ERROR, no conoce número



C: Variables locales vs globales

- Programando: Variable locales, paso de valores

```
#include <stdio.h>
void funcion(int valor)
int main()
{
    int numero= 5;
    funcion(numero);
    return 0;
}
void funcion(int valor)
{
    printf("El numero es %d", valor);
}
```

valor valdrá 5
para este ejemplo



C: Bibliotecas

■ Programando

¿¿#include <stdio.h>??



- Biblioteca de funciones de entrada salida por consola: por ejemplo printf, scanf, gets, puts ...



C: Biblioteca

- Hagamos un programa

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

```
int main()
{
    printf("Bienvenidos a SO\n")
    exit(0);
}
```

- ¿Cómo escribo este programa en un fichero en gnu-Linux?

```
$ gedit
```





C: Biblioteca

- Hagamos un programa



```
exit(0);
```

- Vuelta la Sistema Operativo

C

- Ejecútalo





C

- Para poder ejecutarlo primero tenemos que pasar de un archivo .c, que en el fondo es un archivo de texto a las instrucciones que entiende el procesador traducidas a ceros y unos.
 - Por lo que se compila y se enlaza (link)
- ```
$ gcc -o nombre_del_ejecutable mi_fichero.c
```

## C



## ■ Ejecuta

```
$./nombre_del_ejecutable
```

```
Bienvenidos a SO
```

```
$
```

Aparece ./ para indicar que el fichero ejecutable que estamos buscando se encuentra en la misma carpeta donde estamos





# C

- Cómo saber que un fichero se puede ejecutar  
`$ls -la`  
`-rwxr-xr-x ... nombre_del_ejecutable`  
`-rw-r--r-- ... mi_fichero.c`
- Si no presenta `-rwxr-xr-x` no es ejecutable
- También se puede saber qué tipo de archivo es con la orden `file`  
`$file nombre_del_ejecutable`  
`nombre_del_ejecutable: ELF 64-bit LSB`  
`executable, x86-64, version 1 (SYSV),`  
`dynamically linked ...`



## C: Biblioteca

- Para programar en C también se necesitan las cabeceras (programa.h)
    - Normalmente se encuentran en `usr/include`
    - Se enlazan sin problema, por eso en el ejemplo anterior no hemos tenido que indicar dónde está `stdio.h` y `stdlib.h`
    - Si se encuentran en un sitio raro, tendrás que indicarlo al compilar
- ```
$ gcc -I/usr/sitio_raro/include mi_programa.c
```
- Aún así utilizará también los ficheros `.h` presentes en `usr/include`



C: Biblioteca

- Además también existen las bibliotecas (library)

```
$ ls /usr/lib
```

- Para compilar utilizando una biblioteca se puede poner:

```
$ gcc -o nom_ejecutable programa.c /usr/lib/biblio.a
```

- Si utilizas una biblioteca estándar existen abreviaciones, por ejemplo, para la biblioteca matemática `libm.a`, en lugar de poner su ruta de puede escribir:

```
$ gcc -o nom_ejecutable programa.c -lm
```




C: Biblioteca

- Avancemos un poco, creamos estos dos programas (en ficheros separados)

```
#include <stdio.h>
void primero(int arg)
{
    printf("Estudiantes matriculados en SO %d\n", arg);
}
/*-----*/
#include <stdio.h>
void segundo(char *arg)
{
    printf("Bienvenidos a SO %s\n", arg);
}
```



C

■ ¿Lo entiendes todo?

```
#include <stdio.h>
void primero(int arg)
{
    printf("Estudiantes matriculados en SO %d\n", arg);
}
/*-----*/
#include <stdio.h>
void segundo(char *arg)
{
    printf("Bienvenidos a SO %s\n", arg);
}
```

C



%c	Un único caracter (char)
%d	Un entero (short, int)
%f	Un número en punto flotante con notación decimal (float, double)
%ld	Long integer (long int)
%p	Una posición de memoria en hexadecimal (*puntero)
%s	Cadena de caracteres (char *)
%u	Entero sin signo (unsigned short, unsigned int, unsigned long)



