PRÁCTICA 2: Tipo Abstracto de Dato Pila

OBJETIVOS

- Aprender a utilizar el TAD Pila para resolver problemas.
- Aprender a elegir la estructura de datos apropiada para implementar el TAD Pila.
- Codificar sus primitivas y utilizarlo en un programa principal.
- Aprender a utilizar una librería estática en C.

NORMAS

Los programas que se entreguen deben:

- Estar escritos en C, siguiendo las normas de programación establecidas.
- Compilar sin errores ni warnings incluyendo las banderas `-Wall` y `-pedantic` al compilar.
- Ejecutarse sin problema en una consola de comandos.
- Incorporar un adecuado control de errores. Es justificable que un programa no admita valores inadecuados, pero no que se comporte de forma anómala con dichos valores.
- No producir fugas de memoria al ejecutarse.

PLAN DE TRABAJO

- **Semana 1:** Ejercicio 1. Prueba de la biblioteca de pila e implementación y uso del algoritmo de ordenación de pila.
- **Semana 2:** Ejercicio 2. Algoritmo de recorrido del laberinto en profundidad.
- **Semana 3:** Ejercicio 3. Implementación y prueba de una biblioteca para el TAD Pila.

Cada profesor indicará en clase cómo realizar las **entregas parciales semanales**: papel, e-mail, Moodle, etc.

La entrega final se realizará a través de Moodle, siguiendo escrupulosamente las instrucciones indicadas en el enunciado referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido que se debe entregar debe llamarse `Px_Prog2_Gy_Pz`, siendo `x` el número de la práctica, `y` el grupo de prácticas y `z` el número de pareja (ejemplo de entrega de la pareja 5 del grupo 2161: `P2_Prog2_G2161_P05.zip`).

El fichero comprimido debe contener la siguiente organización de ficheros:

```
--- P2_Prog2_Gy_Pz/
 |--- point.c
 |--- point.h
 |--- map.c
 |--- map.h
 |--- p2_e1a.c
 |--- p2_e1b.c
 |--- p2_e1c.c
 |--- p2_e2.c
 |--- p2_e3.c
 |--- Makefile
 |--- libstack_fDoble.a
 |--- stack_fDoble.c
 |--- stack_fDoble.h
 |--- laberinto_1.txt
 |--- laberinto_2.txt
```

Las fechas de subida a Moodle del fichero son las siguientes:

- Los alumnos de Evaluación Continua, la semana del **21 de marzo** (cada grupo puede realizar la entrega hasta las **23:55 h**. de la noche anterior a su clase de prácticas).
- Los alumnos de Evaluación Final, según lo especificado en la normativa.

Ejercicio 1.

El objetivo de este ejercicio consiste en implementar un algoritmo para ordenar los elementos de una pila.

Algoritmo

El algoritmo recibe una pila y devuelve otra pila con los elementos originales ordenados de menor (bottom) a mayor (top). La pila original se vacía. En el pseudo-código no se ha incluido el oportuno control de errores.

Algorithm 1: Order stack, no error control

```
   input : Stack s
   output: Stack s_o
1  s_o = stack_init ()
2  while stack_isEmpty (s) ≡ False do
3      e = stack_pop (s)
4      while stack_isEmpty (s_o) ≡ False & e < stack_top (s_o) do
5          e_a = stack_pop (s_o)
6          stack_push (s, e_a)
7      end
8      stack_push (s_0, e)
9  end
10 return s_o
```

En el primer apartado del ejercicio, el algoritmo anterior se utilizará para ordenar una pila de puntos. En el segundo apartado se ordenará una pila de enteros.

Recursos adicionales

Debes hacer uso de la librería del TAD Stack `libstack_fDoble.a` y la interfaz pública definida en el archivo de cabecera `stack_fDoble.h` (puedes ver el contenido de este fichero en el apéndice 1.1 de este documento)

Ejercicio 1a.

