

# **Administración de Sistemas Informáticos**

## **Fundamentos de Programación**

**IES GRAN CAPITAN**

**Profesor: José Ramón Albendín Ramírez**

**ÍNDICE****METODOLOGÍA DE LA PROGRAMACIÓN 8**

Introducción.....	8
Conceptos de Algoritmo y Programa.....	8
Pasos para la resolución de un problema .....	9
Datos: tipos y características .....	10
Tipos de datos .....	11
Características de las variables .....	12
Variables simples y compuestas.....	12
Operadores y Expresiones.....	13
Lenguaje e instrucciones.....	15
Evolución y clasificación de los lenguajes de programación.....	15
Métodos de expresión de un algoritmo .....	16

**PROGRAMACIÓN ESTRUCTURADA 19**

Tipos de instrucciones.....	19
1.- Instrucción de declaración de datos .....	21
2.- Instrucción de Entrada.....	21
3.- Instrucción de salida .....	21
4.- Instrucción de asignación .....	22
5.- Instrucción compuesta.....	22
6.- Instrucción selectiva simple .....	22
7.- Instrucción selectiva compuesta. ....	23
8.- Instrucción selectiva múltiple. ....	24
9.- Instrucciones de control repetitivas.....	25
10.- Instrucción de salto.....	27
Elementos y características de un programa .....	28
Variables auxiliares.....	28
Características de un programa.....	29

**PROGRAMACIÓN MODULAR 31**

Introducción.....	31
Clasificación de los módulos .....	32
En función de su situación con respecto al módulo que lo invoca .....	32
Atendiendo al posible retorno de un valor.....	32
En función de cuando ha sido desarrollado .....	32
En función del número de módulos distintos que realizan la llamada.....	32
Ámbito de las variables .....	32
Funciones y Procedimientos .....	33
Declaración de una función de usuario.....	33
Parámetros .....	33
Invocación de una función .....	34
Paso de parámetros a una función .....	34
Recursividad .....	34
Implementación de la recursividad.....	35
Recursividad y variables locales.....	36
Tipos de recursividad.....	36

**ESTRUCTURAS ESTÁTICAS INTERNAS DE DATOS: TABLAS 37**

Conceptos Básicos.....	37
Clasificación de Tablas .....	37
Tablas Unidimensionales.....	37
Tablas bidimensionales .....	38

Tablas multidimensionales.....	39
Operaciones con Tablas .....	39
Declaración de una tabla .....	39
Recorrido o acceso secuencial .....	39
Carga de datos .....	40
Lectura.....	41
Escritura.....	41
Operaciones de búsqueda y ordenación en tablas .....	41
Búsqueda secuencial o lineal un vector desordenado .....	42
Búsqueda secuencial o lineal en un vector ordenado.....	42
Búsqueda binaria o dicotómica en un vector ordenado .....	42
Ordenación de un vector.....	43
<b>ESTRUCTURAS ESTÁTICAS EXTERNAS DE DATOS: FICHEROS</b>	<b>44</b>
Conceptos y definiciones .....	44
Clasificación de registros .....	45
Registros de longitud fija.....	45
Registros de longitud variable.....	45
Operaciones con registros .....	45
Clasificación de ficheros.....	46
Permanentes .....	46
Temporales.....	46
Operaciones con ficheros.....	46
Organización y acceso .....	47
Ficheros de organización secuencial .....	48
Ficheros de organización relativa .....	49
Organización directa .....	49
Organización aleatoria o indirecta.....	50
Variantes de la organización secuencial .....	51
Organización secuencial encadenada .....	51
Organización secuencial indexada .....	51
Organización secuencial indexada-encadenada.....	52
Tratamiento de ficheros secuenciales.....	53
Definición del fichero.....	53
Asignación física.....	53
Apertura del fichero.....	53
Cierre del fichero.....	54
Escritura en el fichero .....	54
Lectura del fichero .....	54
Consultas.....	54
Partición de un fichero .....	56
Rupturas de control.....	56
Actualización.....	56
<b>ESTRUCTURAS DINÁMICAS INTERNAS DE DATOS</b>	<b>58</b>
Punteros.....	58
Operaciones básicas con punteros.....	58
Listas.....	59
Listas enlazadas o encadenadas.....	59
Listas doblemente enlazadas.....	61
Listas circulares .....	61
Pilas.....	62