C

■ ¿Lo entiendes todo?

```
#include <stdio.h>
void primero(int arg)
{
    printf("Estudiantes matriculados en SO %d\n", arg);
}
/*-----*/
#include <stdio.h>
void segundo(char *arg)
{
    printf("Bienvenidos a SO %s\n", arg);
}
```

Escribe en esa posición del texto el número almacenado en arg

Escribe en esa posición del texto la cadena de caracteres almacenada en arg



C

- En C la manera fácil de trabajar con una cadena de caracteres, es decir, una palabra, una frase ... es mediante la utilización d un puntero a la cadena de caracteres:

```
char *variable_cadena;
```

- Es más puedes escribir:

```
char *variable_cadena = "Me gusta SO";
```

- Sólo funciona con cadenas, nunca con un array de enteros

- Hay que tener cuidado de no manipular el puntero, porque en ese caso perderíamos la cadena



C

■ ¿Lo entiendes todo?

```
#include <stdio.h>
void primero(int arg)
{
    printf("Estudiantes matriculados en SO %d\n", arg);
}
/*-----*/
#include <stdio.h>
void segundo(char *arg)
{
    printf("Bienvenidos a SO %s\n", arg);
}
```

La función segundo va a trabajar con una cadena de caracteres, arg indica dónde se encuentra la primera letra de la cadena



C: Biblioteca

- Queremos que estos programas formen parte de una biblioteca, hay que compilarlos pero no crear un ejecutable
 - De todas formas te daría error porque no hay una función que se llame main



Utiliza la orden `man`
para encontrar la
opción de `gcc`



C: Biblioteca

- ¿Lo has encontrado?
`$gcc -c primero.c segundo.c`
- Comprueba que ha creado los ficheros objeto, donde se “guardan las referencias” a las funciones

```
$ls *.o
```

```
primero.o segundo.o
```




C: Biblioteca

- Creamos el archivo cabecera (lib.h)

```
/*
```

```
Esta es la cabecera lib.h que declara las funciones  
primero y segundo
```

```
*/
```

```
void segundo(char *);
```

```
void primero(int);
```

\$gedit lib.h



- Creamos la biblioteca

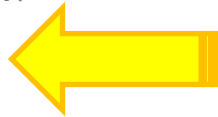
```
$ ar crv mi_biblio.a primero.o segundo.o
```



C: Biblioteca

- Y un programa que llame a uno de estos programas

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



```
int main()
{
    segundo("estudiantes");
    exit (0);
}
```



C: Biblioteca

```
$ gcc -o programa programa.c mi_biblio.a  
$ ./programa  
Bienvenidos a SO estudiantes  
$
```

- Podemos crear la biblioteca siguiendo un estándar, en ese caso el nombre debe empezar por lib

```
$ ar crv libmia.a primero.o segundo.o
```

```
$ gcc -o programa programa.c -L. -lmia  
$ ./programa  
Bienvenidos a SO estudiantes  
$
```



C: parámetros vs referencias

- Vamos a trabajar con C, en algún momento vamos a enfrentarnos al uso de punteros





C: parámetros vs referencias

■ Uso de punteros

Un puntero es una variable, la diferencia es que almacena una posición de memoria



Cuando una función recibe un puntero recibe una **POSICIÓN DE MEMORIA** donde hay algo guardado o es el comienzo de una estructura más compleja



Cuando una función recibe un parámetro recibe un **VALOR**





C: parámetros vs referencias

■ Uso de punteros

```
char *palabra  
int *valor
```

- Un puntero, como una variable debe inicializarse

//Para un array:

//Primero: Declarar e inicializar un array.

```
int array[9]={1,2,3,4,5,6,7,8,9};
```

//Luego le damos la dirección de inicio del array

```
int *puntero = &array[0];
```

