G U I A 3

- Concepto de Variable
- Ubicación de las variables en la memoria
- Concepto de apuntador
- Utilidad de apuntadores
- Caracterización de los apuntadores
- Niveles de direccionamiento
- Apuntadores y arreglos
- Apuntadores y TADs
- Apuntadores y Funciones
- Arreglos de apuntadores
- Funciones que retornan apuntadores

T E O R I A

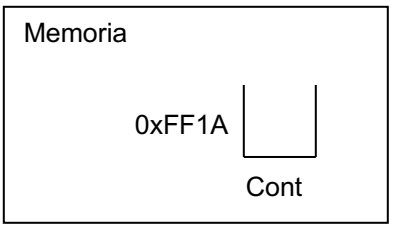
HABLANDO DE APUNTADORES

Cuando se declaran variables en un programa, lo que realmente se está haciendo es colocarle un nombre externo a un área de memoria en donde se pretende almacenar información. Internamente el computador reconocerá esa misma área, para efectos de las operaciones en las cuales se involucre, a través de una dirección de ubicación o dirección de memoria y que, como su nombre lo indica, tiene la misma función que la dirección de nuestra casa. Imagínese usted cómo sería la ubicación de las personas si las calles y las carreras no tuvieran un número que permitiera identificar de manera única cada casa. Con toda seguridad que la ubicación de nosotros, o al menos la referenciación de nuestro lugar de vivienda, sería bien difícil. Tan solo imaginemos que vivimos en una ciudad grande y que necesitamos decir en donde vivimos, si no existen las direcciones se complica la situación. Normalmente la memoria del computador ha de concebirse como una gran ciudad y por ello es tan importante que se tenga en cuenta que la única forma que se tiene, internamente, para referenciar cualquier posición dentro de ella es a través de direcciones de memoria.

Ahora bien, las direcciones de memoria en un computador se manejan en sistema hexadecimal por lo cual es importante que, adicional a esta aplicación, recordemos que la base de dicho sistema es el número 16 y que los dígitos van desde el 0 hasta el 9 y desde la A hasta la F, siendo A igual a10 y F igual a 15. Para una identificación apropiada de las direcciones vamos a anteponerle a cualquier número en hexadecimal, tal como se hace en Lenguaje C, el identificador 0x (un cero seguido de una x). Esto solo para que sepamos, sin mayores explicaciones, en qué momento estamos hablando de un número en sistema hexadecimal y cuando estamos hablando de un número en sistema decimal. De esta manera la orden

int Cont;

Está declarando una variable de tipo entero cuyo nombre externo es *Cont* y cuya dirección de memoria es asignada por el sistema operativo que para el ejemplo es un número compuesto de cuatro dígitos hexadecimales (0xFF1A). No olvide que anteponemos 0x para hacer un poco mas fácil la identificación de las direcciones de memoria.



Ahora bien, las variables que convencionalmente declaramos tienen la capacidad de almacenar un dato con determinado tipo. El tema que nos ocupa es el conocimiento de los apuntadores que, por definición técnica, no son mas que variables que pueden almacenar, a diferencia de las variables convencionales, direcciones de memoria. Sé que en este momento usted podría pensar que

pareciera inoficiosa la función de almacenar direcciones de memoria pero a medida que vayamos avanzando irá entendiendo la gran ventaja que significa para un programador poder contar con esta herramienta.

Los apuntadores o punteros en el Lenguaje C, son variables que "apuntan " , es decir que tienen la capacidad de almacenar la dirección de las ubicaciones en memoria de otras variables, y por medio de ellos tendremos un poderoso método de acceso a todas ellas. En la medida en que uno se va familiarizando con los apuntadores, éstos se van convirtiendo en la mas cómoda y directa de las herramientas para la manipulación y el control de variables que revistan algún nivel de complejidad así como parámetros y argumentos. La programación siempre ha necesitado de estas variables pero normalmente han sido utilizadas por el interior de los compiladores. La declaración de un apuntador es muy sencilla, todo lo que hay que hacer es anteponer un * a la variable que queremos que tenga la capacidad de almacenar una dirección de memoria. El tipo de dato de un apuntador nos indica las características de la variable hacia donde éste puede apuntar y por ello caracteriza de paso al mismo apuntador. De esta manera escribir la instrucción

