INTRODUCCIÓN

La memoria es la parte de de la computadora encargada de almacenar la información que ésta maneja, la misma guarda tanto los programas como los datos implicados en la ejecución de los mismos.

Todos los datos se almacenan a partir de una dirección de memoria y utilizando tantos bytes como sea necesario, es por ello que conviene detenernos brevemente para estudiar algunas cuestiones relativas a las variables y la memoria que ocupan.

Cabe recordar que la memoria es una sucesión de celdas numeradas y que una dirección de memoria no es más que un número entero.

La declaración de una variable supone la reserva de una zona de memoria lo suficientemente grande para albergar su contenido.

Es decir, cuando declaramos una variable de un tipo simple, por ejemplo, int, se reservan 4 bytes de memoria en los que se almacenará el valor de dicha variable.

Tomemos el siguiente ejemplo:

```c
#include <stdio.h>
int main()
{
    int x, y;

    x = 0;
    y = x + 8;

    return 0;
}
```

Este fragmento de código reserva espacio para dos enteros de 4 bytes cada uno, inicialmente, cuando se reserva la memoria, esta contiene un patrón de bits arbitrario.

Al ejecutarse la sentencia x = 0 se interpreta como, "almacenar el valor 0 en la dirección de memoria de la variable x", es decir, suponiendo que se almacenan a partir de la dirección de memoria 100 se interpreta como "almacenar el valor 0 a partir de la posición de memoria 100", e "y" a partir de la posición 104.

Si quisiéramos representar gráficamente esta situación se vería idealmente de la siguiente forma:

Dirección de Memoria	Contenido	Id variable
100	00000000	x
101	00000000	
102	00000000	
103	00000000	
104	00000000	y
105	00000000	
106	00000000	
107	00001000	

Ahora bien ,normalmente no necesitamos saber en qué dirección de memoria se almacena una variable, o por lo menos hasta el momento no lo necesitamos, sino que nos conformamos con representar las variables escalares mediante cajas y su contenido de una forma más cómodamente legible. La representación anterior entonces se simplificará, asi:

a	0
b	8

De las dos gráficas vistas anteriormente podemos inferir entonces que toda variable tiene un identificador, un tamaño, una posición de memoria y un contenido, lo importante es la relación entre estos elementos y cómo se almacenan.

El programador al realizar la declaración de la variable, reserva un espacio de almacenamiento en memoria, conoce el nombre de la misma, pero el Sistema Operativo (S.O.) es quien asigna la dirección de memoria.

Es decir, el S.O. encuentra un espacio libre de memoria y la asocia con el nombre de la variable.

La pregunta que surge en este momento es ,¿Cómo y para qué conocer la dirección de memoria de una variable?.

Punteros

Los punteros son uno de los aspectos más potentes de la programación en C, permiten manipular la memoria del ordenador de forma eficiente. Dos conceptos fundamentales para comprender el funcionamiento de los punteros son:

- El tamaño de las variables y su posición en memoria.
- Todo dato está almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada también como un dato más.

El correcto entendimiento y uso de los punteros es crítico para una productiva programación en C, entre otras razones debido a que proporcionan los medios por los cuales las funciones pueden modificar su argumentos en la llamada y su uso puede mejorar la eficiencia de ciertas rutinas.

Variables puntero

Si una variable va a contener un puntero, entonces tiene que declararse como tal. La declaración de un puntero consiste en un tipo base, un asterisco (*), seguido del nombre de la variable.

La forma general de declarar una variable puntero es:

```
<Especificador_de_tipo><*><Identificador_variable_puntero>
```

Donde <Especificador_de_tipo> es un tipo de dato válido de C y el <Identificador_variable> que es el nombre propio de la variable que funcionará como puntero.

Ejemplo:

```
...
int *p;        //Declaración de un puntero a un entero
char *n;       //Declaración de un puntero a un carácter
float *p1;     //Declaración de un puntero a un float
...
```

Técnicamente, cualquier tipo de puntero puede apuntar a cualquier lugar de la memoria, pero el lenguaje C asume que a lo que apunta es a un objeto de su tipo base, y toda la aritmética asociada está hecha en relación a su tipo base, por lo que es muy importante declarar correctamente el puntero.

