Punteros

6.1.- Introducción

Un puntero es una variable que representa la *posición en la memoria* (en vez del valor) de otro dato, como pude ser una variable o un elemento de un array. Los punteros se usan frecuentemente en C, en particular pueden ser usados para trasvasar información entre una función y sus puntos de llamada, proporcionando una forma de devolver varios datos desde una función a través de los argumentos de la función. Los punteros permiten que referencias a otras funciones puedan ser especificadas como argumentos de una función. Esto tiene el efecto de pasar funciones como argumentos de una función dada.

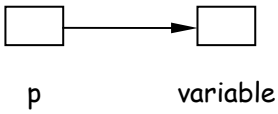
Los punteros están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales de la tabla. Es mas, proporcionan una forma conveniente para representar tablas multidimensionales, permitiendo que una tabla multidimensional sea reemplazada por una tabla de punteros de menor dimensión. Esta característica permite que una colección de cadenas de caracteres sean representadas por una sola tabla, incluso cuando las cadenas pueden tener distinta longitud.

6.2.- Conceptos básicos

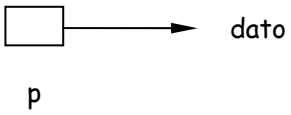
En la memoria del ordenador, cada dato almacenado ocupa una o mas celdas contiguas de memoria (es decir, palabras o bytes adyacentes). El número de celdas de memoria requeridas para almacenar un dato depende de su tipo. Por ejemplo, un carácter se almacenará normalmente en 1 byte (8 bits), un entero utilizará 2 bytes contiguos, un número en coma flotante necesita cuatro bytes contiguos y una cantidad en doble precisión puede requerir ocho bytes contiguos.

Son ejemplos de punteros:

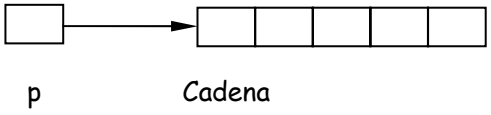
.- Puntero a variable:



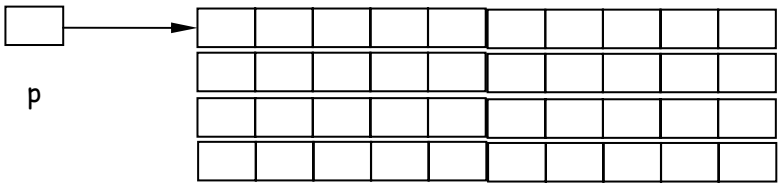
.- Puntero a dato:



.- Puntero a cadena:

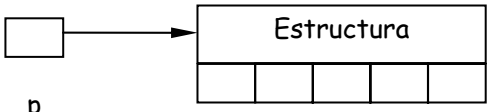


.- Punteros a cadena bidimensional



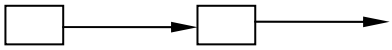
Cadena bidimensional

.- Punteros a estructuras:



Estructura

.- Punteros a puntero:



Las celdas de memoria dentro de un ordenador están numeradas consecutivamente, desde el principio hasta el final del área de memoria. El número asociado con cada celda de memoria es conocido como ***dirección*** de la celda. La mayoría de los ordenadores usan un sistema de numeración decimal para asignar las direcciones de las celdas de memoria. (Algunos ordenadores, sin embargo, utilizan la numeración octal).

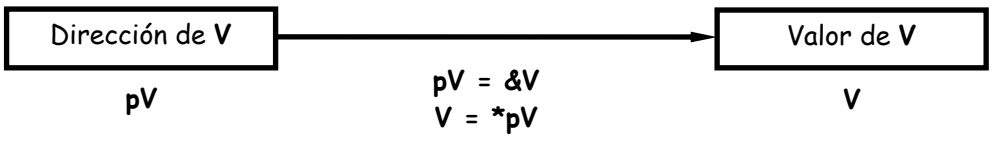
Si se supone que **V** es una variable que representa un determinado dato, el compilador automáticamente asignará celdas de memoria para ese dato. Puede accederse al dato, por tanto, si se conoce su localización, es decir, la ***dirección*** de la primera celda de memoria que ocupa.

La dirección de memoria de **V** queda determinada mediante la expresión **&V**, donde **&** es un operador monario, llamado *operador dirección*, que proporciona la dirección del operando.

Si ahora se asigna la dirección de **V** a otra variable, **pV**, se tendrá que:

pV = & V

Esta nueva variable es un ***puntero*** a **V** puesto que "*apunta*" a la posición de memoria donde se almacena **V**. Debe recordarse, sin embargo, que **pV** representa la ***dirección*** de **V** y no su valor. De esta forma **pV** es referida como una ***variable apuntadora***.



El dato representado por **V** (es decir, el dato almacenado en las celdas de memoria de **V**) puede ser accedido mediante la expresión ***pV** donde * es un operador monario, llamado ***operador indirección*** que opera sólo sobre una variable puntero. *Por lo tanto,* ***pV** y **V** representan el mismo dato (el contenido de las mismas celdas de memoria).* Además se escribe **pV** = **&V** y **U** = ***pV**, entonces **U** y **V** representan el mismo valor; esto es, el valor de **V** se asigna indirectamente a **U** (supuesto, claro está, que **U** y **V** están declaradas como el mismo tipo de datos).

