Ampliación de teoría de punteros y asignación dinámica de memoria

1.- Puntero a puntero

Un puntero almacena la dirección de un objeto, y nada nos hace pensar que ese objeto no pueda ser otro puntero. Por tanto, es posible declarar un puntero que apunta a otro puntero.

La notación de puntero a puntero se realizará mediante un doble asterisco: **

A continuación se muestra un ejemplo donde vemos como se interrelacionan entre sí variables, punteros y punteros a punteros:

```
int n = 1;
int *p1 = &n;
int **p2 = &p1;
```

En la primera línea, asignamos un valor a la variable n y apuntamos con p1 a la dirección de dicha variable. A continuación guardamos la dirección de p1 en p2.

Declaración	Dirección de memoria aleatoria	Valor almacenado en la dirección de memoria
`int n = 1;`	122f HEX	1
`int *p1 = &n;`	133f HEX	122f HEX
`int **p2 = &p1;`	144f HEX	133f HEX

- Si imprimimos por pantalla ‘p2’, imprimiremos la dirección del propio puntero p2
- Si imprimimos por pantalla ‘*p2’, imprimiremos la dirección almacenada en p2, es decir, ‘133f’
- Si imprimimos por pantalla ‘**p2’, imprimiremos el valor almacenado en *p1, es decir, 1

Ejercicio 1

Según el código siguiente:

```
#include <stdio.h>
main(){
  int **p2,*p1,i;

  i=1;
  p1 = &i;
  p2 = &p1;
}
```

¿qué salida obtendremos en cada unos de los apartados siguientes?¿Por qué?:

```
a. printf ("La direccion de i es: %X\n",&i);
b. printf ("El valor de i es: %d\n",i);
c. printf ("El valor de i es: %d\n",*i); // error pq no es un
   puntero
d. printf ("La direccion de p1 es: %X\n",&p1);
e. printf ("El puntero p1 apunta a: %X\n",p1);
f. printf ("El valor de p1 es: %d\n",p1); // error
g. printf ("El valor de p1 es: %d\n",*p1); // correcto
h. printf ("La direccion de p2 es: %X\n",&p2);
i. printf ("El puntero p2 apunta a: %X\n",p2);
j. printf ("El valor de p2 es: %d\n",p2); // error
k. printf ("El valor de p2 es: %d\n",*p2); //error
l. printf ("El valor de p2 es: %d\n",**p2); // correcto
```

2.- Punteros y arrays

¿Qué ocurre cuando creamos un array? La respuesta es bien sencilla. Cuando creamos un array, lo que hacemos es generar un puntero (más bien una constante de puntero), con igual nombre, que apunta a la dirección del primer elemento del array.

Por ejemplo:

```
int array[3], *puntero;
puntero=array;
```

Como decíamos, ‘puntero’ apuntará a la dirección del primer elemento de ‘array’, y si quisiéramos acceder al segundo elemento de ‘array’, deberíamos escribir:

array[1] ó *(puntero+1)

Si llamamos a ‘array’ sin índice, devolveremos la dirección de comienzo del array, es decir, el primer elemento.

Ejercicio 2

Si quisiéramos declarar un vector de enteros y asignar valores, de todas estas declaraciones y asignaciones, ¿cuáles serían correctas?

```
#include <stdio.h>
main() {
      int array;
      array[3]={1,2,3};
      int array[3]={1,2,3};
      array={1,2,3};
      array{1,2,3};
      array[]={1,2,3};
}
```

Ejercicio 3

Si nos fijamos en el siguiente programa:

```
#include <stdio.h>
main() {
      int array[3]={1,2,3};
      int *puntero;
      printf("Debería imprimir esto %d \n",array[0]);
      printf("Debería imprimir esto %d \n",*array);
      printf("Debería imprimir esto %d \n",array);
}
```

¿Cuáles de los printf() que aparecen son correctos?