Colas.....	62
Estructuras dinámicas no lineales .....	63
Árboles.....	63
<b>LENGUAJE C 67</b>	
Introducción.....	67
Características del lenguaje C .....	67
Identificadores.....	67
Tipos de datos básicos.....	68
Tipo entero.....	68
Tipo de dato real.....	68
Tipo de dato carácter.....	69
Tipo de dato vacío .....	69
Modificadores de tipo básico .....	69
Modificador short .....	70
Modificador long.....	70
Modificados signed .....	70
Modificador unsigned.....	70
Declaración de variables .....	70
Operadores .....	71
Indicadores de expresión.....	71
Operadores aritméticos.....	71
Operadores relacionales y lógicos .....	72
Operadores para el tratamiento de bits.....	73
Operadores de asignación.....	73
Operador coma.....	73
Operador de tamaño.....	73
Operador de molde.....	74
Operador condicional.....	74
Orden de prioridad de los operadores .....	74
Expresiones .....	75
<b>ESTRUCTURA DE UN PROGRAMA EN C 76</b>	
Introducción.....	76
Directivas del preprocesador .....	77
Declaraciones globales.....	77
Función main().....	77
Funciones definidas por el usuario .....	78
Comentarios .....	78
Creación de un programa .....	78
Depuración de un programa en C.....	78
Instrucciones de declaración.....	80
Instrucciones de asignación .....	81
Instrucciones de entrada y salida .....	81
Función de entrada (teclado) .....	81
Funciones de salida (pantalla) .....	83
Estructuras de control .....	85
Alternativa simple.....	85
Alternativa doble .....	85
Bloque de sentencias if ... else-if.....	86
Sentencia if ... else abreviada con el operador ? :.....	86
Alternativa múltiple. Sentencia switch() .....	86

Estructura repetitiva while .....	87
Estructura repetitiva do-while .....	88
Estructura repetitiva for .....	88
Sentencia break .....	88
Sentencia continue .....	88
Sentencia goto .....	89
<b>FUNCIONES EN C      90</b>	
Estructura de una función .....	90
Definición de una función .....	90
Llamada a una función .....	91
Declaración de una función. Prototipo .....	91
Parámetros de una función .....	92
Paso de parámetros por valor .....	92
Paso de parámetro por referencia .....	92
Parámetros <i>const</i> de una función .....	93
Ámbito .....	93
Ámbito del programa .....	93
Ámbito de archivo fuente .....	94
Ámbito de una función .....	94
Ámbito de bloque .....	94
Funciones locales .....	94
Clases de almacenamiento .....	94
Variables automáticas .....	94
Variables externas .....	94
Variables registro .....	95
Variables estáticas .....	95
Funciones de biblioteca .....	96
<b>ARRAYS EN C 97</b>	
Introducción .....	97
arrays unidimensionales .....	97
Declaración .....	97
Acceso a los datos .....	97
Carga de datos .....	97
arrays bidimensionales .....	98
Declaración .....	98
Acceso a los datos .....	98
Carga de datos .....	99
Arrays multidimensionales .....	99
arrays de caracteres .....	100
Declaración .....	100
Acceso a datos .....	100
Carga de datos .....	100
Funciones de entrada/salida para cadenas .....	101
Arrays indeterminados .....	101
<b>PUNTEROS Y GESTIÓN DINÁMICA DE MEMORIA      103</b>	
Introducción .....	103
Conceptos básicos .....	103
Definición de punteros y asignación de direcciones .....	103
Indirección .....	104
Operaciones con punteros .....	105

