

# Práctica 1: Introducción al entorno de desarrollo y la programación de sistemas

## Índice

<b>1</b>	<b>Objetivos</b>	<b>1</b>
<b>2</b>	<b>Requisitos</b>	<b>1</b>
<b>3</b>	<b>Ejercicios</b>	<b>2</b>
Ejercicio 1 . . . . .		2
1. Compilación . . . . .		2
2. Herramienta make . . . . .		2
3. Tamaño de variables . . . . .		2
4. Arrays . . . . .		3
5. Punteros . . . . .		3
6. Funciones . . . . .		4
7. Cadenas de caracteres (strings) . . . . .		4
Ejercicio 2 . . . . .		4
Ejercicio 3 . . . . .		6
Ejercicio 4 . . . . .		7
Ejercicio 5 . . . . .		8

## 1 Objetivos

- Familiarizarse con el entorno de desarrollo de aplicaciones C en GNU/Linux.
- Revisar los fundamentos de C
- Familiarizarse con el uso de la función `getopt ( )` para el tratamiento de opciones
- Conocer las funciones esenciales de procesamiento de cadenas de caracteres
- Familiarizarse con el manejo básico del shell e introducirse a su programación.

El archivo [ficheros\\_p1.tar.gz](#) contiene una serie de ficheros para la realización de algunos de los ejercicios de esta práctica.

## 2 Requisitos

Para poder realizar con éxito la práctica el alumno debe haber leído y comprendido los siguientes documentos facilitados por el profesor:

- Transparencias de clase de Introducción al entorno de desarrollo, que nos introduce al entorno GNU/Linux que utilizaremos en el laboratorio, y describe cómo trabajar con proyectos C con Makefile. Además contienen un repaso de los conocimientos de C necesarios para realizar con éxito las prácticas, haciendo especial hincapié en los errores que cometen habitualmente los estudiantes menos experimentados en el lenguaje C.
- Manual del laboratorio titulado [Entorno de desarrollo C para GNU/Linux](#), que describe las herramientas que componen el entorno de desarrollo que vamos a utilizar, así como las funciones básicas de la biblioteca estándar de C que los alumnos deben conocer.
- Presentación “Introducción a Bash”, que realiza una breve introducción al interprete de órdenes (shell) Bash.

## 3 Ejercicios

### Ejercicio 1

En el directorio `ejercicio1` de los ficheros para la práctica ( [ficheros\\_p1.tar.gz](#)) hay una serie de subdirectorios con códigos de pequeños programas de C que pretenden poner de manifiesto algunos de los errores frecuentes que cometen los programadores con poca experiencia con C así como familiarizar al estudiante con las herramientas básicas de compilación que se usan en un entorno Linux.

Para cada directorio se proporciona una serie de tareas y preguntas que deberás responder, para las cuales tendrás que examinar y probar los ejemplos proporcionados. Consulta el [manual del entorno](#) para saber como utilizar el compilador. Puedes utilizar VSCode como editor.

#### 1. Compilación

- Compila el código del ejercicio y ejecútalo
- Obtén la salida de la etapa de pre-procesado (opción `-E` o la opción `--save-temps` para obtener la salida de todas las etapas intermedias) y en un fichero `hello2.i`
- ¿Qué ha ocurrido con la “llamada a `min()`” en `hello2.i`?
- ¿Qué efecto ha tenido la directiva `#include <stdio.h>`?