Ejercicio 4

Si queremos apuntar con un puntero a la dirección del primer elemento del array llamado “array”, ¿qué opción de todas las que aparecen sería la correcta?:

```
#include <stdio.h>
main() {
      int array[3]={1,2,3};
      int *puntero;

      puntero=array;
      puntero=array[];
      puntero=*(array);
      puntero=*array;
      &puntero=array;

}
```

Ejercicio 5

Si nos fijamos en el siguiente programa:

```
#include <stdio.h>
main() {
      int array[3]={1,2,3};
      int *puntero;
      puntero=array;
      printf ("Pero imprime esto %d\n",puntero);
      printf ("Pero imprime esto %d\n",*puntero);
}
```

¿Qué printf() es correcto?

3.- Arrays de punteros

Los punteros también pueden estructurarse en arrays. Por ejemplo, si deseáramos declarar un array de punteros a enteros de 3 elementos, usaríamos la siguiente sintaxis:

```
int *puntero[3],a=1;
```

Y ahora, si quisieramos asignar la dirección de una variable entera a uno de los elementos del array de punteros, por ejemplo al elemento 1, utilizaríamos:

```
puntero[1]=&a;
```

Para acceder al valor de 'a', debemos hacerlo mediante:

```
*puntero[1];
```

Paso de arrays de punteros a una función

Debemos llamar a la función con el nombre del array, sin índices. De esta forma, le pasaremos a la función el puntero que apuntaría al array. Lo que hacemos es pasarle un puntero a un array de punteros a enteros (No confundir con el paso de un puntero a enteros)

4.- scanf

Cada argumento de scanf debe ser un puntero:

- Para variables que no sean string:
    - *Enteros*

      ```
      int x;
      scanf ("%d", &x);
      ```

    - *Char*

      ```
      char letra ;
      scanf ("%c", &letra);
      ```

- Para arrays de variables que no sean string
    - *Arrays unidimensionales*

      ```
      int entero[21];
      scanf("%d",entero);  // solo recoge el 1er elemento del vector
      ```

      es equivalente a:

      ```
      scanf("%d",&entero[0]);
      ```

Ejercicio 6

Fíjese en el siguiente programa:

```
#include <stdio.h>
main() {
 int entero[2]={10,20};
 printf("Antes: %d\n", entero[0]);
 scanf("%d",entero);
 scanf("%d",&entero);
 scanf("%d",entero[0]);
 scanf("%d",&entero[0]);
 printf("Despues: %d", entero[0]);
}
```

Si quiseramos acceder al elemento 0 del array y modificarlo, ¿cual de los métodos de scanf() es el correcto para realizar la operación?

- *Arrays multidimensionales*

```
float **matriz;
scanf("%f",&matriz[i][j])

float **matriz;
scanf("%f",*matriz) // accederíamos al primer elemento de la primera fila

scanf("%f",matriz); // Error
```

Error porque accederíamos a la dirección de memoria del puntero a puntero, es decir, si tenemos que **matriz tiene asignado en memoria la dirección fff2, con esa instrucción accederíamos a esa dirección, y no podríamos escribir ningún valor. Necesitaríamos acceder a la dirección de la primera fila que se haría accediendo mediante *matriz.

- Para variables de cadenas de caracteres:

```
char palabra[21];
scanf("%s", palabra);
```

5.- Asignación dinámica de memoria en Arrays Unidimensionales

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.

<u>Tamaño de la memoria</u>

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

- *<u>Ejemplo 1.- uso de reserva de memoria</u>*

```
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;
}
```

