Unidad 1
Punteros.

Ing José Luis MARTÍNEZ

23 de mayo de 2024

1. Punteros

Los punteros son direcciones a variables, al definirla podemos asociarle un puntero que contenga su ubicación, y desde él acceder a la variable desde cualquier lugar del programa. La programación utilizando punteros tiene varias ventajas

1. Pasar información entre funciones en ambos sentidos
2. Permiten devolver múltiples datos desde una función, a través de sus argumentos
3. Proveen una forma alternativa de acceder a los elementos de un array.
4. Se los utiliza para pasar arreglos y cadenas como argumentos de función
5. Se los utiliza para crear estructuras de datos complejas como son las listas enlazadas, pilas, colas, árboles y grafos
6. Son utilizados para la locación dinámica de memoria

Ejemplo. DUP que dadas los valores de cargas eléctricas discretas y ubicadas en un sistema de coordenadas cartesianas, calcule el campo eléctrico en el origen

```
/* El campo electrico en un punto del espacio debido a una carga electrica
está dado por Ex = k q/|r|^3 (x i);  Ey = k q/|r|^3 (y j)
 r^2 = x^2 + y^2 */

 #include<stdio.h>

 #include<math.h>
 #define k 9e9

 void campo(float *ptrEx, float *ptrEy, int n);
 void calculaCampo(float *ptrEx, float *ptrEy, float q, float x, float y);
 int main(int argc, char* argv[])
 {
     int n;
     float accEx, accEy, Ex, Ey, *ptrEx, *ptrEy;
     ptrEx = &Ex;
     ptrEy = &Ey;
     printf(" Ingrese la cantidad de cargas que va a utilizar: \n");
     scanf("%d", &n);
     campo(ptrEx, ptrEy, n);

     printf("\nEl valor de la suma de las componentes en x es: %f. \n", Ex);
```

```
     printf("\nEl valor de la suma de las componentes en y es: %f. \n", Ey);

     printf("\n El módulo del campo eléctrico es: %f", sqrt(Ex*Ex + Ey * Ey));
   printf("\n El ángulo del campo eléctrico es: %f", atan(Ey/Ex));

     return 0;
 }

void campo(float *ptrEx, float *ptrEy, int n)
{
    float q;
    float x,y, accEx=0.0, accEy=0.0;
    while (n > 0)
    {
        printf("Ingrese el valor de la carga q(%d): ", n);
        scanf("%f", &q);
        printf("Ingrese la coordenada x: ");
        scanf("%f", &x);
        printf("Ingrese la coordenada y: ");
        scanf("%f", &y);
        calculaCampo(ptrEx, ptrEy, q, x, y);
        accEx += *ptrEx;
        accEy += *ptrEy;
        printf("\nEl valor de la suma de las componentes en x es: %f. \n", accEx);
        printf("\nEl valor de la suma de las componentes en y es: %f. \n", accEy);
        --n;
    }
    *ptrEx = accEx;
    *ptrEy = accEy;

}

void calculaCampo(float *ptrEx, float *ptrEy, float q, float x, float y)
{
    float r2= x*x + y*y;
    float modr=sqrt(r2);
    float modr3 = pow(modr,3);
    *ptrEx = (k*q/modr3)*x;
    *ptrEy = (k*q/modr3)*y;

}
```

1.1. Apuntadores NULL y void

Normalmente un puntero apunta a alguna posición específica de memoria, sin embargo se presentan situaciones donde es necesario inicializar el apuntador sin indicar la dirección a la que debe apuntar, por ejemplo en el caso de que la variable aún no ha sido creada. Esto crea un problema porque un apuntador no inicializado tiene un valor aleatorio y puede contener cualquier dirección de memoria, por ello es necesario asegurarse que los punteros utilicen direcciones de memoria válidas.

Para los casos mencionados C cuenta con dos tipos de apuntadores especiales: los apuntadores void y NULL (nulo).

Un apuntador nulo no apunta a ninguna parte, es decir, un apuntador nulo no direcciona ningún da-

to válido en memoria. Este apuntador se utiliza para proporcionar a un programa un medio de conocer cuando una variable apuntador no direcciona a un dato válido. Para declarar un apuntador nulo se utiliza la macro NULL, definida en los archivos de cabecera stdef.h, stdio.h, stdlib.h y string.h. Se debe incluir uno o más de estos archivos de cabecera antes de que se pueda utilizar la macro NULL. Ahora bien, se puede definir NULL en la parte superior de su programa (o en un archivo de cabecera personal) con la línea:

```
#define NULL 0
```

