Programación de dispositivos electrónicos

Unidad 4

Matrices-Estructuras-Ficheros-Punteros

Héctor Albano Diego Codevilla

Matrices o Arrays Multidimencionales

Es posible crear arrays de mas de dos dimensiones, su sintaxis para declararla sera:

tipodevariable nombredelarray [dimensión1] [dimensión2] [...] [dimensiónN];

Por ejemplo si queremos almacenar en una matriz el número de alumnos con que cuenta una academia ordenados en función del nivel y del idioma que se estudia. Tendremos 3 filas que representarán Nivel básico, medio o de perfeccionamiento y 4 columnas que representarán los idiomas (0 = Inglés, 1 = Francés, 2 = Alemán y 3 = Ruso). La declaración de dicha matriz sería:

int alumnosfxniveleidioma [3] [4];

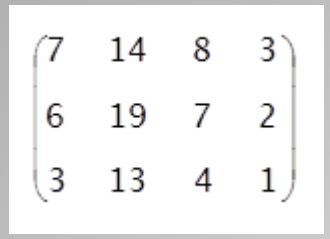
Podríamos asignar contenidos de la siguiente manera:

```
alumnosfxniveleidioma[0] [0] = 7;        alumnosfxniveleidioma[0] [1] = 14;
alumnosfxniveleidioma[0] [2] = 8;        alumnosfxniveleidioma[0] [3] = 3;
alumnosfxniveleidioma[1] [0] = 6;        alumnosfxniveleidioma[1] [1] = 19;
alumnosfxniveleidioma[1] [2] = 7;        alumnosfxniveleidioma[1] [3] = 2;
alumnosfxniveleidioma[2] [0] = 3;        alumnosfxniveleidioma[2] [1] = 13;
alumnosfxniveleidioma[2] [2] = 4;        alumnosfxniveleidioma[2] [3] = 1;
```

Tambien se puede definir la matriz y asignarle valores

int alumnosfxniveleidioma [3] [4] **=** {{7, 14, 8, 3}, {6, 19, 7, 2},{ 3, 13, 4, 1}} ;

La representación gráfica que podríamos asociar a esta asignación de datos sería esta matriz:



$$\begin{pmatrix} 7 & 14 & 8 & 3 \\ 6 & 19 & 7 & 2 \\ 3 & 13 & 4 & 1 \end{pmatrix}$$

También podemos usar arrays de dos dimensiones si queremos guardar una lista de cadenas de texto, como en este

```
#include <stdio.h>
int main()
{
    char mensajeError[5][80] = { "Fichero no encontrado",
    "El fichero no se puede abrir para escritura",
    "El fichero está vacío",
    "El fichero contiene datos de tipo incorrecto",
    "El fichero está siendo usado" };
    printf("El segundo mensaje de error es: %s \n", mensajeError[1]);
    printf("La primera letra del tercer mensaje de error es: %c \n", mensajeError[2][0]);
    return 0;
}
```

Estructuras

DECLARACION DE ESTRUCTURAS

Los vectores son organizaciones secuenciales de variables simples de un mismo tipo, resulta necesario en múltiples aplicaciones, agrupar variables de distintos tipos, en una sola entidad.

Este tipo de variable compuesta, que permite manejar ésta situación típica de las Bases de Datos, se denomina ESTRUCTURA, o registros, ya que tienen muchos aspectos en común con los registros usados en bases de datos. Y siguiendo la misma analogía, cada objeto de una estructura se denomina a menudo campo.

No hay limitaciones en el tipo ni cantidad de variables que pueda contener una estructura (incluso puede contener otra estructura), mientras su máquina posea memoria suficiente como para alojarla. Una estructura no puede contenerse a sí misma como miembro y no pueden existir dos miembros con el mismo nombre, aunque sí pueden coincidir con el nombre de otra variable simple, (o de un miembro de otra estructura), declaradas en otro lugar del programa.

Para usarlas , se deben seguir dos pasos: primero declarar la estructura en sí, es decir, darle un nombre y describir a sus objetos, y luego declarar a uno o más objetos de la estructura.

struct <identificador de la estructura>]
{
<tipo de dato objeto1><nombre objeto1>;
<tipo de dato objeto2><nombre objeto2>;
<tipo de dato objeto3><nombre objeto3>;
….
} <objeto_estructura> ,<objeto_estructura>, …..;

El identificador de la estructura es un nombre opcional para referirse a la estructura.