6.- Asignación dinámica de memoria en arrays multidimensionales

1) Método de acceso por índices

a) Declaración de la matriz

Primero se define un puntero de punteros:

```
float **mat1;
```

b) Reserva de memoria

A continuación, se reserva memoria para los punteros:

```
mat1=(float **) malloc (filas*sizeof(float *));
```

y, por último, para cada puntero, reservamos memoria para sus elementos, que coincidirá con el número de columnas

```
for(i=0;i<filas;i++)
      mat1[i]=(float *) malloc(columnas*sizeof(float));
```

c) Asignación de valores

La asignación de valores a los elementos, se realizará por medio de índices:

```
for(i=0;i<filas;i++)
      for(j=0;j<columnas;j++)
      {
            printf("mat1[%d][%d]= ",i,j);
            scanf("%f",&mat1[i][j]);
      }
```

d) Acceso a los elementos de la matriz

Por tanto, para acceder al elemento (2,5), accederíamos mediante:

```
mat1[2][5];
```

e) Liberación de memoria

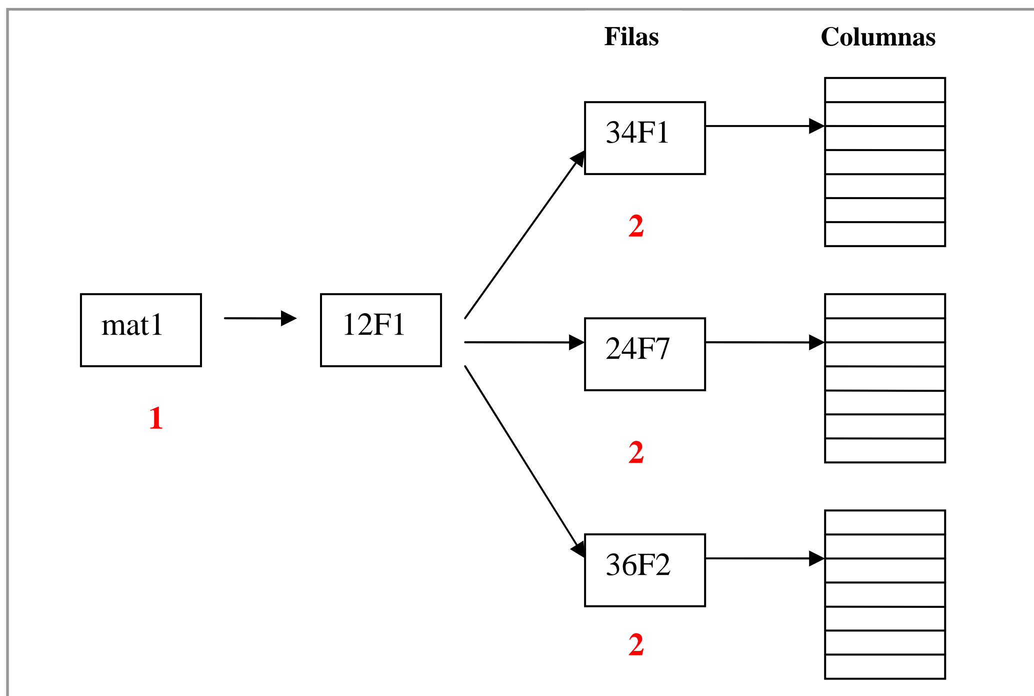
```
for(i=0;i<filas;i++)
   free(*(mat1+i));  // se libera cada uno de los vectores
free(mat1);          // se libera el puntero mat1
```

f) Aspectos a tener en cuenta

`*(*(mat1+i)+j)` es equivalente a **`mat1[i][j]`**
`(*(mat1+i)+j)` es equivalente a **`&mat1[i][j]`**

g) Ejemplo gráfico

Si nos fijamos en la figura:



Podemos comprobar:

1) como se le asigna la dirección de memoria al puntero de punteros mat1:

```
mat1=(float **) malloc (filas*sizeof(float *));
```

2) y, como para cada puntero (fila), reservamos memoria para sus elementos, que coincidirá con el número de columnas

```
for(i=0;i<filas;i++)
    mat1[i]=(float *) malloc(columnas*sizeof(float));
```

2) Método de acceso mediante aritmética de punteros

a) Declaración de la matriz

```
float *mat1;
```

b) Reserva de memoria

```
mat1=(float *)malloc(filas*columnas*sizeof(float *));
```

c) Asignación de valores

```
for(i = 0 ; i < filas ; i++)
      for(j = 0 ; j < columnas ; j++)
      {
              printf("mat1[%d][%d]= ",i,j);
              scanf("%f",mat1 + (i * columnas + j));
      }
```

d) Acceso a los elementos de la matriz

Para acceder a un elemento de la matriz, deberemos hacer:

*(mat1+(i*columnas+j))

Donde `mat1` es el puntero, `i` es el contador de filas, `j` el contador de columnas y `columnas` la variable que indica el número de columnas que tenemos.