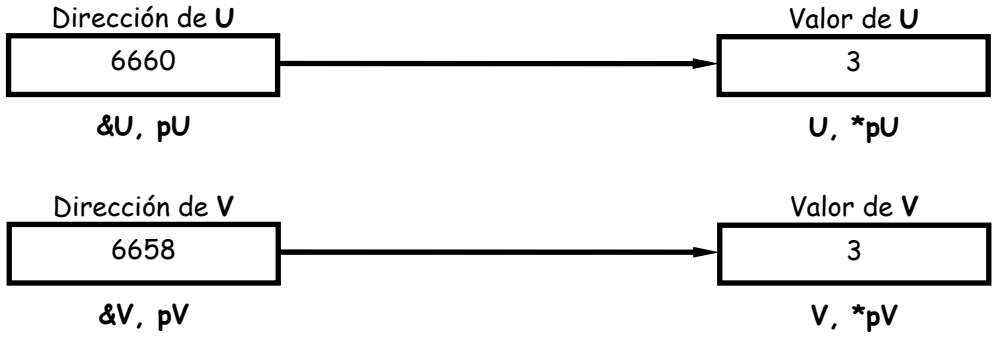
A continuación se muestra un sencillo programa que ilustra la relación entre dos variables enteras, sus correspondientes direcciones y sus punteros asociados:

```
#include <stdio.h>

void main()
{
        int u = 3;
        int v;
        int *pu; //puntero a un entero
        int *pv; //puntero a un entero

        pu = &u; //asigna la dirección de u a pu
        v = *pu; //asigna al valor de u a v
        pv = &v; //asigna la dirección de v a pv

        printf ("\n   u = %d      &u = %d      pu = %d     *pu = %d",u, &u, pu, *pu);
        printf ("\n   v = %d      &v = %d      pv = %d     *pv = %d",v, &v, pv, *pv);

}
```

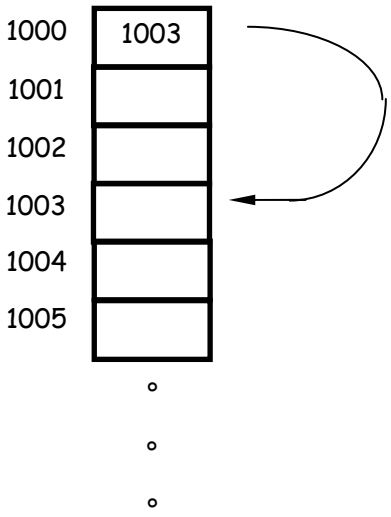
La ejecución del programa ofrecerá el siguiente resultado:

u = 3	&u = 6660	pu = 6660	*pu = 3
v = 3	&v = 6658	pv = 6658	*pv = 3

La relación entre **pu** y **u** y **pv** y **v** se muestra en la figura adjunta:



Un ***puntero*** *es una variable que contiene una dirección de memoria.*Esa dirección es la posición de otro objeto (normalmente otra variable) en memoria. Por ejemplo, si una variable contiene la dirección de otra variable, se dice entonces que la primera variable ***apunta*** a la segunda. La figura adjunta clarifica esta situación:



Memoria

Si una variable va a ser puntero debe declararse como tal. Una declaración de puntero consiste en un tipo base, un ***y el nombre de la variable: La forma general de la declaración de una variable puntero es:

tipo * nombre;

donde ***tipo*** es el tipo base del puntero, que puede ser cualquier tipo válido y el nombre de la variable puntero se especifica con ***nombre.***

Es importante consignar que todas las operaciones de punteros se realizan de acuerdo con sus tipos base ya que el sistema reserva para la variable nombre un número de posiciones de memoria acorde con el tipo de dato al que el puntero apunta.

6.3.- Expresiones de punteros

En general, las expresiones que involucran punteros siguen las mismas reglas que las demás expresiones, si bien hay algunos aspectos especiales de las expresiones de punteros como las asignaciones, las conversiones y su aritmética que se detallan a continuación.

6.3.1.- Asignaciones de punteros

Un puntero puede utilizarse a la derecha de una instrucción de asignación para asignar su valor a otro puntero. Cuando ambos punteros son del mismo tipo, la situación es muy sencilla:

```
#include <stdio.h>

void main()
{
        int x = 99;
        int *p1, *p2;

        p1 = &x;
        p2 = p1;
```

```
    //Imprimir el valor de x dos veces
    printf ("Valores apuntados por p1 y p2:   %d   %d\n",*p1, *p2);

    //Imprimir la dirección de x dos veces
    printf ("Direcciones en p1 y p2:   %p   %p",p1, p2);

}
```

Obteniéndose como salida:

Valores apuntados por p1 y p2: 99 99

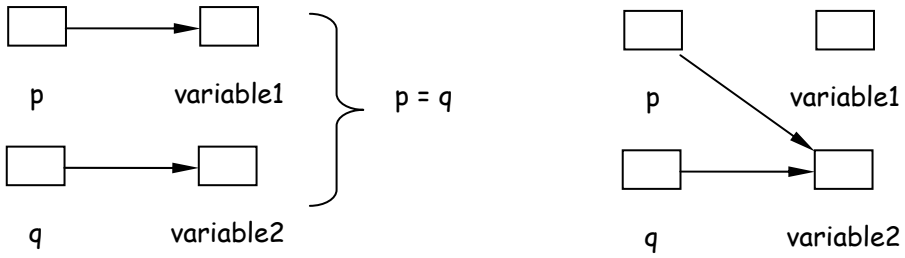
Direcciones en p1 y p2: 19F6 19F6

Se observa que, tras la secuencia de asignaciones

```
p1 = &x;

p2 = p1;
```

tanto **p1** como **p2** apuntan a **x**, por lo tanto tanto **p1** como **p2** apuntan al mismo objeto.

Debe observarse también que se ha utilizado el ***modificador de formato*** de **printf( ) %p** que hace que **printf( )** muestre una dirección en el formato utilizado por el ordenador en cuestión. Lo anterior se ilustra en la siguiente figura:



6.3.2.- Conversiones de punteros

En C se puede asignar un puntero a **void*** a cualquier tipo de puntero. También se puede asignar cualquier otro tipo de puntero a un puntero **void***.

En puntero **void*** se denomina ***puntero genérico*** que se utiliza para especificar un puntero cuyo tipo base se desconoce. El tipo **void*** permite a una función especificar un parámetro que pueda recibir cualquier tipo de argumento puntero sin que se produzca un error de discordancia de tipo.

No se requiere ninguna forma explícita para convertir de o a un puntero **void***

Excepto void* todas las demás conversiones de punteros se deben realizar por medio de un molde explícito, si bien, ***la conversión de un tipo de puntero en otro puede dar lugar a un comportamiento impredecible***, como se muestra en el siguiente programa:

```
#include <stdio.h>
void main()
{
        double x = 100.1, y;
        int *p;
  //La siguiente expresión hace que p puntero a entero, apunte a un double
        p = (int *)&x;
  // se asigna a y el valor de x a través del puntero
        y = *p;
  // Se imprime y sale otra cosa distinta de x = 100.1
        printf ("El valor de x es %f", y);
}
```

6.3.3.- Aritmética de punteros

Existen sólo dos operaciones aritméticas que se pueden usar con punteros: la suma y la resta. Ello es debido a que, cada vez que se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base.

Cuando se aplica a punteros a char dado un carácter ocupa un byte, la aritmética de punteros puede parecer normal. Sin embargo, en el resto de los tipos, los punteros aumentan o decrecen en la longitud del tipo de datos a los que apuntan. Este método garantiza que un puntero siempre apunte a un elemento adecuado del tipo base.

No se está limitado sólo a los operadores de incremento y decremento, por ejemplo, ***se pueden sumar o restar enteros a punteros***:

p1 = p1 + 12

hace que p1 apunte al duodécimo elemento del tipo p1 que está mas allá del elemento al que apunta actualmente.

Además de las operaciones anteriores, sólo puede definirse una operación mas: restar un puntero a otro con el fin de obtener el número de objetos del tipo base que separan ambos punteros. Todas las demás operaciones aritméticas están prohibidas. En particular:

No se pueden multiplicar o dividir punteros

No se pueden sumar o restar punteros

No se les puede aplicar desplazamientos a nivel de bits

No se puede sumar o restar el tipo **float** ***o el tipo*** **double** ***a los punteros***

Como resumen, pueden establecerse cada una de las operaciones permitidas con punteros de la forma siguiente:

a) Suma.

p ← p + 1.

Sumar uno a un puntero significa obtener una dirección igual a la dirección inicial (contenido de p) más un número de bytes igual al número de bytes asociado al tipo del puntero.

Si p es un puntero a carácter, p + 1 significa sumar uno.

Si p es un puntero a entero, p + 1 significa sumar dos bytes.

En general p + 1 es p + sizeof (tipo base) bytes y p + n es p + n * sizeof (tipo base) bytes.

b) Incremento.

p ← p + 1 ó p++

Sumar al contenido de p un número de bytes igual al tipo base.

c) Restar.

p ← p - 1.

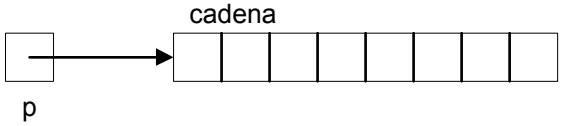
Significa obtener una dirección igual al valor de p menos un número de bytes igual al tipo base del puntero.

d) Decremento.

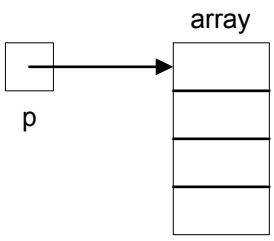
p ← p - 1 ó p—

Restar al contenido del puntero p un número de bytes igual al tipo base.