Los objeto estructura son objetos declarados de ese tipo de la estructura, y su inclusión también es opcional. Si bien, aún siendo ambos opcionales, al menos uno de estos elementos debe existir.

Una vez definida una estructura, es decir, si hemos especificado un nombre para ella, se pueden declarar más objeto estructura en cualquier parte del programa. Para ello usaremos la forma normal de declaración de objetos, es decir:

struct<identificador> <objeto_estructura> ,<objeto_estructura>...;

La primera sentencia es sólo una declaración, es decir que no asigna lugar en la memoria para la estructura, sólo le avisa al compilador como tendrá que manejar a dicha memoria para alojar variables del tipo struct definidas.
En la segunda sentencia, se definen objeto estructura del tipo de la estructura anterior, ésta definición debe colocarse luego de la declaración, y se reserva memoria.
Las dos sentencias pueden combinarse en una sola, dando la definición a continuación de la declaración (en este caso también se reserva memoria).

Las estructuras pueden referenciarse completas, usando su nombre, como hacemos con los objetos que ya conocemos, y también se puede acceder a los elementos definidos en el interior de la estructura, usando el operador de selección (.), un punto.

Es muy importante recalcar que , dos estructuras , aunque sean del mismo tipo, no pueden ser asignadas o comparadas la una con la otra, en forma directa, sino asignando o comparándolas miembro a miembro.

<objeto_estructura>. <nombre objeto>=<valor>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct datos
   {
      char nombre[30];
      int edad;
      float nota;
   };
int main()
{
   struct datos persona;

   strcpy(persona.nombre,"Hector");
   persona.edad = 20;
   persona.nota = 7.5;

   printf("El nonbre es %s\n", persona.nombre);
   printf("La edad es %d\n", persona.edad);
   printf("La nota es %f\n", persona.nota);
   return 0;
}
```

Inicialización de estructuras

Las estructuras pueden ser inicializadas mediante listas de inicialización como con los arreglos. Para inicializar una estructura se escribe en la declaración de la variable a continuación del nombre de la variable un signo igual con los inicializadores entre llaves y separados por coma. Si en la lista aparecen menos inicializadores que en la estructura los miembros restantes son automáticamente inicializados a 0.

Las variables de estructura también pueden ser inicializadas en enunciados de asignación asignándoles una variable del mismo tipo o asignándole valores a los miembros individuales de la estructura.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct datosPersona
        {
        char nombre[30];
        int edad;
        float nota;
         }ficha0= {"Carlos",23};
int main()
{
   printf("El nombre es es %s\n", ficha0.nombre);
   printf("La edad es %d\n", ficha0.edad);
   printf("La nota es %f\n", ficha0.nota);

   struct datosPersona ficha1 = {"Juan",20,7.5};

   printf("El nombre es es %s\n", ficha1.nombre);
   printf("La edad es %d\n", ficha1.edad);
   printf("La nota es %f\n", ficha1.nota);
   return 0;
}
```

Arrays de estructuras

La combinación de las estructuras con los arrays proporciona una potente herramienta para el almacenamiento y manipulación de datos.

Cuando hablamos de arrays dijimos que se podían agrupar, para formarlos , cualquier tipo de variables, esto es extensible a las estructuras y podemos entonces agruparlas ordenadamente, como elementos de un array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct
{
char nombre[50];
int edad;
float nota;
}alumnos[5];
int main()
{
int i;
for(i=0;i<5;i++){
printf("Ingrese el nombre del alumno %d\n",i+1);
gets(alumnos[i].nombre);
printf("Ingrese la edad del alumno %d\n",i+1);
scanf("%d",&alumnos[i].edad);
printf("Ingrese la nota del alumno %d\n",i+1);
scanf("%f",&alumnos[i].nota);
fflush(stdin);
}
printf("El nombre es es %s\n", alumnos[1].nombre);
printf("La edad es %d\n", alumnos[1].edad);
printf("La nota es %f\n", alumnos[1].nota);

alumnos[1].nota=10;

printf("La nota es %f\n", alumnos[1].nota);
return 0;
}
```

Estructuras anidadas

Podemos encontrarnos con un registro que tenga varios datos, y que a su vez ocurra que uno de esos datos esté formado por varios datos más sencillos. Para hacerlo desde C, incluiríamos un "struct" dentro de otro, declarando primero el "interior" y luego el "exterior", así:

