# Ejercicio 1:

## 1. Compilación:

**Compila el código del ejercicio y ejecútalo**
Compilado con `gcc -g -Wall hello2.c -o hello2` y ejecutado con `./hello2`
`-Wall`: Incluye todas las advertencias
`-g`: Generación de información de depuración

**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**
Usando la opción `-E` muestra el contenido de la salida de pre-procesado y `--save-temps` obtiene la salida en ficheros de todas las salidas intermedias
Ejecutado `gcc --save-temps hello2.c -o hello2`

**¿Qué ha ocurrido con la "llamada a `min()`" en *hello2.i*?**
Se ha sustituido en el código con el contenido de la función `a = ( (a<b)?a:b );`

**¿Qué efecto ha tenido la directiva `#include <stdio.h>`?**
Ha incluido todo el código (definiciones de funciones, variables...) en el archivo hello2.i

## 2. Herramienta make

**Examina el makefile, identifica las variables definidas, los objetivos (*targets*) y las regalas.**
Variables definidas:
`CC = gcc`: Define el compilador como GCC
`CFLAGS = -Wall -g`: Opciones de compilación
`LIBS = -lm`: Define la variable `LIBS` con la opción `-lm`, que indica la inclusión de la biblioteca de matemáticas al enlazar

Reglas:
`%.o %.c`: Regla de compilación que indica cómo generar archivos objeto (`*.o`) a partir de archivos fuente C (`*.c`). Utiliza el compilador definido por `CC`y las opciones de compilación definidas por `CFLAGS`
`all : aux.o init.o aux.h`: Regla principal que indica que el objetivo principal es construir el ejecutable llamado "prueba" usando los archivo objeto `aux.o` y `init.o`, así como el encabezado `aux.h`
`gcc $(CFLAGS) -o prueba aux.o init.o $(LIBS)`: Comando para vincular los archivos objeto en el ejecutable "prueba". Las opciones de compilación están definidas por `CFLAGS`, y se incluye la biblioteca de matemáticas especificada por `LIBS`.

Otros:
`.PHONY: clean`: Indica que "clean" no es un archivo real, sino un objetivo ficticio. Esto evita conflictos si hay un archivo llamado "clean" en el directorio.
`clean`: Regla para limpiar los archivos objeto (`*.o`) y el ejecutable ("prueba"). La opción `-` antes de `rm` suprime los mensajes de error si los archivos no existen.

**Ejecuta `make` en la linea de comandos y comprueba las ordenes que ejecuta para construir el proyecto**
Ejecutado `make`

**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?**
Si modificas `aux.c` y ejecutas `make` se volverá a compilar el archivo `aux.c`

**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?**
Tras ejecutar `make clean` se borran todos los archivos objeto (terminados en `.o`) y el ejecutable `prueba`
La directiva `.PHONY: clean` indica que clean no es un archivo, para evitar conflictos y que la tarea se ejecute independientemente de que exista un archivo llamado `clean`
Si comentamos la directiva `.PHONY: clean` y creamos un archivo llamado `clean`, la ejecución de `make clean`no realizara la limpieza correctamente

**Comenta la línea `LIBS = -lm` poniendo delante una almohadilla (#). Vuelve a construir el proyecto ejecutando `make` (haz un *clean* antes si es necesario). ¿Qué sucede? ¿Qué etapa es la que da problemas?**
Al comentar `LIBS = -lm`, se excluye la opción que vincula la biblioteca matemática lo que provoca errores en la etapa de vinculación (linking).

# 3. Tamaño de variables

## main1.c

**¿Por qué el primer `printf()` imprime valores distintos para 'a' con los modificadores `%d` y `%c`?**
El modificador `%d`indica que es un número entero, por lo cual mostrara el número entero contenido en la variable.
El modificador `%c` indica que es un carácter, por lo cual interpretara el valor como un carácter ASCII y mostrara su representación.
**¿Cuánto ocupa un tipo de datos `char`?**
Un tipo `char` ocupa 1 byte = 8 bits
**¿Por qué el valor de 'a' cambia tanto al incrementarlo en 6? (la respuesta está relacionada con la cuestión anterior)**
El tipo de dato `char` ocupa 1 byte, por lo tanto podrá contener los valores [-128, 127]. Al sumarle 6 se produce un desbordamiento, ya que debería dar 128 (valor que no se encuentra en el intervalo)

