TEMA 32

Lenguaje C: Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.

Índice

1. INTRODUCCIÓN

Los tipos de datos tienen gran importancia en C, ya que toda variable es declarada con un tipo de datos determinado.

Los tipos de datos pueden ser según su complejidad:

- **Simples**. Son tipos de datos básicos.
- **Complejos**. Son los que se crean a partir otros tipos de datos.

Una estructura de datos es ”un conjunto de datos homogéneos que se tratan como una sola unidad” [ALGL93].

Un tipo estructurado define una agrupación de componentes de un tipo más simple, podemos clasificarlos en:

- **homogéneos**: Tablas o arrays.
- **heterogéneos**: Estructuras o struct.

Las estructuras de datos pueden ser:

- **Estáticas**

  Tienen un número fijo de elementos que queda determinado desde la declaración de la estructura.

  El espacio ocupado en memoria se define en tiempo de compilación y no puede ser modificado durante la ejecución del programa.

  Corresponden a este tipo los arrays y registros

- **Dinámicas**. Tiene un número de elementos que varía a lo largo de la ejecución del programa. El espacio ocupado en memoria puede ser modificado en tiempo de ejecución. Corresponden a este tipo las listas, árboles y grafos . Estas estructuras no son soportadas en todos los lenguajes.

La elección de la estructura de datos idónea dependerá de la naturaleza del problema a resolver y, en menor medida, del lenguaje. Las estructuras de datos tienen en común que un identificador, nombre, puede representar a múltiples datos individuales.

Según donde resida los datos las estructuras se pueden clasificar en:

- **Internos**. Los datos residen en memoria central de la computadora.
- **Externos**. Los datos residen en un soporte externo.

Ver la figura 1 de la página 213

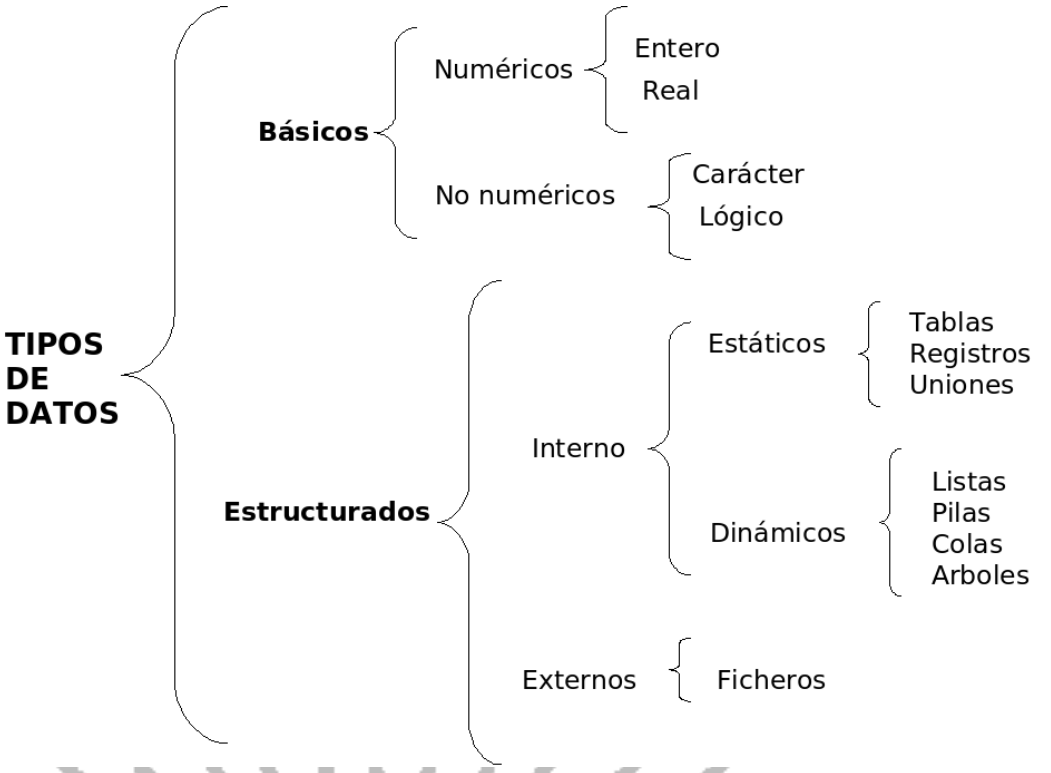


Figura 1: Clasificación general de los tipos de datos

2. ENTRADA SALIDA DE DATOS.

En C no existe ninguna palabra clave para realizar la entrada y salida de datos (E/S). Se realizan a través de funciones de biblioteca (concretamente, la biblioteca stdio.h.

2.1. E/S por consola

Las funciones principales que realizan la entrada y salida sin formato son:

- **getchar():** Lee un carácter del teclado. Espera hasta que se pulsa una tecla y entonces devuelve su valor.
- **putchar():** Imprime un carácter en la pantalla en la posición actual del cursor.
- **gets():** Lee una cadena de caracteres introducida por el teclado y la sitúa en una dirección apuntada por su argumento de tipo puntero a carácter.
- **puts():** Escribe su argumento de tipo cadena en la pantalla seguida de un carácter de salto de línea.

