Ejercicio 1.- Generación de vectores con memoria dinámica con malloc

Este ejercicio consiste en reescribir el programa resuelto en el fichero `malloc1.c`, de manera que, en vez de tener todo el código en un único fichero, y en la función `main()`, se escriben sendas funciones que realicen las tareas más importantes. Se debe escribir el código correspondiente a las funciones que se describen a continuación.

El proyecto constará de tres ficheros. El fichero `main.c` queda como sigue:

```
#include <stdio.h>
#include "vectorDinamica.h"
#include "vectorAux.h"
#define N 5

int
main(void) {
    int vx[N] = {0};
    int i, errNum;
    int *ptr;

    puts("VECTOR CREADO CON MALLOC");
    puts("========================");
    puts("");
    puts("Estudio de las posiciones de memoria donde se almecenan los datos
estáticos");
    puts("y las variables generadas con reserva de memoria dinámica.");
    puts("");

    if ((ptr = creaVectorInt(N, &errNum)) == NULL )
        return errNum;
    else {
        puts ("Variables estáticas:");
          puts ("    int vx[N] = {0};  N=5");
          puts ("    int i;           ");
          puts ("    int *ptr;        ");
          puts ("                     ");
          printf("Dirección último elemento de vx: %p \n", vx+(N-1));
          printf("Dirección comienzo de vx.......: %p \n", vx);
          printf("Dirección de i.................: %p \n", &i);
          printf("Dirección de ptr...............: %p \n", &ptr);

          printf("Dirección último elemento vector dinámico ptr: %p \n",
                                                          ptr+(N-1));
          printf("Dirección comienzo vector dinámico ptr........: %p \n",
                                                          ptr);

          printf("\n\n");
          printf("Carga valores aleatorios:\n");

          if (!cargarValoresAleatorios(ptr, N, 10, 100))
            mostrarValores(ptr, N);
          if (!cargarValoresAleatorios(vx, N, 10, 100))
```

```
            mostrarValores(vx, N);

        printf("\n\n");
        printf("Carga valores teclado:\n");
        if (!cargarValoresTeclado(ptr, N))
            mostrarValores(ptr, N);
        if (!cargarValoresTeclado(vx, N))
            mostrarValores(vx, N);
        free(ptr);
    }

    return 0;
}
```

Creación de un vector de números enteros con memoria dinámica

El fichero `vectorDinamica.c` contiene la función que crea un vector de enteros utilizando memoria dinámica:

```
int *creaVectorInt(size_t tam, int *codigoError);
```

esta función crea, de forma dinámica, un vector de enteros de tamaño `tam` y devuelve la dirección de memoria del primer *byte* del bloque reservado si la función tiene éxito, y `NULL` en caso contrario. Para controlar el correcto funcionamiento se incluye, además, el parámetro `codigoError` que se utiliza para devolver un valor entero (paso por referencia) indicando:

- **-1** Error. Si el valor del parámetro `tam` es incorrecto (menor o igual que cero)
- -2 Error. Se ha producido un error en la invocación a la función de reserva dinámica de memoria
- **0** Éxito. Además la función devuelve, como ya se ha indicado anteriormente, la dirección de memoria del primer *byte* del bloque reservado

El alumno debe escribir el correspondiente fichero `vectorDinamica.h` con el prototipo de la función y correctamente protegido contra dobles inclusiones (`#ifndef … #endif`).

Nota: la funcioón `creaVectorInt` permite crear de forma dinámica un vector de enteros. Observar que con pocas modificaciones se pueden codificar funciones para crear vectores de reales, de cadenas o de tipos definidos por el usuario (por ejemplo `tipoEmpleado`).

El fichero `vectorAux.c` contiene las funciones que permiten la carga (tanto con valores aleatorios como desde teclado) y la visualización de vectores de enteros.