Operadores de punteros

Existen dos operadores especiales de punteros: **&** y *****, son operadores unarios y tienen más prioridad a la hora de evaluarlos que los operadores binarios.

El operador & (anpersand) seguido del nombre de una variable devuelve su dirección de memoria y se interpreta como "la dirección de". Sólo sirve para posiciones de memoria (puede apuntar a variables o vectores, pero no a constantes o expresiones).

El operador (asterisco) ***** seguido del nombre de la variable me da el contenido de una posición de memoria, y se interpreta con "en la dirección de" (generalmente almacenada en un puntero).

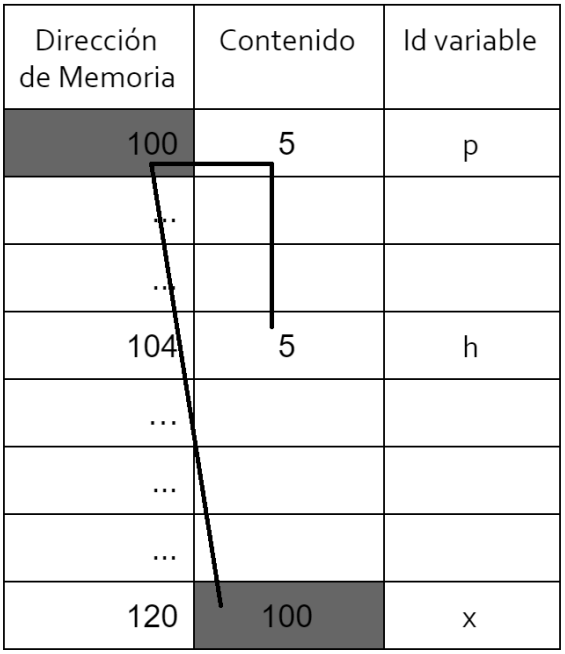
Veamos un ejemplo sencillo para aclarar el uso de estos operadores:

```c
#include <stdio.h>
int main()
{
    int *x; //x es una variable tipo puntero a un int
    int p, h;

    p = 5; // A p se le asigna 5
    x = &p;  // x recibe la dirección de p
    h = *x; /* A h se le asigna el contenido de p indirectamente
               a través de x*/
    printf("El valor de h es: %d", h);//Muestra el valor 5

    return 0;
}
```

Para graficar esta situación vamos a suponer que la direcciones de memoria comienzan en la posición 100, entonces la gráfica se vería idealmente de la siguiente forma:



Dirección de Memoria	Contenido	Id variable
100	5	p
…		
…		
104	5	h
…		
…		
…		
120	100	x

La gráfica es una representación simplificada para que sea más fácil ver el paso a paso de la situación pero recuerden que cada variable reserva el espacio correspondiente al tipo de dato que utilizamos en la declaración de la misma.

Como podemos observar la variable puntero x, contiene la dirección de p, entonces x apunta a la variable p.
Ahora, el contenido de la variable p se puede manipular usando tanto el identificador p como con la notación *x.

Veamos los siguientes ejemplos reutilizando el fragmento de código anterior:

```c
#include <stdio.h>
int main()
{
    int *x;
    int p;

    p = 9;  // A p se le asigna 9
    x = &p; // x recibe la dirección de p
    //Por lo tanto estas dos salidas son equivalentes
    printf("%d\n", p);
    printf("%d\n", *x);
    return 0;
}
```

Mirando este ejemplo podemos afirmar entonces que p=9 es equivalente a hacer *x=9; siempre que las declaraciones sean correctas.

También y como lo venimos haciendo hasta ahora, podemos leer desde el teclado de la siguiente forma:

```
...
scanf("%d", &p); //Leemos p, a través de la dirección
...
```

Cabe recordar en este punto que las variables puntero deben apuntar siempre al tipo de dato correcto. Es decir cuando se declara un puntero a un tipo int, el compilador asume que cualquier dirección que contenga apunta a una variable entera. Debido a que C permite asignar cualquier dirección a una variable puntero, debemos tomar nosotros dicha precaución.