#### 2. Herramienta make

- Examina el makefile, identifica las variables definidas, los objetivos (*targets*) y las reglas.
- Ejecuta `make` en la línea de comandos y comprueba las ordenes que ejecuta para construir el proyecto.
- Marca el fichero `aux.c` como modificado ejecutando `touch aux.c`. Después ejecuta de nuevo `make`. ¿Qué diferencia hay con la primera vez que lo ejecutaste? ¿Por qué?
- Ejecuta la orden `make clean`. ¿Qué ha sucedido? Observa que el objetivo `clean` está marcado como *phony* en la directiva `.PHONY: clean`. ¿por qué? Para comprobarlo puedes comentar dicha línea del makefile, compilar de nuevo haciendo `make`, y después crear un fichero en el mismo directorio que se llame `clean`, usando el comando `touch clean`. Ejecuta ahora `make clean`, ¿qué pasa?
- Comenta la línea `LIBS = -lm` poniendo delante una almoadilla (`#`). Vuelve a contruir el proyecto ejecutando `make` (haz un `clean` antes si es necesario). ¿Qué sucede? ¿Qué etapa es la que da problemas?

#### 3. Tamaño de variables

Compila y ejecuta el código de cada uno de los ejemplos proporcionados y responde a las preguntas proporcionadas para ellos.

- `main1.c`
  - ¿Por qué el primer `printf()` imprime valores distintos para ‘a’ con los modificadores `%d` y `%c`?
  - ¿Cuánto ocupa un tipo de datos `char`?

- ¿Por qué el valor de 'a' cambia tanto al incrementarlo en 6? (la respuesta está relacionada con la cuestión anterior)
- Si un "long" y un "double" ocupan lo mismo, ¿por qué hay 2 tipos de datos diferentes?
- *main2.c*
  - ¿Tenemos un problema de compilación o de ejecución?
  - ¿Por qué se da el problema?. Solucionalo, compila y ejecuta de nuevo.
  - a,b,c, y x están declaradas de forma consecutiva. ¿Son sus direcciones consecutivas?
  - ¿Qué significa el modificar "%lu" en `printf()`?
  - ¿A qué dirección apunta "pc"? ¿Coincide con la de alguna variable anteriormente declarada? Si es así, ¿Coinciden los tamaños de ambas?
  - ¿Coincide el valor del tamaño de `array1` con el número de elementos del array? ¿Por qué?
  - ¿Coinciden las direcciones a la que apunta `str1` con la de `str2`?
  - ¿Por qué los tamaños (según `sizeof()`) de `str1` y `str2` son diferentes?

## 4. Arrays

Compila y ejecuta el código de los ejemplos proporcionados y responde a las preguntas propuestas para cada uno de ellos.

- *array1.c*
  - ¿Por qué no es necesario escribir "&list" para obtener la dirección del array `list`?
  - ¿Qué hay almacenado en la dirección de `list`?
  - ¿Por qué es necesario pasar como argumento el tamaño del array en la función `init_array`?
  - ¿Por qué el tamaño devuelto por `sizeof()` para el array de la función `init_array` no coincide con el declarado en `main()`?
  - ¿Por qué NO es necesario pasar como argumento el tamaño del array en la función `init_array2`?
  - ¿Coincide el tamaño devuelto por `sizeof()` para el array de la función `init_array2` con el declarado en `main()`?
- *array2.c*
  - ¿La copia del array se realiza correctamente? ¿Por qué?
  - Si no es correcto, escribe un código que sí realice la copia correctamente.
  - Descomenta la llamada a la función `tmo` en la función `main()`. Compila de nuevo y ejecuta.
  - El problema que se produce, ¿es de compilación o de ejecución? ¿Por qué se produce?
  - Encuentra un valor de `MAXVALID` superior a 4 con el que no se dé el problema. ¿Se está escribiendo más allá del tamaño del array? Si es así, ¿por qué funciona el código?

## 5. Punteros

Compila y ejecuta el código de los ejemplos y responde a las cuestiones proporcionadas para cada uno de ellos.