Asignación de punteros .....	105
Aritmética de punteros.....	105
Comparación de punteros.....	106
Punteros y Arrays.....	106
Punteros y arrays unidimensionales .....	106
Punteros y cadenas de caracteres.....	108
Punteros y arrays bidimensionales .....	108
Arrays de punteros .....	108
Indirección múltiple.....	109
Funciones de asignación dinámica de memoria .....	109
<b>ESTRUCTURAS COMPUESTAS 113</b>	
Introducción.....	113
Estructuras .....	113
Definición de variables.....	113
Referencia a elementos .....	114
Carga de datos .....	115
Estructuras anidadas.....	115
Array de estructuras.....	116
Punteros y estructuras .....	116
Funciones y estructuras .....	117
Paso de elementos individuales.....	117
Paso de la estructura completa.....	117
Campos de bits .....	118
Uniones .....	119
Enumeraciones .....	120
Tipos de datos definidos por el usuario.....	122
<b>ESTRUCTURAS DINÁMICAS INTERNAS DE DATOS EN C 123</b>	
Introducción.....	123
Listas enlazadas simples .....	123
Creación .....	123
Insertar un nodo en la lista.....	123
Borrar un nodo de la lista.....	124
Recorrido de la lista en sentido ascendente .....	125
Recorrido de la lista en sentido descendente .....	125
Búsqueda de un nodo.....	126
Listas enlazadas dobles .....	126
Creación .....	126
Insertar un nodo en la lista.....	126
Borrar un nodo de la lista.....	127
Recorrido de la lista en sentido ascendente .....	128
Recorrido de la lista en sentido descendente .....	128
Recorrido en ambos sentidos .....	129
Listas circulares.....	129
Creación .....	129
Insertar un nodo en la lista.....	130
Borrar un nodo en la lista.....	131
Recorrido de la lista en sentido ascendente .....	132
Búsqueda de un nodo.....	132
Pilas .....	133
Creación .....	133

Apilar.....	133
Desapilar.....	133
Cima de la pila .....	134
Búsqueda de un nodo.....	134
Colas .....	135
Creación .....	135
Insertar un nodo en la cola .....	135
Borrado de un nodo de la cola.....	136
Recorrido de una cola.....	136
Búsqueda de un nodo en la cola.....	136
<b>FICHEROS EN C</b>	
<b>138</b>	
FLUJOS .....	138
Puntero FILE .....	138
Apertura de un archivo .....	139
Modos de apertura de un archivo .....	140
NULL Y EOF.....	141
Cierre de archivos.....	141
Creación de un archivo secuencial.....	142
Funciones putc () y fputc () .....	142
Funciones getc () y fgetc () .....	142
Funciones fputs () y fgets () .....	143
Funciones fprintf () y fscanf () .....	144
Función feof () .....	145
Función rewind () .....	146
Archivos binarios en C .....	146
Función de salida fwrite () .....	147
Función de lectura f read () .....	148
Funciones para acceso aleatorio.....	149
Función fseek() .....	149
Función ftell () .....	154

# 1

## Metodología de la programación

### INTRODUCCIÓN

Al igual que el comportamiento y el pensamiento humano se rigen por métodos de razonamiento lógicos que nos permiten la ejecución de acciones o tareas concretas, el comportamiento o actuación de un ordenador se rige por lo que se denomina **programación**, entendiendo como tal el desarrollo y puesta en marcha de una solución a un problema concreto, mediante una secuencia de instrucciones o conjunto de acciones lógicas que debe ejecutar el ordenador.

El ordenador es una máquina capaz de recibir datos y proporcionar resultados. Los datos recibidos constituyen la información de entrada, y los resultados ofrecidos, la información de salida. El ordenador realiza un determinado proceso que transforma los datos de entrada en los resultados de salida. Para que un ordenador pueda recibir, procesar y proporcionar información, necesita que se le den órdenes concretas. El conjunto de estas órdenes necesarias para resolver un determinado problema recibe el nombre de **programa** y la ciencia para diseñar, construir y mantener programas es la **programación**.

### CONCEPTOS DE ALGORITMO Y PROGRAMA

Los computadores son máquinas complejas y sofisticadas. No obstante desconocen la cuestión más sencilla que se les pueda plantear. El hardware por sí solo no es suficiente para hacer operativo a los ordenadores, debido a que precisan de órdenes que los hagan funcionar y estas órdenes forman lo que se denomina programa. La preparación de las diferentes instrucciones y operaciones ordenadas que forman nuestro programa necesita una representación detallada para reflejar la secuencia de ejecución de las mismas, para lo cual se utilizan los algoritmos. Una definición más precisa de algoritmo es:

*La determinación exacta y sin ambigüedades de la secuencia de pasos elementales que deben ejecutarse para encontrar la solución de un problema.*

Un algoritmo debe ser:

- **Preciso.**- En cuanto al orden de las operaciones.
- **Finito.**- En cuanto al número de operaciones.
- **Correcto.**- Debe conducir a la solución del problema.

