Programación II
1º Grado Ingeniería Informática

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

Objetivos:

- Aprender a utilizar un TAD ya implementado, usando como documentación su fichero de interfaz
- 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 una "matriz" que contiene datos utilizando TAD. Aunque sabemos que en C es posible tratar las matrices de forma muy directa, como agrupaciones estáticas o dinámicas de datos como

```
float v[2][3];
```

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 la "matriz") 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 la matriz. Siguiendo con nuestro ejemplo, veremos que nuestro TAD "matriz" 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 el `fichero.h`)
- 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 "matriz" usando TAD

Cómo trabajar con tipos de datos opacos:

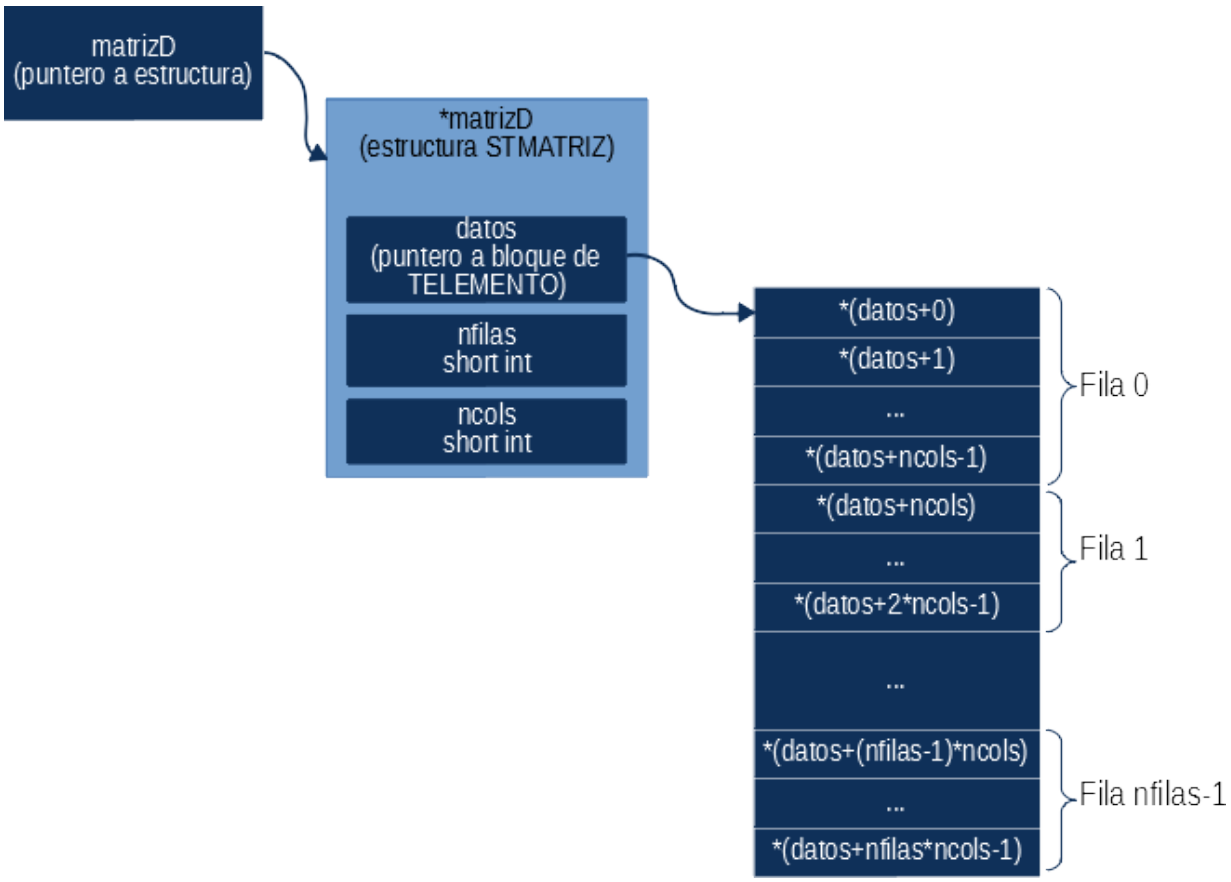
Abordaremos esta tarea directamente a través del ejemplo que consideramos en esta práctica, y que va a ser una "**matriz**" (que denominaremos **matriz dinámica**) que almacena datos de tipo real (**float**).

