Práctica 2: Gestión de Memoria Dinámica y Paso de Parámetros por dirección

.::1 Sesión::.

1. Objetivos

Los objetivos de esta práctica son:

- Comprender el concepto de puntero y la sintaxis de su uso en el lenguaje C.
- Utilizar el paso de parámetros a funciones por dirección por medio de punteros.
- Gestionar memoria dinámica para el manejo de vectores unidimensionales y bidimensionales.

2. Memoria dinámica y paso de parámetros por dirección

2.1.- Memoria dinámica

Hasta ahora, cuando declarábamos un array reservábamos una cantidad fija de memoria. Por ejemplo:

```
float x[50]; // Array de 50 float
char cadena[100];//array de caracteres
```

Pero, ¿qué ocurre si necesitamos una cantidad de memoria variable porque no conocemos el número de elementos máximo que tendrá el array?

Para ello, realizaremos una gestión dinámica de memoria, que se realizará mediante la función *`malloc (stdlib.h):`*

```
void *malloc(tamaño)
```

- *tamaño*: es el tamaño en bytes de la memoria que se quiere reservar.
- *void **: malloc devuelve un puntero a un tipo genérico.

Tamaño de la memoria

Para decidir cual va a ser el tamaño de la memoria que vamos a utilizar en la reserva, usamos **`sizeof.`**

Para reserva en arrays, nos fijaremos únicamente en el número total de elementos y no en el número de dimensiones.

`sizeof` admite como parámetro un determinado tipo de datos y devuelve la cantidad en bytes que necesitará para su almacenamiento en memoria.

Ejemplo:

sizeof(int) dará como resultado 2 bytes que es la representación de un entero en memoria

- *Ejemplo 1.- uso de reserva de memoria*

```
float *buffer; // declaramos un puntero
buffer = (float *) malloc(50*sizeof(float));
      //reservamos memoria para 50 float
```

Comprobación de que la memoria se ha reservado correctamente

Una vez hecha la reserva correspondiente de memoria, deberemos comprobar si dicha reserva se ha efectuado correctamente o no. Para ello nos haremos valer de la siguiente comparación:

```
if(buffer==NULL)
        return -1;
```

Si no se ha podido reservar la memoria, `malloc` devuelve NULL

Liberar la memoria

La memoria que hemos reservado dinámicamente, será más que aconsejable liberarla una vez que no la necesitemos porque no vamos a seguir trabando con ella. La forma para liberarla es la siguiente:

```
free(buffer);
```

- *Ejemplo 2.- uso de reserva y liberación de memoria*

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
      int i, num;
      int* buffer;

      printf(“Cuantos enteros desea almacenar?");
      scanf("%d", &num);

      buffer = (int*)malloc(num*sizeof(int)); //reservamos la memoria

      if(buffer==NULL) //Si no se ha podido reservar la memoria, malloc devuelve
            return -1; //el puntero NULL

      for(i=0; i<num; i++) {
            printf(“Elemento %d:\n", i);
            scanf(“%d", &buffer[i]);
      }

      free(buffer);//Liberamos la memoria reservada cuando no la necesitamos
      return 0;
}
```

2.2.- Paso de parámetros por dirección

Como ya se ha visto en el primer cuatrimestre, el paso de argumentos a una función se puede realizar de dos formas:

- *Paso por valor*: Se le pasa el valor de la variable. Se altera una copia de la variable, no la variable original.
- *Paso por dirección*: El contenido de la dirección puede ser cambiado. Se cambia la variable original.

C permite pasar la dirección de una variable como argumento de una función. Esto posibilita cambiar la variable dentro del cuerpo de la función (Paso por dirección)

El paso por dirección también permite a una función devolver más de un dato, en contraposición con *paso por valor*, en el que únicamente se podía devolver un solo dato con return.

A continuación se muestra un ejemplo de paso de parámetros por dirección a una función:

```
#include <stdio.h>
void func1(int u, int v);
void func2(int *pu, int *pv);
main ()
{
 int u=1;
 int v=3;
 printf("Antes de func1: u=%d, v=%d\n", u, v);
 func1(u, v);
 printf("Despues de func1: u=%d, v=%d\n", u, v);
 printf("Antes de func2: u=%d, v=%d\n", u, v);
 func2(&u, &v);
 printf("Despues de func2: u=%d, v=%d\n", u, v);
 system("PAUSE");
}

void func1(int u, int v)
{
 u=0;
 v=0;
 printf("Dentro de func2: u=%d, v=%d\n", u, v);
}
void func2(int *pu, int *pv)
{
 *pu=0;
 *pv=0;
 printf("Dentro de func2: *pu=%d, *pv=%d\n", *pu, *pv);
}
```

Lo que obtendremos a la salida, será:

```
Antes de func1: u=1, v=3
Dentro de func1: u=0, v=0
Después de func1: u=1, v=3
Antes de func2: u=1, v=3
Dentro de func2: *pu=0,*pv=0
Después de func2: u=0, v=0
```

3. Tareas a realizar

Esta práctica consiste en realizar un programa que realice las siguientes operaciones:

a) Intercambiar en una función los valores de dos variables enteras. A continuación se muestra un ejemplo de ejecución (en negrita aparecen los datos introducidos por el usuario):

```
Introduzca a: 2
Introduzca b: 8
Inicialmente: a = 2, b = 8
Tras intercambiar: a = 8, b = 2
```

b) Invertir un vector de números reales. Para ello en primer lugar se leerán números reales positivos y se almacenarán en un vector. Se pedirá por teclado el número de elementos a introducir.