El siguiente fragmento de código lee un carácter del teclado y lo muestra por pantalla. A continuación lee una cadena (de 10 caracteres incluido el carácter nulo) y también la muestra por pantalla:

```
#include <stdio.h>
main()
{
char cadena[10];
int i;
i=getchar();
putchar(i);
gets(cadena);
puts(cadena);
}
```

Las funciones principales que realizan la entrada y salida con formato, es decir, se pueden leer y escribir en distintas formas controladas, son:

- **printf():** Escribe datos en la consola con el formato especificado.
- **scanf():** Función de entrada por consola con el formato especificado.

En la función printf() (con scanf() no), entre las comillas se pueden poner rótulos literales mezclados con los caracteres de transmisión.

Los caracteres de transmisión son precedidos de un % para distinguirlos de los normales: Caracteres de transmisión Argumento que transmite

CARÁCTER	DESCRIPCIÓN
%c Int	Un carácter simple
%Ns char *	una cadena de caracteres
%Nd %Ni int	un número decimal
%o int	octal sin signo
%x %X int	hexadecimal sin signo
%Nu int	decimal sin signo
%N.Df	Float o double con D decimales, en notación fija
%N.De %N.DE	float o double con D decimales, en notación científica
%N.Dg %N.DG	float o double en notación científica si el exponente es menor de diez a la menos cuatro, o fija en caso contrario.
%p void *	escribe el número que corresponde al puntero
\\	Escribe un signo de %

Donde aparecen las letras N.D o no se pone nada o serán en realidad dos números que dicen que la transmisión total del valor al menos ocupará N posiciones (si el número necesita más de N las tomará, si usa menos las dejara en blancos, a menos que se quiera rellenar con ceros, entonces se pone 0N) y que la parte decimal tendrá como máximo las D posiciones después de un punto.

Normalmente el número se ajusta por la derecha para el campo de N posiciones que le hemos dicho que utilice; si deseamos el ajuste por la izquierda,

se añade un signo menos precediendo al valor N (-N). Una l precediendo a N (p.e. %l5d) significa que transmitiremos un long int : si, por el contrario, es una h significa que transmitiremos un short int.

Existe otro tipo de carácter especial, los caracteres de escape, que tienen un significado especial. Los caracteres de escape son los siguientes:

`\n`	Nueva línea
`\t`	Tabulador
`\b`	Espacio atrás
`\xdd`	Código ASCII en notación hexadecimal (cada d representa un dígito)
`\ddd`	Código ASCII en notación octal (cada d representa un dígito)

La lista de argumentos estará separada por comas. Debe existir una correspondencia biyectiva entre los caracteres de transmisión (aquellos que comienzan con un %) y la lista de argumentos a transmitir. Cabe destacar una diferencia en la lista de argumentos entre las funciones printf() y scanf(). En está ultima función (scanf()), la lista de argumentos va precedida por el operador de dirección(&), puesto que scanf() requiere que los argumentos sean las direcciones de las variables, en lugar de ellas mismas.

(&), puesto que scanf() requiere que los argumentos sean las direcciones de las variables, en lugar de ellas mismas.

2.2. E/S por archivos

En C un archivo puede ser cualquier cosa, desde un archivo de disco a un terminal o una impresora. Se asocia una secuencia con un archivo específico realizando una operación de apertura, una vez que está abierto, la información puede ser intercambiada entre éste y el programa. El puntero a un archivo es el hilo que unifica el sistema de E/S con buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia, el puntero a un archivo identifica un archivo en disco específico y utiliza la secuencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Para obtener una variable de tipo puntero a archivo se debe utilizar una sentencia como la siguiente:

FILE *punt;

La función fopen() abre una secuencia para que pueda ser utilizada y le asocia a un archivo.

Su prototipo es:

FILE *fopen(const char *nombre_archivo, const char *modo);

Donde nombre_archivo es un puntero a una cadena de caracteres que representan un nombre válido del archivo y puede incluir una especificación de directorio. La cadena que apunta modo determina cómo se abre el archivo.