La aritmética de punteros exige que estos punteros y las direcciones obtenidas resultantes correspondan a direcciones de una zona de memoria definida, array, cadena, etc.



p = p +1 El puntero p apunta al siguiente carácter.



p = p +1 El puntero apunta al siguiente elemento del array sea cual sea el tipo de los elementos del array.

6.3.4.- Comparación de punteros

Se pueden comparar dos punteros en una expresión relacional; por ejemplo, dados los punteros **p** y **q** la siguiente instrucción es perfectamente válida:

if(p < q) printf("p apunta a una celda de memoria anterior a q\n");

La comparación suele resultar útil cuando dos punteros apuntan a un objeto común, como puede ser un array.En gestiones de pilas y colas suele utilizarse este concepto.

6.4.- Punteros y arrays

Existe una estrecha relación entre los punteros y los arrays. Puede considerarse el siguiente fragmento de programa:

```
char cad [80], *p1;
p1 = cad;
```

en donde a **p1** le ha sido asignada la dirección del primer elemento del array cad. Para acceder al quinto elemento del array se escribirá:

```
cad [4]
```

o bien:

```
*(p1+4)
```

Ambas instrucciones son análogas. (Recuérdese que los arrays comienzan en el elemento 0); otra forma de acceder al quinto elemento sería sumar 4 al puntero **p1** ya que el nombre del array sin índice devuelve la dirección del comienzo del array (que es el primer elemento).

En esencia, C proporciona dos métodos de acceso a los elementos del array: la aritmética de punteros y la indexación del array. Aunque la notación normal de indexación del array a veces es mas fácil de entender, la aritmética de punteros puede ser mas rápida.

Para ilustrar lo dicho, puede compararse las dos versiones que se muestran a continuación de una función (**putstr**( )) para imprimir los elementos de un array:

1º.- Por indexación de c como array

```
void putstr(char *c)
{
register int i;
for (I = 0; c[i] ; ++i)
        putchar(c[i]);
}
```

2º.- Por acceso de c como puntero

```
void putstr(char *c)
{
while (*c) putchar(*c++);
}
```

Como aplicación de lo expuesto puede considerarse una función ***copiarmatriz*** que permita copiar una matriz de cualquier tipo de datos y de cualquier tipo de dimensiones en otra matriz de análogas características:

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

#define FILAS 2
#define COLS 3

void CopiarMatriz (void *dest, void *orig, int n);

void main()
{
        int Matriz1[FILAS][COLS] = {24, 30, 15, 45, 34, 7};
        int Matriz2[FILAS][COLS], f, c;
        CopiarMatriz (Matriz1, Matriz2, sizeof(Matriz1));
        for (f = 0;f < FILAS;f++)
        {
                for (c = 0;c < COLS;c++ )
                        printf ("%d   ", Matriz2[f][c]);
                printf ("\n");
        }
getchar();
}

void CopiarMatriz (void *dest, void *orig, int n)
{
        char *destino = dest;
        char *origen = orig;

        int i = 0;
                for (i = 0;i < n;i++)
                {
                        destino[i] = origen[i];
                }
}
```

6.4.1.- Punteros a cadenas de caracteres

Puesto que una cadena de caracteres es una matriz de caracteres, es lógico pensar que todo lo expuesto anteriormente es perfectamente aplicable a las cadenas de caracteres. Un puntero a una cadena de caracteres puede definirse de alguna de las dos formas siguientes:

***char** *cadena;
***unsigned char** *cadena;

Para identificar el principio y el final e la cadena , la ***dirección de memoria*** donde comienza la cadena viene dada por el nombre de la matriz que la contiene y el final viene

dado por el carácter **\0** con el que C finaliza todas las cadenas. El siguiente ejemplo define e inicia una cadena de caracteres llamada *nombre*:

```
char * nombre = Francisco Javier";
printf ("%s", nombre);
```

en el que *nombre* es un puntero a una cadena de caracteres al que el compiladfor de C asigna la dirección del comienzo del literal "Francisco Javier". Posteriormente el compilador finaliza la cadena con el carácter \0. Por tanto, la función **printf** sabe que la cadena que tiene que visualizar comienza en la dirección *nombre* y, a partir de aquí, tiene que ir accediendo a posiciones sucesivas de memoria hasta encontrar el carácter \0.

Es importante considerar que *nombre* no tiene una copia de la cadena asignada sino la dirección de memoria del lugar donde la cadena está almacenada y que coincide con la dirección del primer carácter. Esto implica que el puntero se puede reasignar para apuntar a una nueva cadena:

```
char * nombre = Francisco Javier";
printf ("%s", nombre);
nombre = "Carmen";
```

Además, un literal, por tratarse de una constante de caracteres, no se puede modificar, por lo que la secuencia:

```
char * nombre = Francisco Javier";
nombre[9] = '-';
```

producirá un error. Sin embargo, este error no ocurre cuando la cadena de caracteres viene dada por una matriz que no haya sido declarada constante (**const**) como indica el ejemplo:

```
char nombre[ ] = Francisco Javier";
char *pnombre = nombre;
pnombre[9] = '-';
```

Un ejemplo de lo expuesto es el siguiente, que define una función que devuelve como resultado el número de caracteres de una cadena (longitud de la cadena), de forma análoga a la función **strlen** de la biblioteca de C.

