Unidad 2 - Arreglos (Arrays) - Pointers

Introducción. Arreglos (Arrays).

Un array es una estructura homogénea, compuesta por varias componentes, todas del mismo tipo y almacenadas consecutivamente en memoria. Cada componente puede ser accedido directamente por el nombre de la variable array seguido de un subíndice encerrado entre corchetes.

La representación de los arrays se hace mediante variables suscritas o de subíndices y pueden tener una o varias dimensiones (subíndices). A los arrays de una dimensión se les llama también listas y a los de dos dimensiones tablas.

Desde el punto de vista matemático, en más de una ocasión necesitaremos representar variables, tales como:

$$a_1\ a_2\ a_3 \ldots a_i \ldots a_n$$

en el caso de un subíndice o bien

$$\begin{array}{c} a_{11}\ a_{12}\ a_{13} \ldots a_{1j} \ldots a_{1n} \\ a_{21}\ a_{22}\ a_{23} \ldots a_{2j} \ldots a_{2n} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{i1}\ a_{i2}\ a_{i3} \ldots a_{ij} \ldots a_{in} \end{array}$$

si se utilizan dos subíndices. Para realizar esta misma representación bajo C, tendremos que recurrir a los arrays que acabamos de definir y que a continuación se estudian.

Declaración de un array

La declaración de un array específica el nombre del array, el número de elementos del mismo y el tipo de estos.

tipo nombre [**tamaño**];
tipo nombre [**tamaño filas**][**tamaño columnas**]…;
tipo nombre [];

tipo indica el tipo de los elementos del array. Puede ser cualquier tipo excepto void
nombre es un identificador que nombra al array
tamaño es una constante que especifica el número de elementos del array. El tamaño puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando se hace referencia a un array declarado en otra parte del programa

Ejemplos:

int lista[100];

Este ejemplo declara una variable array denominada lista con 100 elementos (del 0 al 99), cada uno de ellos de tipo entero (**int**).

char *vector[];

Este ejemplo declara el tipo y el nombre de un array de punteros de objetos de tipo char.

El lenguaje C no chequea los límites de un array. Es responsabilidad del programador el realizar este tipo de operaciones.

Los array se inicializan en C en forma global. No se pueden inicializar arrays locales. (Realmente se pueden inicializar variables locales, incluyendo arrays, si se declaran como **static**). El formato general de inicialización de un array es similar a otras variables, según podemos ver aquí:

tipo nombre[**tam1**]…[**tamN**] = {lista de valores};

Ejemplo:

int i[10] = {1,2,3,4,5,6,7,8,9,10};

Los array se inicializan por filas, así que si queremos inicializar un array de 2 filas por 3 columnas, lo debemos hacer de la siguiente forma:

int tabla[2][3] = {10,12,14,16,18,20};

los valores quedan dispuestos de la siguiente forma:

	columna 1	columna 2	columna 3
fila 1	10	12	14
fila 2	16	18	20

Cadenas de caracteres.

Una cadena de caracteres es un array unidimensional, en el cual todos sus elementos son de tipo **char**.

Una cadena se puede inicializar asignándole un literal. Por ejemplo:

char cadena[] = "abcd";

Este ejemplo inicializa una cadena con cinco elementos (cadena[0] a cadena[4]). El quinto elemento, es el carácter nulo (\0), con el cual C finaliza todas las cadenas de caracteres.

Si se especifica el tamaño del array y la cadena de caracteres es más larga, los caracteres que sobran se ignoran.

Si la cadena asignada es más corta que el tamaño del array, el resto de los elementos de este se inicializan con el valor nulo (\0).

Ejemplos:

char nombre_apellidos[60];

Este ejemplo se declara la cadena denominada nombre_apellido de 60 caracteres como máximo.

char lista[100][60];

Este ejemplo declara la cadena denominada lista. Esta cadena tiene 100 filas cada una de las cuales es una cadena de caracteres de longitud máxima 60.

Para leer una cadena de caracteres podemos usar la función scanf(), el nombre de la variable a leer no necesita estar precedida por el operador de dirección &, porque el nombre de una cadena es una dirección, la dirección de comienzo del array. Si se lee elemento a elemento, el operador & es necesario.

```
char nombre[41];

scanf("%s",nombre);
```

Función "gets". Leer una cadena de caracteres.

La función gets lee una cadena desde la entrada estándar, stdin, y la almacena en la variable especificada. Esta variable es un puntero a la cadena de caracteres (apunta al primer carácter).

Cabecera: stdio.h
Sintaxis:

```
#include <stdio.h>
char *var;
char gets(var);
```

La función gets a diferencia de la función scanf, permite la entrada de una cadena formada por varias palabras separadas por espacios en blanco, sin ningún tipo de formato. Recordar que para scanf el espacio en blanco actúa como separador de datos en la entrada.

Ejemplo:

```
#include <stdio.h>

char linea[80];
char *resu;

int main(){
    printf("Introduce una cadena: ");
    resu = gets(linea);
    printf("\nLa línea introducida es: \n");
    printf("%s\n", linea);
    printf("\nLa escribo por segunda vez:\n");
    printf("%s\n", resu);

    return 0;
}
```

Función puts. Escribir una cadena de caracteres.

La función puts escribe una cadena de caracteres en la salida estándar stdout, y reemplaza el carácter nulo de terminación de la cadena (\0) por el carácter nueva línea (\n).

Cabecera: stdio.h
Sintaxis:

```
int puts(cadena)
const char *cadena;
```

La función puts retorna un valor 0 si se ejecuta satisfactoriamente, en caso contrario retorna un valor distinto de 0.

Función "fgets". Lee una cadena de caracteres de stdin.

Sintaxis:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Lee una cadena de caracteres desde *stream* (stream = stdin) y se la asigna a la cadena *s.* La función deja de leer cuando se hayan leídos n-1 caracteres o se encuentre un carácter nueva linea, lo que ocurra primero. f*gets* mantiene el carácter nueva linea al final de *s*. Un carácter NULL se agrega a *s* para indicar el fin de la cadena.

Si todo va bien fgets devuelve la cadena apuntada por *s* y devueleve NULL o eof en el caso de algún error.