- *punteros1.c*
  - ¿Qué operador utilizamos para declarar una variable como un puntero a otro tipo?
  - ¿Qué operador utilizamos para obtener la dirección de una variable?
  - ¿Qué operador se utiliza para acceder al contenido de la dirección "a la que apunta" un puntero?
  - Hay un error en el código. ¿Se produce en compilación o en ejecución? ¿Por qué se produce?
- *punteros2.c*
  - ¿Cuántos bytes se reservan en memoria con la llamada a `malloc()`?
  - ¿Cuál es la dirección del primer y último byte de dicha zona reservada?
  - ¿Por qué el contenido de la dirección apuntada por `ptr` es 7 y no 5 en el primer `printf()`?
  - ¿Por qué se modifica el contenido de `ptr[1]` tras la sentencia `*ptr2=15;`?
  - Indica dos modos diferentes de escribir el valor 13 en la dirección correspondiente a `ptr[100]`.

- Hay un error en el código. ¿Se manifiesta en compilación o en ejecución? Aunque no se manifieste, el error está. ¿Cuál es?
- *punteros3.c*
  - ¿Por qué cambia el valor de `ptr[13]` tras la asignación `ptr = &c`?
  - El código tiene (al menos) un error. ¿Se manifiesta en compilación o en ejecución? ¿Por qué?
  - ¿Qué ocurre con la zona reservada por `malloc()` tras a asignación `ptr = &c`? ¿Cómo se puede acceder a ella? ¿Cómo se puede liberar dicha zona?

## 6. Funciones

Compila y ejecuta el código de cada uno de los ejemplos proporcionados y responde a las cuestiones proporcionadas para cada uno de ellos.

- *arg1.c*
  - ¿Por qué el valor de `xc` no se modifica tras la llamada a `sumC`? ¿Dónde se modifica esa información?
  - Comenta las dos declaraciones adelantadas de `sum()` y `sumC()`. Compila de nuevo, ¿Qué ocurre?
- *arg2.c*
  - ¿Por qué cambia el valor de `y` tras la llamada a `suma()`?
  - ¿Por qué en ocasiones se usa el operador `'.'` y en otras `'->'` para acceder a los campos de una estructura?
  - ¿Por qué el valor de `zc` pasa a ser incorrecto sin volver a usarlo en el código?
  - Corrije el código para evitar el error producido en `zc`

## 7. Cadenas de caracteres (strings)

Compila y ejecuta el código de cada uno de los ejemplos proporcionados y responde a las cuestiones proporcionadas para cada uno de ellos.

- *strings1.c*
  - El código contiene un error. ¿Se manifiesta en compilación o en ejecución? ¿Por qué se produce? Soluciona el error comentando la(s) línea(s) afectadas. Vuelve a compilar y ejecutar.
  - ¿En qué dirección está la letra 'B' de la cadena "Bonjour"? ¿Y la de la letra 'j'?
  - Tras la asignación `p=msg2`, ¿cómo podemos recuperar la dirección de la cadena "Bonjour"?
  - ¿Por qué la longitud de las cadenas `p` y `msg2` es 2 tras la línea 30? Se asignan 3 bytes a 'p' que modifican a ambos, pero luego la longitud es sólo 2.
  - ¿Por qué `strlen()` devuelve un valor diferente a `sizeof()`?
- *strings2.c*
  - El código de `copy` no funciona. ¿Por qué?
  - Usa ahora la función `copy2()` (descomenta la línea correspondiente). ¿Funciona la copia?
  - Propón una implementación correcta de la copia.
  - ¿Qué hace la función `mod()`? ¿Por qué funciona?
  - Descomenta la última llamada a la función `mod()`. Compila y ejecuta. ¿Por qué se produce el error?