Carga de un vector con valores aleatorios

```
int cargarValoresAleatorios(int *vector,
                            size_t tam,
                            int rangoInf,
                            int rangoSup);
```

Función que asigna a los elementos del vector de enteros referenciado por el parámetro `vector`, y de tamaño `tam`, valores aleatorios comprendidos en el intervalo indicado por los parámetros `rangoInf` y `rangoSup`. Devuelve un valor entero que indica si la función se ejecuta con éxito o si ha habido algún error en el proceso según el siguiente criterio:

- **-1** Si el valor del parámetro vector no es válido (referencia `NULL`)
- **-2** Si el rango de valores no es válido (`rangoInf` >= `rangoSup`)
- **0** Si se ejecuta correctamente

Carga de un vector con valores desde teclado

```
int cargarValoresTeclado(int *vector, size_t tam);
```

Función que solicita al usuario valores enteros con los que ir cargando los elementos del vector referenciado por el parámetro `vector`, y de tamaño `tam`. Devuelve un valor entero que indica si la función se ejecuta con éxito o si ha habido algún error en el proceso según el siguiente criterio:

- **-1** Si el valor del parámetro vector no es válido (referencia `NULL`)
- **0** Si se ejecuta correctamente

Visualización de un vector de enteros

```
int mostrarValores(int *vector, size_t tam);
```

Función que muestra el contenido del vector de enteros referenciado por el parámetro `vector`, y de tamaño `tam`. Devuelve un valor entero que indica si la función se ejecuta con éxito o si ha habido algún error en el proceso según el siguiente criterio:

- **-1** Si el valor del parámetro vector no es valido (referencia `NULL`)
- **0** Si se ejecuta correctamente

El alumno debe escribir el correspondiente fichero `vectorAux.h` con los prototipos de las funciones y correctamente protegido contra dobles inclusiones (`#ifndef … #endif`).

Para la correcta compilación del proyecto se debe crear el correspondiente fichero Makefile y usar la orden make, tal y como se ha explicado en clase. Opcionalmente se podrá utilizar alguno de los IDE que se han presentado.

Ejercicio 2.- Matrices de registros y matrices de punteros a registros utilizando memoria dinámica

Se debe reescribir el Ejercicio 1 de los propuestos en el PDF Enunciados ejercicios adicionales, en el tema 1 de la asignatura. El título del ejercicio es "Matrices de registros y de punteros a registros". La modificación consiste en sustituir la matriz estática de registros por una matriz dinámica de registros (ver Figura 1.), y la matriz estática de punteros a registros por una matriz dinámica de punteros a registros reservados dinámicamente (ver Figura 2).

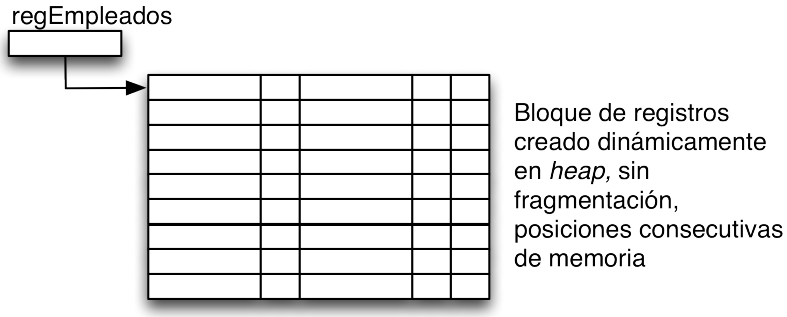

Figura 1. Vector de registros.

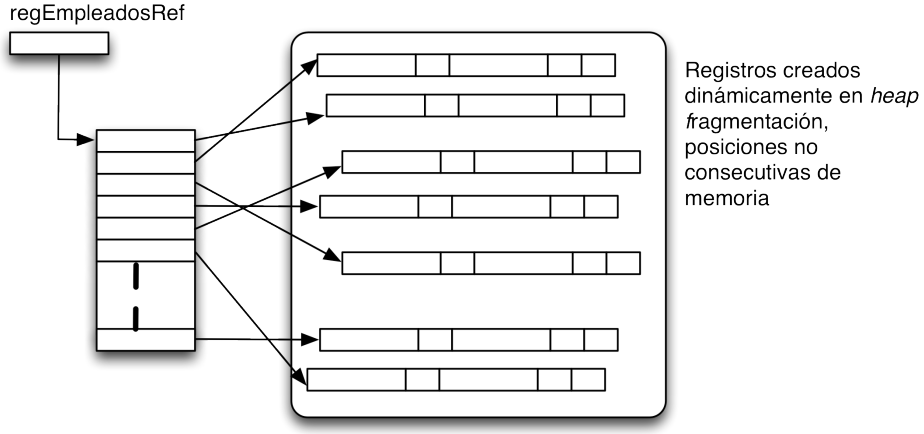

Figura 2. Vector de punteros a registro que referencian registros creados dinámicamente.

Se crearán dos ficheros, uno con la función `main()`, tal y como queda a continuación, y el fichero `empleado.c` con las funciones de creación, manejo y gestión de vectores de registros y vector de punteros a registros.