Ejemplo:

Si deseamos ir a la fila 3, columna 2 y la matriz tiene 5 columnas, entonces ese elemento es el: 3*5+2 = 17. Por tanto, ese elemento se encontrará en la posición 17.

e) Liberación de memoria

```
free(mat1);
```

f) Aspectos a tener en cuenta

`*(mat1+(i*columnas+j))` es equivalente a **`mat1[i*columnas+j]`**

Ejercicio 7

Cuando compilamos el siguiente ejercicio, el compilador nos proporciona un error:

"incompatible types in assignment"

¿Por qué?

```
#include <stdio.h>
main(){
  int mat1[3][3];
  int i,j;

  mat1 = (int **)malloc(3*sizeof(int*));
  for(i=0; i < 3; i++)
       mat1[i] = (int *)malloc(3*sizeof(int));

  for(i=0;i<3;i++)
      for(j=0;j<3;j++)
      {
            printf("mat1[%d][%d]= ",i,j);
            scanf("%d",&mat1[i][j]);
         printf("\n%d\n",mat1[i][j]);
      }
   system("pause");
}
```

Ejercicio 8

Marca la opción incorrecta para declarar un array multidimensional de enteros de dimensión 3 (cuadrada)

a) int mat1[3][3];
b) int **mat1[3][3];
c) int mat1[3];

Ejercicio 9

Si compilamos el siguiente programa, ¿Qué ocurrirá? ¿Por qué?

```
#include <stdio.h>
main(){
  int *mat1;
  int i,j;
// Reserva de memoria para la matriz
  mat1 = (int **)malloc(3*sizeof(int*));
  for(i=0; i < 3; i++)
```

```
       mat1[i] = (int *)malloc(3*sizeof(int));
// fin de reserva

  for(i=0;i<3;i++)
      for(j=0;j<3;j++)
      {
            printf("mat1[%d][%d]= ",i,j);
            scanf("%d",&mat1[i][j]);
            printf("\n%d\n",mat1[i][j]);
      }
   system("pause");
}
```

Ejercicio 10

Marca la opción correcta para declarar un array multidimensional de enteros de dimensión desconocida y por tanto, del cual deberemos reservar memoria dinámica (mediante el método de acceso por índices).

a) `int mat1;`
b) `int **mat1[3][3];`
c) `int **mat1;`

7.- Paso de punteros como parámetros de funciones

Vamos a estudiar cuatro casos acerca de paso de parámetros a funciones y así poder ver las diferencias que existen entre ellas:

	Caso	Declaración de la función	Llamada a la función	¿Qué se modifica?
1	Variable: paso por valor	`void CalculaNomina(float s)`	`CalculaNomina(sueldo);`	Nada
2	Variable: paso por dirección	`void CalculaNomina(float *s)`	`CalculaNomina(&sueldo);`	Valor de sueldo
3	Puntero: paso por valor	`void CalculaNomina(float *s)`	`puntero=&sueldo;` `CalculaNomina(puntero);`	Valor de sueldo
4	Puntero: paso por dirección	`void CalculaNomina(float **s)`	`puntero=&sueldo;` `CalculaNomina(&puntero);`	Valor de sueldo, dirección de puntero

Caso 1: Variable: paso por valor

Para este caso, y visto ampliamente a lo largo del curso, declaramos una función llamada CalculaNomina, que recibirá una variable ordinaria float. Cuando realizamos la llamada a dicha función, le pasamos como argumento, otro float llamado “sueldo”. La función recibirá dicho argumento, hará las modificaciones que crea oportunas, y cómo la función no devuelve nada (fijémonos en el “void” de su declaración), a menos que tenga un “return”, el valor de la variable “s” dentro del main no se verá alterado y seguirá valiendo lo mismo.