Los modos son los siguientes:

r	Abre un archivo de texto para lectura.
w	Crea un archivo de texto par escritura
a	Abre un archivo de texto para añadir
r+	Abre un archivo de texto para lectura escritura
w+	Crea un archivo de texto para lectura escritura
a+	Añade o crea un archivo de texto para lectura escritura

Funciones más importantes:

- **fclose()** : cierra una secuencia que fue abierta mediante una llamada a fopen(). Escribe toda la información que todavía se encuentre en el buffer del disco y realiza un cierre formal del archivo a nivel del sistema operativo. También libera el bloque de control de archivo asociado con la secuencia, dejándolo libre para su reutilización. A veces es necesario cerrar algún archivo para poder abrir otro, debido a la existencia de un límite del sistema operativo en cuanto al número de archivos abiertos. Su prototipo es: int fclose(FILE *fp);
- **putc()** escribe caracteres en un archivo que haya sido abierto previamente para operaciones de escritura, utilizando la función fopen(). Su prototipo es: int putc(int car, FILE *pf);
- **getc()** escribe caracteres en un archivo que haya sido abierto, en modo lectura, mediante fopen(). Su prototipo es: int getc(FILE *pf);La función fputs() escribe la cadena en la secuencia especificada. Su prototipo es: fputs() escribe la cadena en la secuencia especificada. Su prototipo es: int fputs(const char *cad, FILE *pf);La función fgets() lee una cadena de la secuencia especificada hasta que se lee un carácter de salto de línea o hasta que se han leído longitud-1 caracteres.
- **rewind()** inicia el indicador de posición al principio del archivo indicado por su argumento. Su prototipo es: rewind() inicia el indicador de posición al principio del archivo indicado por su argumento.

  Su prototipo es:

  void rewind(FILE *pf);

Existen otras muchas funciones en la biblioteca estándar de C como pueden ser

- **remove():** Borra el archivo especificado.
- **fflush():** Vacía el contenido de una secuencia de salida.
- **fread()**: Lee tipos de datos que ocupan más de un byte. Permiten la lectura de bloques de cualquier tipo de datos.
- **fwrite():** Escribe tipos de datos que ocupan más de un byte. Permiten la escritura de bloques de cualquier tipo de datos.
- **fprintf():** Hace las funciones de printf() sobre un fichero.
- **fscanf():** Hace las funciones de scanf() sobre un fichero.
- **feof():** Detecta el final de un fichero.
- **ferror():** Detecta un error en la lectura/escritura de un fichero.
- **fclose():** Cierra una secuencia que fue abierta mediante una llamada a fopen().
- **putc()**. Eescribe caracteres en un archivo que haya sido abierto previamente para operaciones de escritura, utilizando la función fopen().
- **getc()**: Escribe caracteres en un archivo que haya sido abierto, en modo lectura, mediante fopen(). Su prototipo es:
- **fputs()**: Escribe la cadena en la secuencia especificada. Su prototipo es: int fputs(const char *cad, FILE *pf);
- **fgets()**: Lee una cadena de la secuencia especificada hasta que se lee un carácter de salto de línea o hasta que se han leído longitud-1 caracteres. Su prototipo es: fgets() lee una cadena de la secuencia especificada hasta que se lee un carácter de salto de línea o hasta que se han leído longitud-1 caracteres.

  Su prototipo es:

  int fgets(char *cad, FILE *pf);

3. ESTRUCTURADOS DE DATOS ESTÁTICAS

3.1. Arrays o Tablas

La declaración de un array especifica el nombre del array, el número de elementos del mismo y el tipo de éstos.

Arrays unidimensionales. La declaración de un array de una dimensión, se hace de la forma:

tipo nombre [tamaño];

tipo nombre [ ];

Siendo:

- **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.

Ejemplo:

```
int lista [100];
```

Arrays multidimensionales. La declaración de un array de varias dimensiones se hace de la forma:

```
tipo nombre [expr-cte][expr-cte]
...;
tipo nombre [][expr-cte] ...;
```

La primera .exprpr-cte” puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando hace referencia a un array declarado en otra parte del programa.

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

Para dimensionar un array se pueden emplear constantes o expresiones a base de constantes de cualquier tipo entero.

Para acceder a un elemento de un array, se hace mediante el nombre del array seguido de uno o más subíndices, dependiendo de las dimensiones del mismo, cada uno de ellos encerrado entre corchetes. Un subíndice puede ser una constante, una variable o una expresión cualquiera.

Ejemplo:

```
int tabla [20] [80];
tabla [2] [3] = {{0, 1, 2}, {5, 4, 7}};
```

3.2. Cadenas de caracteres

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

```
char cadena [longitud];
```

Un array de caracteres puede ser inicializado asignándole un literal. Por ejemplo:

```
char cadena [] = "abcd; "
```

Este ejemplo inicializa el array de caracteres ”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 de caracteres y la cadena asignada es más larga que el tamaño especificado, se obtiene un error en el momento de la compilación. Por ejemplo:

```
char cadena [3] = "abcd";
```

Funciones más importantes:

- Operador **sizeof(cadena)**. Este operador da como resultado el tamaño en bytes de su operando. Cuando dicho operando es un array, el resultado es el tamaño total del array.
- Función **gets()**. Leer una cadena de caracteres. La función ”gets” lee una línea de la entrada estándar, ”stdin”, y la almacena en la variable especificada. Esta variable es un puntero a la cadena de caracteres leída.
- 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).

Funciones para manipular cadenas de caracteres. Las declaraciones de las funciones, para manipular cadenas de caracteres, que a continuación se describen, están en el fichero a incluir string.h. La sentencia:

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

Se requiere solamente para declaraciones de función.

