Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Objetivos:

- Aprender a utilizar un TAD ya implementado, usando como documentación su fichero de interfaz (.h)
- Aprender a construir bibliotecas de estructuras de datos, definiendo los tipos de datos como TIPOS DE DATOS OPACOS.
- Aprender a manejar la memoria dinámica.
- Ser capaz de pasar por línea de comandos argumentos al `main`.
- Ser capaz de automatizar la compilación del proyecto a través del `makefile`

Descripción:

En esta práctica veremos cómo manipular y crear un "vector" que contiene datos utilizando TAD. Aunque sabemos que en C es posible tratar los vectores de forma muy directa, como agrupaciones estáticas o dinámicas de datos como

```
float v[10];
```

el uso de TAD es en general una buena práctica de programación que resulta especialmente adecuada para la reutilización de código. El único inconveniente es que la implementación es más costosa, puesto que exige **definir, diseñar y construir** todos los procedimientos y funciones para construir el TAD (en nuestro caso el "vector") y manejar la información que contenga. A cambio, el diseño con TAD conlleva innumerables ventajas puesto que, una vez implementado, su manipulación es muy intuitiva y simple y, sobre todo, independiente de los tipos de datos que agrupe el vector. Siguiendo con nuestro ejemplo, veremos que nuestro TAD "vector" puede agrupar datos de tipo **int**, **char**, **float**, **double**, de una forma muy directa.

La clave de definir un TAD correctamente está en tres aspectos:

- Hacerlo totalmente independiente de los detalles de implementación interna: esto se consigue usando **tipos opacos de datos** (`void *` en el módulo de definición de la biblioteca del TAD `fichero.h`)
- Permitir de manera natural **cambiar el tipo de datos concreto** que almacena el TAD: esto se consigue utilizando una **definición abstracta** (en nuestro caso el nuevo tipo de datos `TELEMENTO`) en el módulo de definición, mediante una sentencia `typedef` (para cambiar el tipo de datos concreto basta modificar el `TELEMENTO` en `fichero.h` y en `fichero.c`)
- Construir un **repertorio de procedimientos y funciones** suficientemente amplio para manipular el TAD, pero en los que no se incluya ningún aspecto concreto (dependencia) con determinadas tareas o tipos de datos. Así garantizamos su completa reutilización.

En esta práctica utilizaremos un TAD para familiarizarnos inicialmente con esta nueva forma de diseñar programas. Seguidamente construiremos los procedimientos/funciones básicas de manipulación para dicho TAD. En prácticas posteriores haremos uso de este para tareas de más alto nivel, y ahí es donde comenzaremos a apreciar las ventajas de reutilización de código.

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Cómo trabajar con tipos de datos opacos:

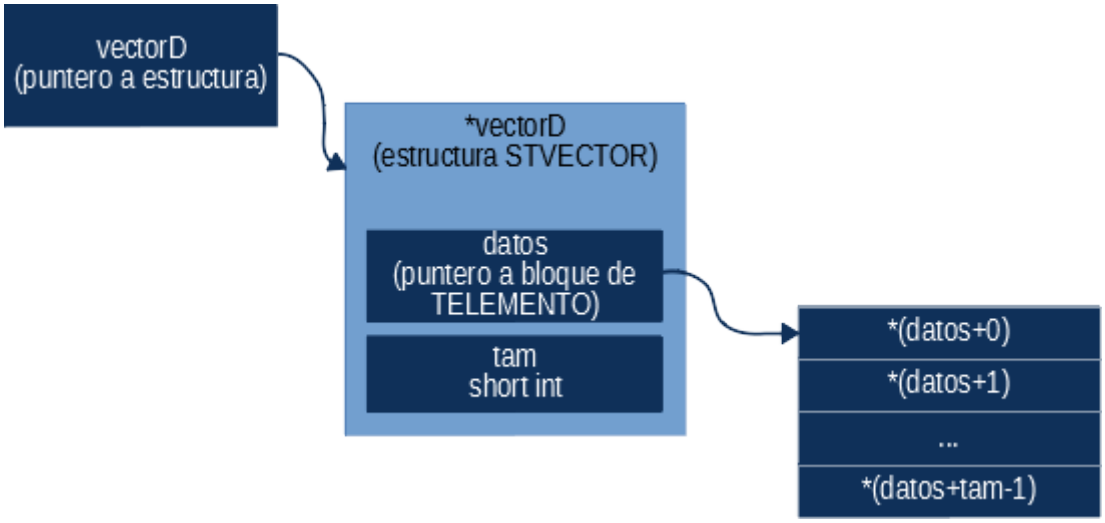
Abordaremos esta tarea directamente a través del ejemplo que consideramos en esta práctica, y que va a ser un "**vector**" (que denominaremos **vector dinámico**) que almacena datos de tipo real (**float**).