*Int num, *punto;*

Nos está indicando que *num* es una variable convencional que puede almacenar un dato entero y que *punto* es una variable que tiene la capacidad de almacenar la dirección de memoria de una variable entera. Hasta este momento no hemos asociado una con la otra. Veamos el siguiente conjunto de instrucciones (Prog01.cpp):

```
Void main()
{
        int num, *punto;
        num = 18 ;
        punto = &num ;
        printf (« %d   %x   %d   %d   %x   %x », num, punto, *punto, &num, &punto);
}
```

En la primera instrucción *int num, *punto;* estamos declarando dos variables, una que se llama *num* y que puede almacenar un dato entero y otra que se llama *punto* que puede almacenar la dirección de memoria de una variable entera. En la instrucción *num = 18 ;* le hemos asignado el valor 18 a la variable *num* lo cual significa que ese es, por ahora, el contenido de dicha variable. En la instrucción siguiente *punto = &num ;* le estamos indicando al Lenguaje C que almacene en la variable *punto* (que es una variable apuntador) la dirección de memoria en donde se encuentra la variable *num* (que es una variable convencional). Por último tenemos una instrucción de impresión o salida que nos permite mostrar cinco datos:

- el primero (o sea *num*) corresponde al contenido de la variable *num* o sea el valor 18
- el segundo (o sea *punto*) corresponde al contenido Hexadecimal de la variable *punto* que corresponde a la dirección en donde se encuentra ubicada la variable *num*
- el tercero (o sea **punto*) corresponde al contenido de la dirección hacia donde apunta la variable *punto*, como la variable *punto* apunta hacia la variable num y ésta almacena el valor 18 entonces en esta orden se imprimirá el valor 18. Cabe anotar que cuando se habla de apuntar se está haciendo referencia al almacenamiento de una dirección de memoria
- el cuarto (o sea &*num*) corresponde a la dirección en donde se encuentra la variable *num* que es ni mas ni menos que el mismo valor que está almacenado en la variable *punto*
- el quinto (o sea &*punto*) corresponde a la dirección en donde se encuentra ubicada en memoria la variable *punto* que también, por ser variable, tiene una ubicación específica en la memoria

Una posible ejecución de este programa podría mostrar lo siguiente:

```
18   fff4    18    fff4    fff2
```

Es importante recordar que cualquier tipo de dato, primitivo o abstracto, puede servir para declarar apuntadores y que dicha declaración habilita al apuntador para que almacene direcciones de memoria SOLO de variables que tengan su mismo tipo. También cabe anotar que la notación que utilizaremos a lo largo de todo este curso corresponde a la notación estándar del lenguaje C según el cual se cumplen los siguiente patrones:

- Cuando se escriba SOLO el nombre de una variable se estará haciendo referencia al contenido de dicha variable
- Cuando se escriba el nombre de una variable precedido por un asterisco se estará haciendo referencia al contenido de la dirección hacia donde apunta dicha variable razón por la cual ésta debe ser un apuntador. De este caso se exceptúa solamente la declaración de variables
- Cuando se escriba el nombre de una variable precedido por un signo & se estará haciendo referencia a la dirección en donde está ubicada en memoria dicha variable

En estas declaraciones sólo decimos al compilador que reserve una posición de memoria para albergar la dirección de una variable , del tipo indicado , la cual será referenciada con el nombre que hayamos dado al puntero .

NIVELES DE DIRECCIONAMIENTO

Cuando se declara un apuntador, éste queda habilitado para almacenar la dirección de memoria en donde se ubica otra variable. Eso fue lo que nos permitió “enlazar” a la variable *num* del ejemplo anterior con la variable *punto*. Sin embargo, debemos tener en cuenta que la variable *punto*, por ser una variable mas y estar ubicada en la memoria, tiene también su propia ubicación. Por tanto, cómo haríamos para apuntar hacia ella? Vamos a tomar como variables convencionales aquellas que son variables sencillas y simples. Las que corresponden a declaraciones como *int num;* por ejemplo. Hacia estas variables se puede apuntar con apuntadores que han sido declarados con el mismo tipo y con un *, por esta razón es que *punto* que fue declarada inicialmente como *int *punto;* pudo almacenar la dirección de la variable *num*. La cantidad de asteriscos en la declaración de una variable apuntador establece la cantidad de “saltos” que habría que dar desde una variable hasta llegar a un dato efectivo teniendo en cuenta que un dato efectivo no es mas que ese dato que se almacena en las variables simples y sencillas. De esta manera si quisiéramos tener una variable que pudiera almacenar la dirección en donde está ubicada la variable punto, entonces necesitaríamos declararla con un doble asterisco de la siguiente forma:

Int ***ppunto*;

Por la forma de su declaración, a esta variable se le puede asignar, como contenido, la dirección en donde se ubica una variable que sea tipo apuntador pero que haya sido declarada con un solo asterisco lo cual significa que *ppunto* puede almacenar la dirección de la variable *punto* lo cual lo podemos hacer con una asignación muy sencilla:

ppunto = &punto;

La variable ppunto es un apuntador de segundo nivel y la variable punto es un apuntador de primer nivel. Es muy importante que se tenga muy en cuenta la notación dado que es allí en donde alguna confusión puede aparecer. De manera que el siguiente programa (Prog02.cpp) es un buen ejemplo de lo que acabamos de decir:

```
Void main()
{
        int num, *punto, **ppunto;
        num = 18;
```

```
        punto = &num;
        ppunto = &punto;
        printf(“ %d %x %x %x \n “, num, punto, ppunto, &num);
        printf(“ %x %d %x %d “, &punto, *punto, &ppunto, **ppunto);
}
```

En este programa estamos declarando tres variables: una variable convencional y dos apuntadores: el primero de primer nivel y el segundo de segundo nivel. Se le asigna el valor 18 a la variable *num*, se le ordena a la variable *punto* que almacene la dirección de memoria en donde se encuentra ubicada la variable *num* y se le indica a la variable *ppunto* (que es una apuntador de segundo nivel) que su contenido sea la dirección de memoria en donde se encuentra ubicada la variable *punto* (que es un apuntador de primer nivel). Luego tenemos la siguiente línea de salida:

```
printf(“ %d %x %x %x \n “, num, punto, ppunto, &num);
```

Ella imprimirá:

num	el contenido de la variable *num*
punto	el contenido de la variable punto o sea la dirección de la variable *num*
ppunto	el contenido de la variable punto o sea la dirección de la variable *punto*
&*num*	la dirección en donde se encuentra ubicada la variable *num*

Asimismo la instrucción

```
printf(“ %x %d %x %d “, &punto, *punto, &ppunto, **ppunto);
```

imprimirá lo siguiente:

&*punto*	la dirección en donde está ubicada la variable *punto*
**punto*	el contenido de la dirección que almacena la variable *punto* o sea el contenido de *num*
&*ppunto*	la dirección en donde se encuentra almacenada la variable *ppunto* que por ser variable también tiene su ubicación propia
***ppunto*	el contenido del contenido de la dirección hacia donde apunto *ppunto* o sea el contenido de la variable *num*

Cuando se habla de apuntadores, debe tenerse especial cuidado con la notación que se utilice ya que, como ha visto, cualquier pequeño símbolo utilizado de manera inapropiada puede generar errores que muchas veces no son tan fáciles de encontrar. Usted tal vez se preguntará que para qué sirven estos niveles de direccionamiento... pues muy sencillo, el nombre de un vector es un apuntador de primer nivel y el nombre de una matriz es una apuntador de segundo nivel. Esto significa que usted podrá construir funciones muy genéricas que trabajen con vectores y matrices sin que tenga que saber exactamente cuál es su dimensión. Eso sí, deben estar bien delimitados para que no sufra los problemas del “desbordamiento” pero si usted ha tomado las precauciones técnicas apropiadas encontrará en los apuntadores la principal herramienta para construir funciones realmente útiles y genéricas. Cuando se pasan como parámetros arreglos a una función y éstos se reciben en apuntadores, éstos en cualquier momento se pueden utilizar con subíndices. El subíndice lo único que indica adicional es la cantidad de bytes (acorde con el tipo de datos) que debe procesar en un momento dado.