Un mismo problema puede resolverse por más de un algoritmo.

En programación, cada uno de los pasos que forman el algoritmo se denomina **instrucción**, y deben de ser escritas en un lenguaje comprensible para el ordenador. Al lenguaje se le llama **lenguaje de programación** y al conjunto de estas instrucciones se le denomina **programa**. Un programa, por tanto, es un algoritmo transcrito, mediante un lenguaje de programación, a instrucciones ejecutables por el ordenador. El objetivo de la programación es establecer una secuencia de acciones (instrucciones) que puedan ser ejecutadas por el ordenador y que realicen el trabajo.

En un programa coexisten siempre las instrucciones que lo forman y los datos que son manipulados por estas instrucciones. Generalmente se suele presentar el esquema más



básico de un programa el que contiene instrucciones necesarias para ejecutar sobre los datos los tres grandes grupos de acciones:



## PASOS PARA LA RESOLUCIÓN DE UN PROBLEMA

La resolución de un problema mediante un programa se compone de varias fases que pueden agruparse en tres bloques diferenciados:

**Análisis.**- En esta fase se estudia el problema examinando su naturaleza y entorno en el que la solución informática se ejecutará:

**Definición del problema.**- Es fundamental para poder llegar a una buena solución el tener perfectamente definido el problema. Hay que recoger información e identificar las necesidades y los objetivos que se persiguen.

**Análisis del problema.**- En esta fase se pretende llegar a conclusiones sobre aspectos relativos al problema: equipo a utilizar (ordenador, periféricos, soportes, etc.), personal implicado en el problema, estudio de los datos y documentos de entrada, estudio de los datos de salida, descomposición del problema en módulos, etc. Se trata en esta fase de marcar, en grandes rasgos, los pasos necesarios para resolver el problema y su entorno.

**Resolución.**- Conocidas las necesidades del problema y un posible enfoque de la solución se pasa a su construcción.

**Programación.**- Se trata de diseñar la solución de una forma más concreta, en forma de algoritmo y mediante la aplicación de un conjunto de técnicas. Se obtiene un conjunto de pasos elementales a seguir por el ordenador para alcanzar la solución, que quedan representados en ordinogramas, pseudocódigo o cualquier otro medio de representación de algoritmos.

**Codificación.**- Es la transcripción del algoritmo desarrollado en la fase anterior a un lenguaje de programación concreto utilizando y sintaxis. Generalmente, es un trabajo fácil y mecánico, aunque requiere buen conocimiento del lenguaje de programación que se utiliza.

**Instalación.**- Una vez obtenido el código del programa, se pasa a su instalación, que comprende las siguientes fases:

**Edición.**- Consiste en la escritura del *programa fuente* almacenándolo en algún soporte permanente. Esto se realiza con un programa llamado editor. Normalmente, edición y codificación son términos sinónimos, ya que en la actualidad se suelen desarrollar simultáneamente. Al resultado se le denomina *programa fuente*.

**Traducción.**- Del programa fuente editada a instrucciones comprensibles por la máquina, obteniéndose un ejecutable. Esto se realiza con unos programas *intérpretes* y *compiladores* que, además, verifican la correcta sintaxis del programa. La principal diferencia entre interpretación y compilación es que la primera se efectúa en el momento de la ejecución del programa (se traduce tantas veces como se ejecute), mientras que la compilación se realiza de forma previa a la ejecución (una vez compilado correctamente el programa no es necesario hacerlo más).