Algunas funciones para manipular cadenas de caracteres son:

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

- **strchr** (cadena, 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').
- **strcmp** (cadena1, 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 a la cadena2 y

  > 0 si la cadena1 es mayor que la cadena2.
- **strcpy** (cadena1, cadena2). Esta función copia la cadena2, incluyendo el carácter de terminación nulo, en la cadena1 y devuelve un puntero a cadena1.
- **strlen** (cadena). Esta función devuelve la longitud en bytes de cadena, no incluyendo el carácter de terminación nulo
- **strncat** (cadena1, cadena2, n). Esta función añade 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.
- **strcmp** (cadena1, cadena2, n). Esta función compara los primeros n caracteres de cadena1 y cadena2 distinguiendo mayúsculas y minúsculas, y devuelve un valor:

  < 0 si la cadena1 es menor que la cadena2

  $= 0$ si la cadena1 es igual a la cadena2 y

  > 0 si la cadena1 es mayor que la cadena2.

  Si n es mayor que la longitud de la cadena1, se toma como valor la longitud de la cadena1.
- **strncpy** (cadena1, cadena2, n). Esta función copia n caracteres de la cadena2, en la cadena1 (sobrescribiendo los caracteres de cadenas) 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 cadena1 es rellenada con caracteres nulo ('\0') hasta la longitud n.
- **strstr** (cadena1, cadena2). Esta función devuelve un puntero a la primera ocurrencia de cadena2 en cadena1 o un valor NULL si la cadena2 no se encuentra en la cadena1.

  char *strstr(char *cadena1, char *cadena2);
- **strlwr** (cadena). Convierte las letras mayúsculas de cadena, en minúsculas. El resultado es la propia cadena en minúsculas.
- **strupr** (cadena). Convierte las letras minúsculas de cadena, en mayúsculas. El resultado es la propia cadena en mayúsculas.

3.3. Estructuras

Una estructura es una agrupación de datos bajo un nombre común. Es un nuevo tipo de datos que puede ser manipulado de la misma forma que los tipos predefinidos como ”float, int, char,” entre otros. Una estructura se puede definir como una colección de datos de diferentes tipos, lógicamente relacionados. En C una estructura sólo puede contener declaraciones de variables. En otros compiladores, este tipo de construcciones son conocidas como ”registros”.

Crear una estructura es definir un nuevo tipo de datos, denominado tipo estructura y 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[, variable] ... ];
```

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

variable.miembro

Ejemplo:

```
struct datos {
  char nombre [42];
 char domicilio [60];
};
```

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.

3.4. Uniones

Una unión es una variable que puede contener miembros de diferentes tipos, en una misma zona de memoria.

La declaración de una unión tiene la misma forma que la declaración de una estructura, excepto que en lugar de la palabra reservada ”struct” se pone la palabra reservada ”unión”. Todo lo expuesto para las estructuras, es aplicable a las uniones, excepto la forma de almacenamiento de sus miembros.

```
union tipo_union
{
 declaraciones de los miembros
};
```

tipo_union es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo unión, podemos declarar una o más variables de ese tipo de la forma:

```
union tipo_union [variable[, variable] ... ];
```

Para referirse a un determinado miembro de la unión, se utiliza la notación:

```
variable.miembro
```

Para almacenar los miembros de una unión, se requiere una zona de memoria igual a la que ocupa el miembro más largo de la unión. Todos los miembros son almacenados en el mismo espacio de memoria y comienzan en la misma dirección. El valor almacenado es sobrescrito cada vez que se asigna un valor al mismo miembro o a un miembro diferente.

Las uniones ofrecen una buena solución cuando se presentan datos alternativos en una misma estructura. Para ello podemos elegir dos estructuras distintas o una que englobe a ambas (union).

Ejemplo:

```
union   {
        struct  {
                int a_servicio;
                long n_nomina;
                }
        struct  {
                char f_contrato [10];
                int t_contrato;
        }
}
```

4. GESTIÓN DE PUNTEROS

4.1. Definición y uso

Un puntero es una dirección de memoria, es decir, la posición donde se guarda un dato en la memoria del ordenador. Puede verse como una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato. Quiere esto 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, incluyendo

estructuras, funciones etc. Los punteros se pueden utilizar para crear y manipular estructuras de datos, para asignar memoria dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

Un puntero se declara igual que una variable pero anteponiendo el operador de indirección (*) al identificador del puntero, el cual significa ”puntero a”. Un puntero siempre apunta a un objeto de un tipo particular. Un puntero no inicializado tiene un valor desconocido.

tipo * var-puntero;

Siendo

- **tipo.** especifica el tipo del objeto apuntado; puede ser cualquier tipo, incluyendo tipos agregados.
- **var-puntero.** nombre de la variable puntero.

Recordemos que el operador * devuelve el dato contenido en una dirección de memoria y que el operador & devuelve la dirección de memoria donde está guardado un dato

Ejemplo:

```
int *pint; /* pint es un puntero a un entero */
char *pnom; /* pnom es un puntero a una cadena de caracteres */
double *p, *q; /* p y q son punteros a reales */
```

El espacio de memoria requerido para un puntero, es el número de bytes necesarios para especificar una dirección máquina. En la familia de micros 8086, una dirección ”near” (dirección con respecto a la dirección base del segmento) necesita 16 bits y una dirección ”far” (dirección segmentada) necesita 32 bits.

4.2. Punteros y arrays.

En C existe, entre punteros y arrays, una relación tal que, cualquier operación que se pueda realizar mediante la indexación de array, se puede realizar también con punteros.

Para clarificar lo expuesto, observar el siguiente programa, realizado primeramente con arrays y a continuación con punteros:

```
/* Escribir los valores de un array. Versión con arrays */

#include <stdio.h>
main()
{
static int lista[] = {24, 30, 15, 45, 34};
int ind;
for (ind = 0; ind<5; ind++)
  printf("\%d", lista[ind]);
}
```

En este ejemplo, la notación utilizada para acceder a los elementos del array estático, es la expresión: lista[ind].

A continuación se expone la versión con punteros:

```
/* Escribir los valores de un array. Versión con punteros */
\include <stdio.h>
main()
{
static int lista$[] = {24, 30, 15, 45, 34};
int ind;
for (ind = 0; ind<5; ind++)
printf("\%d", *(lista+ind));
}
```

Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos del array es: *(lista+ind).

Esto deja constancia de que ”lista” es la dirección de comienzo del array. Si a esta dirección le sumamos 1, o dicho de otra manera si ”ind” vale 1, nos situamos en el siguiente entero de la lista; esto es *(lista+1) y lista[1] representan el mismo valor. Un incremento de uno sobre una dirección equivale a avanzar no un byte, sino un número de bytes igual al tamaño de los objetos que estamos tratando.

Según esto, hay dos formas de referirnos al contenido de un elemento de un array:

```
*(array+indice) o array[indice]
```

Puesto que *(lista+0) es igual que lista[0], la asignación: p = &lista[0] es la misma que la asignación p = lista, donde p es una variable de tipo puntero. Esto indica que la dirección de comienzo de un array es la misma que la del primer elemento. Por otra parte, después de haber efectuado la asignación p = lista, las siguientes expresiones dan lugar a idénticos resultados:

p[ind], *(p+ind), lista[ind], *(lista+ind)

Sin embargo, hay una diferencia entre el nombre de un array y un puntero. El nombre de un array es una constante y un puntero es una variable. Esto quiere decir que las operaciones:

```
lista = p o lista++   no son correctas, y las operaciones
p = lista o p++       sí son correctas.
```

4.3. Arrays de punteros.

Se puede definir un array, para que sus elementos contengan en lugar de un dato, una dirección o puntero.

Ejemplo:

```
int *a[10], v;
```

```
a[0] = &v;
printf("\%d", *a[0]);
```

Este ejemplo define un array de 10 elementos que son punteros a datos de tipo "int" y una variable entera "v". A continuación asigna al elemento a[0], la dirección de "v" y escribe su contenido.

Cuando cada elemento de un array es un puntero a otro array, estamos en el caso de una doble indirección o "punteros a punteros". La característica de poder referenciar un puntero con otro puntero, le da al lenguaje C una gran potencia y flexibilidad para crear estructuras complejas.

Para especificar que una variable es un puntero a un puntero, se procede de la forma siguiente:

tipo **variable;

Ejemplo:

```
int **pp;        /* pp es un puntero a un puntero */
```

Para escribir un programa utilizando la notación de punteros en lugar de la notación array, nos planteamos únicamente la cuestión de cómo escribir la expresión "tabla[f][c]", utilizando la notación de punteros. Pues bien, pensemos que un array de dos dimensiones es un array de una dimensión, donde cada elemento es a su vez un array de una dimensión.

Si elegimos una fila, por ejemplo "tabla[1]", o en notación puntero "tabla+1", interpretamos esta expresión como un puntero a un array de 5 elementos; esto quiere decir que "tabla+1" es un puntero a un puntero, o que el contenido de "tabla+1", *(tabla+l), es la dirección del primer elemento de esa fila, "tabla[1][0]", o en notación puntero "*(tabla+1)+0". Las direcciones "tabla+1" y "*(tabla+1) coinciden, pero ambas expresiones tienen diferente significado. Por ejemplo:

tabla+1+2 se refiere a la fila tabla[3] y

*(tabla+1) +2 se refiere al elemento tabla[1][2]

((tabla+1) +2) es el contenido del elemento tabla[1][2].

5. PUNTEROS A FUNCIONES

Igual que sucedía con los arrays, el nombre de una función representa la dirección de comienzo de la función; por lo tanto, puede ser utilizado para pasarlo a funciones, colocarlo en arrays, etc.

Para declarar un puntero a una función se procede así:

```
tipo (*p_identif)();
```

- tipo es el tipo del resultado devuelto por la función.

- p_identif identifica a una variable de tipo puntero. Esta variable recibirá un puntero a una función, dado por el propio nombre de la función.

En el siguiente ejemplo, se define un puntero ”p” a una función. A continuación, se asigna a ”p” la dirección de la función ”escribir” y se llama a la función mediante la sentencia: (*p) (5);

```
#include <stdio.h>
void escribir(int);
main()
{
void (*p)(int); /*p es un puntero a una función */
p = escribir;   /*p = dirección de la función */
(*p) (5);              /* llamada a la función */
void escribir(int a)   /* función escribir */
printf("\%d\n",a);
}
```

El nombre de una función representa la ”dirección”de comienzo de la misma.

6. ESTRUCTURAS DE DATOS DINÁMICAS

6.1. Definición y uso

La propiedad característica de las estructuras dinámicas es la facultad que tienen para variar su tamaño y hay muchos problemas que requieren de este tipo de estructuras. Esta propiedad las distingue claramente de las estructuras estáticas fundamentales (arrays y estructuras). Por tanto, no es posible asignar una cantidad fija de memoria para una estructura dinámica, y como consecuencia un compilador no puede asociar direcciones explícitas con las componentes de tales estructuras. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una asignación dinámica de memoria; es decir, asignación de memoria para las componentes individuales, al tiempo que son creadas durante la ejecución del programa, en vez de hacer la asignación durante la compilación del mismo.

6.2. Asignación dinámica de memoria.

Cuando se asigna memoria dinámicamente para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Para realizar esta operación disponemos en C de la función malloc().

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

p = malloc(t);
```

Esta función asigna un bloque de memoria de t bytes y devuelve un puntero que referencia el espacio asignado. Si hay insuficiente espacio de memoria o si t es 0, la función retorna a un puntero nulo (NULL).

Para liberar un bloque de memoria asignado por la función "malloc()", utilizaremos la función free().

```
\include <stdlib.h> o <malloc.h>

free(void *p);
```

6.3. Listas lineales

Si deseamos una lista de elementos u objetos de cualquier tipo, originalmente vacía, que durante la ejecución del programa vaya creciendo y decreciendo elemento a elemento, según las necesidades previstas en el programa, entonces tenemos que construir una lista lineal en la que cada elemento apunte o direccione el siguiente. Por este motivo, una lista lineal se la denomina también lista enlazada.

Para construir una lista lineal primero tendremos que definir la clase de objetos que van a formar parte de la misma. De una forma genérica el tipo definido será de la forma:

```
typedef struct id tipo_objeto;
struct id
 {
 /* declaración de los miembros de la estructura */
 tipo_objeto *siguiente;
 };
```

Ejemplo:

```
typedef struct datos elemento;
struct datos
{
 int dato;
 elemento *siguiente;
}
```

Ver la figura 2 de la página 227

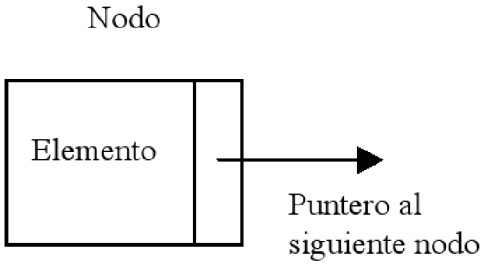


Figura 2: Ejemplo de nodo de la lista

6.4. Pilas.

Una pila es una lista lineal en la que todas las inserciones y supresiones (y normalmente todos los accesos), se hacen en un extremo de la lista. Un ejemplo de esta estructura es una pila de platos. En ella, el añadir o quitar platos se hace siempre por la parte superior de la pila. Este tipo de listas reciben también el nombre de listas LIFO (Last In First Out - último en entrar, primero en salir).

6.5. Árboles

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas. En un árbol existe un nodo especial denominado raíz. Un nodo del que sale alguna rama, recibe el nombre de nodo de bifurcación o nodo rama y un nodo que no tiene ramas recibe el nombre de nodo terminal o nodo hoja.

6.5.1. Árboles binarios.

Un árbol binario es un conjunto finito de nodos que consta de un nodo raíz que tiene dos subárboles binarios denominados subárbol izquierdo y subárbol derecho. Evidentemente, la definición dada es una definición recursiva, es decir, cada subárbol es un árbol binario.

Esta definición de árbol binario, sugiere una forma natural de representar árboles binarios en un ordenador: debemos tener dos enlaces (izdo. y dcho.) en cada nodo, y una variable de enlace raíz que nos direcciona el árbol. Esto es:

```
typedef struct datos nodo;
struct datos
{
  /* declaración de miembros */
  nodo *izdo;
  nodo *dcho;
};
```