**Si un "long" y un "double" ocupan lo mismo, ¿por qué hay 2 tipos de datos diferentes?**
`long`: Es  representa números enteros con 8 bytes
`double`: Es una representa números reales (con decimales) con 8 bytes

## main2.c

**¿Tenemos un problema de compilación o de ejecución?**
El error es de compilación

**¿Por qué se da el problema?. Soluciónalo, compila y ejecuta de nuevo.**
El error se produce en la linea 13 con la instrucción:`int array2[a];`
Esto se debe a que el tamaño del array debe ser asignado por una constante. Se puede solucionar intercambiando por: `array2[7]`

**a,b,c, y x están declaradas de forma consecutiva. ¿Son sus direcciones consecutivas?**
Las direcciones no son consecutivas

**¿Qué significa el modificar `"%lu"` en `printf()`?**
Indica que tiene que imprimir un valor del tipo `unsigned long`

**¿A qué dirección apunta `"pc"`? ¿Coincide con la de alguna variable anteriormente declarada? Si es así, ¿Coinciden los tamaños de ambas?**
`pc` apunta a la dirección de `x`. No coinciden los tamaños, porque `x`es una variable de tipo `char` que ocupa un byte, 8 que `pc` es un puntero que ocupa 8 bytes

**¿Coincide el valor del tamaño de `array1` con el número de elementos del array? ¿Por qué?**
No coincide con el número de elementos, pero si con la suma del tamaño de todos los elementos del array.
4(Tamaño int) ×10 (Número elementos)=40×(Tamaño array)

**¿Coinciden las direcciones a la que apunta `str1` con la de `str2`?**
No, no coinciden las direcciones.

**¿Por qué los tamaños (según `sizeof()`) de `str1` y `str2` son diferentes?**
`str1` esta definido como un puntero, por lo tanto su tamaño viene determinado por el tamaño del puntero a `char`(8 bytes)
`str2` esta definido como un array de carácteres, por lo tanto su tamaño viene determinado por el tamaño de la longitud de la cadena (incluyendo el carácter nulo al final de la cadena)

# 4. Arrays

## array1.c

**¿Por qué no es necesario escribir `"&list"` para obtener la dirección del array `list`?**
No es necesario, porque `list`contiene la dirección base del array (primer elemento)

**¿Qué hay almacenado en la dirección de `list`?**
Se encuentra el primer elemento del array.

**¿Por qué es necesario pasar como argumento el tamaño del array en la función `init_array`?**
Porque en C al pasar un array en una función no se pasa el tamaño del mismo. Esto se debe que al pasar un array por una función como parámetro se degrada como un puntero y se pierde información del array.

**¿Por qué el tamaño devuelto por `sizeof()` para el array de la función `init_array` no coincide con el declarado en `main()`?**
En la función `main()`, `sizeof()` refleja el tamaño que ocupan los elementos del array en memoria, mientras que en `init_array` refleja el tamaño del puntero que apunta a la primera dirección del array. Esto se debe a lo mencionado en el punto anterior.

**¿Por qué NO es necesario pasar como argumento el tamaño del array en la función `init_array2`?**
Esto se debe a que en `init_array2`, se inicializa (PREGUNTAR DUDA)

**¿Coincide el tamaño devuelto por `sizeof()` para el array de la función `init_array2` con el declarado en `main()`?**
No, el tamaño devuelto en `sizeof()` coincide con el tamaño del puntero.

## array2.c

**¿La copia del array se realiza correctamente? ¿Por qué?**
No, porque simplemente se está asignando el puntero `src` a `dst` cuando lo que queremos hacer es copiar el contenido de un array a otro. Además al salir de la función se pierde esa asignación.

**Si no es correcto, escribe un código que sí realice la copia correctamente.**

```
void copyArray(int src[],int dst[],int size)
{
        for (int i = 0; i < size; i++) {
                dst[i] = src[i];
        }
}
```

**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?**
Se produce un error de ejecución llamado segmentation fault, debido a que se intenta acceder a una dirección de memoria que no esta asignada al proceso

**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?**
Con un valor superior al del tamaño del array, puede ocurrir que aunque estemos modificando una dirección que no esta asignada al array, esta si este asignada al proceso, por lo tanto no se produce segmentation fault.