```
#include <stdio.h>

int longcad(char *);

void main()
{
        char *cadena = "abcde";
        printf ("%d\n", longcad(cadena));
}


int longcad(char *cad)
{
        char *p = cad;
        while (*p != '\0')
                p++;
```

```
        return (p - cad);
}
```

Como se ve, la función realiza las siguientes operaciones:

1º.- Asigna a p la dirección del primer carácter de la cadena, que coincide con la dirección de comienzo de la misma e inicia la ejecución del ciclo while

2º.- Cuando se ejecuta la condición del bucle while se compara el carácter *p apuntado por p con el carácter nulo. si *p es el carácter nulo el bucle finaliza, si no se incrementa el valor de p en una unidad para que apunte al siguiente carácter y se vuelve a valuar la condición.

Observese que la expresión *p != '\0' es cierta (valor distinto de 0) cuando *p es un carácter distinto del nulo y falsa en el caso contrario, por lo que el programa anterior podría haberse escrito:

```
int longcad(char *cad)
{
        char *p = cad;
        while (*p )
                p++;
        return (p - cad);
}
```

Además al evaluar exclusivamente el carácter, podría evaluarse el caracter siguiente, por lo cual, el programa anterior quedará:

```
int longcad(char *cad)
{
        char *p = cad;
        while (*p++)
        return (p - cad-1);
}
```

El siguiente ejemplo presenta una función que copia una cadena en otra:

```
#include <stdio.h>
void copicad (char *, char *);

void main()
{
        char cadena1[81], cadena2[81];

        printf ("Introducir una cadena:    ");
        gets(cadena1);

        copicad (cadena2, cadena1);

        printf ("La cadena copiada es:    %s\n",cadena2);

}
```

```
void copicad( char *p, char *q)
{
        while ((*p = *q) != '\0')
        {
                p++;
                q++;
        }
}
```

que, aplicando los criterios anteriores la función copicad podía haberse escrito:

```
void copicad( char *p, char *q)
{
        while (*p++ = *q++)
}
```

6.4.2.- matrices de punteros

Los punteros pueden estructurarse en arrays como cualquier otro tipo de datos. Así, la declaración para un array de punteros a enteros de tamaño 10 es:

int *p[10] ;

Para asignar la dirección de una variable entera llamada **var** al tercer elemento del array se escribe

p[2] = &var ;

Y, para encontrar el valor de **var**, se escribe:

***p[2] ;**

Si se quiere pasar un array de punteros a una función, se puede utilizar el mismo método que se utiliza para otros arrays: llamar simplemente a la función con el nombre del array, sin índices. Por ejemplo, una función que reciba el array **p[10]** sería como esta:

```
Void mostrararray(int *q[ ])
{
int i;
for (i = 0 ; i < 10 ; i++)
        printf("%d%, *q[i]);
}
```

Recuérdese que **q** no es un puntero a entero sino un puntero a un array de punteros a enteros, por lo que hay que declarar el parámetro **q** como un array de punteros a enteros como se ha mostrado en el código anterior.

Los arrays de punteros frecuentemente se utilizan para mantener punteros a mensajes de error. Se puede crear una función que muestre un mensaje dado su número de código de error, como se indica en la función adjunta:

```
Void Errordesintaxis(int num)
{
static char *err[ ] = {
```

```
"No se puede abrir el archive \n",
"Error de lectura\n",
"Error de escritura\n",
"Fallo de dispositivo\n"
}
        printf ("%s", err[num]);
}
```

El array **err** guarda los punteros a cada cadena de error; esto funciona debido a que una constante de cadena que se usa en una expresión produce un puntero a la cadena. Se llama a la función **printf** con un puntero a carácter que apunta al mensaje de error indexado por el número de error pasado a la función.

6.5.- Punteros constantes

Una declaración de un puntero precedida por **const** hace que el objeto apuntado sea tratado como constante, no sucediendo lo mismo con el puntero. Por ejemplo;

Pueden considerarse los dos casos expuestos a continuación:

1º. Objeto constante y puntero variable

```
int a = 10, b = 20;
const int *p = &a;     //el objeto a es constante y el puntero p variable.
*p = 15 ;     //se producirá un error, el objeto apuntado por p debe ser constante.
p = &b;     //la sentencia es correcta; p pasa a apuntar a un nuevo objeto
```

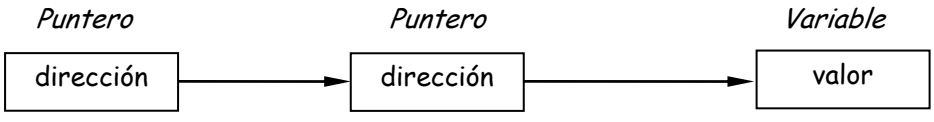
2º. Puntero constante y objeto variable