Podemos pensar inicialmente en dos formas distintas para definir dicho TAD, al que por simplicidad vamos a denominar `matrizD` (matriz dinámica):

a) Como un puntero a un bloque de memoria de NxM números reales, donde N es el número de filas de la matriz y M es el número de columnas de la matriz. En este caso, como el tamaño de la matriz se conoce en tiempo de ejecución, todos los procedimientos de manipulación del TAD deben recibir N y M como parámetros. Esto no es una buena práctica, pues nada evita que desde `main()` se llame a cualquiera de estos procedimientos con unos tamaños N y/o M erróneos.

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

Por tanto, utilizaremos la opción b), que define el TAD `matrizD` como un puntero a una estructura con dos campos: el puntero al bloque de memoria de números reales (`float`) y el tamaño de la matriz cuadrada (número de filas y número de columnas).



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

Empezaremos definiendo los tipos de datos que requiere el TAD:

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

`matrizdinamica.h`	`matrizdinamica.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 (y algún `%f` por `%d` si hay `printf` o `scanf`).

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 las matrices deben estar incluidas en un módulo de biblioteca independiente donde el tipo `matrizD` debe estar definido como <u>opaco</u> (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 (*`matrizdinamica.c`*)*,

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

`matrizdinamica.h`

```
typedef float TELEMENTO;
typedef void * matrizD;
```

`matrizdinamica.c`

```
typedef float TELEMENTO;
typedef struct {
 TELEMENTO *datos;    /*valores de la matriz*/
 short nfilas,ncols; /*tamaño de la matriz (filas, columnas)*/
}STMATRIZ;         /*definición del tipo de datos estructura*/

typedef STMATRIZ *matrizD;  /*puntero a estructura*/
```

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

Práctica 1.0: utilización del TAD

Como práctica inicial de TAD, construiremos un programa principal que **utilice** el TAD matrizdinamica, del cual os facilitaremos:

- el módulo de interfaz, donde están especificados los procedimientos que es posible usar para manejar el TAD o acceder a sus elementos (fichero **`matrizdinamica.h`**)
- el módulo de implementación YA COMPILADO (fichero **`libmatrizdinamica.a`**[1]). Este archivo contiene los ficheros `.o` asociados a la librería. Para consultar el contenido del archivo podéis usar el comando ar: `ar -t libmatrizdinamica.a`

La práctica consistirá en la creación de un módulo principal (**`main.c`**) que utilice el TAD para una tarea sencilla. Deberá mostrarse un menú que permita al usuario realizar las operaciones siguientes:

```
a) Crear matriz m1
s) Salir
```

Observa detenidamente, desde cualquier editor, el contenido de los siguientes ficheros que has extraído de `matrizdinamica.zip`:

- `matrizdinamica.h`
  definición de los tipos de datos y de los prototipos de las funciones
- `main.c`
  uso e invocación de las funciones de manipulación del TAD.

Crea un `makefile` para compilar el programa. Para ello deberás utilizar 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 (en este caso, **`-L . -lmatrizdinamica`**).

[1] Como el funcionamiento de esta librería compilada depende del sistema operativo y su versión, si no funciona podéis crearla a partir del fichero `matrizdinamica.c` que podéis copiar de la página 6 de este documento. Para crear la librería estática necesitáis: crear el fichero `.o` mediante el comando `gcc –static –c –o matrizdinamica.o matrizdinamica.c`. A continuación, con el comando `ar` creamos la librería: `ar –rcs –o libmatrizdinamica.a matrizdinamica.o`

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

Práctica 1.1: implementación del TAD (versión 0):

Como primer paso hacia la implementación del TAD considera nuevamente el programa de la práctica 1.0, en el cual vas a sustituir el módulo de implementación compilado (**`libmatrizdinamica.a`**) por un nuevo módulo con el código fuente necesario que construirás durante la práctica. Ello te permitirá comprender todos los detalles de bajo nivel del TAD. El menú seguirá siendo el anterior:

```
a) Crear matriz m1
s) Salir
```

Crea un nuevo proyecto en Netbeans al que le pondrás como nombre Practica1. A continuación se detalla la implementación de los ficheros de interfaz e implementación del TAD `matrizD`:

Interfaz de usuario: `matrizdinamica.h`

Para escribir el programa sólo es necesario conocer la interfaz de usuario, donde 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 "**`matrizdinamica`**", cuidando que la extensión "**`.h`**" esté seleccionada y pulsamos el botón "Terminar". El contenido del fichero de interfaz **`matrizdinamica.h`** será el siguiente:

`matrizdinamica.h`

```
/*Tipo de datos de los elementos de la matriz*/
typedef float TELEMENTO;
/*tipo opaco, los detalles de implementación están ocultos al usuario*/
typedef void * matrizD;

/*Función crear: asigna memoria y devuelve la asignación a la matriz. Recibe m1
por referencia para devolver al programa principal la dirección de memoria
reservada por este procedimiento*/
void crear(matrizD *m1,short nfilas, short ncolumnas);

/*Función asignar: Llena una posición de la matriz con un valor. Recibe una
copia de la dirección de memoria reservada para la matriz m1*/
void asignar(matrizD *m1, short fila, short columna, TELEMENTO valor);
```

Módulo de implementación: `matrizdinamica.c`

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

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

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 "**`matrizdinamica`**", 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 la matriz y rellenarla de valores, debe ser el siguiente[2]:

matrizdinamica.c

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

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

/*Implementación del TAD matrizD */
typedef struct {
    TELEMENTO *datos;     /*valores de la matriz*/
    short nfilas,ncols;  /*tamaño de la matriz (filas, columnas)*/
}STMATRIZ;                /*definición del tipo de datos estructura*/
typedef STMATRIZ *matrizD;  /*puntero a estructura*/

/*Funciones de manipulación de datos */
/*Función crear: asigna memoria y devuelve la asignación a la matriz*/
void crear(matrizD *m1, short tamf, short tamc)
{
    short i;
    *m1=(matrizD)malloc(sizeof(STMATRIZ));
    (*m1)->datos=(TELEMENTO*)malloc(tamf*tamc*sizeof(TELEMENTO));
    (*m1)->nfilas=tamf;   (*m1)->ncols=tamc;
    for(i=1; i <= tamf*tamc; i++)
       /*Inicialización a 0 de las componentes de la matriz*/
       *((*m1)->datos+i-1) = 0;
}
/*Función asignar: Asigna un valor a una posición de la matriz */
void asignar(matrizD *m1, short fila, short columna, TELEMENTO valor){
      *((*m1)->datos + (fila-1)*(*m1)->ncols + columna-1)=valor;
}
```

[2] 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 "matriz" usando TAD

Con la construcción de los ficheros de interfaz (`matrizdinamica.h`) e implementación (`matrizdinamica.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 "matrizdinamica.h"`

A continuación, declara las variables que vas a utilizar y usa los procedimientos que manipulan la matriz especificados en `matrizdinamica.h`, teniendo en cuenta los tipos de los argumentos. Lo importante es pensar que el tipo de datos que estás usando 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 de la matrizdadinamica, 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 `matrizD`, NO PODEMOS ACCEDER A SUS COMPONENTES, NI CREAR UNA VARIABLE DE ESE TIPO, DESTRUIRLA O MANIPULARLA SI NO TENEMOS PROCEDIMIENTOS EN `matrizdinamica.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.

`main.c`

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

int main(int argc, char** argv) {
      matrizD m1;              /* declaramos la matriz*/
      short nfil,ncol, i, j;  /*variables tamaño y recorrido */
      TELEMENTO valor;        /*El valor a introducir en la matriz*/
      char opcion;            /*La variable del menú*/

      do {
            printf("\n--------------------\n");
            printf("\na) Crear matriz m1");
            printf("\ns) Salir");
            printf("\n--------------------\n");
            printf("\nOpcion: ");
            scanf(" %c",&opcion);
            ...
```

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

```
            ...
            switch(opcion) {
                  case 'a':   /*Crear matriz m1*/
                        printf("Tamanho de la matriz m1 (filas columnas): ");
                        scanf("%hd %hd", &nfil,&ncol);
                        crear(&m1, nfil, ncol);
                        /*Asignar valores a m1*/
                        for (i=1; i<=nfil; i++) {
                              for (j=1; j<=ncol; j++) {
                                    printf("m1(%d,%d): ",i,j);
                                    scanf("%f",&valor);
                                    asignar(&m1, i, j, valor);
                              }
                        }
                        break;
                  case 's':
                        printf ("Saliendo del programa\n");
                        break;
                  default: printf ("Opcion incorrecta\n");
            }
      } while (opcion != 's');
      return (EXIT_SUCCESS);
}
```

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

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

Por último, modificarás main para poder pasar argumentos por línea de comandos. La función `main()` es:

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

donde `argc` indica el número de argumentos y `argv` es una matriz de cadenas de texto.
Si desde la consola ejecutas el programa (cuyo nombre es `ejecutable`) enviando como argumentos de entrada el número de filas y columnas (en este ejemplo 2 y 3) y los valores a almacenar:

```
./ejecutable 2 3 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 9, y esas cadenas se almacenan en el vector `argv[]` de tamaño 9:

`argv[0]`	`"./ejecutable"`	Nombre del programa
`argv[1]`	`"2"`	Número de filas: 2
`argv[2]`	`"3"`	Número de columnas: 3
`argv[3]`	`"3.1"`	Fila 1 (3 columnas)
`argv[4]`	`"4.6"`	
`argv[5]`	`"3.2"`	
`argv[6]`	`"1.6"`	Fila 2 (3 columnas)
`argv[7]`	`"2.1"`	
`argv[8]`	`"0.5"`	

Para poder transformar de cadenas de texto (que es lo que contiene `argv[i]`) a `int` para el caso de los tamaños de la matriz, es necesario usar la función `atoi(cadena de texto)` y para pasar a `float` los valores a almacenar en la matriz, es necesario utilizar la función `atof(cadena de texto).` Estas funciones devuelve el número correspondiente a dicha cadena en formato entero o en formato real, respectivamente. La matriz se debe almacenar en filas, de tal manera que en caso del ejemplo tendríamos conceptualmente una matriz de tamaño 2x3 de la forma $\begin{pmatrix} 3.1 & 4.6 & 3.2 \\ 1.6 & 2.1 & 0.5 \end{pmatrix}$

Recuerda que es necesario validar el número de argumentos. Como estás tratando con matrices, el número de argumentos que se pasen al ejecutable correspondientes a los valores reales a almacenar en la matriz debe ser el producto de los dos primeros (filas x columnas) + 3 (que corresponden al nombre del ejecutable, filas y columnas). En el caso anterior, `argc` debe ser 3 + 2x3, o sea, 9.

Esta forma de ejecución debes probarla desde la ventana de comandos, ya que los IDE no permiten enviar argumentos a main.

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

Práctica 1.2: ampliación de la funcionalidad del TAD (versión 0):

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 **matrizdinamica**.

Función `liberar()`

En este momento podemos plantear una nueva pregunta. Imaginemos que ejecutamos nuestro programa introduciendo una matriz **`m1`** con 4x3 componentes y que a continuación introducimos una nueva matriz con 2x2 componentes.

¿Qué sucede con la memoria que habíamos reservado para la primera matriz? No la hemos liberado y, al mismo tiempo, hemos perdido la referencia a dicha memoria, pues con la segunda definición de la matriz m1 apuntamos a **otra** zona de memoria con 2x2 posiciones consecutivas.

La solución a este problema es liberar la memoria ocupada por la matriz previa 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 de la matriz (esto es, es incorrecto liberar una matriz si antes no la 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 creada la matriz (de otro modo, tendríamos un error de ejecución pues el programa intentaría liberar unas posiciones de memoria no reservadas).

Para escribir esta función puedes basarte en la función `destruirMatriz.c` que has corregido para la práctica P0_E6, pero ten en cuenta que en esa práctica no se trabajaba con TADs opacos, por tanto puede cambiar la forma de hacer referencia al TAD `matrizD`.

Función `recuperar()`

La función `recuperar()` recibirá como argumentos una determinada `matrizD` y la posición (fila y columna) y DEVOLVERÁ al programa principal el valor real (`float`) 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_E6.

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 "matriz" usando TAD

Función `prodescalar()`

La función `prodescalar()` recibirá como argumento un escalar y una determinada `matrizD` y DEVOLVERÁ al programa principal la matriz resultante de multiplicar el escalar por cada elemento de la matriz.

Funciones `nfilas()` y `ncolumnas()`

Es necesario introducir dos funciones en el TAD matriz dinámica que nos permitan obtener el tamaño actual de la matriz, y que así pueda ser utilizado por procedimientos auxiliares que operen con las matrices.

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

Como primer paso para completar el programa, vamos a añadir la función `imprimir()` que imprime la matriz e invocarla justo antes del break; del caso 'a'. Para acceder a las componentes de la matriz usaremos la función del TAD `recuperar()` y para acceder a su tamaño usaremos las funciones `nfilas()` y `ncolumnas()` escritas anteriormente, ya que al ser un tipo opaco no podemos hacer m→nfils ni m→ncols.

NOTA: La función `imprimir()` NO debe formar parte del TAD matriz dinámica, ya que asume un determinado tipo de datos para realizar las operaciones. Se debe añadir como procedimiento adicional dentro del fichero `main.c` del programa.

Entregables

Deberás entregar por el Campus Virtual el ejercicio correspondiente a la práctica 1.2. 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 **apellido1_apellido2.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.

Práctica 1: creación y manipulación de la Estructura de Datos "matriz" 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 tamano)`**: 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 (**`tamano`** 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.