Ejemplo:

```
#include <stdio.h>

int main(void)
{
        char string[42];

        fgets(string, 42, stdin);          //Lee una cadena desde consola
        printf("%s", string);              //Imprime la cadena leída

        getchar();
        return 0;
}
```

Funciones para manipular cadenas de caracteres.

Todas las funciones que a continuación se describen tiene en común el archivo de cabecera **string.h**, el cual se debe incluir en cualquier programa que haga uso de las funciones para manipular cadenas de caracteres.

Las funciones más usadas son:

La función strcat(cadena1,cadena2).

```
char *strcat(cadena1, cadena2);
char *cadena1;
const char *cadena2;
```

Esta función añade la cadena2 a la cadena1, termina la cadena resultante con el carácter nulo y devuelve un puntero a cadena1.

La función strch(cadena, c).

```
char *strchr(cadena, c);
char *cadena;
int c;
```

Esta función devuelve un puntero a la primera ocurrencia de c en cadena o un valor NULL si el carácter no es encontrado. El carácter c puede ser un carácter nulo (\0).

La función strcmp(cadena1, cadena2).

```
int strcmp(cadena1, cadena2);
const char *cadena1;
const char *cadena2;
```

Esta función compara la cadena1 con la cadena2 y devuelve un valor <0 si la cadena1 es menor que la cadena2, =0 si la cadena1 es igual que la cadena2 y >0 si la cadena1 es mayor que la cadena2.

La función strcpy(cadena1, cadena2).

```
char *strcpy(cadena1, cadena2);
char *cadena1
const char *cadena2;
```

Esta función copia la cadena2 incluyendo el carácter de terminación nulo, en la cadena1 y devuelve un puntero a cadena1.

La función strlen(cadena).

```
size__t strlen(cadena);
char *cadena;
```

Esta función devuelve la longitud en bytes de la cadena, no incluyendo el carácter de terminación nulo.

La función strcspn(cadena1, cadena2).

```
size__t strcspn(cadena1, cadena2);
const char *cadena1;
const char *cadena2;
```

Esta función da como resultado la posición (subíndice) del primer carácter de cadena1, que pertenece al conjunto de caracteres contenidos en cadena2.

La función strncat(cadena1, cadena2, n).

```
char *strncat(cadean1, cadena2, n);
char *cadena1;
const char *cadena2;
size__t n;
```

Esta función añade a los primeros n caracteres de cadena2 a la cadena1, termina la cadena resultante con el carácter nulo y devuelve un puntero a cadena1. Si n es mayor que la longitud de cadena2, se utiliza como valor de n la longitud de cadena2.

La función strncmp(cadean1, cadena2, n).

```
int strncmp(cadena1, cadena2, n);
const char *cadena1;
const char *cadena2;
size_t n;
```

Esta función compara los primeros n caracteres de cadenal y cadena2 y devuelve un valor <0 si la cadena1 es menor que la cadena2, =0 si la cadena! es igual que la cadena2 y >0 si la cadenal es mayor que la cadena2.

La función strncpy(cadena1, cadena2, n).

```
char *strncpy(cadena1 , cadena2, n);
char *cadenal;
const char *cadena2;
size_t n;
```

Esta función copia n caracteres de la cadena2, en la cadena1 y devuelve un puntero a cadena1. Si n es menor que la longitud de cadena2, no se añade automáticamente un carácter nulo a la cadena resultante. Si n es mayor que la longitud de cadena2, la cadena resultante es rellenada con caracteres nulos.

La función strrchr(cadena, c).

```
char *strrchr(cadena, c);
const char *cadena;
int c;
```

Esta función devuelve un puntero a la última ocurrencia de c en cadena o un valor NULL si el carácter no es encontrado. El carácter c puede ser un carácter nulo (' \ 0').

La función strspn(cadenal, cadena2).

```
size_t strspn(cadenal, cadena2);
const char *cadenal;
const char *cadena2;
```

Esta función da como resultado la posición (subíndice) del primer carácter de cadena1, que no pertenece al conjunto de caracteres contenidos en cadena2.

La función strstr(cadena1, cadena2).

```
char *strstr(cadena1, cadena2);
const char *cadena1;
const char *cadena2;
```

Esta función devuelve un puntero a la primera ocurrencia de cadena2 en cadenal o un valor NULL si la cadena2 no se encuentra en la cadena 1.

La función strtok(cadena1, cadena2).

```
char *strtok(cadena1, cadena2);
char *cadenal;
const char *cadena2;
```

Esta función lee la cadena1 como una serie de cero o más elementos básicos separados por cualquiera de los caracteres expresados en cadena2, los cuales son interpretados como delimitadores. Una vez leído el primer elemento de cadena1, la siguiente llamada a strtok, para leer el siguiente elemento, se efectúa poniendo en lugar del argumento cadena1, NULL. Observar el ejemplo que se expone a continuación. Esta función devuelve un puntero por cada elemento en cadena1. Cuando no hay más elementos, se devuelve un puntero nulo. Puede ponerse más de un delimitador entre elemento y elemento. También pueden variarse el conjunto de caracteres que actúan como delimitadores, de una llamada a otra.

Ejemplo:

```
#include "stdio.h"
#include "string.h"

char *cadena = "esta cadena, está formada por varias palabras";
char *elemento;

void main()
{
   /* Leer el primer elemento */
   /* Delimitadores: blanco y coma (" ,") */

   elemento = strtok(cadena, " ,");

   /* mientras hay elementos en cadena */
```

```
    while(elemento != NULL)
    {
            printf("%s \n", elemento);

            /* leer el siguiente elemento */

            elemento = strtok(NULL," ,");
    }
}
```

Solución:

```
esta
cadena
está
formada
por
varias
palabras
```

Los argumentos empleados en estas funciones, cadena, cadena1 y cadena2, deben de contener como carácter de terminación el carácter nulo ('\0').

Funciones para Conversion de Datos.

atof.

Convierte una cadena de caracteres a un valor en doble precisión.

```
#include <stdlib.h>

double atof(cadena);
const char *cadena;
```

atoi.