Funciones adicionales de punto:

Añade a la interfaz del TAD Point las funciones `point_euDistance()` que calcula la distancia euclidea entre dos puntos cualesquiera y `point_cmpEuDistance()` que compara dos puntos utilizando sus distancias euclideas al origen de coordenadas $(0, 0)$.

```
/**
 * @brief Calculate the euclidean distance betweeen two points.
 *
 * The euclidean distance is defined as sqrt ((x1-x2)^2 + (y1-y2)^2)
 * where (x1, y1) and (x2, y2) are the coordinate of both points
 *
 * @code
 * // Example of use
 * const Point *p1, *p2;
 * double d;
 * p1 = point_new (x1, y1, BARRIER);
 * p2 = point_new (x2, y2, SPACE);
 * point_euDistance (p1, p2, &d);
 * printf ("%lf", d);
 * // .... additional code ....
 * @endcode
 *
 * @param p1 pointer to point
 * @param p2 pointer to point
 * @param distance addresss
 *
 * @return Returns OK or ERROR in case of invalid parameters
 */
Status point_euDistance (const Point *p1, const Point *p2, double *
   distance);

/**
 * @brief Compares two points using their euclidean distances to the
    point (0,0).
 *
 *
 * @param p1,p2 Points to compare.
 *
 * @return It returns an integer less than, equal to, or greater than
     zero if
 * the euclidean distance of p1 to the origin of coordinates is found
    ,
 * respectively, to be less than, to match or be greater
 * than the euclidean distance of p2. In case of error, returns
    INT_MIN.
 */
int point_cmpEuDistance (const void *p1, const void *p2);
```

Comprobación de la corrección de las funciones

Crea un programa **p2_e1a.c** que realice las siguientes acciones:

- Generar n puntos de coordenadas aleatorias entre 0 y 100 donde n es un parámetro de entrada del programa.
- Imprimir para cada uno de los puntos anteriores la distancia euclidea al origen de coordenadas.
- Comparar las distancias euclideas para todos los pares de puntos.

Ejemplo de salida:

El siguiente ejemplo corresponde a la ejecución del programa para 5 puntos cuando **no** se elige una semilla aleatoria (ver la siguiente ayuda)

```
$ ./p2_e1a 5
Point p[0]=[(6, 3): +] distance: 6.708204
Point p[1]=[(5, 7): +] distance: 8.602325
Point p[2]=[(5, 3): +] distance: 5.830952
Point p[3]=[(2, 6): +] distance: 6.324555
Point p[4]=[(1, 9): +] distance: 9.055385
p[0] <  p[0]: False
p[0] <  p[1]: True
p[0] <  p[2]: False
p[0] <  p[3]: False
p[0] <  p[4]: True
p[1] <  p[1]: False
p[1] <  p[2]: False
p[1] <  p[3]: False
p[1] <  p[4]: True
p[2] <  p[2]: False
p[2] <  p[3]: True
p[2] <  p[4]: True
p[3] <  p[3]: False
p[3] <  p[4]: True
p[4] <  p[4]: False
```

Ayuda: Como generar puntos con coordenads aleatorias

El siguiente extracto de código (**incompleto y sin control de errores**) genera `NPOINTS` puntos con coordenadas aleatorias entre `0` y `MAX_RAND` y calcula sus distancias al origen de coordenadas.

```
#include <math.h>
#include <time.h>
```

```c
#define NPOINTS 5
#define MAX_RAND 10

int main() {
    Point *p[NPOINTS] = {NULL};
    Point *origen;
    int i;
    double d;

    // set the random seed
    srand(time(NULL));

    // coordinate origin point
    origen = point_new (0, 0, BARRIER);

    for (i=0; i < NPOINTS; i++) {
        // point with random coordinates
        p[i] = point_new(rand() % MAX_RAND , rand() % MAX_RAND,
           BARRIER);
        // ........................................
        // ....must be included control error......
        //.........................................

        point_print (stdout, p[i]);
        point_euDistance (p[i], origen, &d);
        fprintf (stdout, "Euclidean distance: %lf", d);
    }
    // .....the code continue....
}
```

Sino quisieses obtener una salida aleatoria comenta la llamada a la función `srand()` en el código anterior.

Ejercicio 1b. Algoritmo de ordenación de una pila de puntos:

Codificación del algoritmo:

Codifique el algoritmo de ordenación de un pila mediante la función

```c
/**
 * @brief: Does an ordered stack with the greatest element at the top.
 *
 * The origenal stack will be emptied.
 *
 * @param sin, input stack
```

```
 *
 * @return The function returns an ordered stack or NULL otherwise
**/
Stack *stack_orderPoints (Stack *sin);
```

La función deberá incluir el oportuno control de errores.