- Entonces para asignar a un puntero la dirección de una variable se utiliza el operador &



C: parámetros vs referencias

Siempre para saber DÓNDE está
almacenada una estructura
***puntero = &estructura;**





C: parámetros vs referencias

- Como el puntero es una variable que guarda una dirección de memoria, el valor de esa dirección de memoria se guarda en otra dirección de memoria
- Si necesitamos acceder a la dirección en la que está guardado el puntero:

```
int valor=0;  
int *puntero1=&valor;  
int **puntero2=&puntero1;
```





C: parámetros vs referencias

■ Un puntero a un puntero

Memoria

Dirección (hex)	Valor
0x0100	0
...	
0x0240	0x0100
...	
...	
...	
0x0460	0x0240

← En la dirección 0x0100 está guardada la variable valor, que vale 0

← En la dirección 0x0240 está guardada la variable puntero a valor, que vale 0x0100, es decir, donde está valor

← En la dirección 0x0460 está guardada la variable puntero al puntero de valor, que vale 0x0240, es decir, donde está el puntero a valor



C: parámetros vs referencias

```
void intercambia (int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```



C: parámetros vs referencias

```
void intercambia (int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

¡¡ No Funciona !!





C: parámetros vs a referencias

- Cuando pasas un parámetro pasas el VALOR
 - Un valor desprovisto de toda referencia a la variable (aunque en el programa se llame igual)
 - Lo que ocurre en el programa (intercambia) es que se está utilizando una variable que también se llama a
 - El programa intercambia lo que haces es intercambiar los valores de a y b, pero este intercambio NO se almacena en memoria
 - ¡¡No sabemos dónde habría que guardarlos!!
- Un ejemplo simplificado:
 - *Intercambia nos dice que tenemos que pintar la habitación roja de amarillo y la amarillo de roja, pero ... ¡¡dónde c*** están esas habitaciones!!*



C: parámetros vs referencias

```
void intercambia (int *a, int *b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```



C: parámetros vs referencias

```
void intercambia (int *a, int *b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

¡¡ No Funciona !!





C: parámetros vs referencias

- Ahora intercambia SI sabe donde se encuentran a y b, pero tmp es un valor
- ***“Un vaso es un vaso y un plato es un plato”***



C: parámetros vs referencias

```
void intercambia (int *a, int *b)
{
    int *tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```




C: parámetros vs referencias

```
void intercambia (int *a, int *b)
{
    int *tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

¡¡ No Funciona !!





C: parámetros vs referencias

- Aunque como variable de la función estamos trabajando con la posición de memoria de a y la posición de memoria de b
`int *a, int *b`
- Dentro del programa estamos trabajando con el valor de a y con el valor de b



C: parámetros vs referencias

```
void intercambia (int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```





C: parámetros vs referencias

- Como paso las referencias desde el programa principal (main)

```
int main()
{
    int a = 5;
    int b = 0;
    printf("Antes del cambio %d\n", a);
    intercambia (&a, &b);
    printf("Despues del cambio %d\n", a);
    exit(0);
}
```

Se definen a y b
como un VALOR

La función necesita la
DIRECCIÓN de a y la de b

C: gestión de ficheros



- Existen tres variables
 - stdin: entrada estándar
 - stdout: salida estándar
 - stderr: salida de error



C: gestión de ficheros