El fichero main.c queda como sigue:

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

```
#include <time.h>
#include "empleado.h"

int
main(void)
{
    tipoEmpleado *regEmpleados;
    tipoEmpleado **regEmpleadosRef;
    int  numEmpleados,errNum;
    clock_t tiempoInicial, tiempoFinal;

    printf("\nNúmero de empleados: ");
    scanf("%d%*c",&numEmpleados);
    printf("\nEl tamaño de un registro es: %ld",sizeof(tipoEmpleado));
    regEmpleados = crearVectorRegistros(numEmpleados, &errNum);
    if (!cargarRegistrosAleatorios(regEmpleados, numEmpleados)) {
        printf("\n\nValores no clasificados vector registros: \n");
        mostrarRegistros(regEmpleados, numEmpleados);
    }
    tiempoInicial = tiempoFinal = clock ();
    ordenarRegistrosApellido(regEmpleados, numEmpleados);
    tiempoFinal = clock ();

    printf("\n\nValores clasificados vector de registros: \n");
    printf("Tiempo tardado: %f\n",
                  (tiempoFinal-tiempoInicial)/(double)CLOCKS_PER_SEC);
    mostrarRegistros(regEmpleados, numEmpleados);

    regEmpleadosRef = crearRegistrosRef(numEmpleados, &errNum);
    if (!cargarRegistrosAleatoriosRef(regEmpleadosRef, numEmpleados)) {
        printf("\nValores no clasificados vector punteros a registros: \n");
        mostrarRegistrosRef(regEmpleadosRef, numEmpleados);
    }
    tiempoInicial = tiempoFinal = clock ();
    ordenarRegistrosRefApellido(regEmpleadosRef, numEmpleados);
    tiempoFinal = clock ();

    printf("\n\nValores clasificados vector punteros a registros: \n");
    printf("Tiempo tardado: %f\n",
                  (tiempoFinal-tiempoInicial)/(double)CLOCKS_PER_SEC);
    mostrarRegistrosRef(regEmpleadosRef, numEmpleados);

    free(regEmpleados);
    if (!liberarMemRegistrosRef(regEmpleadosRef, numEmpleados))
        regEmpleadosRef = NULL;

    return 0;
}
```

El fichero `empleado.h` queda como sigue:

```
#ifndef __EMPLEADO_H
#define __EMPLEADO_H

typedef struct empleados {
    char apellidos[30];
```

```
    char nombre[15];
    int  matricula;
    char bufferRelleno[5];
} tipoEmpleado;

tipoEmpleado *crearVectorRegistros(int numEmpleados, int *errNum);
int cargarRegistrosAleatorios(tipoEmpleado * empleados, int numEmpleados);

tipoEmpleado **crearRegistrosRef(int numEmpleados, int *errNum);
int liberarMemRegistrosRef(tipoEmpleado **empleadosRef, size_t numEmpleados);
int cargarRegistrosAleatoriosRef(tipoEmpleado **empleados, int numEmpleados);

void cargarUnRegistro(tipoEmpleado * empleado);

void mostrarRegistro(tipoEmpleado empleado);
void mostrarRegistrosRef(tipoEmpleado ** refsEmpleados, int numEmpleados);
void mostrarRegistros(tipoEmpleado * empleados, int numEmpleados);

void ordenarRegistrosRefApellido(tipoEmpleado ** refsEmpleados,
                                 int numEmpleados);
void ordenarRegistrosApellido(tipoEmpleado * empleados, int numEmpleados);

#endif
```

En el fichero `empleado.c` se implementarán todas las funciones declaradas.

Creación del vector de registros con memoria dinámica

```
tipoEmpleado *crearVectorRegistros(int numEmpleados, int *errNum);
```

Esta función crea, de forma dinámica, un vector de registros de tipo `tipoEmpleado` y de tamaño `numEmpleados` y devuelve la dirección de memoria del primer *byte* del bloque reservado si la función tiene éxito, y `NULL` en caso contrario. La estrategia seguida es la que se muestra en la Figura 1. Para controlar el correcto funcionamiento se incluye, además, el parámetro `errNum` que se utiliza para devolver un valor entero (paso por referencia) indicando:

- **-1** Error. Si el valor del parámetro `numEmpleados` es incorrecto (menor o igual que cero)
- -2 Error. Se ha producido un error en la invocación a la función de reserva dinámica de memoria
- **0** Éxito. Además la función devuelve, como ya se ha indicado anteriormente, la dirección de memoria del primer *byte* del bloque reservado