Podemos pensar inicialmente en dos formas distintas para definir dicho TAD, al que por simplicidad vamos a denominar `vectorD` (vector dinámico):

a) Como un puntero a un bloque de memoria de N números reales, donde N es el número de componentes del vector. En este caso, como el tamaño del vector se conoce en tiempo de ejecución, todos los procedimientos de manipulación del TAD deben recibir N como parámetro. **Esto no es una buena práctica**, pues nada evita que desde `main()` se llame a cualquiera de estos procedimientos con un tamaño de N erróneo.

b) Por lo dicho en el apartado anterior, el tamaño N del vector debe estar ENCAPSULADO con el TAD `vectorD` pues, una vez que se dice el tamaño del vector, este no cambia durante toda la manipulación de este, y no se debe permitir al usuario que realice operaciones erróneas utilizando los procedimientos de manipulación del vector.

Por tanto, utilizaremos la opción b), que define el TAD `vectorD` como un puntero a una estructura con dos campos: el puntero al bloque de memoria de números reales (`float`) y el tamaño del vector (número de componentes).



Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Empezaremos definiendo los tipos de datos que requiere el TAD:

Vamos a asumir que el vector almacenará valores **float.** Dicho tipo de datos se abstraerá definiendo un **nuevo tipo de datos** (utlizando `typedef`) que deberá incluirse tanto en `vectordinamico.h` como en `vectordinamico.c`:

vectordinamico.h	vectordinamico.c
`typedef float TELEMENTO;`	`typedef float TELEMENTO;`

Esto hace que el TAD sea lo más general posible y que se vea claramente el tipo de datos que CONTIENE. Cambiar este tipo de datos es simplemente cambiar esta línea de código.

Todos los TAD, al igual que las bibliotecas, tienen un fichero de interfaz (`.h`) y un fichero de implementación (`.c` o `.a`). El primero de ellos contiene las definiciones de tipos de datos y los prototipos de las funciones del TAD. El segundo la implementación (código fuente o compilado) de las mismas.

Todas las operaciones que manipulen o traten los vectores deben estar incluidas en un módulo de biblioteca independiente donde el tipo `vectorD` debe estar definido como **opaco** (deben ocultarse los detalles de implementación a los usuarios de la biblioteca). Esto quiere decir que:

a) debe ser declarado como *tipo de datos puntero en el módulo de implementación* (`vectordinamico.c`),

b) quedando como *puntero indefinido* (`void *`) en el *módulo de definición* (`vectordinamico.h`) en el que únicamente se establece el *nombre* del tipo opaco

vectordinamico.h

```
typedef float TELEMENTO;
typedef void * vectorD;
```

vectordinamico.c

```
typedef float TELEMENTO;
typedef struct {
 TELEMENTO *datos;     /*valores del vector*/
 short tam;            /*tamaño del vector*/
}STVECTOR;          /*definición del tipo de datos estructura*/
typedef STVECTOR *vectorD;  /*puntero a estructura*/
```

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Práctica 1.0: implementación del TAD (versión 0):

Crea un nuevo proyecto en Netbeans al que le pondrás como nombre **P1_v0** indicando que sí quieres que cree `main.c`. (recuerda marcar a la derecha el tipo de main: C11). A continuación, se detalla la implementación de los ficheros de interfaz e implementación del TAD `vectorD`:

Interfaz de usuario: `vectordinamico.h`

En la interfaz de usuario se indican los tipos de datos que es posible utilizar y los prototipos de las funciones que los manejan.