Para inicializar una variable apuntador se hace:

```
char *p = NULL;
```

Algunas funciones de C devuelven también el valor NULL si se encuentra un error. Se puede añadir un test para el valor NULL comparando el apuntador con NULL:

```
char *p;
p = malloc(121*sizeof(char));
if (p == NULL)
{
   puts("Error de asignación de memoria");
}
```

también

```
if (p != NULL) ...
/* este if es equivalente a : */
if (p)
```

Otra forma de declarar un apuntador nulo es asignarle un valor 0:

```
int *ptr = (int *) 0; /* ptr es un puntero nulo */
```

La conversión de tipos (casting) anterior (int *), no es necesario, hay una conversión estándar de 0 a una variable apuntador.

```
int *ptr = 0;
```

Nunca se utiliza un apuntador nulo para referenciar un valor. Como antes se ha comentado, los apuntadores nulos se utilizan en un test condicional para determinar si un apuntador se ha inicializado. En el ejemplo,

```
if (ptr)
   printf("Valor de la variable apuntada por ptr es: %d\n",*ptr);
```

se imprime un valor si el apuntador es válido y no es un apuntador nulo.

En C se puede declarar un apuntador de modo que apunte a cualquier tipo de dato, es decir, no se asigna a un tipo de dato específico. El método es declarar el apuntador como un apuntador void*, denominado apuntador genérico.

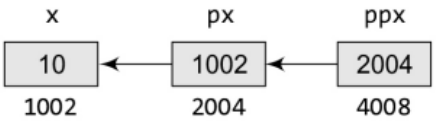
```
void *ptr; /* declara un puntero void, puntero genérico */
```

El apuntador ptr puede direccionar cualquier posición en memoria, pero el apuntador no está unido a un tipo de dato específico. De modo similar, los apuntadores void pueden direccionar una variable float, una char, o una posición arbitraria o una cadena.

> No confundir apuntadores void y NULL. Un apuntador nulo no direcciona ningún dato válido. Un apuntador void direcciona datos de un tipo no especificado. Un apuntador void se puede igualar a nulo si no se direcciona ningún dato válido. NULL es un valor, void es un tipo de dato.

1.2. Punteros a punteros.

un puntero puede apuntar a otro puntero de la forma




```
  int x = 10;
int *px, **ppx;
px = &x;
ppx = &px;
...
...
printf("\n %d", **ppx);
```

1.3. Punteros a arreglos.

Como se ha estudiado un arreglo es una colección de valores de un tipo de datos, donde el nombre del arreglo es la dirección del primer dato y los datos sucesivos se escriben en direcciones sucesivas. En consecuencia se puede acceder a un arreglo mediante punteros. En este caso no necesito definir una variable de tipo puntero porque el arreglo es un puntero en sí mismo.

Ejemplo. Pase un arreglo de caracteres a una función y que desde esta se modifique un elemento

```
int main(int argc, char *argv[])
{
    char c[]= {'m','e','s','a'};

    printf("El valor de c antes de llamar a la funcion cambiaLetra().\n");
    for(int i = 0; i < 4; i++)
        printf("%c", c[i]);
    printf("\n");

    cambiaLetra(c);

    printf("El valor de c despues de llamar a la funcion cambiaLetra().\n");
    for(int i = 0; i < 4; i++)
        printf("%c", c[i]);
    printf("\n");

    return 0;
}

void cambiaLetra(char *c)
{
    c[1] = 'a';
}
```

1.4. Punteros en los arrays de dos dimensiones

Para apuntar a un array bidimensional como tal,o lo que es lo mismo, para apuntar a su inicio, el compilador de C considera que un array bidimensional es en realidad un array de punteros a los arrays que forman sus filas. Por tanto, será necesario un puntero doble o puntero a puntero, que contendrá la dirección del primer puntero del array de punteros a cada una de las filas del array bidimensional o matriz. Si *a*

se ha definido como un array bidimensional, el nombre del array a es un puntero constante que apunta a la primera fila a[0] . El puntero a+1 apunta a la segunda fila a[1] , etc. A su vez a[0] es un puntero que apunta al primer elemento de la fila 0 que es a[0][0] . El puntero a[1] es un puntero que apunta al primer elemento de la fila 1 que es a[1][0] , etc.