# 5. Punteros

## punteros1.c

**¿Qué operador utilizamos para declarar una variable como un puntero a otro tipo?**
Utilizamos el operador `*`

**¿Qué operador utilizamos para obtener la dirección de una variable?**
Utilizamos el operador `&`

**¿Qué operador se utiliza para acceder al contenido de la dirección "a la que apunta" un puntero?**
Utilizamos el operador `*`

**Hay un error en el código. ¿Se produce en compilación o en ejecución? ¿Por qué se produce?**
Se produce un error de ejecución, en concreto un segmentation fault, debido a que se asigna una dirección al puntero `ptr = (int*) 0x600a48;` y esta dirección de memoria no entra en el rango de direcciones del proceso

## punteros2.c

**¿Cuántos bytes se reservan en memoria con la llamada a `malloc()`?**
Memoria reservada = nelem × sizeof(int)
Memoria reservada=127×4
Memoria reservada=508 bytes

**¿Cuál es la dirección del primer y último byte de dicha zona reservada?**
La dirección varía en la ejecución debido a que esta se asigna en tiempo de ejecución. Pero la dirección de memoria del primer byte se puede obtener de la dirección a la que apunta el puntero y la del último sumándole 508 bytes.

**¿Por qué el contenido de la dirección apuntada por `ptr` es 7 y no 5 en el primer `printf()`?**
Esto se debe a que la instrucción `ptr[0]` y `*ptr` acceden a la misma dirección de memoria.

**¿Por qué se modifica el contenido de `ptr[1]` tras la sentencia `*ptr2=15;`?**
Tras la instrucción `ptr2 = ptr`, `ptr2` apuntara a la dirección a la que apunta `ptr`. Y tras la instrucción `ptr2++`, la dirección a la que apunta `ptr2` pasara a apuntar a la siguiente dirección que es la misma que `ptr[1]`

**Indica dos modos diferentes de escribir el valor 13 en la dirección correspondiente a `ptr[100]`**

```
//Primer modo
ptr[100] = 13;

// Segundo modo
*(ptr + 100) = 13
```

**Hay un error en el código. ¿Se manifiesta en compilación o en ejecución? Aunque no se manifieste, el error está. ¿Cuál es?**
El error se produce en ejecución, pero no se manifiesta. Después de liberar la memoria con `free(ptr)` se modifica el valor almacenado en la dirección a la que apunta `ptr`, lo cual es un error, ya que esta dirección ya no esta reservada con `malloc()`

## punteros3.c

**¿Por qué cambia el valor de `ptr[13]` tras la asignación `ptr = &c;`?**
Porque con la asignación `ptr = &c;` se ha modificado la dirección a la que apunta `ptr`, por lo tanto `&ptr[13]` es distinta.

**El código tiene (al menos) un error. ¿Se manifiesta en compilación o en ejecución? ¿Por qué?**
Tras la asignación `ptr = &c` hemos modificado el puntero que apuntaba a una dirección de memoria reservada a que apunte a una variable `c`.
Un error (que no se manifiesta) se produce al acceder a `ptr[13]` tras el cambio de dirección, ya que estamos accediendo a una dirección que no se encuentra reservada.
Y el otro error se produce con la instrucción `free(ptr)`, esta se manifiesta en ejecución y se debe a que se intenta liberar una dirección de memoria que ya no apunta a un bloque asignado dinámicamente.

**¿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?**
Después de la asignación ya no se puede acceder a la dirección reservada por malloc, ni liberar dicha zona. Por eso es importante liberar las zonas de memorias asignadas dinámicamente una vez ya no se necesiten y tener cuidado asignar nuevas direcciones a punteros que tienen una zona reservada.

# Funciones

## arg1.c

**¿Por qué el valor de `xc` no se modifica tras la llamada a `sumC`? ¿Dónde se modifica esa información?**
No se modifica porque `sumC` trabaja con una copia de la estructura `struct _complex_ xc`. Esa modificación solo afecta dentro de la función `sumC`.