```
#include <stdlib.h>
#include <stdio.h>
struct fecha
{
int dia;
int mes;
int anyo;
};
struct
{
char inicial;
struct fecha diaDeNacimiento;
float nota;
} persona;
int main()
{
persona.inicial = 'I';
persona.diaDeNacimiento.mes = 8;
persona.nota = 7.5;
printf("La nota es %f \n", persona.nota);
printf("Nacio en el mes %d", persona.diaDeNacimiento.mes);
return 0;
}
```

Ordenamiento de datos

Es muy frecuente querer ordenar datos que tenemos en un array. Para conseguirlo, existen varios algoritmos sencillos, que no son especialmente eficientes, pero son fáciles de programar. La falta de eficiencia se refiere a que la mayoría de ellos se basan en dos bucles "for" anidados, de modo que en cada pasada quede ordenado un dato, y se dan tantas pasadas como datos existen, de modo que para un array con 1.000 datos, podrían llegar a tener que hacerse un millón de comparaciones. Existen métodos más efectivos, pero más difíciles de programar.

Veremos tres de estos métodos simples de ordenación:

Método de burbuja

(Intercambiar cada pareja consecutiva que no esté ordenada)

```
Para i=1 hasta n-1
      Para j=i+1 hasta n
            Si A[i] > A[j]
                  Intercambiar ( A[i], A[j])
```

(Nota: algunos autores hacen el bucle exterior creciente y otros decreciente, así:)

```
Para i=n descendiendo hasta 1
      Para j=2 hasta i
            Si A[j-1] > A[j]
                  Intercambiar ( A[j-1], A[j])
```

Selección directa
(En cada pasada busca el menor, y lo intercambia al final de la pasada)

```
Para i=1 hasta n-1
      menor = i
      Para j=i+1 hasta n
            Si A[j] < A[menor]
                  menor = j
      Si menor <> i
            Intercambiar ( A[i], A[menor])
```

Inserción directa
(Comparar cada elemento con los anteriores -que ya están ordenados- y desplazarlo hasta su posición correcta)