Convierte una cadena de caracteres aun valor entero.

```
#include <stdlib.h>

double atoi(cadena);
const char *cadena;
```

atol.

Convierte una cadena a un valor entero largo.

```
#include <stdlib.h>
```

```
double atol(cadena);
const char *cadena;
```

strtod.

```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
```

Convierte una cadena a un valor *s* a **double** o **long double**. Si *s* es una secuencia de caracteres que puede ser interpretada como un **double**; la cadena debe coincidir con este formato genérico:

[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]

donde:

[ws] = Espacio en blanco. Opcional.

[sn] = Signo (+ o -). Opcional.

[ddd]=Digitos. Opcional.

[fmt]=e o *E.* Opecional.

[.] = Punto decimal. Opcional.

strtod también reconoce +INF y -INF como más y menos infinito, y +NAN y -NAN para lo que no sea un número.

strtod detiene la conversión de la cadena al primer carácter que no puede ser interpretado como una parte apropiada para reprresentar un valor **double**.

Si *endptr* no es NULL, *strtod* asigna a ***endptr* un puntero al carácter que detuvo la conversión (**endptr* = &*stopper*). *endptr* es útil para la detección de errores.

Esta función devuelve el valor de la cadena *s* como un valor **double**.

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        char input[80], *endptr;
        double value;
        printf("Ingrese un número en punto flotante:");
        gets(input);
        value = strtod(input, &endptr);
        printf("La cadena es %s el número es %lf\n", input, value);
        return 0;
}
```

Estructuras (struct).

Una estructura es una colección de datos elementales de diferentes tipos, lógicamente relacionados. A una estructura se la da también el nombre de registro.

Creación de una estructura.

Crear una estructura es definir un nuevo tipo de datos, denominado tipo estructura, luego de haber definido este nuevo tipo de dato estructura se debe declarar una variable de este tipo. En la definición del tipo estructura, se especifican los elementos que la componen así como sus tipos. Cada elemento de la estructura recibe el nombre de miembro (campo del registro). La sintaxis es la siguiente:

```
struct tipo_estructura {
        declaraciones de los miembros
};
```

tipo_estructura es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo estructura, podemos declarar una variable de ese tipo de la forma:

```
struct tipo_estructura variable;
```

Para referirse aun determinado miembro de la estructura, se utiliza la notación:

```
variable.miembro
```

Ejemplo:

```
#include <stdio.h>

struct ficha            /* definición del tipo estructura ficha */
{
        char nombre[20];
        char apellidos[40];
        long dni;
        int edad;
};

int main(){
        struct ficha proveedor, cliente;

        cliente.edad = 24;
        strcpy(proveedor.apellidos, "Perfecto Rui");

        printf("Edad cliente %d\n", cliente.edad);
        printf("Apellido proveedor: %s", proveedor.apellidos);
```

```
        return 0;
}
```

Este ejemplo define las variables proveedor y cliente, de tipo ficha, por lo que cada una de las variables consta de los miembros: nombre, apellidos, dni y edad.

Una variable que es un miembro de una estructura, puede utilizarse exactamente igual que cualquier otra variable. La salida del ejemplo anterior es:

Edad cliente 24
Apellido proveedor: Perfecto Rui

La declaración de las variables proveedor y cliente, puede realizarse también directamente de la siguiente forma:

```
struct ficha
{
        char nombre[20];
        char apellidos[40];
        long dni;
        int edad;
}  proveedor, cliente;
```

o también, sin dejar constancia del nuevo tipo definido (*unnamed*),

```
struct
{
        char nombre[20];
        char apellidos[40];
        long dni;
        int edad;
} proveedor, cliente;
```

Otra posibilidad es usar **typedef** para definir la estructura:

```
typedef struct
{
        char nombre[20];
        char apellidos[40];
        long dni;
        int edad;
} ficha;
```

En esta caso *ficha* define la estructura unnamed. Para declarar una variable de esta estructura hacemos:

ficha proveedor;

Observe que no utilizamos **struct** delante de ficha cuando usamos **typedef**.

La declaración de un miembro de una estructura no puede contener especificadores de clase de almacenamiento (**extern**, **static**, **auto**, **register**) y no pueden ser inicializadas. Su tipo puede ser cualquiera: fundamental, array, puntero, unión o estructura. Si un miembro se declara de tipo estructura, esta debe ser diferente a la que contiene al miembro, sin embargo, si se puede declarar como un puntero a la estructura que lo contiene.

Ejemplo:

```
struct fecha
{
        int día, mes, anno;
};

struct ficha
{
        char nombre[20];
        char apellidos[40];
        long dni;
        struct fecha fecha_naci;
};

struct ficha persona;
```

Este ejemplo define la variable persona, en la que el miembro fecha_naci es a su vez una estructura.

Los miembros de una estructura son almacenados secuencialmente en el mismo orden en el que son declarados.

Operaciones con estructuras.

Con una variable declarada como una estructura, solamente pueden realizarse tres operaciones:

- tomar su dirección por medio del operador &
- acceder a uno de sus miembros
- asignar una estructura a otra con el operador de asignación.

Estructuras anidadas.

Una estructura puede contener una o más estructuras. Una estructura que va a ser anidada debe ser declarada antes de la declaración de la estructura que la vaya a contener; de otra forma es muy probable que el compilador nos indique un error.

Ejemplo:

```
#include<stdio.h>

typedef struct
```

```
{
        int hora;
        int min;
        int seg;
        int mseg;
}s_timestamp;

typdef struct
{
        int index;
        float value;
        s_timestamp timestamp; //Se puede utilizar debido a que fue declarada anteriormente.
}s_Data;

int main()
{
        s_Data data;

        data.index = 1;
        data.value = 10.5;
        data.timestamp.hora = 10;
        data.timestamp.min = 10;
        data.timestamp.seg = 20;
        data.timestamp.mseg = 222;

        printf("%0.2d:%0.2d:%0.2d:%0.3d > ", data.timestamp.hora, data.timestamp.min,
                                                data.timestamp.seg, data.timestamp.mseg);
        printf("Index: %d", data.index);
        printf(" - Value = %10.3f\n", data.value);

        return 0;
}
```

Arrays de estructuras.

Cuando los elementos de un array son de tipo estructura, el array recibe el nombre de array de estructuras o array de registros. Esta es una construcción muy útil y potente.

Ejemplo:

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

#define SIZE 100

typedef struct
{
        char name[50];
```

```
        int code;
        float grd;
}student;

int main(void)
{
int i;
student stud[SIZE];

for(i = 0; i < SIZE; i++)
{
        printf("\nEnter name: ");
        gets(stud[i].name);

        printf("Enter code: ");
        scanf("%d", &stud[i].code);

        printf("Enter grade: ");
        scanf("%f", &stud[i].grd);

        printf("\nN: %s C: %d G: %.2f\n", stud[i].name, stud[i].code, stud[i].grd);

        getchar();      /*Elimina el carácter nueva linea que queda en el stdin después de
                        ingresar el grado*/
}
return 0;
}
```

Campos de Bits.