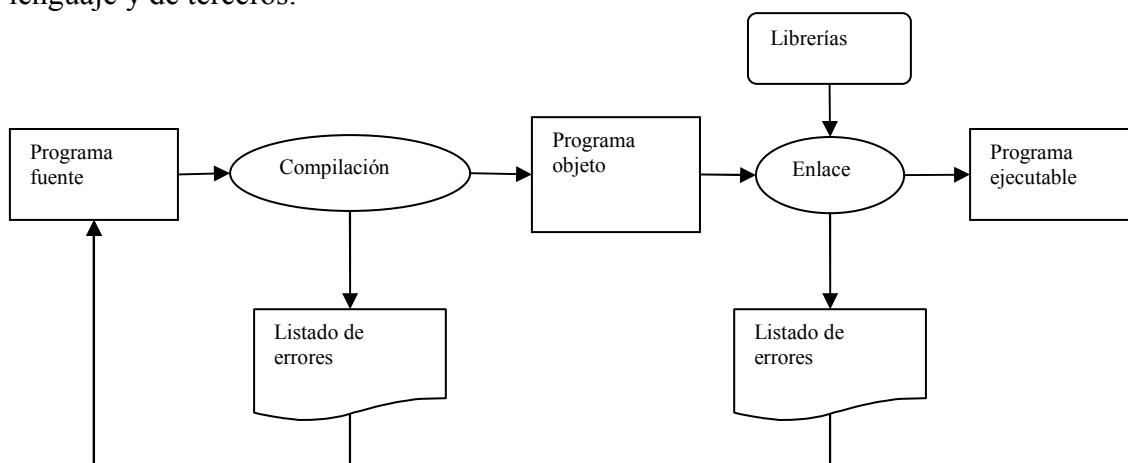
**Intérprete.**- Traduce cada instrucción antes de ejecutarla.

Ventajas: Para probar el programa el programador no tiene que realizar ninguna operación adicional.

**Inconvenientes:** Una vez terminado e instalado el programa, antes de cada ejecución debe ser nuevamente interpretado, con la consiguiente lentitud de la ejecución. También el programa fuente ha de estar presente en el ordenador en el que se ejecuta, por lo que puede ser inspeccionado y modificado por el usuario.

**Compilador.-** Traduce el programa, obteniéndose finalmente un programa que ya es directamente ejecutable por el ordenador. Las ventajas e inconvenientes son las contrarias del intérprete.

El proceso de obtención de un programa ejecutable mediante un compilador suele constar de 2 partes: *compilación* propiamente dicha y *enlace* con las librerías del lenguaje y de terceros.



**Pruebas.-** Al programa se le va sometiendo, mediante un completo juego de datos de ensayo, a pruebas para verificar su correcto funcionamiento en todas las situaciones posibles. A esta etapa se le denomina también **depuración** porque en ella deben aparecer todos los posibles errores de funcionamiento. Si se sigue la técnica de programación modular las pruebas se realizan durante la fase de codificación, en la que cada módulo es independiente del resto y por tanto, depurable de forma aislada.

**Documentación.-** Una vez terminado y probado suficientemente, el programa debe de documentarse con listados del código fuente, la descripción completa del algoritmo, tablas de decisión, organigramas o pseudocódigo, documentos de salida y entrada, etc.

**Mantenimiento.-** La vida del programa continúa una vez que está en explotación. Como mínimo, serán necesarias modificaciones y actualizaciones impuestas por cambios en el entorno o por nuevas necesidades del usuario. Esta etapa se hace a veces imposible de realizar por personas distintas a los programadores originales si no se ha documentado correctamente el programa o aplicación.

## DATOS: TIPOS Y CARACTERÍSTICAS

Ya se ha visto que un programa no es más que una secuencia ordenada de instrucciones que actúan sobre datos. Estos datos que se manipulan en un programa deben, necesariamente, de estar organizados de tal forma que se puedan localizar y acceder a ellos sin problema. Es decir, los datos deben de estar dispuestos en *estructuras*.

El correcto diseño de las estructuras de datos es, al menos, tan importante como el diseño de los programas que las acceden. Estas estructuras se almacenan en la memoria principal. Cualquier dato, para que pueda ser accedido por cualquier instrucción, debe de ser llevado previamente a la memoria principal, de igual forma, cualquier resultado, aunque entre desde un periférico para salir por otro, es necesario que pase por memoria.