**Comenta las dos declaraciones adelantadas de `sum()` y `sumC()`. Compila de nuevo, ¿Qué ocurre?**
Se produce un error de compilación, porque se realizan llamadas a `sumC`y `sum` antes de que hayan sido declaradas.

## arg2.c

**¿Por qué cambia el valor de `y` tras la llamada a `sum()`?**
Cambia el valor de `y` porque en la función `sum()` los argumentos se pasan por referencia gracias al uso del operador `*`. Por lo que la función **no** trabaja sobre una copia.

**¿Por qué en ocasiones se usa el operador '.' y en otras '->' para acceder a los campos de una estructura?**
El operador `.` se utiliza cuando se accede a los campos de una estructura a través de una variable directa. Mientras que el operador `->`se utiliza cuando se accede a los campos de na estructura a través de un puntero a esa estructura.

**¿Por qué el valor de `zc` pasa a ser incorrecto sin volver a usarlo en el código?**
Esto se debe a como se maneja la estructura de retorno. Dentro de `sumC()` se crea la estructura `_complex_ r`y se devuelve la dirección de `r` el problema con esto es que `r` se crea en la pila de función y se libera cuando la función termina. Por lo tanto, cuando se devuelve la dirección no apunta a nada.

**Corrige el código para evitar el error producido en `zc`**
Se puede solucionar el error creando un puntero `r` y asignarle memoria dinámicamente.

```
struct _complex_ * sumC( struct _complex_* a,  struct _complex_* b)
{
        struct _complex_ *r = malloc(sizeof(struct _complex_));
        r->re = a->re + b->re;
        r->im = a->im + b->im;

        // We modify the first argument
        a->re = 12.5;
        a->im = 13.4;
        return r;
}
```

# Cadenas de caracteres (strings)

## 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 la letra `'j'`?**
La dirección de 'B' es `p`
**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.**
Porque el último carácter (final de string `\0` ) no cuenta a la hora de determinar la longitud.

**¿Por qué `strlen()` devuelve un valor diferente a `sizeof()`?**
Porque `strlen()` devuelve la longitud de la cadena
Mientras que `sizeof()` devuelve el espacio en memoria que esta a asignada al puntero o array.

## strings2.c

**El código de `copy` no funciona. ¿Por qué?**
El problema radica en como se pasan los parámetros a la función. Se esta modificando localmente el puntero `dst`, pero no esta afectando al puntero original

**Usa ahora la función `copy2()` (descomenta la línea correspondiente). ¿Funciona la copia?**
El propósito de la función `copy2()` no es copiar el contenido, sino asignar el puntero `orig` al puntero `dst`. Aunque los punteros apuntan al mismo lugar no copia el contenido en si.

**Propón una implementación correcta de la copia**
Implementación propuesta en la función `copy3()`

```
void copy3(char* org, char* dst) {
            // Reserva memoria (podría funcionar sin el malloc)
        dst = malloc(strlen(org) + 1);
        for (int i = 0; i < strlen(org) + 1; i++) {
                dst[i] = org[i];
        }
}
```

**¿Qué hace la función `mod()`? ¿Por qué funciona?**
La función `mod()` copia el contenido de un string a otro, pero pasando de minúsculas a mayúsculas. Esto se debe al `-32`y la representación ASCII.

**Descomenta la última llamada a la función `mod()`. Compila y ejecuta. ¿Por qué se produce el error?**
En este caso, es probable que el error se deba a intentar modificar una cadena literal, que está almacenada en una región de memoria de solo lectura.

# Ejercicio 2

Compilación con: `gcc -g -w -o primes primes.c`
Depurado con `gdb -tui primes`
Código tras modificaciones:

```
/**
* 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 = 0;
	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 (x == 2) {
			result[i] = x;
			i++;
			x++;
		}
		else {
			if (is_prime(x)) {
				result[i] = x;
				i++;
			}
			x += 2;
		}
	}
	return;
}

int is_prime(int x) {
	if (x == 2){
		return 1;
	}
	else if (x % 2 == 0) {
```

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

# Ejercicio 3

## Código

Contenido fichero `getopt.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "defs.h"

char *progname;