Al contrario de la mayoría de los lenguajes de computadora, C tiene un método predefinido para acceder a un único bit en un byte. Este método puede ser mu útil por una serie de razones:

- Si el almacenamiento es limitado, se pueden almacenar varias variables de tipo boole (verdadero o falso) en un byte.
- Ciertas interfaces de dispositivo transmiten información que se codifica en bits en un byte.
- Ciertas rutinas de encriptación necesitan acceder a los bits en un byte.

Aunque se pueden realizar todas estas operaciones con operadores bitwise, un campo de bit puede añadir más estructuración y eficacia al código. El campo de bits puede también hacer más transportable un programa.

El método que C usa para acceder a los bits se basa en la estructura. Un campo de bit es justamente un tipo especial de estructura que define la longitud en bits que tendrá cada elemento. El formato general de una definición de campo de bit es:

```
struct nombre de estructura de tipo {
        tipo nombre_1 : longitud;
        tipo nombre_2 : longitud;
```

```
        tipo nombre _3 : longitud;
};
```

Se debe declarar un campo de bits como **int**, **unsigned** o **signed**. Se debería declarar los campos de bits de longitud 1 como unsigned porque un bit único no puede tener signo.

No se permiten arrays de campo de bits, punteros a campo de bits o funciones que retornen un campo de bits.

El tamaño de un campo de bits no debe sobrepasar el espacio físico, es decir, el lugar ocupado por un entero.

Ejemplos:

```
struct palabra{
        unsigned short car_ascii        : 7;        /*bits 0 a 6*/
        unsigned short bit_paridad      : 1;        /*bit 7*/
        unsigned short operación        : 5;        /*bits 8 al 12*/
        unsigned short                  : 2;        /*bits de relleno*/
        unsigned short bit_signo        : 1;        /*bit 15*/
};
```

El campo de bits anterior utiliza un **short** (16 bits).

```
typedef struct{
        unsigned char bit0:  1;
        unsigned char bit1:  1;
        unsigned char bit2:  1;
        unsigned char bit3:  1;
        unsigned char bit4:  1;
        unsigned char bit5:  1;
        unsigned char bit6:  1;
        unsigned char bit7:  1;
} flag;

flag flag1;     /*Declaramos variable de tipo flag. Campo de bits de un byte de longitud*/

flag1.bit0 = 1;         /*En esta linea de sentencia hacemos uno el bit0 de la flag1*/
```

Uniones.

Como una estructura una unión contiene uno o más elementos, los cuales pueden ser de diferentes tipos. La propiedades de una unión son casi idénticas a las de las estructuras; se permiten la mismas operaciones que las de las estructuras. Su diferencia radica en que los miembros de una estructura se guardan en diferentes direcciones, mientras que los de una unión se guardan en la misma dirección.

Creación de una unión.

Una unión se define de la misma forma que con una estructura, solo que para una unión usamos **union** en lugar de **struct**. Cuando declaramos una unión el compilador reserva suficiente espacio de memoria para poder alojar el elemento que tenga mayor tamaño.

Ejemplo:

```
#include <stdio.h>

typedef union
{
        char ch;
        int i;
        double d;
} sample;

int main(void)
{
        sample s;

        printf("Size: %u\n", sizeof(s));
        return 0;
}
```

En este caso la salida es:

Size: 8

El compilador reserva el espacio para guardar el elemento de mayor tamaño que para este ejemplo es el **double**.

Ejemplo:

```
#include <stdio.h>

typedef union
{
        char ch;
        int i;
        double d;
} sample;

int main(void)
{
        sample s;

        s.ch = 'a';
        printf("%c %d %f\n", s.ch, s.i, s.d);

        s.i = 64;
        printf("%c %d %f\n", s.ch, s.i, s.d);
```

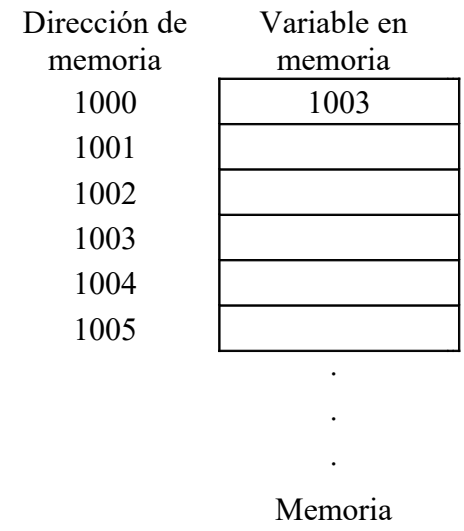
```
        s.d = 12.48;
        printf("%c %d %f\n", s.ch, s.i, s.d);

        return 0;
}
```

Aquí el primer printf muestra correctamente el valor del caracter asignado a *s.ch*, los valores de *s.i* y *s.d* no tienen sentido, luego de asignar el valor 64 a *s.i* el valor que tenía *s.ch* se pierde debido a que este esta en el mismo espacio de memoria de *s.i*, entonces el segundo printf ahora muestra correctamente el valor de 64 que se asignó a *s.i*, por último se asigna 12.48 a *s.d* lo que provoca que se pierda el valor asignado a *s.i*, el último printf muestra entonces el valor asignado a *s.d* correctamente.

Punteros.