Veamos un ejemplo, con un código que si bien no da error, no produce el resultado deseado, en este caso el lenguaje C, mostrará un mensaje de advertencia:

```
#include <stdio.h>
int main()
{
    float x, y;
    int *p;

    x = 12.45;
    p = &x;
    y = *p; //y es float, le asigna la dirección del puntero a int
    printf("%f\n", y);

    return 0;
}
```

En este fragmento de código no se asignará el valor de "x" a "y", tal y como se ve. Debido a que p se declara como un puntero a un entero y no tiene en cuenta que hay que tomar tantos bytes como corresponden a un float.

Expresiones de punteros

Las expresiones que involucran punteros se ajustan a las mismas reglas que cualquier otra expresión en C, teniendo en cuenta algunos aspectos especiales.

Inicialización de punteros

Si se usa un puntero antes de darle valor probablemente falle el programa y el sistema operativo, por eso es tan importante la inicialización de ellos. Se puede hacer de dos formas: la primera es asignando la dirección de una variable y la segunda es inicializarlo en un valor nulo.

Ejemplo:

```
void main ()
{
  int var, *Ap1, *Ap2, *Ap3;
  Ap1=&var; //Su contenido depende de la dirección de var
  Ap2=NULL;
  Ap3=0;
  //Tanto NULL como cero son equivalentes
}
```

Asignación de punteros

Como en el caso de cualquier otra variable, un puntero puede utilizarse a la derecha de una expresión de asignación para asignar su valor a otro puntero.

Veamos el siguiente ejemplo:

```
#include <stdio.h>
int main()
{   int x;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;//Asignación de un puntero a otro
    /*A continuación se muestran las direcciones contenidas en p1 y p2.
     Tanto p1 como p2 apuntan ahora a x*/
    printf("%d\n", p1);
    printf("%d", p2);
    return 0;
}
```

En este fragmento de código se declaran, un entero y dos punteros a enteros respectivamente. Los punteros se comportan como el resto de variables, no tienen valor inicial al comienzo del programa. Luego siguiendo las líneas de código se asigna la dirección de x a las dos variables p1 y p2. La dirección de una variable existe desde el principio de un programa, y por tanto esta asignación es correcta a pesar de que todavía no hemos almacenado nada en la variable x.

Aritmética de punteros

Si declaramos una variable como int n=2 y posteriormente hacemos n++, debería resultar claro que lo que ocurre es que aumenta en una unidad el valor de la variable n, pasando a ser 3. Pero ¿qué sucede si hacemos esa misma operación sobre un puntero?

Para entender qué ocurre en la aritmética de punteros, observemos el siguiente fragmento de código:

```
...
 int valor, *p;

 p=&valor;
 *p=2;
 p++;
...
```

Luego de ejecutar estas líneas de programa, lo que ha ocurrido no es que el contenido de a lo que apunta p sea 3. Eso se consigue modificando "*p", es decir, deberíamos escribir:

```
...
(*p)++;
...
```

En la situación anterior hemos incrementado el valor de "p". Como "p" es un puntero lo que en realidad estamos modificando es la dirección a la que apunta "p".

Analicemos entonces las siguientes situaciones: sea p1 un puntero con un valor actual de 2000 y asumiendo que los enteros ocupan 4 bytes de longitud, luego de la siguientes asignaciones:

```
...
p1++;
p1--;
...
```

al incrementar o decrementar p1 contendrá 2004 o 1996 respectivamente. Es decir, que cada vez que se incrementa o decrementa p1, apunta al siguiente o anterior entero.

En general, cada vez que se realiza este tipo de operaciones a un puntero, el mismo apunta a la posición de memoria del siguiente o anterior elemento de su tipo base.

Estas operaciones no están limitadas a los operadores de incremento y decremento. Se pueden sumar o restar constantes. Ejemplo:

```
...
p1=p1+9;
...
```

Este fragmento de código hace que p1 apunte al noveno elemento del tipo base al que apunta actualmente p1.

No olvidar que manipular posiciones de memoria puede dar lugar a fallos muy difíciles de descubrir o incluso a que el programa se interrumpa con un aviso indicando "desbordamiento", porque estamos accediendo a zonas de memoria que no hemos reservado.

Comparación de punteros

No es muy común hacer esta operación y generalmente se utiliza cuando dos o más punteros apuntan a un objeto en común, pero igual se pueden comparar dos punteros en una expresión relacional.