Comprobación de la corrección de la función:

Crea un programa **p2_e1b.c** que utilice la función `stack_orderPoints()`. El programa deberá:

- Generar n puntos de coordenadas aleatorias entre 0 y 10 donde n es un parámetro de entrada del programa.
- Calcular la distancia euclidea de los puntos al origen de coordenadas
- Almacenar los puntos en una pila.
- Imprimir la pila con los puntos.
- Ordenar los elementos de la pila de menor a mayor según la distancia euclidea de los puntos al origen de coordenadas.
- Imprimir la pila original (vacía) y la pila ordenada.

El programa deberá gestionar correctamente la memoria. Utiliza `valgrind` para verificar que el programa no tiene fugas de memoria.

El programa utilizará la biblioteca estática `libstack_fDoble.a` para gestionar la pila. En el anexo 1.3 puedes encontrar cómo crear y usar librerías estáticas en tu proyecto.

El programa deberá usar llamadas a `stack_print` para mostrar el contenido de las pilas. Puedes encontrar un anexo sobre el uso de esta función en el anexo 1.4.

Ejemplo de salida:

El siguiente ejemplo corresponde a la ejecución del programa para 5 puntos cuando **no** se elige una semilla aleatoria (ver la ayuda anterior)

```
$ ./p2_e1b 5
Point p[0]=[(6, 3): +] distance: 6.708204
Point p[1]=[(5, 7): +] distance: 8.602325
Point p[2]=[(5, 3): +] distance: 5.830952
Point p[3]=[(2, 6): +] distance: 6.324555
Point p[4]=[(1, 9): +] distance: 9.055385
Original stack:
SIZE: 5
[(1, 9): +]
```

```
[(2, 6): +]
[(5, 3): +]
[(5, 7): +]
[(6, 3): +]
Ordered stack:
SIZE: 5
[(1, 9): +]
[(5, 7): +]
[(6, 3): +]
[(2, 6): +]
[(5, 3): +]
Original stack:
SIZE: 0
```

Ejercicio 1c. Generalización de la función y aplicación a una pila de enteros

El ejercicio anterior presentaba una limitación: sólo funcionaba con pilas con elementos de tipo `Point`. Sin embargo, C es lo suficientemente flexible para poder hacer implementaciones más genéricas. Esto lo podremos usar gracias a los punteros a funciones.

Implemente una funcion `stack_order()` que ordene los elementos de una pila. La función deberá incluir como uno de sus parámetros de entrada un puntero a función con la referencia a la función de comparación de los elementos de la pila.

```
stack_order (Stack *s, int (*f_cmp)(const void *, const void *));
```

Comprueba la corrección de la función anterior substituyendo en el programa **p2_e1b.c** la llamada a la función `stack_orderPoints()` por `stack_order()`

Pila de enteros

Crea un programa **p2_e1c.c** que realice las siguientes acciones:

- Generara n enteros aleatorios en el itérvalo $[0, 10]$, siendo n un parámetro de entrada al programa
- Almacenar los enteros en una pila.
- Imprimir la pila.
- Ordenar los elementos de la pila de menor a mayor.
- Imprimir la pila original (vacía) y la pila con los enteros ordenada.

Para realizar la comparación de los elementos puedes utilizar la siguiente función de comparación para enteros:

```
int int_cmp(const void *c1, const void *c2) {
 if (!c1 || !c2)
   return INT_MIN;

 return (*(int *)c1 - *(int *)c2);
}
```

¿Sabrías diseñar, implementar y utilizar las funciones apropiadas para ordenar una pila de cadenas de caracteres?

Ejercicio 2.

En este ejercicio vamos a implementar el algoritmo de búsqueda en profundidad en un laberinto utilizando el TAD Pila.

Algorithm 2: DFS, without error control

```
   input  : Map: m
   output: Point: p
 1 p = NULL
 2 p_i = map_getInput (m)
 3 p_o = map_getOutput (m)
 4 s = stack_init ()
 5 stack_push (s, p_i)
 6 while p ≠ p_o & stack_isEmpty (s) ≡ False do
 7 |   p = stack_pop (s)
 8 |   if point_getVisited (p) ≡ False then
 9 |   |   point_setVisited (p, True)
10 |   |   forall p_n ∈ Neighboors (p) do
11 |   |   |   if point_getVisited (p_n) ≡ False then
12 |   |   |   |   stack_push (s, v)
13 |   |   |   end
14 |   |   end
15 |   end
16 end
17 stack_free (s)
18 if p ≡ p_o then
19 |   return p
20 end
21 return NULL
```

Para ello, vamos a utilizar la biblioteca del TAD Pila **libstack_fDoble.a** y los TAD Map y Point que has implementado en la práctica anterior.