Caso 2: Variable: paso por dirección

Esta vez, declararemos un puntero en la declaración de la función. Con esto, lo que querremos decir, es que dicha función, espera recibir una dirección para almacenarla en el puntero.
Realizamos la llamada a la función, y le pasamos como argumento la dirección de la variable ordinaria sueldo (con el operador &). La función recibirá dicho argumento, hará las modificaciones que crea oportunas y cuando devolvamos el control al main, habremos modificado el valor de la variable ordinaria “sueldo”, ya que dentro de la función habremos trabajado apuntando a su dirección de memoria, y por tanto, tales cambios se realizarán de forma global para esa variable.

Caso 3: Puntero: paso por valor

Para este caso, declararemos también un puntero en la declaración de la función. Lo que cambia ahora es que previamente a la llamada a la función, utilizaremos un puntero para almacenar la dirección de la variable sueldo, y será dicho puntero el que pasaremos a la función. Se debe observar como el paso del puntero se realiza sin el operador “&” ya que espera una dirección, que será la que le proporciona puntero, que a su vez será la dirección de la variable sueldo.

A continuación se muestra un ejemplo

```
#include <stdio.h>

void func2(int *pu);
main ()
{
  int u=1,*puntero;
  puntero=&u;
  printf("La direccion de u es: u=%X \n", &u);
  func2(puntero);
  system("PAUSE");
}

void func2(int *pu)
{
    printf("pu apunta a %X \n", pu);
}
```

Salida por pantalla:

```
-La dirección de u es: u=22FF74
-pu apunta a 22FF74
```

Caso 4: Puntero: paso por dirección

En este caso, declaramos un puntero a puntero en la declaración de la función, como ya hemos visto en el primer punto de este tema, y previo a la llamada de la función, declaramos un nuevo puntero que será el que le pasaremos a la función y apuntamos con él a la dirección de la variable "sueldo". A continuación, llamamos a la función pasándole la dirección a la que apunta "puntero".
"s" por tanto apuntará a la dirección de memoria de "puntero", pudiendo hacer dentro de la función que "puntero" apunte a otro objeto. (ver punto 1.- Puntero a puntero)

8.- Paso de arrays unidimensionales como parámetros de funciones

Para pasar un array a una función, podemos usar directamente el nombre de dicho array (sin indices ni corchetes), y de esta forma, el array completo se pasará a la función.

En la declaración de la función, debe aparecer una declaración de dicho array que recibirá la función sin especificar que tamaño tendrá, es decir, con corchetes vacíos [].

Al contrario que para una variable ordinaria, a una función lo que se le pasa es la dirección de memoria del primer elemento del array. Por tanto, decimos que los arrays siempre son pasados por referencia, y por tanto, cualquier modificación que hagamos sobre ellos dentro de una función, será permanente.

Ejemplo:

a) Declaración de la función:

```
void LeerVector(float vector[])// Declaración de la función
{
…
}
```

…

b) Llamada a la función dentro del main

```
main()
{
…
    LeerVector(vector); // Llamada a la función
…
}
```

9.- Paso de arrays multidimensionales como parámetros de funciones

Tal y como hemos visto, este tipo de arrays son algo más complicados de entender, ya que cada uno de los elementos del array también es un array. Veamos un ejemplo:

Ejemplo:

Main

```
…
float **mat1;
LeerMatriz(mat1);
…
```

Declaración de la función

```
void LeerMatriz(float **m){
…
}
```

Hemos definido en el main un puntero de punteros llamado “**mat1” y a su vez hemos declarado una función llamada LeerMatriz, que esperará la dirección de memoria donde se almacena el primer elemento del array “mat1”.

Para que la función reciba ese dato, deberemos llamarla con el nombre del array entre paréntesis y sin ningún *.