/**
 * @brief Print usage of the tool
 */
void usage() {
    printf("Usage: %s [ options ] title \n", progname);
    printf("\n\toptions:\n");
    printf("\t\t-h: display this help message\n");
    printf("\t\t-e: print even numbers instead of odd (default)\n");
    printf("\t\t-l length: length of the sequence to be printed\n");
    printf("\t\ttitle: name of the sequence to be printed\n");
}

/**
 * @brief Prints a sequence of odd or even numbers in stdout
 * @param length
 * @param type
 * @param title
 */
void display_numbers(int length, parity_t type, char *title) {
    int i, n;
    int first = (type == ODD) ? 1 : 2;

    printf("Title: %s\n", title);

    for (i = 0, n = first; i < length; i++, n += 2) {
        printf("%d ", n);
    }

    printf("\n");
}
```

```
int main(int argc, char *argv[]) {
    int opt;
    struct options options;

    progname = argv[0];

    /* Initialize default values for options */
    options.par_mode = ODD;
    options.length = 10;
    options.title = NULL;

    /* Parse command-line options */
    while ((opt = getopt(argc, argv, "hel:")) != -1) {
        switch (opt) {
            case 'h':
                usage();
                exit(0);
            case 'e':
                options.par_mode = EVEN;
                break;
            case 'l':
                options.length = atoi(optarg);
                break;
            default:
                exit(EXIT_FAILURE);
        }
    }

    /* There should be one extra argument (the title of the sequence) */
    if (optind >= argc) {
        fprintf(stderr, "Invalid title\n");
        usage();
        exit(EXIT_FAILURE);
    }

    /* Fill options.title with the corresponding element of argv */
    options.title = argv[optind];

    /* Call display_numbers */
    display_numbers(options.length, options.par_mode, options.title);

    return 0;
}
```

Los demás archivos no se modifican por lo tanto no los incluyo

## Preguntas

**1. ¿Qué cadena de caracteres debes utilizar como tercer argumento de `getopt()`?**
La cadena de caracteres del tercer argumento de `getopt` llamada `optstring` especifica las opciones que el programa acepta, si al carácter de la opción le sigue el carácter `:` especifica que esa opción requiere argumento.
**2. ¿Qué línea de código utilizas para leer el argumento `title`?**
`options.title = argv[optind];`

# Ejercicio 4

Responde 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.*

Si se hiciese uso de la asignación `campo=cadena_existente` se le asignaría a `campo` el puntero de `cadena_existente`, mientras que lo que queremos es que se copie el contenido de `cadena_existente` a `campo`. Y `strcpy()` acepta dos parámetros `origen`y `destino` y simplemente copia el contenido, pero necesitamos asignar memoria dinámica a la cadena. Por ello se hace uso de `clone_string()` este crea un puntero, le asigna memoria y copia el contenido de la cadena a donde apunta ese puntero para luego devolver el puntero.

**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: Consúltese el valor y las direcciones de las variables del programa usando el depurador.**

***##################TODO#########################***

# Contenido final defs.h:

```
#ifndef DEFS_H
#define DEFS_H

#define MAX_LOGIN_NAME  50
#define MAX_PASSWD_LINE  255

/**
 * Structure that describes an entry in /etc/passwd
 * Check "man 5 passwd" for details
 */
typedef struct passwd_entry {
      char login_name[MAX_LOGIN_NAME];
      char* optional_encrypted_passwd;
      int uid;
      int gid;
      char* user_name;
      char* user_home;
      char* user_shell;
} passwd_entry_t;

/**
 * Index of the various fields in passwd_entry_t
 * Helper data type to simplify the parser's implementation.
 */
typedef enum {
      LOGIN_NAME_IDX=0,
      ENCRYPTED_PASS_IDX,
```

```
	UID_IDX,
	GID_IDX,
	USER_NAME_IDX,
	USER_HOME_IDX,
	USER_SHELL_IDX,
	NR_FIELDS_PASSWD
} token_id_t;

/**
 * Output modes supported by the program
 */
typedef enum {
	VERBOSE_MODE,
	PIPE_MODE,
	COMMA_MODE
} output_mode_t;

/**
 * Structure to hold the "variables" associated with
 * command-line options
 */
struct options {
	FILE* outfile;
	output_mode_t output_mode;
	char* infile;
};