```
int a = 10, b = 20;

int   *cons p = &a;   // El objeto a es variable y el puntero p constante
*p = 15 ;   //la sentencia es correcta ya que el objeto apuntado por p es variable
p = &b;   //se producirá un error ya que el puntero p es constante.
```

6.6.- Indirección múltiple

Puede hacerse que un puntero apunte a otro puntero que, a su vez, apunte a un valor de destino. Esta situación se denomina ***indirección múltiple*** o ***punteros a punteros***. Los punteros a punteros pueden resultar confusos. La siguientefigura ilustra este concepto:

Indirección simple



Indirección múltiple

Como puede verse, el valor del puntero normal es la dirección del objeto que contiene el valor deseado. En el caso del puntero a puntero, el primer puntero contiene la dirección del segundo puntero el cual contiene la dirección del objeto que contiene el valor deseado.

La indirección múltiple puede llevarse al nivel que se desee, si bien existen muy pocos casos en que se necesite mas de un puntero a puntero. De hecho, la indirección excesiva es difícil de seguir y muy propensa a errores.

Una variable que es puntero a puntero debe declararse como tal, lo que se hace adicionando un * delante del nombre de la variable. Así, la siguiente declaración indica al compilador de **balance** es un puntero a puntero de tipo float:

float ****balance;**

Para acceder al valor de destino indirectamente apuntado por un puntero a puntero hay que poner dos veces el oerador asterisco como en este ejemplo:

```
#include <stdio.h>
void main()
{
int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf ("%d", **q);
}
```

Aquí se declara **p** como un puntero a un entero y **q** como un puntero a puntero a entero. La impresión **printf**( ) imprime el vallor 10 de la variable **x** en la pantalla.

6.7.- Inicialización de punteros

Después de declarar un puntero local no estático, pero antes de asignarle un valor, contiene un valor desconocido. Si se intenta utilizar el puntero antes de darle valor seguramente se producirá un error, bien en el programa, bien en el sistema operativo.

Existe una convención entre los programadores de C cuando trabajan con punteros, consistente en que un puntero que asigne a una dirección no válida debe ser asignado al valor nulo (que es un 0). Se usa el nulo por que C garantiza que no existe ningún objeto en la dirección nula . Así cualquier puntero que sea nulo indica que no apunta a nada y no debe ser usado.

Como se ha dicho, la forma de dar a un puntero el valor nulo es asignándole un 0:

char *p = 0;

Además, muchos archivos de cabecera de C, como stdio.h, definen la macro NULL como una constante a puntero nulo, por lo que, también está admitida una instrucción como:

```
p = NULL;
```

Un ejemplo que utiliza punteros nulos es el que se describe en el siguiente programa en la funció **busca**:

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

int busca (char *p[ ], char *nombre);
char *nombres[ ] = {"Hector", "Ramon", "Felipe", "Daniel", "Juan", NULL};
void main()
{
        if (busca (nombres,"Daniel") != -1)
                printf ("Daniel está en la lista.\n");

        if (busca (nombres,"Jaime") == -1)
                printf ("Jaime no se encuentra en la lista.\n");
}

int busca (char *p[ ], char *nombre)
{
        register int t;
        for (t = 0; p[t] ; ++t)
                if (!strcmp(p[t], nombre )) return t;
                return -1; // no se ha encoontrado el nombre
}
```

La función **busca** () tiene dos parámetros: el primero, **p** es una matriz de punteros **char *** que apuntan a cadenas que contienen nombres. El segundo, **nombre** es un puntero a una cadena que contiene el nombre que se busca. La función recorre la lista de punteros buscando una cadena que coincida con la apuntada por **nombre**. El bucle **for** dentro de **busca** () se ejecuta hasta que se encuentra una coincidencia o hasta que se encuentra el puntero nulo.

Es práctica común el inicializar los punteros **char *** de forma que apunten a constantes de cadena, como se ha hecho en el programa anterior y en la siguiente instrucción:

```
char* p = "Hola amigo";
```

p, como se ve, es un puntero no una matriz , por lo que la cadena anterior no puede esatr guardada en **p**, la realidad es que el compilador de C maneja las cadenas de caracteres de forma muy peculiar: crea lo que se llama una ***tabla de cadenas*** en la que se guardan las constantes de cadena utilizadas por el programa , por lo que la anterior instrucción pone en el puntero **p** la dirección que tiene la cadena "Hola amigo" en la tabla de cadenas. Hasta el final del programa **p** puede utilizarse como cualquier cadena.

6.8.- Punteros a funciones

Dado que una función tiene una dirección física de memoria de su comienzo, su dirección puede asignarse a un puntero. Una vez que un puntero apunta a una función, este

puntero se podrá utilizar para invocar a la función. Los punteros a funciones también se pueden pasar como argumentos de otras funciones.