Ver la figura 3 de la página 228

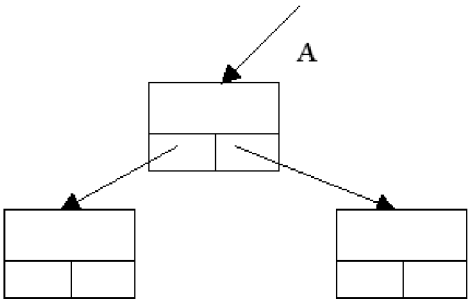


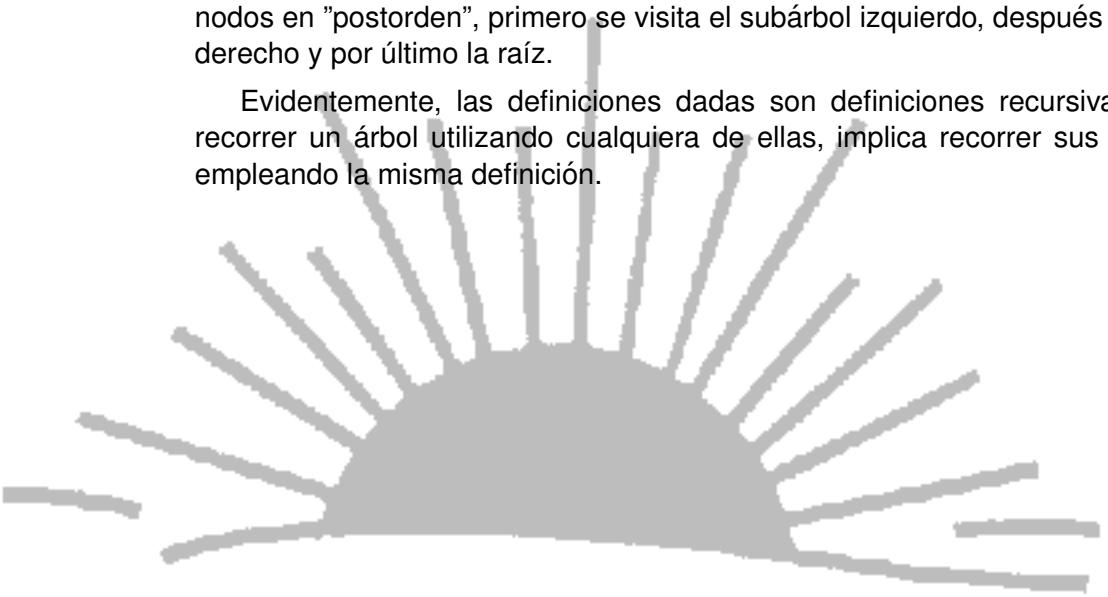
Figura 3: Ejemplo de nodo del arbol binario

Si el árbol está vacío, ”raíz” es igual a NULL; en caso contrario, ”raíz” es un puntero que direcciona la raíz del árbol, e ”izdo” y ”dcho” son punteros que direccionan los subárboles izquierdo y derecho de la raíz, respectivamente.

Hay varios algoritmos para el manejo de estructuras en árbol. Una idea que aparece repetidamente en estos algoritmos es la noción de recorrido de un árbol. Este es un método para examinar sistemáticamente los nodos de un árbol, de forma que cada nodo sea visitado solamente una vez.

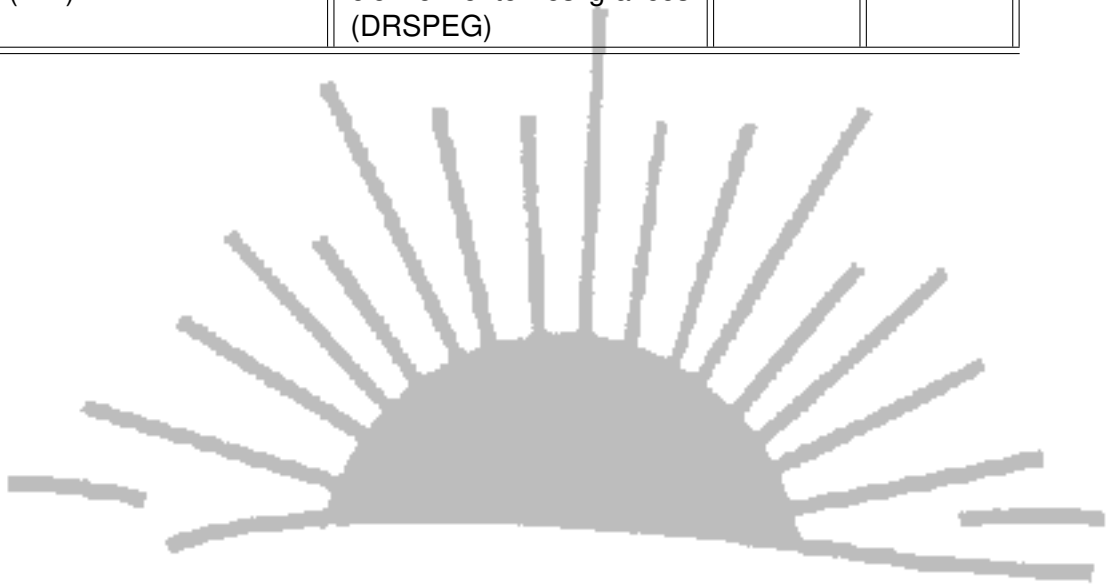
Pueden utilizarse tres formas principales para recorrer un árbol binario: ”preorden”, ”inorden” y ”postorden”. Cuando se visitan los nodos en ”preorden”, primero se visita la raíz, después el subárbol izquierdo y por último el subárbol derecho. Cuando se visitan los nodos en ”inorden”, primero se visita el subárbol izquierdo, después la raíz y por último el subárbol derecho. Cuando se visitan los nodos en ”postorden”, primero se visita el subárbol izquierdo, después el subárbol derecho y por último la raíz.