Estoy seguro que usted sabe que las cadenas en Lenguaje C finalizan en el caracter ‘\0’ y también estoy seguro que mas de una vez usted se preguntó para qué servía un carácter que no se podía imprimir ni hacer, aparentemente nada con él. Pues bien, la gran utilidad que tiene el carácter nulo es que permite construir funciones que reciban cadenas que finalicen con él sin importar su longitud. Estas funciones tendrán como parámetro principal un apuntador de primer nivel que se ubica en el primer byte de la primera posición del vector y de allí en adelante lo puede recorrer, si lo queremos, hasta que llegue al final de la misma en donde estará el carácter nulo.

Así como un tipo de dato abstracto nos permite declarar apuntadores, los TADs (tipos abstractos de datos) también nos lo pueden permitir. Veamos el siguiente ejemplo:

```
struct fecha
{
        int dd;
        int mm;
        int aaaa;
}

struct fecha hoy, *today;
```

Con este conjunto de instrucciones estamos creando un TAD llamado fecha y declarando dos variables: una variable convencional llamada hoy y un apuntador de primer nivel a variables tipo fecha llamado ayer. Las siguientes pueden ser asignaciones válidas (Prog03.cpp):

```
        hoy.dd = 15;
        hoy.mm = 02;
        hoy.aaaa = 2004;

        today = (struct fecha *) malloc (sizeof(struct fecha));
        today = & hoy;

        printf (" %d / %d / %d", hoy.dd, (*today).mm, hoy.aaaa);
```

Este conjunto de instrucciones nos permite mostrar la forma de manejar variables que han sido declaradas a través de TADs. En particular la instrucción:

```
        today = (struct fecha *) malloc (sizeof(struct fecha));
```

Asigna un espacio de memoria para crear el apuntador *today* en ella. La función *malloc* para ser utilizada exige la inclusión de la librería *stdlib.h*. Luego de creado este espacio de memoria entonces se puede ejecutar la orden *today = & hoy;* que almacena en la variable *today* la dirección de memoria en donde se encuentra ubicada la variable *hoy*. A pesar de que la sintaxis pareciera compleja, en realidad no lo es. La función malloc necesita como parámetro el tamaño del tipo de dato que debe asignar y se le antepone el tipo de dato a asignar para que devuelva este tipo de dirección.

Es importante anotar que cuando se incremente el contenido de una variable apuntador se le está diciendo que apunte hacia la dirección siguiente en memoria, por lo tanto de allí en adelante esta variable estará apuntado hacia una dirección diferente a la original. También vale la pena anotar que se puede evaluar si el espacio de memoria fue asignado a través de malloc o no. Cómo se hace? Luego de realizar la asignación dinámica de memoria se puede preguntar si el contenido del apuntador es igual a null. En caso de que sea cierto significa que no se pudo asignar el espacio de memoria solicitado. Una instrucción válida podría ser:

```
        if (today == null)
```

Que se traduciría en preguntar si fue asignado correctamente el espacio de memoria que se solicitó o no. Cabe anotar que en cualquier momento se pueden declarar arreglos de apuntadores. Una forma de encontrar una gran aplicación a los apuntadores radica en la flexibilidad que estos brindan a la construcción de funciones generales. Supongamos que necesitamos construir una función que retorne la longitud de una cadena que finaliza en carácter nulo. Su versión, basada en apuntadores (Prog04.cpp), podría ser la siguiente:

```
Int long_cad (char *p)          La función retornará un valor entero que corresponde a la longitud
{                               de la cadena. Su único parámetro es la cadena
        int cont=0;             Inicializamos un contador en 0 el cual "medirá" la longitud
        while (*p)              Mientras no se llegue al caracter nulo (que por defecto es falso)
        {
                cont++;         Incremente el valor del contador
                p++;            e incremente la posición del apuntador
        }
return(cont);                   Al finalizar, retorne el valor que se encuentra en la variable cont
}
```

Para utilizar esta función todo lo que tenemos que hacer es llamarla enviándole una cadena concreta, enviándole el nombre de un vector que contenga una cadena (asegurándonos que ésta finalice en \0) o enviándole el nombre de otro apuntador de primer nivel que apunte hacia una cadena. De manera que los siguientes podrían ser llamados válidos:

```
long_cad("hola");       Se le envía una cadena concreta
long_cad(v);            Se le envía nombre de vector que fue declarado char v[20] por ejemplo
long_cad(x);            Se le envía nombre de apuntador que fue declarado char *x
```