```
include <stdio.h>
main()
{
    FILE *archivo1, *archivo2;
    char c;
    archivo1 = fopen("prueba.txt","r");
    archivo2 = fopen("copia.txt","w");
    if ((archivo1 == NULL) || (archivo2 ==NULL))
        printf ("Error al abrir ficheros");
    else
    {
        while ((c=getc(archivo1)) != EOF)
        {
            putc (c, archivo2);
        }
        if (fclose(archivo1)!=0)
            printf ("Error al cerrar el fichero prueba");
        if (fclose(archivo2)!=0)
            printf ("Error al cerrar el fichero copia");
    }
}
```





C: gestión de ficheros

- `include <stdio.h>`
 - Se necesita esa biblioteca estándar
- `FILE *archivo1, *archivo2;`
 - La única manera de gestionar un archivo es mediante un puntero al archivo, es decir, conocer dónde empieza el archivo
- `archivo1 = fopen("prueba.txt", "r");`
 - Para trabajar con el archivo hay que abrirlo (**fopen**)
 - r: sólo lectura (el fichero debe existir previamente)
 - w: sólo escritura
- `archivo1 == NULL`
 - Hemos metido la pata, el archivo no existe, no continuamos con el algoritmo
- `fclose(archivo1)`
 - Al terminar el algoritmo tenemos que cerrar el archivo
 - Si es igual a cero es que todo ha funcionado



C: gestión de ficheros

- `c=getc(archivo1)`
 - Para leer carácter a carácter del archivo se utiliza la función **getc**
 - Si el carácter es EOF se ha llegado al final del archivo (End Of File)
- `putc (c, archivo2);`
 - Para escribir carácter a carácter en un archivo



C: gestión de ficheros

```
include <stdio.h>
main()
{
    FILE *archivo1, *archivo2;
    char c;
    archivo1 = fopen("prueba.txt","r");
    archivo2 = fopen("copia.txt","w");
    if ((archivo1 == NULL) || (archivo2 ==NULL))
        printf ("Error al abrir ficheros");
    else
    {
        while ((c=getc(archivo1)) != EOF)
        {
            putc (c, archivo2);
        }
        if (fclose(archivo1)!=0)
            printf ("Error al cerrar el fichero prueba");
        if (fclose(archivo2)!=0)
            printf ("Error al cerrar el fichero copia");
    }
}
```





C: gestión de ficheros

- ¿Serías capaz de hacer un programa que escribiera los datos de 3 archivos prueba en un único archivo copia?



C vs C++

- Yo es que sé programar en C++ y por lo tanto no sé programar en C





gdb

- Para depurar en GNU-Linux se puede utilizar cualquier programa de depuración al que estéis acostumbrados
- Uno de ellos es gdb:
 - GNU Project Debugger
 - Suele venir por defecto, te lo puedes instalar:
`$ sudo apt-get install gdb`

gdb



■ Este programa compila pero no funciona

```
int main()
{
    int a[4]={1,2,3,4};
    printf("Antes de entrar %i\n", a[0]);
    gira (a);
    printf("Despues del cambio %i\n",a[0]);
    exit(0);
}

void gira (int *a)
{
    int c[4];
    int i;
    int j=0;
    for (i=0; i<4; i++) c[i]=a[i];
    for (i=4; i>0; i++)
    {
        a[i-1]=c[j];
        j++;
    }
}
```





gdb

- Para utilizar gdb sobre un programa que no funciona correctamente (aunque si compila)

```
$ gcc -g programa.c
```

```
$ gdb
```

Mirar en man que significa -g
Se crea a.out, que es el ejecutable por defecto de gcc con toda la información de depuración

- Ahora indicamos que queremos ejecutar el programa

```
(gdb) file a.out
```

```
(gdb) run
```

Puedes hacer un list antes de run para comprobar que se ha cargado el programa



gdb

- Aparece el siguiente mensaje

Program received signal SIGSEV, Segmentation fault.
0x000000000000400646 in gira (a=0x7ffffffe000) at c:29

```
29      a[i-1]=c[j];
```

- Parece que hay un problema en la asignación de a a c. Podemos ver los valores de esas variables con print

```
(gdb) print a[0]
```

```
(gdb) print c[0]
```

```
(gdb) print i
```

```
(gdb) print j
```





gdb

- Los valores de i y j no tienen mucho sentido, vamos a poner un breakpoint al empezar el segundo for (línea 28)