Puede usar los operadores ==, !=, <, >, <= y >= para comparar los operandos de cualquier tipo de puntero. Estos operadores comparan las direcciones proporcionadas como si fueran enteros sin signo.

Por ejemplo dados dos punteros p1 y p2, la siguiente expresión es totalmente válida:

```
...
if (p1==p2)
    {
        printf("Los punteros apuntan a la misma direccion");
    }
...
```

Consideraciones importantes

Utilizamos punteros para acceder a la información a través de su dirección de memoria.

Aunque & y * representa AND a nivel de bits y multiplicación cuando se usan como operadores punteros tienen mayor prioridad que todos los operadores aritméticos.

La dirección de memoria se puede obtener al usar el operador & (anpersand) delante de cualquier variable.

El puntero debe contener una dirección a un elemento del mismo tipo que la variable apuntada, de lo contrario puede dar resultados no deseados.

Las direcciones de memoria pueden utilizarse en su forma hexadecimal para ello debo utilizar el especificador de formato %p

Las direcciones de memoria dependen de la arquitectura de la computadora y de la gestión que el sistema operativo haga de ella.

Desde C no es posible indicar numéricamente una dirección de memoria para guardar información (esto se hace a través de funciones específicas).

Punteros a funciones

Como ya se mencionó en el apartado de funciones, en general, se pueden pasar argumentos a las subrutinas de dos formas. La primera forma se denomina pasaje por valor. Este método pasa el valor de un argumento como parámetro de una subrutina. De esta forma los cambios en los parámetros de la subrutina no afectan a las variables que se usan en la llamada.

El pasaje por referencia es la segunda forma de pasar argumentos a una subrutina. En este método, se pasa la dirección del argumento como parámetro en la subrutina, de esta forma, al usar la dirección dentro de dicha subrutina los cambios hechos a los parámetros afectan a la variable usada como argumento de la función.

Consideremos el siguiente ejemplo utilizando pasaje por valor:

```c
#include <stdio.h>
void doble(int);
int main()
{   int a = 4;
    printf("Inicialmente la variable vale %d\n", a);
    doble(a);//llama a la función con el parámetro a
    printf("Finalmente la variable vale %d\n", a);
    return 0;
}
void doble(int x)
{
    x *= 2;
    printf("El doble es %d\n", x);
}
```

La salida de este fragmento de código es la siguiente:

```
Inicialmente la variable vale 4
El doble es 8
Finalmente la variable vale 4
```

Como se puede ver, el valor de la variable pasada a la función "doble" se modificó dentro de dicha función pero no tuvo impacto en la variable "a" que se utilizó como parámetro en la subrutina.

Recuerde que lo que se pasa a la función es una copia del valor del argumento. Lo que ocurre dentro de la función no tiene efecto sobre la variable utilizada en la llamada.

Veamos ahora la misma situación a través del paso por referencia utilizando un puntero como parámetro de la función, donde se pasa la dirección como parámetro de la subrutina, de esta forma es posible alterar el valor de la variable que fue llamada desde el exterior.

Cabe destacar que cualquier función que utilice parámetros de tipo puntero debe ser llamada con la dirección de lo o los parámetros que llaman a la función para ser ejecutada.

Consideremos el mismo ejemplo mediante el uso de punteros, pasaje por referencia:

```
#include <stdio.h>
void doble(int *);//El parámetro de la función es un puntero a int
int main()
{
    int a = 4;
    printf("Inicialmente la variable vale %d\n", a);
    doble(&a); //Llama a la función con la dirección de "a"
    printf("Finalmente la variable vale %d\n", a);
    return 0;
}
void doble(int *x)
{
    *x *= 2; //El * delante de "x" accede a la variable apuntada
}
```

La clave en el pasaje por referencia, está en el uso del tipo de variable puntero o en el correcto pasaje haciendo referencia a la dirección de la variable involucrada.

Punteros y arrays

Existe una estrecha relación entre los punteros y los arrays, a tal punto que el nombre del vector es en sí mismo un puntero a la primera posición del vector. Todas las operaciones que utilizan vectores e índices pueden realizarse mediante punteros.