Ahora vamos a construir una función que reciba dos cadenas y determine si son exactamente iguales, utilizando el mismo recurso de los apuntadores. Esta función retornará 1 si son exactamente iguales las dos cadenas o retornará 0 si no lo son (Prog05.cpp). Una propuesta de esta podría ser

```
int comp_cad(char *cad1, char *cad2)    La función retornará un entero y recibe como parámetros
{                                       dos cadenas, tal como se enunció
while(*cad1&&*cad2)                     Mientras el contenido de las cadenas sea diferente de
        {                               carácter nulo
                cad1++;                 Avance el apuntador a una cadena
                cad2++;                 Avance al apuntador a la otra cadena
        }
if ( !(*cad1 && *cad2 )                 Si al salirse del ciclo ambas cadenas están en \0
        return(1);                      Quiere decir que ambas son exactamente iguales
return(0)                               De lo contrario son diferentes
}
```

Por ultimo, una forma sencilla y fácil de asociar subíndices a cadenas es el arreglo de apuntadores acorde con el cual éste puede ser inicializado de la siguiente forma:

```
char *p[7]= {"lunes","martes","miércoles","jueves","viernes","sábado","domingo"};
```

Los arreglos de apuntadores son una de las herramientas mas eficaces para el manejo de conjuntos de cadenas (Prog06.cpp). Finalmente cuando en la declaración de una función se coloca un * antecediendo el nombre de la misma, esto significa que la función retorna un apuntador y por lo tanto se debe recibir en una variable del tipo especificado. Esto quiere decir que si la función tiene un prototipo como el siguiente

```
char *concatenar (char *cad1, char *cad2)
```

Al momento de llamarse esta función se debe recibir, el "valor" que retorne en una variable cuya declaración haya sido

```
char *p;
```

o sea en un apuntador de primer nivel, al igual que la función. Su uso es muy sencillo y para verificarlo lo ilustraremos con un ejemplo(Prog07.cpp):

```
#include <stdio.h>          Se incluye la librería de entrada y salida
char *funcion(int n);       Se escribe el prototipo de la función que se va a utilizar. Nótese
                            que retorna un apuntador a caracteres
void main()                 Función Principal
{
        char *p;            Se declara un apuntador de primer nivel
        p = funcion(1);     Se llama a la funcion y el valor que retorna se almacena en p
        printf("%s", p);    Se muestra el contenido de p
}

char *funcion(int n)        La funcion que se va a utilizar retorna un apuntador a caracteres
{
        char *x;            Se declara un apuntador local a caracteres
        n++;                Se incrementa el parámetro de la función (este no se usa)
        x = "hola";         Se asigna una cadena al apuntador local a caracteres
        return(x);          Se retorna la cadena a donde fue llamada la funcion
}
```

Como puede ver es muy sencillo, desde todos los flancos posibles, la utilización de los apuntadores y la flexibilidad que le brinda a nuestros programas puede ser mayor que, incluso, la que nosotros mismos imaginamos. Particularmente la construcción de funciones apoyadas en apuntadores es mucho mas flexible. No se olvide que en cualquier momento una función puede llamar a otra.

P R E G U N T A S

- De acuerdo a lo visto, cómo podría usted definir en sus propias palabras lo que es un apuntador
- Qué es una dirección de memoria y porqué es tan importante cuando se declaran variables
- Qué tipos de datos puede almacenar un apuntador
- Enuncia al menos dos ventajas por las que sea realmente útil hacer uso de los apuntadores
- Cuando se declara un apuntador, este solo puede apuntar al tipo de dato con que se declaró?
- Qué es un apuntador de primer nivel y hacia donde puede apuntar
- Qué es una apuntador de segundo nivel y hacia donde puede apuntar
- Tienen los apuntadores direcciones de memoria en donde se ubican?
- Acorde con la teoría vista, hasta donde pueden llegar los niveles de direccionamiento
- Qué relación existe entre un vector y un apuntador de primer nivel
- Qué relación existe entre una matriz y un apuntador de segundo nivel
- Puede un apuntador, apuntar a un dato tipo TAD
- Para qué sirve la función malloc
- Para qué sirve la función sizeof
- Cuando se le pasa el nombre de un vector a una función, cómo se debe declarar el parámetro que lo recibe?
- Enuncie alguna situación en donde pueda ser útil un arreglo de apuntadores