Un puntero es una variable que contiene una dirección de memoria. Normalmente esta dirección es una posición de otra variable en la memoria. Aunque podría ser la dirección de un puerto o de la RAM de propósito general. Si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda. La figura siguiente ilustra esta situación.



Variables puntero.

Si una variable va a contener un puntero, entonces se debe declarar como tal. La declaración de un puntero consiste en un tipo base, el (*), y el nombre de la variable. El formato general para la declaración de una variable puntero es

tipo *nombre_de_variable;

donde tipo puede ser cualquier tipo base en C y nombre_de_variable es el nombre de la variable puntero. El tipo base del puntero define qué tipo de variables puede apuntar el puntero. Por ejemplo, estas sentencias declaran punteros a enteros y caracteres:

```
char *p;
int *temp, *start;
```

p	es un puntero a caracter.
temp y start	son punteros a enteros.

Los operadores para puntero.

Estos son dos operadores especiales de puntero: * y &. El **&** es un operador monario que devuelve la dirección de memoria de su operando. (Un operador monario sólo requiere un operando.) Por ejemplo,

```
cont_dir = &cont;
```

asigna la dirección de memoria de la variable cont en cont_dir. Esta dirección es una posición interna en la computadora de la variable. La dirección no tiene nada que ver con el valor de cont. Se puede recordar el funcionamiento de **&** como devolviendo **«la dirección de»** la variable a la que precede. Por tanto, se puede leer la sentencia de asignación como que **«cont_dir»** recibe la dirección de cont.

Para entender esta asignación más claramente, digamos que la variable cont está situada en la dirección 2000. Después de la asignación, cont_dir tendrá el valor 2000.

El segundo operador, *****, es el complemento de **&**. Es un operador monario que devuelve el valor de la variable situada en la dirección que sigue. Por ejemplo, si cont_dir contiene la dirección de memoria de la variable cont, entonces

```
val = *cont_dir;
```

asignará el valor de cont en val. Entonces, si cont originalmente tenía el valor 100, después de esta asignación, val tendrá el valor 100 ya que es el valor guardado en la posición 2000, que es la dirección de memoria que asignó a cont_dir. Recordar que el funcionamiento del ***** como **«en la dirección»**. Así, en este caso, se leería la sentencia dada como **«val recibe el valor de la dirección cont_dir»**.

Desafortunadamente, en C, el signo de multiplicación y el de **«en la dirección»** es el mismo. Al escribir los programas, tener en cuenta que estos operadores no tienen relación. **&** y ***** tienen una precedencia más alta que el resto de operadores aritméticos excepto el menos notorio, que están igualados.

Aquí se muestra un programa que usa dos sentencias de asignación dando a imprimir el número 100 en la pantalla:

```
#include <stdio.h>

void main()
{
        int *cont_dir, cont, val;

        cont = 100;
        cont_dir = &cont;          /* toma la dirección del contador */
        val =*cont_dir;            /* toma el valor de esta dirección */
        printf("%d", val);         /* visualiza 100*/

}
```

La importancia del tipo base.

Como vimos anteriormente, se puede asignar a val el valor de cont indirectamente usando un puntero a cont. En este momento nos podemos preguntar: ¿Cómo sabe C cuantos bytes copiar a val

desde la dirección a la que apunta cont_dir? La respuesta a esta pregunta esta en que el tipo base del puntero determina el tipo de datos que el compilador asumirá que apunta el puntero. En este caso, como cont_dir es de tipo entero, entonces el compilador copiará a val el tamaño que ocupa el entero (por lo general 4 bytes) desde la dirección a la que apunta cont_dir. Si el tipo de cont_dir hubiese sido **char** entonces el compilador copiará un byte a val.

Por esto hay que asegurarse que las variables puntero siempre apunten al tipo correcto de datos. Por ejemplo, cuando se declara un puntero de tipo **int**, el compilador supone que cualquier dirección que guarde el puntero apuntará a una variable entera. El siguiente fragmento de programa estamos cometiendo un error debido a que queremos asignar la dirección de un **float** a un puntero a **int**, el compilador nos mostraría un error. Luego asignamos el valor de lo que en la dirección de un entero a un **float**, es probable que el compilador no indique algún error.

```
#include <stdio.h>

int main()
{
        float x=10.1, y;
        int *p;

        p = &x;             /*Dependiendo del compilador esto podría marcar error o una
                              advertencia. Se asigna la dirección de un float a un int* */
        y = *p;
        printf("%f", y);

        return 0;
}
```

Si el programa compila (lo que puede pasar en algunos compiladores) no se asignará el valor de x a y. Como el programa declara p como un puntero entero, el compilador transferirá sólo los bytes de información a y, y no los cuatro bytes que constituirían a un número en punto flotante.

Asignaciones de punteros.

Como cualquier otra variable, se puede usar un puntero en la parte derecha de una sentencia de asignación para asignar el valor del puntero a otro puntero, por ejemplo:

```
#include <stdio.h>

int main()
{
        int x;
        int *p1, *p2;

        x = 101;
        p1 = &x;                        // Asigna a p1 la dirección de x que contiene el
                                        // valor 101
        p2 = p1;                        // p2 toma la dirección de p1
        printf("La dirección:  %p", p2);  // Visualiza la dirección en hexadecimal de p2
```

```
        printf("El valor de x es %d\n", *p2); // Visualiza el valor de lo que esta en la dirección
                                              // de p2
        return 0;
}
```

Aritmética de punteros.

Se pueden usar sólo dos operaciones aritméticas sobre punteros: **+** y **-** . Para entender como funciona supongamos que p1 es puntero a un **int** (ocupa 4 bytes de memoria), con el valor actual de 2000 (este es el valor de la dirección, recuerde que es un puntero). Después de la expresión

```
p1++;
```

el contenido de p1 será ahora 2004. Como se pudo observar no fue 2001. Esto se debe a que cada vez que la computadora incrementa a p1, apuntará al siguiente entero. Lo mismo ocurre con los decrementos.

Cada vez que la computadora incrementa o decrementa un puntero, apuntará a la dirección de memoria del elemento siguiente en función del tipo base.