## Ejercicio 2

El programa *primes* cuyo código fuente se muestra a continuación, ha sido desarrollado para calcular la suma de los *n* primeros números primos. Lamentablemente, el programador ha cometido algunos errores. Utilizando el depurador de C `gdb` el alumno debe encontrar y corregir los errores. Compilar directamente en línea de comandos: `gcc -g -w -o primes primes.c`

```
/**
 * This program calculates the sum of the first n prime
 * numbers. Optionally, it allows the user to provide as argument the
 * value of n, which is 10 by default.
```

```

*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/**
 * This function takes an array of integers and returns the sum of its n elements.
 */
int sum(int *arr, int n);

/**
 * This function fills an array with the first n prime numbers.
 */
void compute_primes(int* result, int n);

/**
 * This function returns 1 if the integer provided is a prime, 0 otherwise.
 */
int is_prime(int x);

int main(int argc, char **argv) {

    int n = 10; // by default the first 10 primes
    if(argc == 2) {
        atoi(argv[2]);
    }
    int* primes = (int*)malloc(n*sizeof(int));
    compute_primes(primes, n);

    int s = sum(primes, n);
    printf("The sum of the first %d primes is %d\n", n, s);

    free(primes);
    return 0;
}

int sum(int *arr, int n) {
    int i;
    int total;
    for(i=0; i<n; i++) {
        total += arr[i];
    }
    return total;
}

void compute_primes(int* result, int n) {
    int i = 0;
    int x = 2;
    while(i < n) {
        if(is_prime(x)) {
            result[i] = x;
            i++;
            x += 2;
        }
    }
    return;
}

int is_prime(int x) {
    if(x % 2 == 0) {
        return 0;
    }

```

```

}
for(int i=3; i<x; i+=2) {
    if(x % i == 0) {
        return 0;
    }
}
return 1;
}

```

### Ejercicio 3

En este ejercicio, trabajaremos el uso de `getopt()` una herramienta esencial para el procesamiento de opciones en línea de comando. El objetivo del ejercicio es completar el código del fichero `getopt.c` para que sea capaz de procesar las opciones `-e` y `-l` tal y como indica el uso del programa, que puede consultarse con la opción `-h`:

```

$ make
$ ./getopt -h
Usage: ./getopt [ options ] title

options:
  -h: display this help message
  -e: print even numbers instead of odd (default)
  -l lenght: lenght of the sequence to be printed
  title: name of the sequence to be printed

```

Una vez completado, el programa deberá imprimir una secuencia de `lenght` números (10 por defecto; podemos cambiarlo con la opción `-l`) impares (por defecto) o pares si se incluye la opción `-e`. Los argumentos `-l lenght` y `-e` son opcionales, pero el argumento `title` siempre debe estar presente en la línea de comando.

Ejemplos de salidas para diferentes combinaciones de entrada:

```

$ ./getopt hola
Title: hola
1 3 5 7 9 11 13 15 17 19

./getopt -l 3 hola1
Title: hola1
1 3 5

./getopt -l 4 -e hola2
Title: hola2
2 4 6 8

```

Es necesario familiarizarse con la función `getopt()` consultando la página de manual de `getopt()`: `man 3 getopt`

- `int getopt(int argc, char *const argv[], const char *optstring);`

La función suele invocarse desde `main()`, y sus dos primeros parámetros coinciden con los argumentos `argc` y `argv` pasados a `main()`. El parámetro `optstring` sirve para indicar de forma compacta a `getopt()` cuáles son las opciones que el programa acepta –cada una identificada por una letra–, y si éstas a su vez aceptan parámetros obligatorios u opcionales.

Deben tenerse en cuenta las siguientes consideraciones:

1. La función `getopt()` se usa en combinación con un bucle, que invoca tantas veces la función como opciones ha pasado el usuario en la línea de comandos. Cada vez que la función se invoca y encuentra una opción, `getopt()` retorna el carácter correspondiente a dicha opción. Por lo tanto, dentro del bucle suele emplearse la construcción *switch-case* de C para llevar a cabo el procesamiento de las distintas opciones. Es aconsejable no

realizar el procesamiento de nuestro programa dentro del bucle, sino únicamente procesar las opciones y dar valor a variables/flags que serán utilizadas en el resto de nuestro código para decidir el comportamiento que debe tener.