```
Para i=2 hasta n
      j=i-1
      mientras (j>=1) y (A[j] > A[j+1])
            Intercambiar ( A[j], A[j+1])
            j = j - 1
```

Ficheros o archivos

Podemos clasificar los archivos según varias categorías:

- **Dependiendo de la dirección del flujo de datos:**

De entrada: los datos se leen por el programa desde el archivo.

De salida: los datos se escriben por el programa hacia el archivo.

De entrada/salida: los datos pueden se escritos o leídos.

- **Dependiendo del tipo de valores permitidos a cada byte:**

De texto: sólo están permitidos ciertos rangos de valores para cada byte. Algunos bytes tienen un significado especial, por ejemplo, el valor hexadecimal 0x1A marca el fin de fichero. Si abrimos un archivo en modo texto, no será posible leer más allá de un byte con ese valor, aunque el fichero sea más largo.

Binarios: están permitidos todos lo valores para cada byte. En estos archivos el final del fichero se detecta de otro modo, dependiendo del soporte y del sistema operativo. La mayoría de las veces se hace guardando la longitud del fichero. Cuando queramos almacenar valores enteros, o en coma flotante, o imágenes, etc., deberemos usar este tipo de archivos.

- **Según el tipo de acceso:**

Archivos secuenciales: no permiten acceder al punto exacto donde se guarda la información sin antes haber partido desde el principio y sin haber leído toda la información, hasta el punto que estábamos buscando.

Archivos de acceso aleatorio: permiten acceder a cualquier punto de ellos para realizar lecturas y/o escrituras.

- **Según la longitud de registro:**

Longitud variable: en realidad, en este tipo de archivos no tiene sentido hablar de longitud de registro, podemos considerar cada byte como un registro. También puede suceder que nuestra aplicación conozca el tipo y longitud de cada dato almacenado en el archivo, y lea o escriba los bytes necesarios en cada ocasión. Otro caso es cuando se usa una marca para el final de registro, por ejemplo, en ficheros de texto se usa el carácter de retorno de línea para eso. En estos casos cada registro es de longitud diferente.

Longitud constante: en estos archivos los datos se almacenan en forma de registro de tamaño constante. En C usaremos estructuras para definir los registros. C dispone de funciones de biblioteca adecuadas para manejar este tipo de ficheros.

Es posible crear archivos combinando cada una de estas categorías

Buffers:

Como el acceso a los ficheros es lento comparado con el acceso a memoria. Generalmente, no se accede a ficheros externos cada vez que se realiza una operación de lectura o escritura. En su lugar, se mantiene una copia de una parte del fichero en la memoria, se realizan las operaciones de lectura/escritura que sea posible dentro de esa zona, y cuando sea necesario, porque alguna operación acceda a posiciones fuera de la zona almacenada, se vuelca esa zona al fichero y se lee otro tramo del fichero en memoria.

A estas zonas se le llaman *buffers*, y mejoran sensiblemente el acceso a los ficheros en lo que respecta a la velocidad.

Cuanto más grande es un buffer, mejor será el tiempo de acceso al fichero. En el caso ideal, el tamaño del buffer es mayor o igual que el del fichero, y todas las operaciones de lectura y escritura del fichero se realizan en memoria, de modo que sólo es necesario hacer una lectura del fichero y, si se ha modificado, una escritura.

Pero no todo son ventajas. Cuando se trabaja con buffers, las actualizaciones físicas del fichero están diferidas, en relación a las actualizaciones hechas por el programa, de modo que el fichero no siempre tiene una información actualizada. Esto plantea dos problemas:

Si la aplicación termina de forma inesperada.

Cuando un fichero deba ser accedido por varios usuarios de forma simultánea, se pueden presentar problemas de concurrencia.

Escritura en un fichero de texto

Para manejar ficheros, siempre deberemos realizar tres operaciones básicas:

- Abrir el fichero.
- Leer datos de él o escribir datos en él.
- Cerrar el fichero.

Eso sí, no siempre podremos realizar esas operaciones, así que además, tendremos que comprobar los posibles errores. Por ejemplo, puede ocurrir que intentemos abrir un fichero que realmente no exista, o que queramos escribir en un dispositivo que sea sólo de lectura.

```
#include <stdio.h>
int main()
{
FILE* fichero;
fichero = fopen("prueba.txt", "wt");
 fputs("Esto es una línea\n", fichero);
 fputs("Esto es otra", fichero);
 fputs(" y esto es continuación de la anterior\n", fichero);
 fclose(fichero);
 return 0;
}
```

- **FILE** es el tipo de datos asociado a un fichero. Siempre aparecerá el asterisco a su derecha.
- Para abrir el fichero usamos "**fopen**", que necesita dos datos: el nombre del fichero y el modo de lectura. El modo de lectura estará formado por varias letras, de las cuales por ahora nos interesan dos: "w" indicaría que queremos escribir (write) del fichero, y "t" avisa de que se trata de un fichero de texto (text). Como abrimos el fichero para escribir en él, se creará el fichero si no existía, y **se borrará** su contenido si ya existía (más adelante veremos cómo añadir a un fichero sin borrar su contenido).
- Para **escribir** en el fichero y para leer de él, tendremos órdenes muy parecidas a las que usábamos en pantalla. Por ejemplo, para escribir una cadena de texto usaremos "fputs", que recuerda mucho a "puts" pero con la diferencia de que no avanza de línea después de cada texto (por eso hemos añadido \n al final de cada frase).
- Finalmente, **cerramos** el fichero con "fclose".

Lectura de un fichero de texto

Si queremos leer de un fichero, los pasos son muy parecidos, sólo que lo abriremos para lectura (el modo de escritura tendrá una "r", de "read", en lugar de "w"), y leeremos con "fgets".