Representamos un array de dos dimensiones

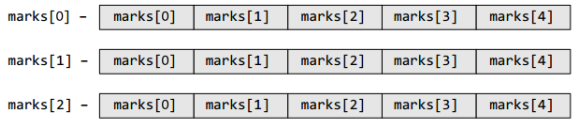


Figura 1: Array 2D

Representación en memoria del Array 2D

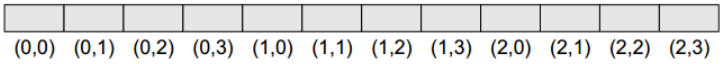


Figura 2: Array 2D

Ejemplo Array bidimensional, punteros y posiciones de memoria. Dada la declaración float A[5][3] que define un array bidimensional de cinco filas y tres columnas, se tiene la siguiente estructura:

Puntero a puntero fila	**Puntero a fila**	**Array bidimensional** float A[4][3]		
A →	A[0] →	A[0][0]	A[0][1]	A[0][2]
A+1 →	A[1] →	A[1][0]	A[1][1]	A[1][2]
A+2 →	A[2] →	A[2][0]	A[2][1]	A[2][2]
A+3 →	A[3] →	A[3][0]	A[3][1]	A[3][2]
A+4 →	A[4] →	A[4][0]	A[4][1]	A[4][2]

A es un puntero que apunta a un array de 5 punteros A[0], A[1], A[2], A[3], A[4] .
A[0] es un puntero que apunta a un array de tres elementos A[0][0], A[0][1], A[0][2] .
A[1] es un puntero que apunta a un array de tres elementos A[1][0], A[1][1], A[1][2] .
A[2] es un puntero que apunta a un array de tres elementos A[2][0], A[2][1], A[2][2] .
A[3] es un puntero que apunta a un array de tres elementos A[3][0], A[3][1], A[3][2] .
A[4] es un puntero que apunta a un array de tres elementos A[4][0], A[4][1], A[4][2] .
A[i][j] es equivalente a las siguientes expresiones:

- $*(A[i]+j)$ el contenido del puntero a la fila i más el número de columna j.
- $*((*(A+i))+j)$. Si se cambia $A[i]$ por $*(A+i)$ se tiene la siguiente expresión anterior.
- $*(\&A[0][0]+3*i+j)$.

Ejemplo Escriba un programa que acceda a los miembros de un arreglo mediante aritmética de punteros.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int matrizA[3][3];
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
```

```
            printf("Ingrese la fila %d y la columna %d. \n", i, j);
            scanf("%d", &matrizA[i][j]);
        }
    }

    for (int i=0; i<3; i++)
    {
        printf("\nEsta en la fila i = %d \n", i);
        for(int j=0; j<3 ; j++)
        {
            printf(" \t columna j = %d, matrizA[%d][%d] = %d", j, i, j, matrizA[i][j]);
        }
    }

    printf("\n\t Prueba con *(matrizA[i]+j)\n");
    for (int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("%d \t", *(matrizA[i]+j) );
        }
        printf("\n");
    }

    printf("\n\t Prueba con *(*(matrizA + i) + j)\n");
    for(int i=0; i<3;i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("%d \t", *(*(matrizA +i)+j));

        }
        printf("\n");
    }

    printf("\n\t Prueba con *(&A[0][0]+ 3*i+j)\n");
    for(int i=0; i<3 ; i++)
    {
        for(int j=0; j<3;j++)
        {
            printf("%d \t",*(&matrizA[0][0]+ 3*i+j));
        }
        printf("\n");
    }

    return 0;
}
```

A es un puntero que apunta a A[0].
A[0] es un puntero que apunta a A[0][0].
Si A[0][0] se encuentra en la dirección de memoria 100 y teniendo en cuenta que un *float* ocupa 4 bytes, la siguiente tabla muestra un esquema de la memoria:

Contenido de puntero a puntero fila	Contenido de puntero a fila	Direcciones del array bidimensional float A[4][4]		
*A = A[0]	A[0] = 100	A[0][0] = 100	A[0][1] = 104	A[0][2] = 108
*(A+1) = A[1]	A[1] = 112	A[1][0] = 112	A[1][1] = 116	A[1][2] = 120
*(A+2) = A[2]	A[2] = 124	A[2][0] = 124	A[2][1] = 128	A[2][2] = 132
*(A+3) = A[3]	A[3] = 136	A[3][0] = 136	A[3][1] = 140	A[1][2] = 144
*(A+4) = A[4]	A[4] = 148	A[4][0] = 148	A[4][1] = 152	A[4][2] = 156

1.5. Paso de arreglos a funciones por referencia

Considere un arreglo de dos dimensiones

```
int mat[5][5];
```

Para declarar un puntero de un array bidimensional, se escribe

```
int **ptr
```

Aquí **ptr es un arreglo de punteros (a un arreglo de una dimensión) mientras que

```
int mat[5][5];
```

es un arreglo bidimensional. Por lo tanto no son lo mismo y no son intercambiables. Los elementos del arreglo se pueden acceder mediante:

```
mat[i][j]
*(*(mat + i) + j)
*(mat[i]+j);
```

Para especificar en el prototipo de función que vamos a utilizar punteros podemos hacer

```
void funcionArray(int ptr[][columnas]);
```

o también

```
void funcionArray(int (*)[10]);
void funcionArray(int (*ptr)[10]);
```

Observe que en ptr[][columnas] el primer corchete vacío está indicando que se trata de un puntero, las sintaxis (*)[10] y (*ptr)[10] indican que son arreglos bidimensionales de 10 columnas, no confundir con *ptr[10] que es un arreglo unidimensional de 10 elementos.

El siguiente código ilustra el uso de punteros a un arreglo bidimensional.