La dirección de una función se obtiene ***utilizando el nombre de la función sin paréntesis ni argumentos*** (es decir, es parecido a la forma de obtener la dirección de una array cuando se utiliza el nombre del array, sin límites). Un programa que ilustra la utilización de punteros a funciones es el siguiente:

```
/*Punteros a funciones:
Función para comparar dos cadenas proporcionadas por el usuario */
#include <stdio.h>
#include <string.h>
void comprobar( char* a, char* b, int   (*cmp) (const char*, const char*));

int main(void)
{
        char c1[80], c2[80];
        int (*p)(const char*, const char*); //puntero a función

        p = strcmp; //se asigna la dirección de la función strcmp al puntero p

        printf ("introducir dos cadenas de caracteres\n");
        gets(c1);
        gets(c2);

        comprobar (c1,c2,p); //Pasa la dirección de la función strcmp mediante p

        return 0;

}

void comprobar( char* a, char* b, int   (*cmp) (const char*, const char*))
{
        printf ("Comprobando la igualdad....:   ");
        if (!(*cmp)(a,b)) printf ("Iguales");
        else printf ("Diferentes");
}
```

Ejemplos típicos de utilización de punteros a funciones son aquellos programas en los que se elige una opción entre un grupo de opciones, por ejemplo, desde un menú en la pantalla.

Si a cada selección le corresponde una función de tratamiento, las llamasdas se podrían programar mediante sentencias **if** anidadas o la sentencia **switch** pero es mucho más rápido y cómodo utilizar punteros a funciones, sobre todo cuando hay muchos casos. Esta es la forma mas general, ya que sólo es necesario definir la matriz con las funciones que se emplean

En general, la forma de declarar un puntero a una función es:

***tipo* (*p) ();**

donde p es el nombre del puntero y tipo es el tipo de valor que devuelve la función . En la declaración del puntero los paréntesis son necesarios porque:

***tipo* *p();**

es una declaración de una función que devuelve un puntero.

Además debe tenerse en cuenta que las funciones a las que se quiera invocar utilizando un puntero deben declararse previamente.

La llamada a la función tendrá la siguiente forma:

(*p)( )

El siguiente programa es ilustrativo de la utilización de punteros a funciones:

```
#include <stdio.h>

int fun1(), fun2(), fun3();

int (*pf)();

void main()
{
        pf = fun1;
        printf ("Direccion de fun1 = %p\n",pf);
        printf ("Direccion de fun1 = %p\n",fun1);

        pf = fun2;
        printf ("Direccion de fun2 = %p\n",pf);
        printf ("Direccion de fun2 = %p\n",fun2);

        pf = fun3;
        printf ("Direccion de fun3 = %p\n",pf);
        printf ("Direccion de fun3 = %p\n",fun3);

}

int fun1()
{
}

int fun2()
{
}

int fun3()
{
}
```

Una función se puede pasar como argumento a otra función, incluyendo el nombre de la función en la lista de parámetros y declarando el parámetro formal, dentro de la función, como un puntero a funciones. Un ejemplo sería:

```
#include <stdio.h>
void ejecuta( int (*f)( ));
void main()
{
ejecuta(listar);
ejecuta(ordenar);
}
void ejecuta( int (*f)( ));
{
(*f)( );
}
```

La función **main** llama a la función **ejecuta( )** con un argumento, el nombre de la función a ejecutar. La función **ejecuta( )** define el parámetro que recibe como un puntero a funciones de tipo entero, es decir, funciones que devuelven un valor entero. La llamada a la función se realiza con la única instrucción que hay en la función **ejecuta( )**, que representa la forma mas general para invocar a una función desde un puntero.

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

void comprobar (char* a, char* b, int (*cmp)(const char*, const char*));
int numcmp(const char* a, const char* b);

void main()
{
        char c1[80], c2[80];

        printf("Introducir dos valores o cadenas:\n");
        gets(c1);
        gets(c2);

        if(isdigit(*c1))
                {
                        printf("\nProbando la igualdad de los valores....: ");
                        comprobar (c1, c2, numcmp);
                }

        else
                {
                        printf("\nProbando la igualdad de las cadenas....: ");
                        comprobar (c1, c2, strcmp);
                }
}
```

```
void comprobar (char* a, char* b, int (*cmp)(const char*, const char*))

{
        if (!(*cmp)(a, b)) printf("Iguales");
        else printf("Diferentes");
}