2. Un aspecto particular de la función `getopt()` es que establece el valor de distintas variables globales tras invocarse, siendo las más relevantes las siguientes:
  - `char* optarg`: almacena el argumento pasado a la opción actual reconocida, si ésta acepta argumentos. Si la opción no incluye un argumento, entonces `optarg` se establece a `NULL`
  - `int optind`: representa el índice del siguiente elemento en el `argv` (elementos que quedan sin procesar). Se usa frecuentemente para procesar argumentos adicionales del programa que no están asociados a ninguna opción. Veremos un ejemplo de ello en la práctica.

Para completar el código, incluye las opciones `-l` y `-e` en la llamada a `getopt()` y completa la estructura *switch-case* para modificar los valores por defecto de la variable `options`. Para leer el valor numérico asociado a la opción `-l`, deberás utilizar la variable global `optarg`, teniendo en cuenta que esta variable es una cadena de caracteres (tipo `char*`) y, sin embargo, queremos almacenar la opción como un número entero (tipo `int`). Consulta el uso de la función `strtol()` en el manual (`man 3 strtol`) para saber cómo realizar esa conversión.

Asimismo, dado que el argumento `title` no será procesado por `getopt()` (pues no está precedido por una marca de opción al estilo `-l`), deberemos continuar el procesamiento de la cadena de entrada tras el bucle `for`. Para ello, se usará la variable `optind` junto con `argv` para almacenar el valor de la cadena de caracteres que será el título de nuestra secuencia.

Completa el código y responde a las siguientes preguntas:

1. ¿Qué cadena de caracteres debes utilizar como tercer argumento de `getopt()`?
2. ¿Qué línea de código utilizas para leer el argumento `title`?

## Ejercicio 4

Estudiar el código y el funcionamiento del programa `show-passwd.c`, que lee el contenido del fichero del sistema `/etc/passwd` e imprime por pantalla (o en otro fichero dado) las distintas entradas de `/etc/passwd` —una por línea—, así como los distintos campos de cada entrada. El fichero `/etc/passwd` almacena en formato de texto plano información esencial de los usuarios del sistema, como su identificador numérico de usuario o grupo así como el programa configurado como intérprete de órdenes (*shell*) predeterminado para cada usuario. Para obtener más información sobre este fichero se ha de consultar su página de manual: `man 5 passwd`

El modo de uso del programa puede consultarse invocándolo con la opción `-h`:

```
$ ./show-passwd -h
Usage: ./show-passwd [ -h | -v | -p | -o <output_file> ]
```

Las opciones `-v` y `-p`, permiten configurar el formato en el que el programa imprime la información de `/etc/passwd`. Las citadas opciones activan respectivamente el modo *verbose* (por defecto) o *pipe*. La opción `-o`, que acepta un argumento obligatorio, permite seleccionar un fichero para la salida del programa alternativo a la salida estándar.

Uno de los principales objetivos de este ejercicio es que el estudiante se familiarice con tres funciones muy útiles empleadas por el programa `show-passwd.c`, y cuya página de manual debe consultarse:

- `int sscanf(const char *s, const char *format, ...);`

Variante de `scanf()` que permite leer con formato a partir de un buffer de caracteres pasado como primer parámetro (`s`). La función almacena en variables del programa, pasadas como argumento tras la cadena de formato, el resultado de convertir los distintos “tokens” de `s` de ASCII a binario.

- `char *strsep(char **stringp, const char *delim);`

Permite dividir una cadena de caracteres en *tokens*, proporcionando como segundo parámetro la cadena delimitadora de esos tokens. Como se puede observar en el programa `show-passwd.c`, esta función se utiliza para extraer los distintos campos almacenados en cada línea del fichero `/etc/passwd`, que están separados por `:`. La función `strsep()` se usa típicamente en un bucle, que para tan pronto como el token devuelto es `NULL`. El primer argumento de la función es un puntero por referencia. Antes de comenzar el bucle, `*stringp` debe apuntar al comienzo de la cadena que deseamos procesar. Cuando `strsep()` retorna, `*stringp` apunta al resto de la cadena que queda por procesar.