El lenguaje C también permite sumar o restar enteros a los punteros. Por ejemplo:

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

hará que p1 apunte al noveno elemento del tipo base, después del que apunta actualmente.

Además de sumar o restar un entero a un puntero no se puede realizar ninguna otra operación como lo son la multiplicación y división; no se pueden sumar o restar dos punteros; no se pueden aplicar los operadores de desplazamiento y enmascaramiento, y no se pueden sumar o restar los tipos **float** o **double** a los punteros.

Ejemplo:

```
#include <stdio.h>
int main()
{
        char c;                 // Ocupa un byte en memoria
        int i;                  // Ocupa 4 bytes en memoria
        char *pC;
        int *pI;

        pC = &c;                // Se asigna la dirección de c
         pI = &i;               // Se asigna la dirección de i

        printf("Direccion de pC: %p\n", pC);
        printf("Direccion de pI: %p\n", pI);

        pC = pC + 1;
        pI = pI + 1;

        printf("Direccion de pC: %p\n", pC);
```

```
        printf("Direccion de pI: %p\n", pI);

        getchar();
        return 0;
}
```

Los dos primeros primeros printf muestran la direcciones actuales de las variables c e i; después de incrementar estas direcciones vemos que lo dos últimos printf muestran las direcciones anteriores incrementadas en 1 para el caso del tipo **char** que ocupa un byte en memoria, y la del **int** incrementada en 4 bytes por que este ocupa 4 bytes en memoria.

Comparaciones de punteros.

Los punteros se pueden compara en una expresión relacional. Por ejemplo, si tenemos dos punteros p y q, es válida la siguiente expresión:

```
p < q
```

Punteros y arrays.

Existe una relación estrecha entre los punteros y arrays. Consideremos las siguientes expresiones:

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

Este fragmento pone a p1 la dirección del primer elemento del array str. *En C, un nombre de array sin un índice es la dirección de comienzo del array.* Es decir, el nombre del array es un puntero a un array. El mismo resultado (puntero al primer elemento del array) se puede generar del siguiente modo:

p1 = &str[0]; equivale a *p1 = str;*

Para acceder al quinto elemento del array se puede escribir:

```
str[4];
```

o

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

Ambas sentencias devolverán el quinto elemento.

Como se puede ver el lenguaje C permite dos métodos de acceso a los elementos de un array. Este hecho es importante porque la aritmética de punteros puede ser más importante que los índices de arrays. Como la velocidad es una consideración frecuente en programación, hace que el uso de punteros para acceder a los electos de un array sea común en programas C.

Para decir verdad el uso de punteros para acceder a los elementos de un array es más eficaz cuando se quiere acceder a este en forma estrictamente ascendente o descendente. Sin embargo si se quiere acceder a estos en forma aleatoria es mejor la indexación ya que generalmente será tan rápido como la evaluación de una expresión compleja de puntero y porque es más fácil programar y entender.

Indexando un puntero.

En C se puede indexar un puntero como si fuera un array. Esta posibilidad indica de nuevo la estrecha relación entre punteros y arrays. Por ejemplo, este fragmento es perfectamente válido e imprime los números l a 5 en la pantalla:

```
/* Indexando un puntero como un array. */

#include <stdio.h>

int main()
{
        int i[5] = {1, 2, 3, 4, 5};
        int *p, t;

        p = i;
        for(t = 0; t < 5; t++)
                printf("%d ", p[t]);

        printf("\n");

        for(t = 0; t < 5; t++)
                printf("%d ", *(p+t));

        return 0;
}
```

En este caso los dos for mostraran la misma salida observándose así que sentencia **p[t]** es idéntica a (**p+t)**.

Punteros de tipo void.

Este es un puntero de tipo genérico ya que puede apuntar a cualquier tipo de datos. Cualquier puntero pude ser convertido (**cast**) a **void***, y vuelto a su tipo original sin perder información. Ahora para punteros distintos de **void***, si queremos asignar un puntero a uno de otro tipo, tenemos que convertir (**cast**) este al tipo al cual queremos asignar.

Ejemplo:

```
char *p1;
int *p2;
```

```
void *p3;
...
p2 = p3;
p3 = p2;
p2 = p1;            // Error p1 es char* y p2 es int*
p2 = (int*)p1;      // Esto es corrrecto estamos haciendo un cast al tipo int* el cual es p2
```

Ejemplo:

```
#include <stdio.h>
int main(void)
{
        void *ptr;
        char s[] = "abcd";
        int i = 10;

        ptr = &i;
        *(int*)ptr += 20;       // Como ptr es void* hacemos un cast a int* para poder acceder
        printf("%d\n", i);
        ptr = s+2;              // Asignamos a ptr la dirección del elemento s[2]
        (*(char*)ptr)++;        // Hacemos un cast a char* para poder opererar como char*
        printf("%s\n", s);

        getchar();
        return 0;
}
```

Usando const con punteros.

Si queremos prevenir el cambio del valor de un variable apuntada por un puntero debemos agregar **const** delante del tipo en la declaración del puntero. Ejemplo:

```
int j, i = 10;
const int *ptr;

ptr = &i;
*ptr = 30;          // No se puede ejecutar, *ptr es una constante
ptr = &j;           // Si podemos cambiar la dirección
```

Si queremos prevenir que un puntero apunte a otra variable debemos agregar **const** después del tipo en la declaración del puntero. Ejemplo:

```
int i, j;
int* const ptr = &i;   // Inicializamos el puntero
ptr = &j;              // No se puede asignar una nueva dirección
*ptr = 30;             // Si podemos asignarle un valor
```

También podemos proteger la dirección y el valor al que apunta esa dirección, para esto debemos agregar **const** adelante y después del tipo en la declaración del puntero. Ejemplo:

```
int i, j;
const int* const ptr = &i;     // Inicialización del puntero
ptr = &j;                      // No se puede la dirección es const
*ptr = 30;                     // No se puede la variable es const
```

Como podemos ver no podemos hacer mucho con esta declaración, solo conseguir el valor del puntero. Podemos decir también que es una declaración rara para usar en la práctica. De todas formas es una declaración que suele usarse en la programación de microcontroladores en donde aveces la cantidad de RAM disponible no es mucha, el uso de **const** según la configuración del compilador hace que esta información se guarde en la flash, y así, liberar espacio de RAM.