T A L L E R

Basado en el concepto de apuntadores y tratando al máximo de manejar, dentro de las funciones, a éstos o sea procurando prescindir de cualquier otro tipo de variable (excepto las que sean absolutamente necesarias) desarrollar las siguientes funciones. Tenga en cuenta que cuando se hace referencia a una cadena siempre se asume que son cadenas que finalizan con carácter nulo:

1 Construir una función que permita copiar el contenido de una cadena en otra de manera que ambas cadenas, al final de la función queden, exactamente iguales
2 Construir una función que reciba dos cadenas y permita concatenar el contenido de la segunda al final de la primera
3 Construir una función que reciba una cadena y un carácter y retorne Verdadero si el carácter se encuentra en la cadena. Falso en caso contrario
4 Construir una función que reciba una cadena y un carácter y retorne la posición (en memoria) de la primera ocurrencia del carácter
5 Construir una función que compare lexicográficamente dos cadenas, es decir, que determine cual de las dos estaría de primero en un diccionario si se buscan en éste de manera secuencial. Como quien dice, si se ordenan las dos cadenas, cual de las dos estaría de primero
6 Construir una función que reciba una cadena y convierta todas las mayúsculas, que aparecen en ella, a minúsculas
7 Construir una función que reciba una cadena y convierta todas las minúsculas, que aparecen en ella, a mayúsculas
8 Construir una función que reciba una cadena y un carácter y retorne la posición (en memoria) de la última ocurrencia de dicho carácter en la cadena
9 Construir una función que reciba una cadena y la retorne completamente invertida
10 Construir una función que reciba una cadena y un carácter y llene dicha cadena con el carácter que recibe respetando la posición del carácter nulo
11 Construir una función que reciba dos cadenas y nos permita determinar si la segunda cadena está contenida en la primera cadena
12 Construir una función que reciba dos cadenas y nos permita determinar si TODOS los caracteres de la segunda cadena aparecen en la primera cadena
13 Construir una función que reciba dos cadenas y nos permita determinas si ALGUNO de los caracteres de la segunda cadena aparecen en la primera cadena
14 Construir una función que reciba dos cadenas y compare si son iguales, asumiendo que mayúsculas y minúsculas tienen el mismo significado
15 Construir una función que reciba dos cadenas y un número entero y que concatene al final de la primera cadena, los n primeros caracteres de la segunda que recibe como parámetro
16 Construir una función que reciba una cadena, un número entero y un carácter y llene las primeras n posiciones de la cadena con el carácter
17 Construir, acorde con el definición que viene en el compilador de lenguaje C (librería string.h), la función strok
18 Construir una función que reciba una cadena y determina si es palíndrome o sea si es una cadena que leída de izquierda a derecha y viceversa es igual omitiendo espacios en blanco y vocales tildadas
19 Construir una función que reciba una cadena y la retorne pero sin vocales
20 Construir una función que reciba una cadena y la retorne encriptada acorde con la siguiente ecuación: $f(x) = x + 1$

Apoyados en un TAD que llamaremos FECHA y que nos permitirá almacenar apropiadamente los datos de una fecha y teniendo en cuenta los arreglos de apuntadores, construir las siguientes funciones

1 Construir una función que reciba un número entre 1 y 7 y retorne el nombre del día correspondiente

2 Construir una función que reciba un número entre 1 y 12 y retorne el nombre del mes correspondiente
3 Construir una función que reciba el número del año y lo retorne en formato de texto (tipo monto escrito)
4 Construir una función que reciba un número cualquier (positivo) y lo retorne como monto escrito. (Para esta función se puede apoyar en otras subfunciones que usted mismo construya)
5 Construir una función que reciba una fecha y retorne la cadena que describe textualmente la fecha. Por ejemplo se recibe “01/10/1970” y se retorna “1o de Octubre de 1970”
6 Construir una función que reciba dos fechas y retorne la cantidad de días que hay entre las mismas
7 Construir una función que reciba una fecha y retorne 1 si la fecha es válida y 0 si no lo es
8 Construir una función que reciba una fecha y retorne su descripción totalmente en texto. Por ejemplo si recibe “07/07/1973” deberá retornar “Siete de Julio de Mil Novecientos Setenta y Tres”