Carga del vector de registros con valores aleatorios

```
int cargarRegistrosAleatorios(tipoEmpleado * empleados, int numEmpleados);
```

Esta función carga los elementos del vector de registros referenciado por el parámetro `empleados` con valores aleatorios, y de tamaño `numEmpleados`, devolviendo un valor entero que indica si ha tenido o no, según los siguientes criterios:

- **-1** Si el valor del parámetro vector no es válido (referencia `NULL`)
- **0** Si se ejecuta correctamente

Creación del vector de punteros a registro con memoria dinámica

```
tipoEmpleado **crearRegistrosRef(int numEmpleados, int *errNum);
```

Esta función crea, de forma dinámica, un vector de punteros a registro de tipo `tipoEmpleado` y de tamaño `numEmpleados`, y a continuación reservará el mismo número de bloques de memoria de tamaño registro `tipoEmpleado`, almacenando sus referencias en las celdas del vector de punteros. La función debe devolver la dirección de memoria del primer *byte* del bloque de punteros reservado si tiene éxito, y `NULL` en caso contrario. La estrategia seguida es la que se muestra en la Figura 2, el bloque de punteros almacena las direcciones al primer *byte* de cada registro reservado. Para controlar el correcto funcionamiento se incluye, además, el parámetro `errNum` que se utiliza para devolver un valor entero (paso por referencia) indicando:

- **-1** Error. Si el valor del parámetro `numEmpleados` es incorrecto (menor o igual que cero)
- -2 Error. Se ha producido un error en la invocación a la función de reserva dinámica de memoria del bloque de punteros a registro
- -3 Error. Se ha producido un error en la invocación a la función de reserva dinámica de memoria para alguno de los bloques que almacenará un registro del tipo `tipoEmpleado`, y cuyas referencias se almacenan en el bloque de punteros. En caso de que se produzca este error se debe liberar la memoria reservada con éxito para todos los bloques de los registros anteriores, además del propio bloque de punteros a registro, devolviendo un `NULL`
- **0** Éxito. Además la función devuelve, como ya se ha indicado anteriormente, la dirección de memoria del primer *byte* del bloque de punteros a registro reservado

Liberación de bloques de memoria dinámica reservada para los registros y el vector de punteros a registro

```
int liberarMemRegistrosRef(tipoEmpleado **empleadosRef, size_t numEmpleados);
```

Esta función libera toda la memora reservada, según la estrategia de la Figura 2, primero los bloques de memoria que almacenan los registros, y después el bloque de memoria que almacena los punteros a registro. La función devolverá un valor entero indicando:

- **-1** Error. Si el valor del parámetro `empleadosRef` es incorrecto (referencia `NULL`)
- **0** Éxito. Libera correctamente toda la memoria reservada

Carga los registros referenciados por punteros del vector de punteros con valores aleatorios

```
int cargarRegistrosAleatoriosRef(tipoEmpleado **empleados, int numEmpleados)
```

Esta función carga los registros referenciados por los punteros almacenados en el bloque de punteros, de tamaño `numEmpleados`, y referenciado por el parámetro `empleados` con valores aleatorios devolviendo un valor entero que indica si ha tenido o no, según los siguientes criterios:

- **-1** Si el valor del parámetro vector no es válido (referencia `NULL`)
- **0** Si se ejecuta correctamente

Nota importante: las funciones `cargarRegistrosAleatoriosRef` y `cargarRegistrosAleatorios` se pueden implementar usando una genérica `cargarRegistroAleatorio` como la presentada en clase de teoría, para evitar repetir código fuente.

Resto de funciones

El resto de funciones se puede reutilizar directamente (copiar y pegar en este caso) de la práctica implementada en el Tema 1. Este aspecto es de gran interés porque muestra la versatilidad de la sintaxis del lenguaje, permitiendo la reutilización de funciones creadas para procesar matrices estáticas en programas en los que se utilizan matrices dinámicas.