Para crear un fichero de interfaz en un proyecto NetBeans debemos abrir el menú contextual (botón derecho del ratón) sobre el apartado "Header Files" del proyecto y seleccionar "New→C Header File…". A continuación, indicamos el nombre "**`vectordinamico`**", cuidando que la extensión "**`.h`**" esté seleccionada y pulsamos el botón "Terminar". El contenido del fichero de interfaz **`vectordinamico.h`** será el siguiente (podéis borrar las líneas 17-27 que hacen referencia a C++ y las líneas iniciales de comentarios):

```
1  #ifndef VECTORDINAMICO_H
2  #define VECTORDINAMICO_H
3
4  /*Tipo de datos de los elementos del vector*/
5  typedef float TELEMENTO;
6  /*tipo opaco, los detalles de implementacion estan ocultos al usuario*/
7  typedef void * vectorD;
8  /*Funcion crear: asigna memoria y devuelve la asignacion al vector. Recibe v
9   *por referencia para devolver al programa principal la direccion de memoria
10  *reservada por este procedimiento*/
11 void crear(vectorD *v, short longitud);
12 /*Funcion asignar: Llena una posicion del vector con un valor. Recibe una copia
13  *de la direccion de memoria reservada para el vector v*/
14 void asignar(vectorD *v, short posicion, TELEMENTO valor);
15
16 #endif /* VECTORDINAMICO_H */
```

Módulo de implementación: `vectordinamico.c`

Los detalles de la implementación o construcción del tipo de datos `vector` están en el fichero `vectordinamico.c`. Estos detalles de implementación siempre se ocultarán al usuario del TAD, que únicamente necesita la interfaz `vectordinamico.h` para poder escribir sus programas.

Para crear un fichero de implementación en un proyecto NetBeans debes abrir el menú contextual en los "Source Files" del proyecto y elegir "New→C Source File…". A continuación, especifica el nombre "**`vectordinamico`**", cuidando que la extensión "`.c`" esté seleccionada y pulsa el botón "Terminar".

El contenido de la primera versión de este archivo, con las únicas funciones de crear el vector y rellenarlo de valores, debe ser el siguiente[1]:

[1] Para que funcione correctamente la función `malloc()` y no se produzcan warnings de compilación, debes incluir la librería `stdlib.h`. La librería `stdio.h` es necesaria para poder usar `printf()`.

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

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

/*Se vuelve a definir el tipo de datos que contiene el vector*/
typedef float TELEMENTO;

/*Implementacion del TAD vectorD */
typedef struct {
    TELEMENTO *datos; /*valores del vector*/
    short tam;        /*tamanho del vector*/
} STVECTOR;           /*definicion del tipo de datos estructura*/
typedef STVECTOR *vectorD; /*puntero a estructura*/

/*Funciones de manipulacion de datos */
/*Funcion crear: asigna memoria y devuelve la asignacion al vector*/
void crear(vectorD *v, short longitud) {
    short i;
    *v = (vectorD) malloc(sizeof (STVECTOR));
    (*v)->datos = (TELEMENTO*) malloc(longitud * sizeof (TELEMENTO));
    (*v)->tam = longitud;
    for (i = 0; i < longitud; i++)
        /*Inicializacion a 0 de las componentes del vector*/
        *((*v)->datos + i) = 0;
}