Codificación del algoritmo

Codifica el algoritmo de búsqueda en profundidad mediante la función

```
/**
 * @brief: Makes a search from the origin point to the output point
 * of a map using the depth-first search algorithm and the ADT Stack
 *
 * The function prints each visited point while traversing the map
 *
```

```
 * @param mp, Pointer to map
 * @param pf, File descriptor
 *
 * @return The function returns the output map point o NULL otherwise
**/
Point * map_dfs (FILE *pf, Map *mp);
```

La función deberá incluir el oportuno control de errores.

Comprobación de la corrección de la función

Crea un programa **p2_e2.c** que utilice la función `map_dfs()` del TAD Map para recorrer un laberinto contenido en un fichero pasado como argumento al programa.

El programa utilizará la biblioteca estática `libstack_fDoble.a` para gestionar una pila de puntos. El programa deberá gestionar correctamente la memoria. Utilice `valgrind` para verificar que no tiene fugas de memoria

Ejemplo de salida:

En el siguiente ejemplo se muestra una ejecución del programa, en el que se recorre el mapa contenido en laberinto_1.txt:

```
$ ./p2_e2 laberinto_1.txt
Maze:
4, 5
[(0, 0): +][(1, 0): +][(2, 0): +][(3, 0): +][(4, 0): +][(0, 1): +]
[(1, 1): .][(2, 1): .][(3, 1): i][(4, 1): +][(0, 2): +][(1, 2): o]
[(2, 2): .][(3, 2): .][(4, 2): +][(0, 3): +][(1, 3): +][(2, 3): +]
[(3, 3): +][(4, 3): +]
DFS traverse:
[(3, 1): i][(3, 2): .][(2, 2): .][(1, 2): o]
```

Ejercicio 3.

En este ejercicio se desea implementar el TAD Stack como un array dinámico de elementos (genéricos), en un fichero `stack_fDoble.c`. La interfaz pública del TAD la puedes encontrar en el fichero `stack_fDoble.h` (anexo 1.1).

El tipo de dato a guardar será un array dinámico de punteros void, de forma que se puedan almacenar, gestionar e imprimir elementos de cualquier tipo

```
#define INIT_CAPACITY 2  // init stack capacity
#define FCT_CAPACITY 2   // multiply the stack capacity

struct _Stack {
    void  **item; /*!<Static array of elements*/
    int top; /*!<index of the top element in the stack*/
    int capacity; /*!<xcapacity of the stack*/
};
```

Una pila se inicializará con un tamaño `INIT_CAPACITY` e incrementará su tamaño según el factor `FCT_CAPACITY` cuando esté llena.

Debes implementar tanto la interfaz pública, como las funciones privadas que consideres necesarias.

Debes probar todos tus ejercicios con tu implementación del TAD Stack.

Ayuda:

La función `realloc` permite modificar el tamaño de un espacio de memoria reservado dinámicamente (con `malloc`, `calloc` o `realloc`). En este ejercicio se reservará memoria inicialmente para el número de elementos indicado por `INIT_CAPACITY` y su tamaño se irá incrementando cuando se quieran añadir nuevos elementos a una pila que ya esté llena. Para tener control de la capacidad se ha añadido el atributo `capacity` a la estructura `struct _Stack`. Esto se hace de esta forma para abordar uno de los problemas que surgen en la reserva de memoria dinámica: cuánto se debe incrementar la capacidad de un contenedor, de forma que no sobre mucho espacio pero tampoco se invoque a la función *realloc* continuamente (lo que sería poco eficiente).