```
#include <stdio.h>
int main()
{
int arr[2][2]={{1,2}, {3,4}};
int i, (*parr)[2];
parr = arr;
for(i = 0; i < 2; i++)
{
 for(j = 0; j < 2 ;j++)
 printf(" %d", (*(parr+i))[j]);
}
return 0;
}
```

Ejemplo. Escriba un programa en C que modifique por referencia un arreglo de 2D.

```
/* Pasar un array 2D por referencia*/

#include<stdio.h>

void cargaArray2D(float (*D2)[3]);

int main(int argc, char *argv[])
{
    float array2D[3][3]={0};

    printf("El array2D antes de entrar a la funcion. \n");

    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            printf("%f  ", array2D[i][j]);
        }
        printf("\n");
    }

    printf("\n");

    cargaArray2D(array2D);

    printf("El array2D despues de entrar a la funcion. \n");

    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            printf("%f  ", array2D[i][j]);
        }
        printf("\n");
    }

    printf("\n");


    return 0;
}

void cargaArray2D(float (*D2)[3])
{
    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            D2[i][j] = (float) i+j;
        }

    }
```

```
}
```

1.6. Puntero a funciones

Cuando se declara una matriz se asume que la dirección es la del primer elemento; del mismo modo, se asume que la dirección de una función será la del segmento de código donde comienza la función. Es decir, la dirección de memoria a la que se transfiere el control cuando se invoca (su punto de comienzo).

Un apuntador a una función es simplemente un apuntador cuyo valor es la dirección del nombre de de la función. Dado que el nombre en sí mismo es un apuntador; un puntero a una función es un apuntador a un apuntador constante.

Técnicamente un apuntador a función es una variable que guarda la dirección de comienzo de la función. La mejor manera de pensar en el apuntador a función es considerándolo como una especie de "alias" de la función, aunque con una importante cualidad añadida:

- pueden ser utilizados como argumento de otras funciones.

Por lo que un apuntador a función es un artificio que el lenguaje C utiliza para poder enviar funciones como argumento de una función, la gramática del lenguaje C no permite en principio utilizar funciones en la declaración de parámetros. Cabe aclarar que no está permitido hacer operaciones aritméticas con este tipo de apuntador.

La sintaxis general para la declaración de un apuntador a una función es:

```
Tipo_de_retorno (*PunteroFuncion) (<lista de parámetros>);
```

Lo anterior es útil cuando se deben usar distintas funciones, quizás para realizar tareas similares con los datos. Por ejemplo, se pueden pasar como parámetros los datos y la función que serán usados por alguna función de control. La biblioteca estándar de C tiene funciones para ordenamiento (qsort()) y para realizar búsqueda (bsearch()), a las cuales se les pueden pasar funciones como parámetros. La declaración de un apuntador a función se realiza de la siguiente forma:

```
int (* pf)();
```

Aquí * pf es un apuntador a una función que no envía parámetros y retorna un tipo de dato entero. Observe que se ha declarado el apuntador y en este momento no se ha dicho a qué variable va a apuntar. Supongamos que se tiene una función int f(); entonces simplemente se debe de escribir:

```
pf= f;
```

Para que la variable pf apunte al inicio de la función f(). Los apuntadores a funciones se declaran en el área de prototipos, por ejemplo:

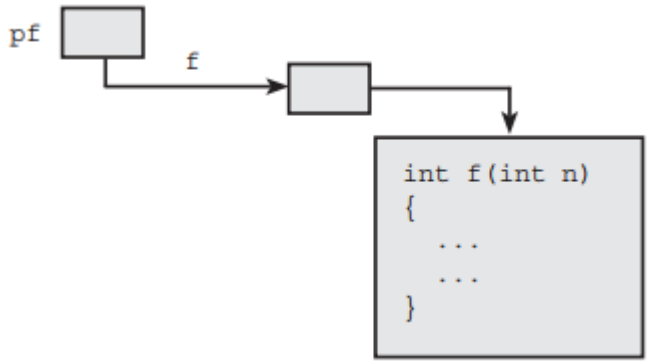
```
int f(int);
int (*fp) (int) =f;
```

En consecuencia, en el cuerpo del programa se pueden tener asignaciones como:

```
ans=f(5);
ans=pf(5);
```

Ejemplo de apuntador a funciones

```
int f(int); /* declara la función f */
int (*pf)(int); /* define puntero pf a función int con argumento
int */
pf = f; /* asigna la dirección de f a pf */
```



Ejemplo 2 de apuntador a funciones

```
#include <stdio.h>

void printMensaje(int dato );
void printNumero(int);
void (*funcPuntero)(int);

int main(){
    printMensaje(1);
    funcPuntero=printMensaje;
    funcPuntero(2);
    funcPuntero(3);
    funcPuntero=printNumero;
    funcPuntero(4);
    printNumero(5);
    getchar();
}
void printMensaje(int dato){
   printf("Esta es la funcion printMensaje:%d\n",dato);
 }
void printNumero(int dato){
   printf("Esta es la funcion printNumero: %d\n",dato);
 }
```

En este programa se declaran dos funciones, printMensaje() y printNumero(), y después (*funcPuntero), que es un apuntador a una función que recibe un parámetro entero y no devuelve nada (void). Las dos funciones declaradas anteriormente se ajustan precisamente a este perfil, y por tanto pueden ser llamadas por este apuntador.

En la función principal, llamamos a la función printMensaje() con el valor uno como parámetro, en la línea siguiente asignamos al puntero a función (*funcPuntero) el valor de printMensaje() y utilizamos el apuntador para llamar a la misma función de nuevo. Por tanto, las dos llamadas a la función printMensaje() son idénticas gracias a la utilización del puntero (*funcPuntero).

Dado que hemos asignado el nombre de una función a un apuntador a función, y el compilador no da error, el nombre de una función debe ser un apuntador a una función. Esto es exactamente lo que sucede.

Un nombre de una función es un apuntador a esa función, pero es un apuntador constante que no puede ser cambiado. Sucede lo mismo con los vectores: el nombre de un vector es un apuntador constante al primer elemento del vector.

El nombre de una función es un apuntador a esa función, podemos asignar el nombre de una función a un apuntador constante y usar el apuntador para llamar a la función. Pero el valor devuelto, así como el número y tipo de parámetros, deben ser idénticos. Muchos compiladores de C y C++ no avisan sobre las diferencias entre las listas de parámetros cuando se hacen las asignaciones. Esto se debe a que las asignaciones se hacen en tiempo de ejecución, cuando la información de este tipo no está disponible para el sistema.

Ejemplo de apuntador a función con dos argumentos

```
   #include <stdio.h>

int MiFuncionSuma(int,int);
int MiFuncionResta(int,int);
int MiFuncionModulo(int,int);
int main(){
    int (*ptrFn) (int,int);
    ptrFn = MiFuncionSuma;
    printf("La suma es : %d\n", ptrFn(5, 6));
    ptrFn = MiFuncionResta;
    printf("La resta es : %d\n", ptrFn(5, 6));
    ptrFn = MiFuncionModulo;
    printf("El Modulo es : %d\n", ptrFn(5, 6));
    getchar();
}
int MiFuncionSuma(int a,int b){
    return a + b;
}
int MiFuncionResta(int a,int b){
    return a - b;
}
int MiFuncionModulo(int a,int b){
    return a % b;
}
```

Como se ha visto, primero se definen las funciones, con o sin parámetros, luego las variables del tipo apuntador a función, por último se enlazan las variables a la dirección de memoria del código de las funciones antes de invocarlas. El enlace se debe hacer a la función, en consecuencia se puede enlazar colocando solo el nombre de la función o usando el operador de dirección & seguido del nombre de la función. Por último, se trabaja únicamente con la función definida.

Ejemplo donde la función tiene como parámetro a un apuntador a una función. Primero se define la variable tipo apuntador a función:

```
   int (*ptrFn) (int, int);
```

luego se define una variable tipo apuntador a función que tiene tres parámetros, el primer parámetro es el apuntador tipo función:

```
   void (*ptrFnD)(int (*ptrFn)(int, int), int, int);
```

Se declaran apuntadores a funciones.