Observemos el siguiente fragmento de código:

```
...
int lista[5]={12,14,10,6,8},*p;
p=lista;//Se le asigna la dirección del primer elemento de lista
...
```

A partir de esta misma declaración y asignación, para acceder por ejemplo al quinto elemento de la lista, se escribe:

```
...
lista[4] /*o bien*/ *(p+4)
...
```

Ambas sentencias devuelven el quinto elemento. Recuerden que los arrays comienzan en cero, es por eso que se usa 4 para indexar el quinto elemento y se suma 4 al puntero para el mismo fin.

Notar que para incrementar la posición del puntero el datos utilizando la suma, se escribe entre paréntesis, es decir no es lo mismo *(p+4) que *p+4, la primera opción incrementa la posición, la segunda incrementa al contenido. ¡Ojo!

Recordar que el nombre del array sin indice devuelve la dirección de comienzo del mismo, por lo cual, teniendo en cuenta el fragmento anterior, para acceder a la dirección se puede escribir:

```
...
lista  /*o bien*/  p
...
```

Una advertencia, al ejecutar los distintos programas propuestos, puede que obtengas un resultado diferente en tu ordenador. La asignación de direcciones de memoria a cada objeto de un programa es una decisión que adopta el compilador con cierta libertad.

Ejemplo de recorrido de un vector utilizando un índice.

```c
#include <stdio.h>
int main()
{
    int v[5] = {1, 2, 3, 4, 5};

    printf("Informe del vector");
    for (int i = 0; i < 5; i++)
    {
        printf("\n%d", v[i]);
    }
    return 0;
}
```

El mismo ejemplo de recorrido de un vector pero esta vez utilizando un puntero.

```c
#include <stdio.h>
int main()
{
    int v[5] = {1, 2, 3, 4, 5};
    printf("Informe del vector");
    for (int i = 0; i < 5; i++)
    {
        printf("\n%d", *(v + i));
    }
    //Cada *(v+i) el puntero señala a la siguiente posición
    return 0;
}
```

Note que en este último fragmento no declaramos un puntero ya que el vector se comporta como "un puntero" a la primera posición del mismo.

La dualidad puntero/vector

Si bien existe una dualidad entre el puntero y el nombre del vector hay que tener en cuenta que no son lo mismo. El lenguaje C permite considerarlos como una misma cosa en muchos contextos, hay algunas diferencias, pero esta ambigüedad me permite realizar las misma tareas teniendo mucha precaución a la hora de utilizarlos.

Consideremos como ejemplo las siguientes declaraciones:

```c
...
int vec[5];
int valor;
int *p;
int *p1;
...
```

A los punteros se les debe asignar explícitamente algún valor:

```c
...
p = NULL;    //Apunta a nada
p = &valor;  //Dirección de memoria de una variable
```

```
p = vec;     //Dirección de memoria de comienzo del vector
p = &vec[2]; //Dirección de memoria de un elemento del vector
p = p1;      // Dirección de memoria apuntada por otro puntero
p = vec + 2; //Dirección calculada mediante aritmética de punteros
...
```

Cuando declaras una variable del tipo array se reserva memoria automáticamente, pero no puede ser redimensionada, una sentencia como la que se muestra a continuación es ilegal:

```
...
vec = p;
...
```

Las funciones que admiten el paso de un array como parámetro o argumento, admiten también un puntero y viceversa. De ahí que se consideren equivalentes.

Reformulamos los ejemplos de recorrido de un vector utilizando funciones con punteros:

```
#include <stdio.h>
void Informe1(int[]); //Utilizando un índice
void Informe2(int *); //Utilizando un puntero
int main()
{
    int v[5] = {1, 2, 3, 4, 5};
    printf("Informe del vector con un indice");
    Informe1(v);
    printf("Informe del vector con punteros");
    Informe2(v);
    return 0;
}
void Informe1(int a[])
{
    for (int i = 0; i < 5; i++)
            printf("\n%d", a[i]);
}
void Informe2(int *a)
{
    for (int i = 0; i < 5; i++)
           printf("\n%d", *(a + i));
}
```

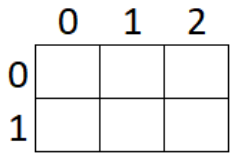
Para pasar todo un arreglo a una función, el nombre del arreglo debe aparecer solo, sin corchetes, ni índices. El correspondiente argumento formal o prototipo debe ser declarado como un arreglo, si se trata de un vector se escribe con un par de corchetes que pueden estar vacíos.