```
Introduzca la dimension del vector:
Introduzca el vector:
3.45
5.34
7.89
Vector introducido: 3.45 5.34 7.89
Vector invertido: 7.89 5.34 3.45
```

c) Sumar dos matrices cuadradas. Se deberá solicitar la dimensión de las matrices de forma que se reservará sólo la memoria necesaria para almacenar los elementos de las matrices (memoria dinámica).

d) Multiplicar una matriz por un escalar entero.

e) Crear un menú para elegir cada una de las opciones que se pueden ejecutar en el programa.

4. Programa a entregar

1.- Mediante el software DevCpp, realizar un programa que contenga todas las funciones descritas a continuación.

a) Función **void Intercambiar(int *p1, int *p2);** que debe intercambiar el valor de dos valores enteros pasados como parámetros (paso por dirección).

b) Para invertir un vector de números reales deberán realizarse los siguientes pasos:

i) Reservar memoria estática para el vector de reales en el que se guardarán los números introducidos por el usuario. Se considerará que el número máximo de elementos del vector es 50.

ii) Definir la función **void LeerVector(float *v, int n)** que leerá números reales y los almacenará en el vector *v* pasado como parámetro hasta la dimension n.

iii) Reservar la memoria necesaria para el vector de reales en el que se guardarán los elementos de forma invertida.

iv) Definir la función **void InvertirVector(float *v, float *inv, int n)** que copia en el vector *inv* los *n* elementos de *v* de forma inversa.

v) Definir la función **void MostrarVector(float *v, int n)** para mostrar los elementos del vector *v* pasado como parámetro. El parámetro *n* indica el número de elementos del vector.

vi) Una vez efectuadas todas las operaciones es necesario liberar la memoria previamente reservada.

c) **Función SumarMatrices(float **ma, float **mb, float **sum, int n)** que se utilizará para realizar la suma de dos matrices. Se recomienda seguir los siguientes pasos:

i) Reservar memoria de forma dinámica para tres matrices: las dos matrices a sumar más la matriz donde se almacenará el resultado. Para ello previamente será necesario conocer la dimensión de las matrices, que será introducida por teclado.

ii) Crear la función **void LeerMatriz(float **m, int n)** para leer una matriz por teclado de dimensión *n*.

iii) Definir la función **void SumarMatrices(float **ma, float **mb, float **sum, int n)** que realizará la suma de las matrices *ma* y *mb* y almacenará el resultado en la matriz *sum*.

iv) Definir la función **void MostrarMatriz(float **m, int n)** que muestra en pantalla la matriz *m* de dimensión *n*.

v) Eliminar la memoria de las matrices previamente reservada.

d) Función **void MatrizPorEscalar(float **m,float **m3,int e,int n),** que realizará la multiplicación de una matriz por un escalar. Se recomienda:

i) Reservar memoria de forma dinámica para dos matrices: la matriz a multiplicar por el escalar, más la matriz donde se almacenará el resultado. Para ello previamente será necesario conocer la dimensión de las matrices, que será introducida por teclado.

ii) Llamar a la función **void LeerMatriz(float **m, int n)** para leer una matriz por teclado de dimensión *n*.
iii) Crear la función **void MatrizPorEscalar(float **m,float **m3,int e,int n)**
iv) Mostrar la matriz resultado mediante la función **void MostrarMatriz(float **m, int n)**

e) Función principal del programa (**main**).

i) El programa deberá mostrar el siguiente menú, ejecutándose cada una de las funciones antes definidas según indique el usuario.

```
Practica 1

1.- Intercambiar dos numeros
2.- Invertir un vector de números reales
3.- Sumar dos matrices cuadradas
4.- Multiplicar una matriz por un escalar
5.- Salir

Elija una opcion:
```

Notas:

- Para la asignación dinámica de memoria debe utilizarse la función **void *malloc (tamaño)** (cabecera *stdlib.h*). *tamaño* es el número de bytes de memoria que se quiere reservar. La función devuelve un puntero vacío (*void **) al espacio de memoria reservado. Este puntero se convertirá al puntero del tipo que se desee. En caso de que no haya memoria disponible la función devuelve NULL.

- Para liberar memoria reservada dinámicamente debe utilizarse la función **void free(void *puntero)** (cabecera *stdlib.h*). Esta función libera la memoria a la que apunta *puntero* previamente reservada.