APÉNDICES

Apéndice 1.1

Contenido del fichero `stack_fDoble.h`:

```
// [ALL]
/**
 * @file  stack_fDoble.h
 * @author Prog2
 * @date 8 Mar 2021
 * @version 1.0
 * @brief Stack library
 *
 * @details Stack interface
 *
 * @see http://www.stack.nl/~dimitri/doxygen/docblocks.html
 * @see http://www.stack.nl/~dimitri/doxygen/commands.html
 */


#ifndef STACK_FDOBLE_H
#define STACK_FDOBLE_H

#include "types.h"
#include <stdio.h>


/**
 * @brief Structure to implement a stack. To be defined in stack_fp.c
 *
 **/
typedef struct _Stack Stack;



/**
 * @brief Typedef for a function pointer to print a stack element at
    stream
 **/
typedef int (*P_stack_ele_print)(FILE *, const void*);


/**
 * @brief This function initializes an empty stack.
 *
```

```
 * @return   This function returns a pointer to the stack or a null
    pointer
 * if insufficient memory is available to create the stack.
 *  */
Stack * stack_init ();

/**
 * @brief  This function frees the memory used by the stack.
 * @param s A pointer to the stack
 *  */
void stack_free (Stack *s);

/**
 * @brief This function is used to insert a element at the top of the
     stack.
 *
 * A reference of the element is added to the stack container and the
     size of the stack is increased by 1.
 * Time complexity: O(1). This function reallocate the stack capacity
     when it is full.
 * @param s A pointer to the stack.
 * @param ele A pointer to the element to be inserted
 * @return This function returns OK on success or ERROR if the stack
    is full.
 *  */
Status stack_push (Stack *s, const void *ele);

/**
 * @brief  This function is used to extract a element from the top of
     the stack.
 *
 * The size of the stack is decreased by 1. Time complexity: O(1).
 * @param s A pointer to the stack.
 * @return This function returns a pointer to the extracted element
    on success
 * or null when the stack is empty.
 * */
void * stack_pop (Stack *s);


/**
 * @brief  This function is used to reference the top (or the newest)
     element of the stack.
 *
 * @param s A pointer to the stack.
 * @return This function returns a pointer to the newest element of
    the stack.
```

```
 * */
void * stack_top (const Stack *s);

/**
 * @brief Returns whether the stack is empty
 * @param s A pointer to the stack.
 * @return TRUE or FALSE
 */
Bool stack_isEmpty (const Stack *s);

/**
 * @brief This function returns the size of the stack.
 *
 * Time complexity: O(1).
 * @param s A pointer to the stack.
 * @return the size
 */
size_t stack_size (const Stack *s);

/**
 * @brief  This function writes the elements of the stack to the
    stream.
 * @param fp A pointer to the stream
 * @param s A pointer to the element to the stack
 * @return Upon successful return, these function returns the number
    of characters writted.
 * The function returns a negative value if there was a problem
    writing to the file.
 *  */
int stack_print(FILE* fp, const Stack *s,  P_stack_ele_print f);

#endif /* STACK_FDOBLE_H */
```

Apéndice 1.2.

Creación y uso de bibliotecas estáticas

Introducción Las bibliotecas son una forma sencilla y versátil de modularizar y reutilizar código. Una biblioteca es un archivo que contiene varios ficheros objetos (`.o`) y que, por tanto, puede ser usado como un único programa en la fase de enlazado con otros programas.

Los sistemas `UNIX` permiten crear dos tipos diferentes de bibliotecas, las bibliotecas estáticas y las bibliotecas dinámicas. En este documento se aborda el proceso de creación, compilación y uso de una biblioteca estática.

Creación de una biblioteca estática Una biblioteca estática consta de:

- Un conjunto de ficheros objetos (`.o`) empaquetados en un único archivo de *filetype* `.a`.
- Un conjunto de ficheros (`.h`) con la interfaz de la biblioteca.

Para crear una biblioteca estática debemos:

- Compilar cada uno de los objetos que forman la biblioteca
- Empaquetar los `.o` en un único archivo por medio la utilidad **ar**

Por convenio, los nombres de todas las bibliotecas estáticas comienzan por `lib` y tienen `.a` por extensión.

Ejemplo:

Supongamos que disponemos de los ficheros `stack_fp.c` y `stack_types.c` que contienen el código de las funciones que deseamos incluir en la biblioteca `libstack_fp.a`. Para obtener los ficheros objeto compilamos los ficheros fuente:

```
$ gcc -c stack_fp.c
$ gcc -c stack_types.c
```

A continuación empaquetamos los ficheros obtenidos `.o` con ar (un empaquetador similar a `tar`).

```
$ ar rcs libstack_fp.a stack_fp.o stack_types.o
```

El significado de las opciones que se le dan a `ar` es el siguiente:

Flag	Significado
r	Reemplaza los ficheros si ya existían en el paquete.
c	Crea el paquete si no existe.
s	Construye un índice del contenido.
t	Lista el contenido de un paquete (o biblioteca).
x	Extrae un fichero de un paquete (o biblioteca).