```
#include <stdio.h>
int main()
{
FILE* fichero;
char nombre[80] = "c:\\autoexec.bat";
char linea[81];
fichero = fopen(nombre, "rt");
if (fichero == NULL)
{
printf("No existe el fichero!\n");
exit(1);
}
fgets(linea, 80, fichero);
puts(linea);
fclose(fichero);
return 0;
}
```

- En el nombre del fichero, hemos indicado un nombre algo más complejo. En estos casos, hay que recordar que si aparece alguna barra invertida (\), deberemos duplicarla, porque la barra invertida se usa para indicar ciertos códigos de control. Por ejemplo, \n es el código de avance de línea y \a es un pitido. El modo de lectura en este caso es "r" para indicar que queremos leer (read) del fichero, y "t" avisa de que es un fichero de texto.

- Para **leer** del fichero y usaremos "fgets", que se parece mucho a "gets", pero podemos limitar la longitud del texto que leemos (en este ejemplo, a 80 caracteres) desde el fichero. Esta cadena de texto **conservará** los caracteres de avance de línea.

- Si **no se consigue** abrir el fichero, se nos devolverá un valor especial llamado NULL

- La orden "**exit**" es la que nos permite abandonar el programa en un punto.

Modos de apertura

Estas son las letras que pueden aparecer en el modo de apertura del fichero, para poder añadir datos a un fichero ya existente

Tipo	Significado
r	Abrir sólo para lectura.
w	Crear para escribir. Sobreescribe el fichero si existiera ya (borrando el original).
a	Añade al final del fichero si existe, o lo crea si no existe.
+	Se escribe a continuación de los modos anteriores para indicar que también queremos modificar. Por ejemplo: r+ permite leer y modificar el fichero.
t	Abrir en modo de texto.
b	Abrir en modo binario.

Leer y escribir letra a letra

Si queremos leer o escribir sólo una letra, tenemos las órdenes "fgetc‟ y "fputc"

```
letra = fgetc(fichero);
fputc (letra, fichero);
```

Lectura hasta el final del fichero

Normalmente no querremos leer sólo una frase del fichero, sino procesar todo su contenido. Para ayudarnos, tenemos una orden que nos permite saber si ya hemos llegado al final del fichero. Es "feof" (EOF es la abreviatura de End Of File, fin de fichero).

```
#include <stdio.h>
int main()
{
FILE* fichero;
char nombre[80] = "c:\\autoexec.bat";
char linea[81];
fichero = fopen(nombre, "rt");
if (fichero == NULL)
{
printf("No existe el fichero!\n");
exit(1);
}
while (! feof(fichero))
{
fgets(linea, 80, fichero);
if (! feof(fichero))
puts(linea);
}
fclose(fichero);
return 0;
}
```

Esa será la estructura básica de casi cualquier programa que deba leer un fichero completo, de principio a fin: abrir, comprobar que se ha podido acceder correctamente, leer con "while !(feof(...))" y cerrar.

Ficheros binarios

Se puede manejar ficheros que contengan información de cualquier tipo. En este caso, utilizamos "fread" para leer un bloque de datos y "fwrite" para guardar un bloque de datos. Estos datos que leamos se guardan en un buffer (una zona intermedia de memoria). En el momento en que se lean menos bytes de los que hemos pedido, quiere decir que hemos llegado al final del fichero.

En general, el manejo de "fread" es el siguiente:

cantidadLeida = fread(donde, tamañoDeCadaDato, cuantosDatos, fichero);

Por ejemplo, para leer 10 números enteros de un fichero (cada uno de los cuales ocuparía 4 bytes, si estamos en un sistema operativo de 32 bits), haríamos:

```
int datos[10];
resultado = fread(&datos, 4, 10, fichero);
if (resultado < 10)
    printf("Había menos de 10 datos!");
```

Al igual que ocurría con "scanf", la variable en la que guardemos los datos se deberá indicar precedida del símbolo &. Si se tratara de una cadena de caracteres (bien porque vayamos a leer una cadena de texto, o bien porque queramos leer datos de cualquier tipo pero con la intención de manejarlos byte a byte), como char dato[500] no será necesario indicar ese símbolo &, como en este ejemplo:

```
char cabecera [40];
resultado = fread(cabecera, 1, 40, fichero);
if (resultado < 40)
printf("Formato de fichero incorrecto, no está toda la cabecera!");
    else
printf("El byte en la posición 5 es un %d", cabecera[4]);
```

Punteros

Un puntero es una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato en un arreglo.

Esto quiere decir, que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o una función. Los punteros se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencias en las llamadas a funciones.

Declaración de un puntero:

```
tipo de variable apuntada   *nombre_del_puntero;
```

```
int *pint ;
double *pfloat ;
char  *letra , *codigo , *caracter ;
```

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.

Inicialización de un puntero:

```
int var1 ;          /* declaro ( y creo en memoria ) una variable entera ) */
int *pint ;         /* declaro ( y creo en memoria ) un puntero que contendrá la dirección de una
                        variable entera */
pint = &var1 ;      /* escribo en la dirección de memoria donde está el puntero la dirección de la
                        variable entera */
```

OBJETO APUNTADO POR UN PUNTERO

La operación contraria es obtener el objeto referenciado por un puntero, con el fin de manipularlo, ya sea modificando su valor u obteniendo el valor actual. Para manipular el objeto apuntado por un puntero usaremos el operador de indirección, que es un asterisco (*). El operador de indirección sólo está permitido usarlo con punteros, y podemos leerlo como "objeto apuntado por".

La expresión " *nombre_del_puntero " , implica "contenido de la variable apuntada por el mismo".

Así por ejemplo serán expresiones equivalentes :

```
y = var1 ;
y = *pint ;
printf("%d" , var1 ) ;
printf("%d" , *pint) ;
```

```
#include <stdio.h>
main()
{
char var1 ;                /*una variable del tipo caracter */
char *pchar;               /* un puntero a una variable del tipo caracter */
pchar = &var1 ;            /*asignamos al puntero la direccion de la variable */
for (var1 = 'a'; var1 <<= 'z'; var1++)
printf("%c", *pchar) ;  /* imprimimos el valor de la variable apuntada */
return 0 ;
}
```

Hay un error, que se comete con bastante frecuencia, y es cargar en la dirección apuntada por un puntero a un tipo dado de variable, el contenido de otro tipo de las mismas, por ejemplo :

```
double d = 10.0 ;
int i = 7 , *pint ;
pint = &i ;
*pint = 10 ;   /* correcto,equivale a asignar a i el valor 10 */ ;
*pint = d ;    /* ERROR se pretende cargar en una variable entera un valor double */
pint = &d ;    /* INCORRECTO se pretende apuntar a una variable double con un
                  puntero declarado como apuntador a int */
pint = 4358 ;  /* ?????? */
```

El primer error, la asignación de un double, produce la pérdida de información dada por la conversión automática de tipo de variable. El segundo produce un llamado de atención rotulado como "asignación sospechosa de un puntero". Resumiendo, las variables ó constantes cargadas por de referenciación de un puntero, deben coincidir en tipo con la declaración de aquel.

La asignación de una constante a un puntero, y no a la variable apuntada por él, es un serio error, ya que debe ser el compilador, el encargado de poner en él el valor de la dirección, aquel asi lo declara dando un mensaje de "conversión de puntero no transportable ". Si bien lo compila puede "colgarse" la máquina o lo que es peor destruirse involuntariamente información contenida en un disco, etc.

Hay un sólo caso en el que esta asignación de una constante a un puntero es permitida, muchas funciones para indicar que no pueden realizar una acción o que se ha producido un error de algún tipo, devuelven un puntero llamado "Null Pointer", lo que significa que no apunta a ningún lado válido, dicho puntero ha sido cargado con la dirección NULL (por lo general en valor 0), así la asignación: pint=NULL; es válida y permite luego operaciones relacionales del tipo: if(pint).... ó if(print != NULL) para convalidar la validez del resultado devuelto por una función.

PUNTEROS Y ARRAYS

Hay una relación muy cercana entre los punteros y los arrays . Ya vimos previamente que el nombre de un array era equivalente a la dirección del elemento [0] del mismo . La explicación de esto es ahora sencilla: el nombre de un array, para el compilador C, es un puntero inicializado con la dirección del primer elemento del array. Sin embargo hay una importante diferencia entre ambos.
Veamos algunas operaciones permitidas entre punteros :

ASIGNACION