Ejercicio 3.- Matrices bidimensionales utilizando memoria dinámica

En este caso se debe partir de los tres ejercicios resueltos, cuyos proyectos están disponibles en la página de la asignatura, para la creación y procesamiento de matrices bidimensionales utilizando memoria dinámica:

- dinamicaMatrices: código fuente de un proyecto donde las matrices bidimensionales se implementan usando un bloque de bytes que está referenciado por un puntero del tipo base de los elementos a almacenar (tipoBase *pt; donde tipoBase es float o int)
- dinamicaMatricesBidimensionales: código fuente de un proyecto análogo al anterior (el main es idéntico y las funciones tienen el mismo prototipo), donde las matrices bidimensionales se implementan como un único bloque de bytes referenciado por un puntero de tipo fila o vector de tipoBase (tipoBase (*pt)[COL], donde la constante COL es el tamaño de la fila y tipoBase vuelve a ser float o int)
- dinamicaMatricesPunteros: código fuente de un proyecto análogo a los anteriores, donde las matrices bidimensionales se implementan como un único bloque de bytes referenciado por un puntero a puntero de tipoBase que contiene los punteros a los bloques de memoria (reservados dinámicamente también) que representan las filas de la matriz bidimensional (tipoBase **pt, con tipoBase float o int)

En estos proyectos la única función de procesamiento de matrices bidimensionales implementada es la de sumar dos matrices. El código se explica en clase de teoría. Ahora se propone al alumno que añada las siguientes funciones:

Multiplicación de matrices bidimensionales

```
matIntRef multiplicarMatInt(matIntRef a, matIntRef b, intRef errNum);
```

Para cada uno de los proyectos se debe añadir esta función que realizará la misma tarea en los tres casos. Recibe dos matrices (en cada caso del tipo correspondiente al proyecto en cuestión) y devuelve el resultado del producto de ambas matrices. La matriz resultado debe ser creada

dinámicamente y de las dimensiones adecuadas. Se puede seguir el esquema que se presenta en la función `sumarMatInt()` entregada. El tercer parámetro, `errNum`, es un valor entero que se pasa por referencia y que sirve para indicar si se ha tenido éxito o no en la tarea, según los siguientes criterios:

- Valores negativos de **-1** a -4. Error controlado por la invocación a las funciones `fallaMatriz` (se usará para comprobar la validez de `a` y `b`), y que indica fallo en los punteros o en las dimensiones almacenadas en la estructura, y `crearMatInt` que indica fallo en las reservas de memoria al crear la matriz resultado
- **-5** Error. El número de columnas de la matriz a no coincide con el número de filas de la matriz b, por lo que no se puede realizar el producto. La función devuelve NULL
- **0** Éxito. Todos los parámetros son correctos, se ha reservado memoria con éxito para la matriz resultado y se ha realizado el producto de ambas matrices correctamente. Se devuelve la dirección de memoria al primer byte del bloque de memoria que almacena la estructura que representa la matriz resultado de la operación

Nota importante: obsérvese que no se distingue entre los tres casos porque los prototipos de las funciones son idénticos en los tres proyectos, lo que cambia es la definición de los tipos `matInt` y `matIntRef` que representan la estructura que soporta la matriz y el punteros a dicha estructura respectivamente.

Obtención de la columna que contiene el valor máximo de la matriz bidimensional

```
int *obtenerColumnaMaxMatInt(matIntRef mat, intRef errNum);
```

Esta función recibe una matriz bidimensional, referenciada por el parámetro `mat`, ya cargada con valores en sus elementos y buscará, recorriendo dicha matriz bidimensional por columnas, el valor máximo de los elementos de la misma, y devolverá en un vector de enteros (en este caso, o de `float` para matrices del tipo `matFloatRef`) que se debe crear dinámicamente, la columna que contiene ese valor máximo, es decir, debe copiar en dicho vector todos los valores almacenados la columna localizada. En el caso de que existan varios valores iguales al máximo, deberá devolver la columna correspondiente al último encontrado. El segundo parámetro es un valor entero que se pasa por referencia para devolver un código de error según los siguientes criterios:

- Valores negativos de **-1** a -4. Error controlado por la invocación a las funciones `fallaMatriz` (se usará para comprobar la validez de `mat`), y que indica fallo en los punteros o en las dimensiones almacenadas en la estructura
- -5 Error en la reserva de memoria dinámica del vector que almacena los elementos de la columna localizada
- 0 Éxito. Tarea completada con éxito, la función devuelve la dirección de memoria al primer byte del bloque de memoria que representa el vector solución

Nota importante: obsérvese que no se distingue entre los tres casos porque los prototipos de las funciones son idénticos en los tres proyectos, lo que cambia es la definición de los tipos `matInt` y `matIntRef` que representan la estructura que soporta la matriz y el punteros a dicha estructura respectivamente.