Uso de una biblioteca desde una terminal Para utilizar las funciones de la biblioteca estática en una aplicación (por ejemplo en un programa `main.c`) es necesario enlazar las funciones incluidas en la biblioteca cuando creamos el ejecutable de la aplicación. Para ello debemos indicar al compilador qué biblioteca queremos utilizar y el lugar donde se halla.

Ejemplo:

Supongamos que hemos escrito un programa `main.c` que hace uso de las funciones incluidas en la biblioteca `libstack_fp.a` creada en el apartado anterior. Para crear el ejecutable `main` debemos:

- Compilar el programa `main.c`:

```
$ gcc -c main.c
```

Notad que al incluir el fichero `main.c` llamadas a las funciones cuyos prototipos estan declarados en los ficheros `.h` de la interfaz de la biblioteca (p.e., `#include stack_fp.h`), estos deberían estar un directorio incluido en el `path` del compilador. De no ser así debe indicarse al compilador dónde se encuentran los ficheros de la interfaz mediante la opción `-I`:

```
$ gcc -c main.c -Idir_lib
```

donde `dir_lib` es el directorio donde hemos guardado los ficheros `.h` de la biblioteca.

- Enlazar el programa con la biblioteca indicando donde está (`dir_lib`) y cuál es su nombre (`stack_fp`):

```
$ gcc -o main main.o -Ldir_lib -lstack_fp
```

Consideraciones:

En el ejemplo se supone que la biblioteca y su fichero de interfaz se encuentran en un directorio llamado dir_lib.

- La opción `-I` se usa para indicar donde se encuentran los ficheros de la interfaz de la biblioteca (en este caso, *stack_fp.h* y *stack_types.h*).
- La opción `-L` indica el directorio donde se encuentra la biblioteca (en este caso *libstack_fp.a*).
- La opción `-l` indica los nombres de la bibliotecas que se van a usar. Sin embargo los nombres de las bibliotecas no deben escribirse ni con el prefijo lib ni con la extensión .a ya que el compilador espera que se sigan las normas de nombrado anteriormente citadas.

Creación y uso de una biblioteca con Makefile A continuación se muestra un fichero `makefile` que permite automatizar todo el proceso, tanto la creación de la biblioteca como del programa. En este caso se supone que todos los ficheros fuentes se encuentran en el directorio actual.

```
CC=gcc
CFLAGS=-Wall -ggdb
IFLAGS=-I./
LDFLAGS=-L./
LDLIBS=-lstack_fp
### -lm enlaza la biblioteca matematica, -pthread enlaza la biblioteca
    de hilos
LIBS = -lm -pthread

all: libstack_fp.a main

#########################################################################
# $@ es el item que aparece a la izquierda de ':'
# $< es el primer item en la lista de dependencias
# $^ son todos los archivos que se encuentran a la derecha de ':' (
   dependencias)
#########################################################################


main: main.o libstack_fp.a
    $(CC) -o $@ $< $(LDFLAGS) $(LDLIBS) $(LIBS)

main.o: main.c
    $(CC) -c -o $@ $< $(CFLAGS) $(IFLAGS)

#########################################################################
###### Crea la biblioteca con las fuentes:  stack_fp.c stack_types.c
#########################################################################

stack_fp.o: stack_fp.c stack_fp.h stack_types.h
```

```
    $(CC) -c -o $@ $< $(CFLAGS)

stack_types.o: stack_types.c stack_types.h
    $(CC) -c -o $@ $< $(CFLAGS)

libstack_fp.a: stack_fp.o stack_types.o
    $(AR) rcs $@ $^

libs: libstack_fp.a

clean:
    rm  -f *.o *.a
```