```
float var1 , conjunto[] = { 9.0 , 8.0 , 7.0 , 6.0 , 5.0 );
float *punt ;
punt = conjunto ;    /* equivalente a hacer : punt = &conjunto [0] */
var1 = *punt ;
*punt = 25.1 ;
```

Es perfectamente válido asignar a un puntero el valor de otro, el resultado de ésta operación es cargar en el puntero punt la dirección del elemento [0] del array conjunto, y posteriormente en la variable var1 el valor del mismo (9.0) y para luego cambiar el valor de dicho primer elemento a 25.1.
Veamos cual es la diferencia entre un puntero y el nombre de un array: el primero es una variable , es decir que puedo asignarlo , incrementarlo etc , en cambio el segundo es una constate, que apunta siempre al primer elemento del array con que fue declarado, por lo que su contenido no puede ser variado.

ASIGNACION ERRONEA

```
int conjunto[5] , lista[] = { 5 , 6 , 7 , 8 , 0 ) ;
int *apuntador ;
apuntador = lista ;      /* correcto */
conjunto = apuntador;    /* ERROR ( se requiere en Lvalue no constante ) */
lista = conjunto ;       /* ERROR  ( idem ) */
apuntador = &conjunto    /* ERROR no se puede aplicar el operador & (dirección) a una constante */
```

INCREMENTO O DECREMENTO DE UN PUNTERO

```
int *pint , arreglo_int[5] ;
double *pdou , arreglo_dou[6] ;
pint = arreglo_int ;       /* pint apunta a arreglo_int[0] */
pdou = arreglo_dou ;    /* pdou apunta a arreglo_dou[0] */
pint += 1 ;                /* pint apunta a arreglo_int[1]  */
pdou += 1 ;               /* pdou apunta a arreglo_dou[1]  */
pint++ ;                  /* pint apunta a arreglo_int[2] */
pdou++ ;                   /* pdou apunta a arreglo_dou[2] */
```

Hemos declarado y asignado dos punteros , uno a int y otro a double. Luego apuntamos a los elementos [0] de los arrays. En las dos instrucciones siguientes incrementamos en uno dichos punteros. En ambos casos se incrementa contenido del puntero (dirección del primer elemento del array) en un número igual a la cantidad de bytes que tiene la variable con que fue declarado.

Es decir que el contenido de pint es incrementado en dos bytes(un int tiene 2 bytes) mientras que pdou es incrementado 8 bytes(por ser un puntero a double), el resultado entonces es el mismo para ambos, ya que luego de la operación quedan apuntando al elemento siguiente del array , arreglo_int[1] y arreglo_dou[1].

Será muy fácil "barrer" arrays, independientemente del tamaño de variables que lo compongan, permitiendo por otro lado que el programa sea transportable a distintos hardwares sin preocuparnos de la diferente cantidad de bytes que pueden asignar los mismos a los distintos tipos de variables.

Los dos instrucciones siguientes, vuelven a a incrementarse los punteros, apuntando ahora a los elementos siguientes de los arrays. Ocurre lo mismo al decrementar el puntero.

ARITMETICA DE DEREFERENCIA

Debido a que los operadores * y ++ o -- tienen la misma precedencia y se evalúan de derecha a izquierda, y los paréntesis tienen mayor precedencia que ambos.

```
int *p , a[] = { 0 , 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 } ;
int var ;
p = a ;
```

El puntero está apuntando a a[0].

```
*p = 27 ;
```

Asignamos al elemento apuntado por p (a[0]) un valor constante.

```
var = *p ;
```

var sería asignada contenido de a[0]), y p seguiría apuntando al mismo elemento.

```
var = *( p + 1 ) ;
```

var se le asigna el contenido del elemento siguiente al apuntado por p o sea (a[1]). Lo interesante de remarcar acá es que p, en sí mismo, no varía luego de esta sentencia y continúa apuntando a a[0].