```
(gdb) break 28
```

```
(gdb) run
```

```
Breakpoint 1 at 0x0400620 file programa.c line 28
```

```
28      for(i=4; i>0; i++)
```

- Ejecutamos paso a paso

```
(gdb) step
```

```
29      a[i-1] = c[j];
```




gdb

- Como el fallo ha sido en la línea 29, comprobamos los valores de i y de j (print)
- Recorremos el bucle paso a paso hasta volver a colocarnos en la línea 29

```
(gdb) step
```

```
29      a[i-1] = c[j];
```

```
(gdb) print j
```

```
$1 = 1
```

```
(gdb) print i
```

```
$2 = 5
```





Crear un Script

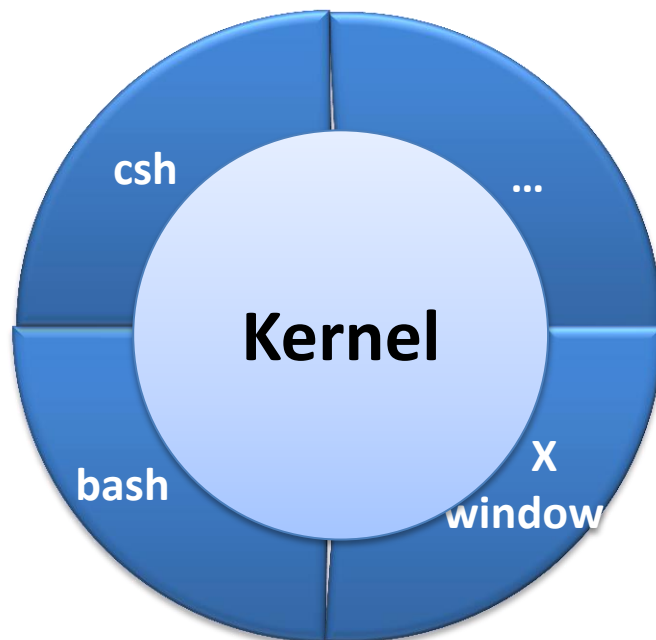


Crear un Script

- Shell es un programa que actúa de interfaz entre el usuario y el kernel
- Un script es un programa escrito en un entorno de ejecución para automatizar la ejecución de unas tareas.
 - El script es interpretado más que compilado
 - Por lo tanto las primitivas de este lenguaje son tareas elementales o llamadas a APIs
- En gnu-linux puede escribirse en python, en bash ...
 - Bourne again shell (Bash): es un programa cuya función consiste en interpretar órdenes. Está basado en la shell de Unix y es compatible con POSIX.



Crear un script





Crear un script

```
#!/bin/bash
```

```
#Comentario
```

```
echo "Como te llamas: "
```

```
read nombre
```

```
echo "Bienvenido a SO, $nombre"
```

¿Qué hace?





Crear un script

```
#!/bin/bash
```

Indica que es un ejecutable
escrito en bash

```
#Comentario
```

```
echo "Como te llamas: "
```

```
read nombre
```

Espera una
entrada

```
echo "Bienvenido a SO, $nombre"
```

Escribir por
pantalla




Crear un script

- Dónde hay que guardarlo:
 - Se puede guardar en cualquier directorio donde tengas permiso
- Qué extensión hay que ponerle:
 - La que tú quieras que te resulte útil, por ejemplo, si está en *bash* usa *sh*.
- Desde dónde puedo ejecutarlo:
 - Desde cualquier directorio, siempre que des el *path*



Crear un script

■ Cómo lo ejecuto




Puedo probar:
`$/nombre_del__script`

A 3D rendered white character is sitting on the ground, hunched over with its hands covering its face in a thinking or distressed pose. Three small blue circles lead from the character's head to a large blue thought bubble.

Crear un script

■ Cómo lo ejecuto

A 3D white figure sitting on the ground, holding its head in its hands, appearing to be in deep thought or frustration.

Puedo probar:
`$/nombre_del__script`

¡No funciona!





Crear un script

- No tienes permiso para ejecutarlo

Puedo probar:

```
$ chmod +x nombre_del__script.sh  
$ ./nombre_del__script
```





Crear un script

- If – else