#endif
```

## Contenido final show-passwd.c

```
#include <stdio.h>
#include <unistd.h> /* for getopt() */
#include <stdlib.h> /* for EXIT_SUCCESS, EXIT_FAILURE */
#include <string.h>
#include "defs.h"

/**
 * @brief  Make a copy of existing string allocating memory accordingly
 *
 * @param original
 * @return new string that is a clone of original
 **/
static inline char* clone_string(char* original)
{
	char* copy;
	copy=malloc(strlen(original)+1);
	strcpy(copy,original);
	return copy;
}


/**
 * @brief Parse /etc/passwd file and returns its contents as an array of entries.
 * The number of entries in the array is returned vie the nr_entries parameter.
```

```
 *
 * @param options
 * @param nr_entries
 * @return array of entries (passwd_entry_t*) and entry count (nr_entries parameter)
 */
passwd_entry_t* parse_passwd(struct options* options, int* nr_entries)
{
	FILE* passwd;
	char line[MAX_PASSWD_LINE+1];
	passwd_entry_t* entries;
	passwd_entry_t* cur_entry;
	int line_count=0;
	char* token;
	char* lineptr;
	token_id_t token_id;
	int entry_idx;
	int entry_count=0;
	int cur_line;

	if ((passwd=fopen(options->infile,"r"))==NULL) {
		fprintf(stderr, "%s could not be opened: ", options->infile);
		perror(NULL);
		return NULL;
	}

	/* Figure out number of lines */
	while (fgets(line, MAX_PASSWD_LINE + 1, passwd) != NULL){
		line_count++;
		/* Discard lines that begin with # */
		if (line[0]!='#')
			entry_count++;
	}

	/* Rewind position indicator*/
	fseek(passwd,0,SEEK_SET);

	entries=malloc(sizeof(passwd_entry_t)*entry_count);
	/* zero fill the array of structures */
	memset(entries,0,sizeof(passwd_entry_t)*entry_count);

	/* Parse file */
	entry_idx=0;
	cur_line=1;
	while (fgets(line, MAX_PASSWD_LINE + 1, passwd) != NULL) {

		/* Discard lines that begin with # */
		if (line[0]=='#'){
			cur_line++;
			continue;
		}

		/* Point to the beginning of the line */
		lineptr=line;
		token_id=LOGIN_NAME_IDX;
		cur_entry=&entries[entry_idx];
```

```
            while((token = strsep(&lineptr, ":"))!=NULL) {
                  switch(token_id) {
                  case LOGIN_NAME_IDX:
                        strcpy(cur_entry->login_name,token);
                        break;
                  case ENCRYPTED_PASS_IDX:
                        cur_entry->optional_encrypted_passwd=strdup(token);
                        break;
                  case UID_IDX:
                        if (sscanf(token,"%d",&cur_entry->uid)!=1) {
                              fprintf(stderr, "Warning: Wrong format for UID field
in line %d of /etc/passwd\n",cur_line);
                              cur_entry->uid=0;
                        }

                        break;
                  case GID_IDX:
                        if (sscanf(token,"%d",&cur_entry->gid)!=1) {
                              fprintf(stderr, "Warning: Wrong format for GID field
in line %d of /etc/passwd\n",cur_line);
                              cur_entry->gid=0;
                        }
                        break;
                  case USER_NAME_IDX:
                        cur_entry->user_name=strdup(token);
                        break;
                  case USER_HOME_IDX:
                        cur_entry->user_home=strdup(token);
                        break;
                  case USER_SHELL_IDX:
                        /* remove new line from token */
                        token[strlen(token)-1]='\0';
                        cur_entry->user_shell=strdup(token);
                        break;
                  default:
                        break;
                  }
                  token_id++;
            }
            if (token_id!=NR_FIELDS_PASSWD) {
                  fprintf(stderr, "Could not process all tokens from line %d of
/etc/passwd\n",entry_idx+1);
                  return NULL;
            }
            cur_line++;
            entry_idx++;
      }
      (*nr_entries)=entry_count;
      return entries;
}

/**
 * @brief Free up the array of entries, including the
 * memory of dynamically allocated strings
```