Punteros y cadenas.

Como un nombre de array sin un índice es un puntero al primer elemento del array, cuando se usan las funciones de cadenas que se discutieron en los capítulos anteriores, lo que sucede realmente es que la computadora sólo pasa un puntero a las cadenas de las funciones y no a la propia cadena. Para ver cómo funciona, ésta es una forma en la que se escribiría la función strlen( ):

```
int strlen(char *s)
{
        int i = 0;

        while (*s) {
                i++;
                s++,
        }

        return i;
}
```

Recordar que un nulo, que es un valor falso, termina todas las cadenas en C. Por tanto, una sentencia como

```
while(*s);
```

se cumplirá hasta que la computadora alcanza el final de la cadena. Aquí strlen() devolverá cero si se ha llamado con la cadena de longitud cero, o de lo contrario, puede retornar la longitud de la cadena.

En ese momento, probablemente le gustaría saber cómo llamar strlen() con una constante de cadena como argumento. Por ejemplo, podría querer saber cómo funciona este fragmento.

```
printf("longitud de PRUEBA es %d", strlen("PRUEBA"));
```

La respuesta es que cuando se usa una constante de cadena, la computadora sólo pasa un puntero de la constante a strlen(). C almacena automáticamente la cadena real. Más generalmente,

cuando se usa una constante de cadena en cualquier tipo de expresión, la computadora trata la constante como un puntero al primer carácter de la cadena. Por ejemplo, este programa imprime la frase «**este programa funciona**» en la pantalla:

```
#include <stdio.h>

int main()
{
        static char *s  = "este programa funciona";

        printf(s);

        return 0;
}
```

Los caracteres que componen la constante de una cadena se almacenan en una tabla de cadena especial que tiene C. Sólo se utiliza un puntero para esta tabla.

Obteniendo la dirección de un elemento de un array.

Hasta ahora simplemente se ha tratado con la asignación de la dirección del primer elemento de un array a un puntero. Sin embargo, se puede asignar la dirección de un elemento específico de un array aplicando el **&** a un array indexado. Por ejemplo, este fragmento pone la dirección del tercer elemento de x en p:

```
p = &x[2];
```

Un sitio en que esta práctica es especialmente útil es en encontrar una subcadena. Por ejemplo, este programa imprime el resto de una cadena, que se introdujo por el teclado, desde el punto en que la computadora encuentra el primer espacio:

```
#include <stdio.h>

/* Visualiza la cadena después de que se encuentra el primer espacio. */

int main()
{
        char s[8];
        char *p;
        int i ;

        printf("introduce una cadena: ");
        gets(s);

/* Encuentra el primer espacio o el fin de la cadena */
```

```
        for (i = 0; s[i] && s[i] !='  '; i++);
        p = &s[i];
        printf(p);

        return 0;
}
```

Este programa funciona porque p apunta tanto al espacio como a un nulo (en caso de que la cadena no contenga espacios). Si existe un espacio entoces imprimirá el resto de la cadena. Si es un nulo, printf() no imprimirá nada. Por ejemplo, si se introduce «**eh aquí**», se visualizará «**aquí**».

Punteros y estructuras.

Un puntero a una estructura se usa normalmente como se usaría un puntero a una variable. Ejemplo:

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

typded struct
{
        char nombre[50];
        int edad;
}estudiante;

int main(void)
{
        estudiante *pEstu, sEstu;

        pEstu = &sEstu;         //Pasamos la dirección de sEstu al puntero
        strcpy((*pEstu).nombre, "ELNombre"); //A la dirección de nombre copiamos ElNombre
        (*pEstu).edad = 22;             //A la dirección de edad asignamos 22
        printf("Nombre: %s  - Edad: = %.2f\n", sEstu.name, sEstu.edad);
        return 0;
}
```

Otra alternativa para acceder a los elementos de una estructura cuando esta es declarada como puntero es utilizar el operador "**->**", entonces el ejemplo anterior lo podemos escribir como sigue:

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

typded struct
```

```
{
        char nombre[50];
        int edad;
}estudiante;

int main(void)
{
        estudiante *pEstu, sEstu;

        pEstu = &sEstu;         //Pasamos la dirección de sEstu al puntero
        strcpy(pEstu->nombre, "ELNombre"); //A la dirección de nombre copiamos ElNombre
        pEstu->edad = 22;               //A la dirección de edad asignamos 22
        printf("Nombre: %s  - Edad: = %.2f\n", sEstu.name, sEstu.edad);
        return 0;
}
```

Arrays de punteros.

Se pueden hacer arrays de punteros como de cualquier otro tipo de datos. La declaración para un array de punteros **int** de tamaño 10 es

```
int *x[10];
```

Para asignar una dirección de una variable entera llamada var al tercer elemento del array de punteros, se escribirá:

```
x[2] = &var;
```

Para encontrar el valor de var, se escribirá:

```
value = *x[2];
```

Punteros a punteros.

El concepto de arrays de punteros es directo ya que el array mantiene su significado claro. Sin embargo, se puede confundir los punteros a punteros.

Un puntero a un puntero es una forma de direccionamiento indirecto múltiple, o una cadena de punteros. Como se ve en la figura, en el caso de un puntero normal, el valor del puntero es la dirección de la variable que contiene el valor deseado. En el caso de un puntero a puntero, el primer puntero contiene la dirección del segundo puntero, que apunta a la variable que contiene el valor deseado.

Se puede llevar direccionamiento indirecto múltiple a cualquier extensión deseada, pero hay pocos casos donde más de un puntero a puntero es necesario, o incluso es bueno de usar. La dirección indirecta en exceso es difícil de seguir y es propensa a errores conceptuales. (No confundir direccionamiento indirecto múltiple con listas enlazadas, que se usan en aplicaciones como las bases de datos.)

Se debe declarar una variable que es un puntero a puntero como tal. Se puede hacer poniendo un asterisco adicional delante del nombre de la variable. Por ejemplo, esta declaración le dice al compilador que nuevobalance es un puntero aun puntero de tipo **float**:

```
float **nuevobalance;
```

Inicialización de punteros.