```
var = *( p++ );
```

A var se le asigna el valor de lo apuntado por p y luego incremente éste para que apunte al próximo elemento. Así en var quedaría 0 (valor de a[0]) y p apuntaría finalmente a a[1]. Si en vez de esto hubiéramos preincrementado a p tendríamos:

```
var = *( ++p ) ;
```

Se lee como: apunte con p al próximo elemento y asigne a var con el valor de éste. En este caso var sería igualada a 10 (a[1]) y p quedaría apuntando al mismo.

En las dos operaciones anteriores los paréntesis son superfluos ya que al analizarse los operadores de derecha a izquierda , daría lo mismo escribir :

```
var = *p++ ;    /* sintácticamente igual a var = *(p++) */
var = *++p ;    /*       "         "   " var = *(++p) */
```

PUNTEROS Y VARIABLES DINAMICAS

Las variables menos duraderas son de tipo local. En la función main() sus variables locales ocupan memoria durante toda la ejecución del programa. Si por ejemplo debemos recibir una serie de datos de entrada, digamos del tipo double y la cantidad de datos es indefinida, y queremos ver si aparece una ó más veces con el mismo valor, pondremos alguna limitación, grande digamos 5000 valores como máximo, debemos definir entonces un array de doubles capaz de albergar a cinco mil de ellos, por lo que el mismo ocupará del orden de los 40k de memoria.
Si definimos este array en main(), ese espacio de memoria permanecerá ocupado hasta el fín del programa, aunque luego de aplicarle el algoritmo de cálculo ya no lo necesitemos más. Una solución posible sería definirlo en una función llamada por main() que se ocupara de llenar el array con los datos, procesarlos y finalmente devolviera algún tipo de resultado, borrando con su retorno a la masiva variable de la memoria.
Sin embargo en C existe otra forma, los programas ejecutables creados con estos compiladores dividen la memoria disponible en varios segmentos, uno para el código (en lenguaje máquina), otro para albergar las variables globales, otro para el stack (a través del cual se pasan argumentos y donde residen las variables locales) y finalmente un último segmento llamado memoria de apilamiento ó amontonamiento (Heap). El Heap es la zona destinada a albergar a las variables dinámicas, es decir aquellas que crecen y decrecen a lo largo del programa.
Para crearlas definimos primero un puntero al tipo de la variable deseada:

```
double *p ;
```

Luego se reserva una cantidad dada de bytes en el Heap, para lo cual se efectúa una llamada a alguna de las funciones de librería, dedicadas al manejo del mismo. La más tradicional es malloc() (su nombre deriva de memory allocation), a esta función se le da como argumento la cantidad de bytes que se quiere reservar, y nos devuelve un pointer apuntando a la primer posición de la "pila" reservada . En caso que la función falle en su cometido (el Heap está lleno) devolverá un puntero inicializado con NULL.

p = malloc(8);

Acá hemos pedido 8 bytes (los necesarios para albergar un double) y hemos asignado a p el retorno de la función, es decir la dirección en el Heap de la memoria reservada.

Para no recordar el tamaño de las variables y para que el programa se ejecute independientemente del compilador podemos usar la función sizeof, para indicar la cantidad de bytes requerida:

p = malloc(sizeof(double)) ;

En caso de haber hecho previamente un uso intensivo del Heap, se debería averiguar si la reserva de lugar fué exitosa:

if(p == NULL)

rutina_de_error() ;

si no lo fue estas sentencias me derivan a la ejecución de una rutina de error que tomará cuenta de este caso. Por supuesto podría combinar ambas operaciones en una sola:

if((p = malloc(sizeof(double))) == NULL) {

printf("no hay mas lugar en el Heap ..... Socorro !!") ;

exit(1) ;}

se ha reemplazado aquí la rutina de error, por un mensaje y la terminación del programa, por medio de exit() retornando un código de error.

Si ahora quisiera guardar en el Heap el resultado de alguna operación, sería tan directo como,

*p = a * (b + 37) ;

y para recuperarlo , y asignárselo a otra variable bastaría con escribir:

var = *p ;

Es importante no olvidarnos que tenemos que explícitamente liberar la memoria, a través de invocar la función free() pasando como parámetro el puntero. Ej free(p);