Responda a las siguientes preguntas:

1. Para representar cada una de las entradas del fichero `/etc/passwd` se emplea el tipo de datos `passwd_entry_t` (estructura definida en `defs.h`). Nótese que muchos de los campos almacenan cadenas de caracteres definidas como arrays de caracteres de longitud máxima prefijada, o mediante el tipo de datos `char*`. La función `parse_passwd()`, definida en `show-passwd.c` es la encargada de inicializar los distintos campos de la estructura. ¿Cuál es el propósito de la función `clone_string()` que se usa para inicializar algunos de los citados campos tipo cadena? ¿Por qué no es posible en algunos casos simplemente copiar la cadena vía `strcpy()` o realizando una asignación `campo=cadena_existente;`? Justifique la respuesta.
2. La función `strsep()`, utilizada en `parse_passwd()`, modifica la cadena que se desea dividir en tokens. ¿Qué tipo de modificaciones sufre la cadena (variable `line`) tras invocaciones sucesivas de `strsep()`? **Pista:** Consúltase el valor y las direcciones de las variables del programa usando el depurador.

Realice las siguientes modificaciones en el programa `show-passwd.c`:

- Consulte la página de manual de la función `strdup` de la biblioteca estándar de C. Intente utilizar esta función como reemplazo de `clone_string()`.
- Añada la opción `-i <inputfile>` para especificar una ruta alternativa para el fichero `passwd`. Hacer una copia de `/etc/passwd` en otra ubicación para verificar el correcto funcionamiento de esta nueva opción.
- Implemente una nueva opción `-c` en el programa, que permita mostrar los campos en cada entrada de `passwd` como valores separados por comas (CSV) en lugar de por `:`.

## Ejercicio 5

En este ejercicio vamos a practicar la programación en `bash` que haga uso de la orden interna `read` (consulta `help read`) para procesar ficheros línea a línea:

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [nam
```

Este comando lee una línea de la entrada estándar, la descompone en palabras, y asigna la primera palabra a la primera variable de la lista de nombres, la segunda a la segunda variable y así sucesivamente.

Si queremos usar un delimitador especial para separar palabras podemos hacerlo asignando valor a la variable `IFS` antes de usar la operación `read`. Por ejemplo, para leer palabras separadas por `:` usaríamos la forma:

```
while IFS=':' read var1 var2 ... ;
do
    # cualquier cosa con $var1, $var2
done
```

Y si no queremos leer de la entrada estándar, podemos redirigir la entrada de todo el bucle a un fichero:

```
while IFS=':' read var1 var2 ... ;
do
    # cualquier cosa con $var1, $var2
done < fichero
```



Utilizar `read` para crear un pequeño script que haga lo mismo que el programa anterior `show-passwd` (con sus opciones por defecto), es decir:

- lea el fichero `/etc/passwd`
- parsee sus entradas formadas por líneas con palabras separadas por `:`
- muestre cada entrada por la salida estándar con el mismo formato que el programa `show-passwd`.

Para obtener salida con formato en `bash` consultar la opción `-e` de `echo` (`man echo`). Alternativamente puede usarse la utilidad `printf` (`man 1 printf`).

Una vez hecho esto, modificar el script para que sólo se muestren aquellas entradas del fichero `/etc/passwd` en las que el `home` del usuario sea un subdirectorio de `/home`. Para ello resultará muy útil el uso del comando `dirname` (`man dirname`) y el uso de la estructura de control de flujo `if` junto con el programa `test` o el programa `[`.

Finalmente, intenta obtener una orden `bash`, combinando los comandos `cut` y `grep`, que permita obtener del fichero `/etc/passwd` todos los `homes` que empiecen por `/home`. Consulta las páginas de manual de `cut` y `grep` y revisa el uso de pipes (`|`) para combinar comandos del shell.