Después de declarar un puntero, pero antes de asignarle un valor, contendrá un valor desconocido.
Si se trata de usar el puntero antes de asignarle un valor (darle una dirección), probablemente el sistema operativo de la computadora nos indicará un error.

Por convención se daría a un puntero que apunta a ninguna parte el valor nulo (NULL) para significar que el puntero apunta a nada. Sin embargo, llegar a ser un puntero con un valor nulo no hace más seguro su uso. Si se usa un puntero nulo en el lado izquierdo de una sentencia de asignación, se correría todavía el riesgo de estropear el programa o el sistema operativo.

Como C supone que un puntero nulo está sin usar, se puede usar el puntero nulo para hacer muchas de las rutinas de puntero más fácil de codificar y más eficaces. Por ejemplo, se podría usar un puntero nulo para enmascarar el final de un array de punteros. Haciendo así que una rutina que accede al array sepa que ha alcanzado el final cuando se encuentra el valor nulo. El bucle **for** mostrado aquí ilustra esta aproximación:

```
/* Busca un nombre suponiendo que el último elemento de p es nulo. */

for(t = 0; p[t]; t++){
        if(!strcmp(p[t], name))
                break;
}
```

El bucle funcionará hasta que la rutina encuentre lo que busca o el puntero nulo. Debido a que el final del array está marcado con un nulo, la condición que controla el bucle caerá cuando la rutina alcance el nulo.

Es una práctica común en los programas profesionales escritos en C inicializar las cadenas. Otra variación de la inicialización de cadenas es su tipo de declaración de cadena:

```
char *p = "hola amigos\n";
```

Como se puede ver, el puntero p no es un array. La razón de que este tipo de inicialización funcione tiene que ver con la manera en que funcionan los compiladores: por ejemplo, Turbo C crea lo que se llama tabla de cadenas, que se usan internamente para guardar las constantes de cadena de un programa. Por tanto, esta sentencia de declaración pone la dirección de la cadena «**hola amigos**», almacenada en la tabla de cadena, en el puntero p. A través del programa, p puede usarse como otra cadena. Por ejemplo, el siguiente programa es válido:

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

char *p = "hola amigos";

int main()
{
        int t;
```

```
        /* imprime la cadena hacia delante y hacia atrás */

        printf(p);
        for(t = strlen(p)-1; t > -1; t--)
                printf("%c", p[t]);
}
```

Problemas Unidad 2.

Uso arrays:

1. Realizar un programa que asigne datos a una matriz unidimensional a de 10 elementos, y a continuación escribir el contenido de dicha matriz.
2. Realizar un programa que lea 10 valores enteros y los guarde en un array. Luego el programa debe verificar si el array es simétrico, esto es si el primer elemento del array es igual a último, si el segundo elemento es igual al penúltimo, y así con el resto.
3. Realizar un programa que lea las notas correspondientes a 10 alumnos de un determinado curso, las almacenen en un array y nos de como resultado la nota correspondiente del curso.
4. Realizar un programa que asigne datos a una matriz de enteros t de dos dimensiones (5x3) y a continuación escriba las sumas correspondientes a las filas de dicha matriz.
5. Realizar un programa que almacene en una matriz las notas de los alumnos correspondientes a los cursos a, b y c de Ingeniería, y a continuación nos permita consultar dicha matriz indicando cuantos el porcentaje de aprobados. Suponga 30 alumnos por curso, cargue las nota generando números aleatorios que se encuentren entre 3.0 y 10. Para aprobar considere una nota mayor o igual a 6.00.
6. Realizar un programa que lea una lista de valores enteros. A continuación y sobre la lista encontrar los valores máximo y mínimo, y el índice en el cual se encuentra los mismos. Considere una lista de 25 elementos y cargue la matriz con números aleatorios.

Uso cadenas:

7. Examinar una cadena de caracteres. Leer una cadena de caracteres y a continuación escribir por cada carácter, su dirección, el carácter y su valor ASCII.
8. Escribir un programa que cree una cadena compuesta por los caracteres del alfabeto inglés todos en mayúsculas y en minúsculas.
9. Escribir un programa que lea una línea de la entrada y la almacene en un array de caracteres. A continuación, convertir los caracteres escritos en mayúscula a minúscula, y los escritos en minúscula a mayúscula.

Uso punteros:

10. Realizar un programa que use un puntero para leer un **double** y de como resultado la parte fraccionaria del número ingresado.
11. Realizar un programa que use tres variables de tipo puntero para leer tres enteros. El programa debe forzar al usuario a entrar los datos en forma descendente.
12. Realiza un programa que lea 100 valores enteros positivos de forma aleatoria. Luego el programa debe reemplazar los valores duplicado en el array por el valor -99. Utilice aritmética de punteros para realizar esta operación.

Complementarios:

13. Realizar un programa que lea 20 cadenas (menores a 100 caracteres) y las almacene en un array de dos dimensiones. Luego el programa debe leer una cadena, si esta se encuentra en el array debe de eliminarse. Para elimina la cadena deben moverse las cadena de más abajo un lugar hacia arriba.
14. Realizar un programa que lea una lista de alumnos y su correspondiente nota final de curso, dando como resultado el tanto por ciento de alumnos aprobados y libres.

15. Consideremos una biblioteca compuesta por libros y revistas. Por cada libro o revista figura la siguiente información:

    Número de referencia.
    Titulo.
    Nombre de autor.
    Editorial.
    Clase de publicación (libro o revista).
    Número de edición (solo libros).
    Año de publicación (solo libros).
    Nombre de la revista.

    Realizar un programa que nos permita:

    a) Almacenar la información correspondiente a la biblioteca en un array.
    b) Listar dicha información.

16. Utilizando el método de clasificación de la burbuja. Realizar un programa que ordene alfabéticamente una lista de cadenas de caracteres.
17. Usar el método de ordenación de la burbuja para ubicar de mayor a menor y viceversa en una lista, 100 números. Presentar en pantalla en filas de 10 columnas los valores desordenados y ordenados. Cargue los valores de forma aleatoria.