int numcmp(const char* a, const char* b)
{
        if(atoi(a) == atoi(b)) return 0;
        else return 1;
}
```

6.9.- Paso de estructuras a funciones

6.9.1.- Paso de miembros de estructuras a funciones

Cuando se pasa un miembor de una estructura a una función, se está realmente pasando el valor de ese miembro a la función. No es relevante que el valor se este obteniendo del miembro de una estructura. Así, para utilizar como parámetro el miembro ***elemento*** de la estructura ***estructura*** en la función ***funcion***:

funcion(... , estructura.elemento, ...);

Si se quiere pasar la ***dirección*** de un miembro individual de la estructura, se debe colocar el operador **&** antes del nombre de la estructura. Así, en el caso anterior:

funcion(... ,&estructura.elemento, ...);

Obsérvese que el operador **&** precede al nombre de la estructura, no al nombre del miembro individual. Debe considerarse también que si elemento es de tipo caráctrer, el propio nombre ya indica la dirección, por lo que, en este caso, el operador **&** es innecesario.

6.9.2.- Paso de estructuras a funciones

Cuando se utiliza una estructura como argumento de una función, se pasa la estructura íntegra mediante el uso del método estándar de llamada por valor. Esto significa,} por supuesto, que todos los cambios realizados en los contenidos de la estructura dentro de la función a la que se pasa no afectan a la estructura utilizada como argumento. Por ejemplo, si se quere utilizar la estructura ***estructura*** del tipo de estructura ***ficha*** como parámetro de la función ***funcion***:

funcion (..., struct ficha estructura, ...);

Cuando se utilice una estructura como parámetro, se debe recordar que el tipo de argumento debe cuadrar con el tipo del parámetro, así, si se van a declarar parámetros que son estructuras , se debe hacer global la declaración del tipo de estructura, de forma que pueda usarse en todas las partes del programa.

6.10.- Punteros a estructuras

C permite punteros a estructuras igual que permite punteros a cualquier otro tipo de variables. Sin embargo, hay algunos aspectos especiales que afectan a los punteros a estructuras, que se describen a continuación.

6.10.1.- Declaración de un puntero a una estructura

Al igual que los demás punteros, los punteros a estructuras se declaran poniendo * delante del nombre de la variable estructura. Por ejemplo, asumiendo que se ha definido previamente la estructura **dir**, lo siguiente declara **pdir** como un puntero a un dato de ese tipo:

struct dir * pdir;

6.10.2.- Uso de punteros a estructuras

Existen dos usos principales de los punteros a estructuras: generar un paso por referencia de una estructura a una función y crear listas enlazadas y otras estructuras de datos utilizando la "***asignación dinámica de memoria***" (tema 8). Ahora se trata exclusivamente el primer uso.

Existe un importante inconveniente en el paso de cualquier estructura a una función, excepto para las mas simples: la ***sobrecarga*** que supone introducir la estructura en la pila cuando se realiza la llamada a la función. (Recuérdese que los argumentos se pasan a la funcióna través de la pila). Para estructuras simples, de pocos miembros, la sobrecarga no es muy significativa, pero si la estructura contiene muchos miembros o si alguno de esos miembros es un array, el tiempo de ejecución puede degradarse a niveles inaceptables. La solución es pasar a la función un puntero a la estructura

Cuando se pasa a una función un puntero a una estructura, sólo se introduce en la pila la dirección de la estructura, esto significa que la llamada a la función es muy rápida. Una segunda ventaja es que, en algunos casos, el paso del puntero permite a la función modificar el contenido de la estructura utilizada coimo argumento.

Para encontrar la dirección de una variable de estructura se coloca el operador **&** antes del nombre de la estructura. Por ejemplo, el siguiente fragmento de programa:

```
struct cuenta
{
        float anotacion;
        char concepto[80];
} caja;
struct cuenta *p;   // declaración de un puntero a la estructura
```

Lo que sigue coloca la dirección de la estructura ***caja*** en el puntero ***p***:

```
p = &caja;
```

Para acceder a los miembros de una estructura a través de un puntero a esa estructura se debe utilizar el operador **->**. Por ejemplo, para referenciar el campo ***anotacion*** se usa:

p -> anotacion;

A **->** se le denomina ***operador flecha*** y consiste en el signo menos seguido del un signo de mayor. Cuando se accede a un miembro de una estructura a través de un puntero a la estructura, se usa el operador flecha en lugar del operador punto.

Para ver cómo se puede usar un puntero a una estructura, puede considerarse el programa que sigue, que imprime en la pantalla las horas, los minutos y los segundos utilizando un temporizador software

```
#include <stdio.h>

#define DESFASE 128000

struct tiempo
{
        int horas;
        int minutos;
        int segundos;
};


void mostrar (struct tiempo *t);
void actualizar (struct tiempo *t);
void retardo (void);

int main(void)
{
        struct tiempo ahora;

        ahora.horas = 0;
        ahora.minutos = 0;
        ahora.segundos = 0;

        while(1)
  {
                actualizar(&ahora);
                mostrar(&ahora);
        }
        return (0);
}


void actualizar (struct tiempo *t)
{
        t->segundos++;
        if (t->segundos == 60)
```

```
                {
                        t->segundos = 0;
                        t->minutos++;
                }

        if (t->minutos == 60)
                {
                        t->minutos = 0;
                        t->horas++;
                }

        if (t->horas == 24)t->horas = 0;
        retardo();
}



void mostrar(struct tiempo *t)
{
        printf ("%02d: ",t-> horas);
        printf ("%02d: ",t-> minutos);
        printf ("%02d\n\n ",t-> segundos);
}


void retardo (void)
{
        long int t;
        for (t=1; t < DESFASE; t++);
}
```

Como se observa en el programa, la sentencia:

```
t -> segundos++;
```

indica al compilador que tome la dirección de t (la dirección de la estructura apuntada por t) y asigne al miembro ***segundos*** el volor que tenía incrementado en una unidad.

En resumen, recuérdese que se ha de utilizar el operador punto para acceder a los miembros de estructuras, cuando se trabaja directamente con ellas. Cuando se tiene un puntero a una estructura se ha de utilizar el operador flecha.