Los arreglos multidimensionales se definen prácticamente de la misma manera que los unidimensionales, hay que tener en cuenta que la memoria es lineal, y así se almacena el array. Nosotros podemos almacenar un array en línea y acceder a él a saltos, convirtiéndolo así en un array de n dimensiones.

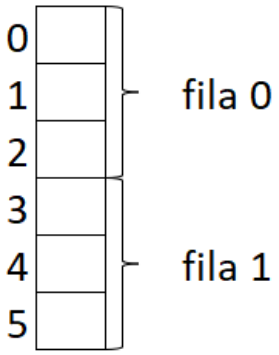
Si pensamos entonces en lo expuesto en el párrafo anterior podemos afirmar que, una matriz multidimensional se declara como una serie consecutiva, siguiendo el tipo base. Por ejemplo, para declarar una matriz bidimensional de tamaño fijo de enteros de 2 filas por 3 columnas, debería escribir la siguiente declaración:

```
int matrix[2][3];
```

Gráficamente o visualmente esta estructura se vería de la siguiente forma:



Si tenemos en cuenta que la memoria es lineal nuestra gráfica se vería con un diseño diferente:



La matriz se puede identificar mediante la dirección de comienzo del primer elemento, la relación entre punteros y matrices es estrecha como ya se mencionó en el uso de vectores. Teniendo en cuenta las gráficas anteriores podemos identificar un elemento de la matriz de formas diferentes dependiendo de la declaración utilizada:

```
matrix[1][0];
*(matrix+3);
```

El uso de punteros para manejar matrices es, en general, más eficiente y rápido que el uso de índices. Cuál es el método más adecuado (punteros o índices) depende del tipo de acceso. Si se va a acceder a los elementos de la matriz de forma aleatoria, es mejor utilizar índices. Sin embargo, si el acceso va a ser secuencial, es más adecuado usar punteros. Este segundo caso es típico en las cadenas de caracteres que desarrollaremos más adelante.

Los siguientes fragmentos de código muestran por pantalla la matriz completa el primer ejemplo la recorre secuencialmente de forma lineal, en este caso es menos visual ya que tiene todo el aspecto de un vector. La segunda forma hace el recorrido tal y como vemos gráficamente una matriz y utilizando aritmética de punteros, (es importante previamente asignarle valor al puntero):

Ejemplo 1: Muestra la matriz completa tal y como se asigna en "memoria":

```
...
int a[2][3]={{1,2,3},{4,5,6}};
int *p;
p = &a [0][0];//p apunta al primer elemento de la matriz
for (int i = 0; i < 6; i++)
     {
     printf("-%d-\n",*p+i);
     }
...
```

Ejemplo 2:

```
...
int a[2][3]={{1,2,3},{4,5,6}};
int *p;
p = &a [0][0];//p apunta al primer elemento de la matriz
for(int i=0;i<2;i++){
           for (int j=0;j<3;j++)
           {
           printf("%d ",*(p+(i*3)+j));
           }
           printf("\n");
     }
...
```

Punteros y arrays de caracteres

Ya hemos visto el uso que se le puede dar a un puntero como si de un array se tratase, entonces usando esta misma lógica podemos utilizar un array de caracteres usando punteros.

Muchas operaciones de cadenas en C se realizan normalmente usando punteros y aritmética de punteros porque tiende a acceder a las cadenas en un modo puramente secuencial. Son ejemplo de ello, strcmp(), strcpy(), etc.

Veamos el siguiente ejemplo, una función que toma como parámetro de entrada un puntero a una cadena y la muestra por pantalla:

```c
#include <stdio.h>
void mostrar(char *);
int main()
{
    char nombre[20]; //Declaramos un array de caracteres
    printf("Ingresar un nombre ");
    gets(nombre);
    mostrar(nombre); //Llamamos a la función con el char
    return 0;
}
void mostrar(char *nom)
{
    printf("%s", nom); //Escribimos en pantalla nombre
}
```

Otro ejemplo de una función que muestra por pantalla una serie de cadenas de caracteres utilizando como parámetro un puntero a la matriz de caracteres:

```c
#include <stdio.h>
void mostrar(char (*)[20]);//Prototipo de la función
int main()
{
    char nombre[2][20] = {"Lulu", "Carla"};
    mostrar(nombre);
    return 0;
}
```

```
void mostrar(char (*nom)[20])//Desarrollo
{
    for (int i = 0; i < 2; i++)
    {
        printf("%s\n", *(nom+i));
    }
}
```

Cabe recordar que en C todas las cadenas terminan con el valor nulo (\0), esto indica el final de la cadena.