Si ya disponemos del fichero de la biblioteca (`libstack_fp.a`) y unicamente queremos crear el ejecutable, nuestro `Makefile` se reduciría a las siguientes sentencias:

```
CC=gcc
CFLAGS=-Wall -ggdb
IFLAGS=-I./
LDFLAGS=-L./
LDLIBS=-lstack_fp
# -lm enlaza la biblioteca matematica, -pthread enlaza la biblioteca
   de hilos
LIBS = -lm -pthread

all: main

#########################################################################
# $@ es el item que aparece a la izquierda de ':'
# $< es el primer item en la lista de dependencias
# $^ son todos los archivos que se encuentran a la derecha de ':' (
   dependencias)
#########################################################################


main: main.o libstack_fp.a
    $(CC) -o $@ $< $(LDFLAGS) $(LDLIBS) $(LIBS)

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

Uso de una biblioteca con Netbeans Cuando para compilar el programa anterior (`main.c`) utilicemos el entorno de Netbeans, debemos modificar las propiedades del proyecto donde reside el programa `main.c` para enlazarlo con la biblioteca estática (`libstack_fp.a`).

Recordad que la biblioteca consta del fichero con la biblioteca (`.a`) y su interfaz (`.h`). Por tanto habrá que:

- Seleccionar en la interfaz gráfica de Netbeans el directorio donde se halla la biblioteca. Para ello seleccionar la ventana donde se halla mediante la secuencia

  File -> Project Properties -> Linkers -> Libraries -> Add Library File

- Seleccionar el directorio donde se halla la interfaz

  File -> Project Properties -> C Compiler -> Include Directories

y una vez seleccionados los cambios se deben aplicar seleccionando `Apply`.

Información extraída parcialmente de http://arco.inf-cr.uclm.es/~dvilla/doc/repo/librerias/librerias.html

Apéndice 1.3.

Punteros a función

Introducción El lenguaje C permite manejar un tipo de dato, puntero a función, que ofrece un gran potencial, ya que podremos usar funciones de manera dinámica y automatizada, es decir, que dependiendo de las necesidades de nuestro programa, podremos hacer llamadas a unas funciones u otras, dependiendo de lo indicado en esa variable. Es decir, al igual que tenemos punteros que apuntan a variables (y pueden cambiar durante el tiempo de ejecución), podemos hacer llamadas a una función, la cual viene indicada por un puntero.

Como ejemplo, podemos ver la función `stack_print` que usa las funciones definidas en `file_utils.h`. En esta biblioteca, hemos definido nuevos tipos de dato (que no son más que punteros a los tipos que ya conocemos, int, float, etc.) y una serie de funciones para manejarlos (init, copy, compare, print, etc.). Si usamos este tipo de datos, podemos hacer que nuestros programas sean genéricos y, con solo cambiar a la función que se apunta (dependiendo del tipo de dato), el programa funcione igualmente.

La función `stack_print` tiene el siguiente prototipo:

```
/**
 * @brief  This function writes the elements of the stack to the
    stream.
 * @param fp A pointer to the stream
 * @param s A pointer to the element to the stack
 * @param A pointer to the elements' print function
 * @return Upon successful return, these function returns the number
    of characters written.
 * The function returns a negative value if there was a problem
    writing to the file.
 *  */
int stack_print(FILE* fp, const Stack *s,  P_stack_ele_print f);
```

Es decir, recibe el fichero donde se vaya a imprimir (`stdout`, fichero, etc.), un puntero a la pila a imprimir y un puntero a la función de impresión del tipo de elemento almacenado en la pila. Este último argumento, viene definido previamente en el archivo de cabecera del TAD pila:

```
/**
 * @brief Typedef for a function pointer to print a stack element at
    stream
 **/
typedef int (*P_stack_ele_print)(FILE *, const void*);
```

Y nos indica que es una función que devuelve un `int`, recibe un puntero a fichero (de nuevo,

`stdout`, fichero, etc.) y un puntero a void. Este último argumento es muy importante, ya que el hecho que sea **`void *`** nos va a permitir que podamos pasar diferentes tipos de datos, como hemos venido haciendo hasta ahora.

Todas las funciones que se pasen como argumento en `stack_print` deben cumplir con este prototipo. Así, en `file_utils.h`, como puedes ver, todas las funciones de impresión mantienen los mismos argumentos, a pesar de manejar tipos de dato diferentes.

Por ejemplo, si nuestra pila (llamada s1) contuviese elementos de tipo:

`int` la llamada a la función para imprimir la pila sería:

```
stack_print(stdout, s1, int_print);
```

`float` la llamada a la función para imprimir la pila sería:

```
stack_print(stdout, s1, float_print);
```