```
double (*fp) (int n);
float (*p) (int i, int j);
void (*sort) (int* ArrayEnt, unsigned n);
unsigned (*search)(int BuscarClave,int* ArrayEnt,unsigned n);
```

El primer identificador, fp, apunta a una función que devuelve un tipo double y tiene un único parámetro de tipo int. El segundo apuntador, p, apunta a una función que devuelve un tipo float y acepta dos parámetros de tipo int. El tercer apuntador, sort, es un apuntador a una función que devuelve un tipo void y toma dos parámetros: un apuntador a int y un tipo unsigned. search es un apuntador a una función que devuelve un tipo unsigned y tiene tres parámetros: un int, un apuntador a int y un unsigned.

En las declaraciones de los siguientes ejemplos (en todos ellos fptr es un apuntador a función de tipo distinto de los demás).

Prototipo de Función	**Funcionamiento**
void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve void.
void (*fptr)(int);	fptr es un puntero a función que recibe un int como parámetro y devuelve void.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un int y un char como argumentos y devuelve un int.
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a int y char como argumentos, y devuelve un puntero a int.
	CUANDO EL VALOR DEVUELTO POR LA FUNCIÓN ES A SU VEZ UN PUNTERO (A FUNCIÓN, O DE CUALQUIER OTRO TIPO), LA NOTACIÓN SE COMPLICA UN POCO MÁS:
int const * (*fptr)();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un int constante.
float (*(*fptr)(char))(int);	fptr es un puntero a función que recibe un char como argumento y devuelve un puntero a función que recibe un int como argumento y devuelve un float.
void * (*(*fptr)(int))[5];	fptr es un puntero a función que recibe un int como argumento y devuelve un puntero a un array de 5 punteros-a-void (genéricos).
char (*(*fptr)(int, flat))();	fptr es un puntero a función que recibe dos argumentos (int y float), devolviendo un puntero a función que no recibe argumentos y devuelve un char.
long (*(*(*fptr)())[5])();	fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven long.
void (*fptr)();	fptr es un puntero a una función, sin parámetros, que devuelve void.
void (*fptr)(int);	fptr es un puntero a función que recibe un int como parámetro y devuelve void.
int (*fptr)(int, char);	fptr es puntero a función, que acepta un int y un char como argumentos y devuelve un int.
int* (*fptr)(int*, char*);	fptr es puntero a función, que acepta sendos punteros a int y char como argumentos, y devuelve un puntero a int.
int (* afptr[10])(int);	a fptr es una matriz de 10 apuntadores a función

1.7. Arreglos (arrays) de apuntadores

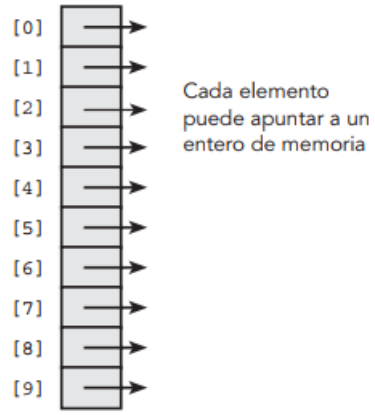
Si se necesita reservar muchos apuntadores a muchos valores diferentes, se puede declarar un arreglo de apuntadores. Un arreglo de apuntadores es un arreglo que contiene como elementos apuntadores, cada

uno de los cuales apunta a un tipo de dato específico. La línea siguiente reserva un arreglo de diez variables apuntador a enteros:

```
int *ptr[10]; /* reserva un array de 10 apuntadores a enteros */
ptr[5] = &edad; /* ptr[5] apunta a la dirección de edad */
ptr[4] = NULL; /* ptr[4] no contiene dirección alguna */
//Otro ejemplo de arreglos de apuntadores, en este caso de caracteres es:
char *puntos[25]; /* array de 25 apuntadores a carácter */
//De igual forma, se podría declarar un apuntador a un arreglo de enteros.
int (*ptr10) [ ];
//y las operaciones de dicha declaración paso a paso son:
(*ptr10) //es un puntero, ptr10 es un nombre de variable.
(*ptr10) [ ] //es un puntero a un array
int (*ptr10) [ ] //es un puntero a un array de int
```

La inicialización de un arreglo de apuntadores a cadenas se puede realizar con una declaración similar a esta:

```
char *nombres_meses[12] = { "Enero", "Febrero", "Marzo",
                            "Abril", "Mayo", "Junio",
                            "Julio", "Agosto", "Septiembre",
                            "Octubre", "Noviembre",
                            "Diciembre" };
```



Ejemplo. El siguiente programa muestra un apuntador que recorre una cadena de caracteres y convierte cualquier carácter en minúsculas a caracteres mayúsculas. Nota: Corregir para que el programa sea más seguro con la violación de segmento de memoria.

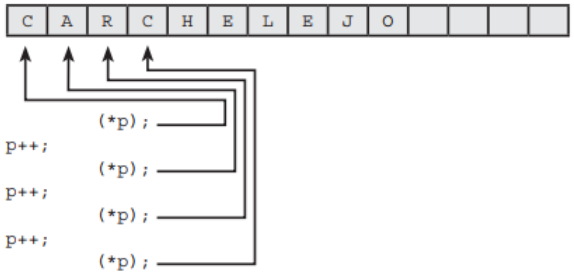
```
#include <stdio.h>

void main()
{
   char *p;
   char CadenaTexto[81];
   puts("Introduzca cadena a convertir:");
   gets(CadenaTexto);
   /* p apunta al primer carácter de la cadena */
   p = &CadenaTexto[0]; /* equivale a p = CadenaTexto */
   /* Repetir mientras *p no sea cero */
   while (*p)
   {
      /* restar 32, constante de código ASCII */
```