```
 *
 * @param entries
 * @param nr_entries
 */
static void free_entries(passwd_entry_t* entries, int nr_entries)
{
	int i=0;
	passwd_entry_t* entry;

	for (i=0; i<nr_entries; i++) {
		entry=&entries[i]; /* Point to current entry */
		free(entry->optional_encrypted_passwd);
		free(entry->user_name);
		free(entry->user_home);
		free(entry->user_shell);
	}

	free(entries);
}

/**
 * @brief This function takes care of invoking the parser,
 * and then displays the contents of /etc/passwd based on the
 * selected output mode
 *
 * @param options
 * @return int
 */
static int show_passwd(struct options* options)
{
	int nr_entries;
	int i;
	passwd_entry_t* entries=parse_passwd(options,&nr_entries);

	if (!entries)
		return EXIT_FAILURE;

	for (i=0; i<nr_entries; i++) {
		passwd_entry_t* e=&entries[i]; /* Point to current entry */
		switch (options->output_mode) {
		case VERBOSE_MODE:
			fprintf(options->outfile,"[Entry #%d]\n",i);
			fprintf(options->outfile,"\tlogin=%s\n\tenc_pass=%s\n\t"
				"uid=%d\n\tgid=%d\n\tuser_name=%s\n\t"
				"home=%s\n\tshell=%s\n",
				e->login_name, e->optional_encrypted_passwd,
				e->uid, e->gid,e->user_name,
				e->user_home, e->user_shell);
			break;
		case PIPE_MODE:
			fprintf(options->outfile,"|%s|%s|%d|%d|%s|%s|%s|\n",
				e->login_name, e->optional_encrypted_passwd,
				e->uid, e->gid,e->user_name,
				e->user_home, e->user_shell);
			break;
```

```
              case COMMA_MODE:
                    fprintf(options->outfile,"%s,%s,%d,%d,%s,%s,%s\n",
                           e->login_name, e->optional_encrypted_passwd,
                           e->uid, e->gid,e->user_name,
                           e->user_home, e->user_shell);
              }

       }
       free_entries(entries,nr_entries);
       return EXIT_SUCCESS;
}

int main(int argc, char *argv[])
{
       int retCode, opt;
       struct options options;

       /* Initialize default values for options */
       options.outfile=stdout;
       options.output_mode=VERBOSE_MODE;
       options.infile="/etc/passwd";

       /* Parse command-line options */
       while((opt = getopt(argc, argv, "hvpo:i:c")) != -1) {
              switch(opt) {
              case 'h':
                    fprintf(stderr,"Usage: %s [ -h | -v | -p | -o <output_file>
]\n",argv[0]);
                    exit(0);
              case 'v':
                    options.output_mode=VERBOSE_MODE;
                    break;
              case 'p':
                    options.output_mode=PIPE_MODE;
                    break;
              case 'o':
                    if ((options.outfile=fopen(optarg,"wx"))==NULL) {
                          fprintf(stderr, "The output file %s could not be opened: ",
                                 optarg);
                          perror(NULL);
                          exit(EXIT_FAILURE);
                    }
                    break;
              case 'i':
                    options.infile=strdup(optarg);
                    break;
              case 'c':
                    options.output_mode=COMMA_MODE;
                    break;
              default:
                    exit(EXIT_FAILURE);
              }
       }

       retCode=show_passwd(&options);
```

```
	exit(retCode);
}
```

# Ejercicio 5

## Script bash:

```
#!/bin/bash

passwd_file='/etc/passwd'
n_entry=0;

while IFS=':' read login_name_idx encypted_pass_idx uid_idx gid_idx user_name_idx
user_home_idx user_shell_idx nr_fields_passwd;
do
	if [ $(dirname "$user_home_idx") == "/home" ]; then
		echo "Entry[#$n_entry]
		login=$login_name_idx
		enc_pass=$encypted_pass_idx
		uid=$uid_idx
		gid=$gid_idx
		user_name=$user_name_idx
		home=$user_home_idx
		shell=$user_shell_idx"
		((n_entry++))
	fi
done < $passwd_file;
```

## Orden bash:

```
cut -d ":" -f 6 /etc/passwd | grep /home
```