Evidentemente, las definiciones dadas son definiciones recursivas, ya que, recorrer un árbol utilizando cualquiera de ellas, implica recorrer sus subárboles empleando la misma definición.



7. CONTEXTUALIZACION

CICLO FORMATIVO	MODULO	CURSO	CUERPO
Administración de sistemas informáticos (ASI)	Fundamentos de Programación (FPR)	1	PS
Desarrollo de Aplicaciones informáticas (DAI)	Programación en Lenguajes Estructurados (PLE)	1	PS
Desarrollo de Aplicaciones informáticas (DAI)	Diseño y realización de servicios de presentación en entornos gráficos (DRSPEG)	2	PT



BIBLIOGRAFIA

[AHU98] AHO, HOPCROFT, y ULLMAN. *Estructuras de datos y algoritmos*. Addison Wesley, 1998.

[ALGL93] EDUARDO ALCALDE LANCHARRO y MIGUEL GARCIA LOPEZ. *Metodología de la programación*. Mcgraw-hill, 2 edition, 1993. ISBN:84-7615-913-7.

[BB97] G. BRASSARD y P. BRATLEY. *Fundamentos de Algoritmia*. Prentice Hall, 1997. Algoritmica, Costes.

[CC05] MARIA ASUNCION CRIADO CLAVERO. *Programacion de Lenguajes Estructurados*. Ra-ma, 2005. ISBN: 84-7897-682-5. CICLOS. Precio: 34,90 Euros.

[GOT97] BYRON GOTTFRIEND. *Programación en C*. Mcgraw-hill, 1997. ISBN:84-481-1068-4. Teoria.

[HSR96] E. HOROWITH, S. SAHNI, y S. RAJASEKARAN. *Computer Algoritms/C++*. Computer Science Press, 1996. Algoritmica, Costes.

[JA03] LUIS JOYANES AGUILAR. *Fundamentos de programación. Algoritmos, Estructura de datos y Objetos*. Mcgraw-hill, 2003. ISBN: 84-481-3664-3. Libro de consulta profesor. Metodología.

[JACS02] LUIS JOYANES AGUILAR y ANDRES CASTILLO SANZ. *Programación en C. Libro de Problemas.* Mcgraw-hill, 2002. ISBN: 84-481-3622-5. Sólo hay problemas.

[KR98] BRIAN W. KERNIGHAN y DENNIS M. RITCHIE. *El Lenguaje de Programacion C.* Prentice-Hall Hispanoamericana, 2 edition, 1998. ISBN 968-880-205-0. ANSI C.

[MOL01] FRANCISCO JAVIER MOLDES. *Guia práctica para usuarios. Lenguaje C*. Anaya, 2001. Teoria.

[PL94] J.M. PÉREZ LOBATO. *Metodología de la programación*. Alhambra-Longman, 1994.

[QC99] ENRIQUE QUERO CATALINAS. *Programación en Lenguaje C.* Paraninfo, 1999. ISBN:84-283-2489-1. Libro sólo de ejercicios.

[QC03] ENRIQUE QUERO CATALINAS. *Programación en lenguajes estructurados*. Paraninfo, 2003. ISBN: 8497320034. CICLOS. LIBRO ALUMNO. MUY BUENO. 28.75EUR.

[QC04] ENRIQUE QUERO CATALINAS. *Fundamentos de programación*. Paraninfo, 2004. ISBN:849732000X. 26EUR. CICLOS. LIBRO ALUMNO. MUY BUENO.

[SC96] M. ANGELES SÁNCHEZ y FELIX CHAMORRO. *Programación Estructurada y Fundamentos de Programación*. Mcgraw-hill, 1996. ISBN: 84-481-0557-5. Teoria. Alumnos Ciclos formativos.

[SCH94] HERBERT SCHILDT. *C. Guia de autoenseñanza.* Mcgraw-hill, 1994. ISBN: 84-481-3204-1 Precio: 38,95 EUR.

[SCH01] HERBERT SCHILDT. *Manual de referencia de C*. Mcgraw-hill, 2001. Consulta.

[SPC05] M. SANTOS, I. PATIÑO, y R CARRASCO. *Fundamentos de programación.* Ra-ma, 2005. ISBN: 8478976787. CICLOS. Precio: 19 Euros.

[UL97] ALFONSO UREÑA LOPEZ. *Fundamentos de Informática*. Ra-ma, 1997.