```
#!/bin/bash
NUM=5
echo "Piensa un numero"
read n
if [ $n -eq $NUM ] then
    echo "Has acertado"
elif [ $n -gt $NUM ] then
    echo "Te has pasado"
else
    echo "No has llegado"
fi
```



Crear un script

■ NUM

– Es una variable que se va a utilizar a lo largo del script

■ La estructura if-else es como se indica en la transparencia anterior:

– la condición va siempre entre corchetes

```
if [ _$n -eq $NUM _ ] then
```

– Toda la estructura acaba con **fi**

– Además:

- *lt*, *gt*, *le*, *ge*, *eq* y *ne* son comparaciones aritméticas
- para comparar cadenas de texto =, !=, < y >



Crear un script

```
#!/bin/bash
# bash while
COUNT=5
while [ $COUNT -gt 0 ]; do
    echo "El valor de cuenta es: " $COUNT
    let COUNT=COUNT-1
done
```



Crear un script

Es demasiada información
Dónde encuentro las respuestas?





Crear un script

man bash?





Crear un script

- En el script también se pueden usar órdenes del *shell*

```
#!/bin/bash  
cat prueba.txt
```




Crear un script

```
#!/bin/bash
file="prueba.txt"
if [ -e $file ]; then
    cat $file
else
    touch $file
fi
```



Crear un script

■ Qué se puede saber de un fichero:

-d directoryname	Check for directory existence
-e filename	Check for file existence
-f filename	Check for regular file existence not a directory
-r filename	Check if file is a readable
-s filename	Check if file is nonzero size
-w filename	Check if file is writable
-x filename	Check if file is executable



Crear un script

- ¿Cómo conseguiríamos que el nombre del fichero entre por teclado?



Crear un script

- Como hacerlo sin *read*

```
#!/bin/bash
if [ -e $1 ]; then
    cat $1
else
    touch $1
fi
```



A diagram illustrating the execution of the script. A blue rounded rectangle contains the text `./script.sh` followed by `prueba.txt`. The `prueba.txt` is enclosed in a red dashed oval. A red arrow points from the label `$1` above to the `prueba.txt` argument.

`./script.sh` `prueba.txt`



Crear un script

- \$1 es el primer parámetro, \$2 es el segundo ...





Crear un script

```
#!/bin/bash  
for i in ls *.txt  
do cat $i  
done
```

- ¿Qué hace este código?





Crear un script

- `for – do – done` : Son palabras reservadas para hacer un `for`
- `ls *.txt`
 - Pruébalo en línea de comandos ¿qué hace?
- `cat $i`
 - Si sabes lo que hace `ls`, ¿qué es lo que guarda `$i`?
- Por eso veo el contenido de todos los ficheros uno detrás de otro ...





Crear un script

```
#!/bin/bash  
ls *.txt > info.txt
```

- ¿Qué hace este código?





Crear un script

- En este caso el operador `>` envía la salida que esperaríamos ver por pantalla al fichero
 - Una lista de todos los ficheros txt de la carpeta
- Entonces ...

```
for i in ls *.txt
do cat $i
done > "copia.txt"
```





Crear un script

- Normalmente el resultado de una orden se almacena en STDOUT
- Si el resultado es error se almacena en STDERR
 - Puede ocurrir que queramos almacenar **SÓLO** el resultado de la salida error
 - *orden incorrecta* **2**> STDERR
 - También puede almacenarse en un archivo



Crear un script

- ¿Y si queremos un único fichero para el error y la salida por pantalla?

\$ Orden **2> STDERR_STDOUT 1>&2**

— Ejemplo

- Tenemos archivo1.txt y no archivo 2.txt

\$ ls archivo1.* archivo2.* **2> STDERR_STDOUT 1>&2**

- En STDERR_STDOUT tenemos:

ls: no se puede acceder a archivo2.: No existe el fichero
o el directorio*

archivo1.txt