/*Funcion asignar: Asigna un valor a una posicion del vector*/
void asignar(vectorD *v, short posicion, TELEMENTO valor) {
    *((*v)->datos + posicion) = valor;
}
```

Con la construcción de los ficheros de interfaz (`vectordinamico.h`) e implementación (`vectordinamico.c`) queda concluido el diseño e implementación del TAD.

Programa principal: `main.c`

En primer lugar, recuerda que para utilizar la biblioteca que acabas de escribir, debes incluirla en main como:
`#include "vectordinamico.h"`

A continuación, declara las variables que vas a utilizar y usa los procedimientos que manipulan el vector especificados en `vectordinamico.h`, teniendo en cuenta los tipos de los argumentos. Lo importante es pensar que el tipo de datos que estás usando (TELEMENTO) es como cualquier otro de los tipos de datos estándar (int, float, char, etc.) y que por tanto lo usarás de la misma forma. Cada tipo de datos tiene definidas las operaciones que lo manipulan y, en el caso del vector dinámico, por ahora sólo tiene definidas las operaciones `crear()` y `asignar()`. A medida que vayas avanzando irás añadiéndole más funcionalidades.

ES MUY IMPORTANTE DARSE CUENTA DE QUE COMO NO SABEMOS CÓMO ESTÁ CONSTRUIDO EL TIPO DE DATOS `vectorD`, NO PODEMOS ACCEDER A SUS COMPONENTES, NI CREAR UNA VARIABLE DE ESE TIPO, DESTRUIRLA O MANIPULARLA SI NO TENEMOS PROCEDIMIENTOS EN `vectordinamico.h` QUE NOS PERMITAN HACERLO. ES DECIR, LA MANIPULACIÓN Y EL ACCESO AL TAD O A SUS ELEMENTOS SE REALIZA **ÚNICA Y EXCLUSIVAMENTE** A TRAVÉS DE LOS PROCEDIMIENTOS DEFINIDOS EN EL TAD.

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

```c
#include <stdio.h>
#include <stdlib.h>
#include "vectordinamico.h"

int main(int argc, char** argv) {
    vectorD vector; /*declaramos el vector*/
    short longitud, i; /*variables tamanho y recorrido*/
    TELEMENTO valor; /*valor a introducir en el vector*/
    char opcion; /*variable del menu*/

    do {
        printf("\n--------------------\n");
        printf("\na) Crear vector v");
        printf("\ns) Salir");
        printf("\n--------------------\n");
        printf("\nOpcion: ");
        scanf(" %c", &opcion); //OJO: espacio antes de %c para vaciar buffer de teclado
        switch (opcion) {
            case 'a': /*Crear vector v*/
                printf("Longitud del vector v: ");
                scanf("%hd", &longitud);
                crear(&vector, longitud);
                /*Asignar valores a v*/
                for (i = 0; i < longitud; i++) {
                    printf("vector(%hd): ", i);
                    scanf("%f", &valor);
                    asignar(&vector, i, valor);
                }
                break;
            case 's':
                printf("Saliendo del programa\n");
                break;
            default: printf("Opcion incorrecta\n");
        }
    } while (opcion != 's');
    return (EXIT_SUCCESS);
}
```

A continuación, desde Netbeans, compila todo, depura y ejecuta el programa, comprobando que funcione correctamente.

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Práctica 1.1: utilización del TAD a través de una librería compilada

Ahora vamos a trabajar en la consola de Cygwin y vamos a ver cómo se crearía una librería compilada a partir de nuestro archivo de implementación (`vectordinamico.c`) para ocultar al usuario los detalles de implementación.

Para ello, en nuestra carpeta de trabajo, crearemos la carpeta **`P1_v1`** y copiaremos en ella los archivos del proyecto **`P1_v0`** con los que acabamos de trabajar: `main.c`, `vectordinamico.h` y `vectordinamico.c`. Nos cambiamos de carpeta de trabajo, situándonos en la carpeta `P1_v1`.

Creación de una librería compilada para ocultar los detalles de implementación

En primer lugar, vamos a ver cómo se crea una librería que se puede proporcionar al usuario para ocultar todos los detalles de implementación, es decir, para ocultar totalmente el archivo `vectordinamico.c`.

- Como el funcionamiento de la librería compilada depende del sistema operativo y su versión, la vamos a crear a partir del fichero `vectordinamico.c` que habéis creado. Para crear la librería estática necesitáis crear el fichero .o mediante el `comando:`

```
gcc -static -c -o vectordinamico.o vectordinamico.c
```

A continuación, con el comando `ar` creamos la librería (con extensión .a):

```
ar -rcs -o libvectordinamico.a vectordinamico.o
```

Podéis consultar el contenido de una librería compilada del siguiente modo:

```
ar -t libvectordinamico.a
```

La práctica consistirá en la compilación y ejecución del programa que has creado, pero SIN UTILIZAR vectordinamico.c en la compilación (de hecho, una vez compilado en la librería, puedes borrar de esta carpeta vectordinamico.c y vectordinamico.o), sino utilizando únicamente la librería compilada `libvectordinamico.a`.

Descarga el `makefile` que tienes junto con este guion, necesario para compilar este proyecto a partir de una librería propia. Para ello, fíjate que se han incluido en la fase de enlazado las opciones `-L` (para indicar la carpeta donde está la librería) y `-l` para indicar el nombre de la librería modificando el contenido del makefile base como se indica a continuación:

```
 …
#carpeta de las librerías estáticas propias (si están en la actual, ponemos .)
LIB_FILES_DIR = .
#opciones de compilación, indica dónde están nuestra librería estática (si es una
#librería estándar, no es necesario)
LIBRARIES= -L $(LIB_FILES_DIR)
#si incluye una librería, en este caso la de vector dinámico
LIBS=-lvectordinamico
…
#REGLA 1: genera el ejecutable, dependencia de los .o
$(OUTPUT): $(OBJS)
	$(CC) -o $(OUTPUT) $(OBJS) $(LIBRARIES) $(LIBS)
…
```

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Práctica 1.2: utilización de los argumentos de main.

Ahora vamos a modificar la función main.c para poder pasar argumentos por la línea de comandos. **Cread una carpeta llamada P1_v2 y copiad los archivos main.c**, **vectordinamico.c** y **vectordinamico.h de la carpeta P1_v0 y el fichero makefile de la carpeta P1_v1**. En este makefile, debéis eliminar todas las referencias a la librería compilada (lo que está en rojo en la página anterior) y añadir el fichero vectordinamico.c a los fuentes: **`SRCS = main.c vectordinamico.c`**.

Los archivos los podéis editar en Netbeans o CLion simplemente arrastrándolos encima del editor, si os es más cómodo para trabajar, pero la compilación debéis hacerla fuera (o en el terminal de Netbeans).

La función main() tiene el siguiente formato:

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

donde `argc` indica el **número de argumentos** y `argv` es un **vector de cadenas de texto**.

Si desde la consola ejecutas el programa (cuyo nombre es `ejecutable`) enviando como argumentos de entrada los valores a almacenar:

```
./ejecutable 3.1 4.6 3.2 1.6 2.1 0.5
```

Los argumentos de `main` tomarán los siguientes valores:

- `argc`, que es el número de cadenas que se escriben en la línea de comandos, tomará el valor de 7, y esas cadenas se almacenan en el vector `argv[]` de tamaño 7:

`argv[0]`	`"./ejecutable"`	Nombre del programa
`argv[1]`	`"3.1"`	Componentes del vector (6 componentes)
`argv[2]`	`"4.6"`	
`argv[3]`	`"3.2"`	
`argv[4]`	`"1.6"`	
`argv[5]`	`"2.1"`	
`argv[6]`	`"0.5"`	

Para poder transformar cadenas de texto (que es lo que contiene `argv[i]`) a `float`, que son los valores a almacenar en el vector, es necesario utilizar la función `atof(cadena de texto)`. Esta función devuelve el número correspondiente a dicha cadena en formato real. En el caso del ejemplo tendríamos conceptualmente un vector de tamaño 6 de la forma (3.1 4.6 3.2 1.6 2.1 0.5).

Recuerda que es necesario validar el número de argumentos. Como estás tratando con vectores, el número de argumentos que se pasen al ejecutable correspondientes a los valores reales a almacenar en el vector debe ser mayor o igual que 2 (que corresponden al nombre del ejecutable, y al menos a una componente del vector). En el caso anterior, `argc` será 1+6=7, por lo que el vector será de longitud 6.

Como todavía no tenemos la función imprimir, para comprobar que estás rellenando el vector correctamente, puedes ir imprimiendo por pantalla los valores a medida que los vas asignando a cada componente del vector.

Práctica 1.3: ampliación de la funcionalidad del TAD:

Crea desde Netbeans el proyecto **`P1_v3`** y copia en ella los archivos .c y .h de **`P1_v2`**. En esta parte de la práctica os facilitaremos las especificaciones informales para **ampliar la funcionalidad** del TAD, de modo que se incremente el repertorio de procedimientos que lo manipulen y realicen tareas algo más complejas que las incluidas hasta el momento. Estas funciones que incorporarás al TAD se utilizarán en posteriores prácticas, que van a basarse en el TAD **vectordinamico**.

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

De nuevo, puedes usar Netbeans para editar los ficheros, pero debes utilizar el makefile anterior para compilar tu proyecto desde el terminal.

Función `liberar()`

En este momento podemos plantear una nueva pregunta. Imaginemos que ejecutamos nuestro programa introduciendo un vector `v` con 40 componentes y que a continuación introducimos un nuevo vector con 10 componentes.

¿Qué sucede con la memoria que habíamos reservado para el primer vector? No la hemos liberado y, al mismo tiempo, hemos perdido la referencia a dicha memoria, pues con la segunda definición del vector v apuntamos a **otra** zona de memoria con 10 posiciones consecutivas.

La solución a este problema es liberar la memoria ocupada por el vector previo **ANTES** de realizar una nueva asignación de memoria.

Además, hemos de tener en cuenta que esta liberación sólo se puede realizar si antes se ha hecho la creación del vector (esto es, es incorrecto liberar un vector si antes no lo habíamos creado). Por tanto, desde el punto de vista del uso del TAD, el programa debe ser cauto en sólo liberar si antes había sido creado el vector (de otro modo, tendríamos un error de ejecución pues el programa intentaría liberar unas posiciones de memoria no reservadas).

Para escribir esta función puedes inspirarte en la función `destruirMatriz.c` que has corregido para la práctica P0_E5, pero ten en cuenta que en esa práctica no se trabajaba con TADs opacos y por tanto puede cambiar la forma de hacer referencia al TAD (en ese caso la variable era una estructura y ahora es un puntero a estructura).

Añade la llamada a esta función como una opción de tu menú: **b) Liberar vector.**

Función `recuperar()`

La función `recuperar()` recibirá como argumentos un determinado `vectorD` y la posición y DEVOLVERÁ al programa principal el valor real (`TELEMENTO`) que se encuentra en dicha posición. Será el programa principal el que se encargue de imprimirlo en pantalla. Debes fijarte que la componente que se quiera extraer EXISTA. Puedes tomar como ejemplo la función `obtenerElemento.c` corregida en la práctica P0_E5.

Observa que todas estas funciones tienen algunos argumentos en los cuales deben volcar resultados y otros argumentos que actúan únicamente como valor de entrada (sólo lectura). Esto afecta a cómo debes pasar los argumentos (con puntero al correspondiente tipo de datos o sin puntero).

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Función `longitudVector()`

Es necesario introducir una función en el TAD vector dinámico que nos permitan obtener el tamaño actual del vector, y que así pueda ser utilizado por procedimientos auxiliares que operen con los vectores, como por ejemplo la función `imprimir()`.

main.c (cambios): Función `imprimir()`

Como primer paso para completar el programa, vamos a añadir al menú la opción de `imprimir()`, que imprime el vector. Para acceder a las componentes del vector usaremos la función del TAD `recuperar()` y para acceder a su tamaño usaremos la función `longitudVector()` escrita anteriormente, ya que al ser un tipo opaco no podemos hacer v→tam.

main.c (cambios): Función `sumaEscalar()`

La función `sumaEscalar()` recibirá como argumento un `vectorD` y DEVOLVERÁ al programa principal el vector resultante de multiplicar un escalar solicitado al usuario dentro de dicha función por cada elemento del vector. Fíjate que como esta función no forma parte del repertorio del TAD, tendrás que usar las funciones `crear()`, `recuperar()`, `longitudVector()` y `asignar()` para realizar la operación solicitada.

NOTA: Las funciones `sumaEscalar()` e `imprimir()` NO deben formar parte del TAD vector dinámico, ya que asume un determinado tipo de datos para realizar las operaciones. Se debe añadir como procedimiento adicional dentro del fichero `main.c` del proyecto todas las funciones que no sean estrictamente necesarias para manejar el TAD.

Entregables

Deberás entregar por el Campus Virtual el ejercicio correspondiente a la práctica 1, versión 3 (P1_v3). Las fechas de entrega se especifican en el Campus Virtual, y las instrucciones para generar el fichero son las siguientes:

- Deberás subir un único fichero comprimido con el nombre **Apellido1Apellido2_1.zip**.
- Incluye ÚNICAMENTE los ficheros fuente (.c) y de cabecera (.h), así como su correspondiente makefile.

Para ser válida, la práctica debe compilar directamente con el makefile (sin opciones) en Linux/Cygwin, en otro caso será evaluada con la calificación de 0. Este criterio se mantendrá en el resto de las prácticas de la asignatura.

Ejemplo de ejecución:

```
$ ./ejecutable 3.1 4.6 3.2 1.6 2.1 0.5

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
```

```
Opcion: c
Los elementos del vector son: [ 3.1 4.6 3.2 1.6 2.1 0.5]

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
Opcion: b
Vector liberado

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
Opcion: c
Error en imprimir: el vector no existe

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
Opcion: a
Introduce el tamanho del vector: 4
Elemento (0) del vector: 1
Elemento (1) del vector: 2
Elemento (2) del vector: 3
Elemento (3) del vector: 4

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
Opcion: c

Los elementos del vector son: [ 1.0 2.0 3.0 4.0]

a) Crear vector
b) Liberar vector
c) Imprimir vector
d) Suma escalar a vector
s) Salir
Opcion: d
Introduce número para sumar al vector: 2

Los elementos del vector son: [ 3.0 4.0 5.0 6.0]
```

Práctica 1: creación y manipulación de la Estructura de Datos "vector" usando TAD

Anotaciones sobre reserva dinámica de memoria en C

Conceptos previos:

- Función **`sizeof`**: calcula en tiempo de ejecución el tamaño en bytes de cualquier tipo de dato. **`sizeof(tipo_de_dato)`**
- Puntero Nulo: la constante **`NULL`** contiene por definición el valor 0, que corresponde a una posición de memoria donde no puede existir ningún dato válido. Suele utilizarse para indicar que un puntero no está inicializado, o que no dispone de ningún dato al que hacer referencia. Es una buena técnica de programación inicializar todos los punteros a **`NULL`**.

Algunas funciones para la gestión dinámica de memoria:

- **`malloc(int tamanho)`**: devuelve un puntero de tipo **`void`** que contiene la dirección de memoria a partir de la cual se ha reservado un bloque de memoria del tamaño requerido (**`tamanho`** se expresa en bytes). Este puntero **`void`** hay que moldearlo para que apunte al tipo de dato para el que se hace la reserva, para asegurarnos así que funcionará correctamente la aritmética de punteros.
- Si no es posible realizar la reserva, **`malloc()`** devuelve **`NULL`**.
- Ejemplo: Reserva dinámica de una matriz de n componentes enteras, mostrando el puntero void moldeado a int *: **`p=(int *) malloc(n*sizeof(int));`**
- **`free(void *pMemoria)`**: pasándole el puntero devuelto por **`malloc()`**, libera la memoria cuando ya no la necesitemos, para su uso posterior en el programa.

BUENAS PRÁCTICAS DE PROGRAMACIÓN

1. SIEMPRE inicializa los punteros a **`NULL`**. De este modo te aseguras de que, si se utilizan posteriormente sin darles otro valor, se generará un error de ejecución, que es más fácil de detectar que el error lógico que tendríamos en caso de no inicializarlos.

2. SIEMPRE comprueba si la reserva dinámica tiene éxito. Una comprobación un tanto drástica, pero muy efectiva, es:

```
m=…malloc(…); /*Intento de reserva dinámica*/
if (!m)       /*Si no se ha podido reservar, m vale NULL*/
   exit(EXIT_FAILURE);   /*Salir del programa*/
```

3. Cuando vayas a recorrer una agrupación (array) con un puntero **`p`**, debes mantener otro puntero **`pInicio`** apuntando al inicio de dicha agrupación, para poder volver a reapuntar **`p`** en cualquier momento a un lugar conocido.