```
        if ((*p >= 'a') && (*p <= 'Z'))
           *p++ = *p-32;
        else
           p++;
     }
     puts("La cadena convertida es:");
     puts(CadenaTexto);
}
```

1.8. Apuntadores constantes frente a apuntadores a constantes

Un apuntador constante es un apuntador que no se puede cambiar, pero que los datos apuntados por el apuntador si se pueden cambiar. Un apuntador a constante se puede cambiar para apuntar a una constante diferente, pero los datos apuntados por el apuntador no se pueden cambiar.



*p++ se utiliza para acceder de modo incremental en la cadena.

Para crear un apuntador constante se debe utilizar el siguiente formato:

```
<tipo de dato > *const <nombre puntero> = <dirección de variable >;
```

Ejemplo apuntadores constantes:

```
int x;
int y;
int *const p1 = &x;
```

p1 es un apuntador de constantes que apunta a x, por lo que p1 es una constante, pero *p1 es una variable por lo que se puede cambiar el valor de *pl pero no el valor de pl.

```
*p1 = y;  // correcto
pl = &y;  // incorrecto
```

Para crear un apuntador constante a una cadena:

```
char *const nombre = "Luis";
*nombre = "Francisco";  // legal
nombre = &otro_nombre;  // ilegal
```

Apuntador a constante El formato para definir un apuntador a una constante es:

```
const <tipo de dato elemento> *<nombre puntero> = <dirección de constante >;
```

Por ejemplo:

```
const int x = 25;
const int y = 50;
const int *p1 = &x;
```

p1 se define como un apuntador a la constante x en consecuencia, se puede hacer que p1 apunte a otra constante.

```
p1 = &y;
```

Cualquier intento de cambiar el contenido almacenado en la posición de memoria pl no está permitida

```
*p1 = 15;
```

Una definición de un apuntador constante tiene la palabra reservada const delante del nombre del apuntador, mientras que el apuntador a una definición constante requiere que la palabra reservada const se encuentre antes del tipo de datos.

La creación de un apuntador a una constante cadena se puede hacer del modo siguiente: *const char *apellido = Ramirez"*;

En el prototipo de la siguiente función se declara el argumento como apuntador a una constante:

```
float cargo(const float *v);
```

Apuntadores constantes a constantes El último caso a considerar es crear apuntadores constante a constantes:

```
const <tipo de dato elemento> *const <nombre puntero> = <dirección de constante >;
```

Esta definición se puede leer como "un tipo constante de dato y un apuntador constante". Por ejemplo:

```
const int x = 25;
const int *const p1 = &x;
```

que indica: "p1 es un apuntador constante que apunta a la constante entera x". Cualquier intento de modificar p1 o bien *p1 producirá un error de compilación.

Consejo de Programación

Si sabe que un apuntador siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defínalo como un apuntador constante.
Si sabe que el dato apuntado por el apuntador nunca necesitará cambiar o nunca debe cambiar, defina el apuntador como un apuntador a una constante.

Un apuntador a una constante es distinto de un apuntador constante. El siguiente ejemplo muestra las diferencias. Este trozo de código define cuatro variables: un apuntador p, un apuntador constante cp y un apuntador pc a una constante y un apuntador constante cpc a una constante.

int *p;	puntero a un int
++(*p);	incremento del entero *p
++p;	incrementa un puntero p
int *const cp;	puntero constante a un int
++(*cp);	incrementa el entero *cp
++cp;	no válido: puntero cp es constante
const int * pc;	puntero a una constante int
++(*pc);	no válido: int * pc es constante
++pc;	incrementa puntero pc
const int * const cpc;	puntero constante a constante int
++(*cpc);	no válido: int *cpc es constante
++cpc;	no válido: puntero cpc es constante

El espacio en blanco no es significativo en la declaración de apuntadores. Las declaraciones siguientes son equivalentes.