Punteros y estructuras

Igual que creamos punteros a cualquier tipo de dato básico, lo mismo podemos hacer si se trata de un tipo de datos no tan sencillo, como un "struct". Del mismo modo que con otros tipos de datos, las estructuras tienen una dirección, asumiendo que esta es el comienzo de su almacenamiento.

Veremos que los punteros a estructuras son de uso muy frecuente para el manejo de estas; en especial cuando se pasan como argumentos a funciones, o son devueltas por estas.

Declaración de punteros a struct

En general la sintaxis para declaración de punteros a estructuras sigue la misma sintaxis general:

```
struct<Identificador_Etiqueta><*><Identificador_variable_puntero>;
```

Si consideramos declaración:

```
...
struct datos
{
    int Legajo;
    int Edad;
};
...
```

A continuación declaramos las variables del tipo struct mencionado anteriormente:

```
...
int main ()
{
struct datos alumno; //Declaración de dato tipo struct
struct datos *p;     //Declaración de un puntero al mismo tipo
...
return 0;
}
```

Podemos hacer la siguiente inicialización, tal y como se ve en el siguiente ejemplo:

```
...
*p = &alumno;
...
```

Los campos de la estructura pueden referenciarse mediante el operador de acceso punto (.) de la siguiente forma:

```
...
(*p).Legajo =1234;
(*p).Edad = 24;
...
```

Los paréntesis son necesarios, ya que el operador de acceso (.) tiene mayor precedencia que el de indirección (*).

Sin embargo, la forma de acceder a los datos en un struct puede cambiar ligeramente reemplazando el operador o la expresión anterior por una notación alternativa más corta, a través de operador (->) de la siguiente forma:

```
...
p->Legajo = 1234;
p->Edad = 24;
...
```

Veamos que ocurre con el acceso a los campos si se trata de estructuras anidadas, observemos la siguiente declaración:

```
...
struct Coordenadas
{
    int x;
    int y;
};

struct Linea
{
    struct Coordenadas p1;
    struct Coordenadas p2;
};

int main()
{
    struct Linea Line, *plin;
    plin = &Line;
...
return 0;
}
```

De esta declaración se desprenden las siguientes formas de acceder a los campos usando la notación clásica (sin punteros) y usando punteros. Existen variantes de estos usos pero estas dos son preferibles por legibilidad:

```
Line.p1.x = 1; //Acceso sin punteros
plin->p1.x = 3; //Acceso con punteros
```

Veamos el ejemplo completo muy sencillo en primer lugar sin usar punteros:

```
#include <stdio.h>
struct Datos {
    int Legajo;
    int edad;
};
```

```c
int main()
{
    struct Datos persona;

    persona.Legajo=1234;
    persona.edad=20;
    printf("Legajo: %d \nEdad: %d", persona.Legajo, persona.edad);

    return 0;
}
```

Veamos la misma situación usando punteros:

```c
#include <stdio.h>
struct Datos {
    int Legajo;
    int edad;
};
int main()
{
    struct Datos persona, *p;
    p=&persona;
    p->Legajo=1234;
    p->edad =20;
    printf("Legajo: %d \nEdad: %d", p->Legajo, p->edad);

    return 0;
}
```

Analicemos ahora una situación completa pasando una estructura a una función utilizando punteros:

```c
#include <stdio.h>
struct Datos
{
    int Legajo;
    int edad;
};
void mostrar(struct Datos *);
```

```c
int main()
{
    struct Datos persona;
    persona.Legajo =1234;
    persona.edad =20;
    mostrar(&persona);//llama a la función con la dirección
return 0;
}
void mostrar(struct Datos *pun)
{
    printf("Legajo: %d \nEdad: %d", pun->Legajo, pun->edad);
}
```