```
int* p;
int * p;
int *p;
```

1.9. Problemas con punteros.

1. Dado el siguiente programa que construye una biblioteca de funciones para calcular las áreas de las figuras geométricas.

Encabezado

```
/* Encabezado Areas.h */
#ifndef AREAS_H_INCLUDED
#define AREAS_H_INCLUDED
#include<math.h>
#define PI 3.141592654

/* Prototipos de funciones */
float rectangulo(float a, float b);

float trianguloBH(float a, float b);

float trianguloABC(float a, float b, float c);

float circulo(float r);

#endif // AREAS_H_INCLUDED
```

Biblioteca de funciones.

```
/* Biblioteca de funciones Areas.c, para el
cálculo de areas de distintas figuras geométricas */

#include "Areas.h"
//#include<math.h>
//#define PI 3.141592654

/* Rectangulo */

float rectangulo(float a, float b)
{
    return a*b;
}

/* Triangulo del que sabemos la altura */

float trianguloBH(float a, float b)
{
    return a*b/2;
}

/* triangulo del que conocemos la longitud de sus lados, se aplica la formula
de Herón  A= sqrt(s(s-a)(s-b)(s-c)), s=(a+b+c)/2 */
float trianguloABC(float a, float b, float c)
{
    float s;
    s=(a + b + c)/2;
```

```
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

/* Circulo */
float circulo(float r)
{
    return PI * r * r;
}
```

Función de implementación

```
/* usoAreas.c, utiliza la biblioteca Areas */

#include<stdio.h>
#include "Areas.h"

int main(int argc, char *argv[])
{
    float areaRec, areaCirc;
    float ladoA, ladoB, ladoC, radio;
    /*
     insertar la parte de programa faltante
    */
    areaRec = rectangulo(ladoA, ladoB);
    printf("El area del rectangulo es: %.2f\n", areaRec);
    areaCirc = circulo(radio);
    printf("El area del circulo es: %.2f\n", areaCirc);
    return 0;
}
```

a) Modifique el problema para utilizar punteros .

2. Encontrar los errores en la siguiente declaración de apuntadores:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= 'C';
float x;
void* r = &x
```

3. ¿Qué diferencias se pueden encontrar entre un apuntador a constante y una constante apuntador?

4. Un arreglo unidimensional se puede indexar con la aritmética de apuntadores. ¿Qué tipo de apuntador habría que definir para indexar un arreglo bidimensional?

5. En el siguiente código se accede a los elementos de una matriz. Acceder a los mismo elementos con aritmética de apuntadores.

```
#define N 4
#define M 5
int f,c;

double mt [N][M];
```

```
. . .
for (f = 0; f < N; f++)
{
   for (c = 0; c < M; c++)
      printf("%lf ", mt[f][c]);
   printf("\n");
}
```

6. Escribir una función con un argumento de tipo apuntador a double y otro argumento de tipo int. El primer argumento se debe de corresponder con un arreglo y el segundo con el número de elementos del arreglo. La función ha de ser de tipo apuntador a double para devolver la dirección del elemento menor.

7. Dada la siguiente función:

```
double* gorta(double* v, int m, double k)
{
   int j;
      for (j = 0; j<m ; j++)
         if (*v == k)
            return v;
      return 0,
}
```

   a) ¿Hay errores en la codificación? ¿De que tipo? Realice la corrección si corresponde.

   b) Dadas las siguientes definiciones:

```
double w[15], x, z;
void *r;
```

   ¿Es correcta la siguiente llamada a función?

```
r = gorta(w,10,12.3);
```

   ¿Y estas otras llamadas?

```
printf("%lf",*gorta(w,15,10.5));
z = gorta(w,15,12.3);
```

8. Escribir un programa en el que se lean 20 líneas de texto, cada línea con un máximo de 80 caracteres. Mostrar por pantalla el número de vocales que tiene cada línea. Utilizar aritmética de apuntadores.

9. Se quiere evaluar las funciones $f(x)$, $g(x)$ y $z(x)$ para todos los valores de x en el intervalo $0 \leq x < 3{,}5$ con incremento de 0,2. Escribir un programa que evalúe dichas funciones, utilizar un arreglo de apuntadores a función. Las funciones son las siguientes:

$$f(x) = 3 * e^{(x-1)} - 2x \tag{1}$$

$$g(x) = -x * \sin(x) + 1{,}5 \tag{2}$$

$$z(x) = x^2 - 2x + 3 \tag{3}$$

10. Dado el siguiente arreglo de punteros, escriba una función que permita acceder a los meses.

```
char *nombres_meses[12] = { "Enero", "Febrero", "Marzo",
                            "Abril", "Mayo", "Junio",
                            "Julio", "Agosto", "Septiembre",
                            "Octubre", "Noviembre",
                            "Diciembre" };
```

11. Escriba una función en C que acceda a las funciones del ejemplo 9 mediante punteros a funciones.