## Tipos de datos

Estudiaremos los datos en función de dos clasificaciones distintas. Según el valor que tienen pueden ser:

**Numéricos.**- Cuando se almacena un dato numérico en memoria, lo que realmente se registra es su valor binario. Los diferentes lenguajes de programación pueden subdividir en rangos los posibles valores de los datos numéricos para optimizar la cantidad de bytes de memoria que utilizan para guardar un número y la velocidad para accederlo. Así, hay lenguajes que distinguen entre datos enteros, reales, de precisión simple, de precisión doble..., ocupando cada uno de ellos una cantidad diferente de bytes en memoria. Otros lenguajes, en cambio, utilizan siempre la misma cantidad de bytes de memoria para guardar números, independientemente de su valor.

**No numéricos.**- Pueden ser:

Alfanuméricos.- Un dato alfanumérico es un conjunto de caracteres que puede tener letras, cifras y caracteres especiales. Los datos alfanuméricos se almacenan en memoria representando cada carácter en un byte (8 bits). Cada carácter del alfabeto tiene un código numérico que es realmente el que se registra en memoria. Así, un dato alfanumérico de 15 caracteres de longitud, ocupa 15 bytes.

Lógicos.- Solo disponibles en algunos lenguajes, los datos lógicos ocupan 1 byte de memoria y solo pueden tomar dos posibles valores: verdadero o falso. Generalmente, esto se implementa almacenando uno de dos números: 0 (falso) o 1 (verdadero).

Otros.- Fechas, bloques de código, objetos...

Según su variabilidad los datos pueden clasificarse en:

**Constantes.**- Una constante es un dato cuyo valor permanece inalterado a lo largo del programa. Normalmente, el valor del dato está escrito en el propio código fuente. Una constante se expresa escribiendo explícitamente su valor. La mayoría de los lenguajes de programación permiten diferentes tipos de constantes, siendo las más generales las numéricas, alfanuméricas (o de caracteres) y las booleanas (o lógicas). En general: Una constante numérica se utiliza poniendo su valor, sin más. Por ejemplo: 8  
Una constante alfanumérica se utiliza poniendo su contenido entre comillas. Por ejemplo: “Juan Hidalgo Pérez”; “C/Alta, 28”, “14011”, “CÓRDOBA”.

**Variables.**- Una variable es una zona de memoria, con un nombre dado por el programador y utilizado para su acceso, capaz de guardar un valor que puede ser cambiado por otro a lo largo de la ejecución de un programa. Todas las variables tienen un nombre, al que se le suele denominar **identificador** porque sirve para referenciar o identificar su contenido. A la operación de dar un valor a una variable se le denomina **asignación**.

Podemos imaginar una variable como una caja en la que guardamos un dato. El conjunto de cajas (variables) se guarda en las estanterías de un almacén (memoria). Cada caja (variable) tiene un nombre o “referencia” (identificador) que utilizaremos para accederla y tomar su contenido o poner un contenido diferente en su interior.

Acceso a una variable.- Es la operación de direccionar los bytes de memoria que ocupa. A una variable se accede para cambiarle el valor o para tomar el valor que tiene.

Asignación.- Es la operación de dar valor a una variable se suele realizar con un *operador de asignación*. Ejemplos:

En lenguaje C: a = “Pepe”

En Pseudocódigo: a ← “Pepe”

En la parte derecha del operador de asignación puede figurar cualquier expresión que dé como resultado uno del mismo tipo de la variable. Algunos lenguajes no utilizan operadores de asignación para dar valores a variables, sino una instrucción de asignación. Además del operador o instrucción de asignación, a una variable se le puede

asignar un valor con instrucciones de lectura desde teclado o desde un soporte de almacenamiento externo.

**Toma de valor.-** Es la operación de leer el valor contenido en la variable. Se accede simplemente poniendo su nombre en el programa. Por ejemplo

$N \leftarrow N + 1$  tomar el valor actual de N, le suma 1 y lo asigna nuevamente a N.

### **Características de las variables**

Toda variable, en todos los lenguajes, tiene las siguientes características:

**Nombre.-** El identificador de la variable. Cada lenguaje de programación impone sus reglas estrictas para nombrar variables: longitud máxima de caracteres del nombre, caracteres utilizables, letras mayúsculas o minúsculas, etc.

**Tipo.-** Conjunto de valores que puede contener. Las variables, al igual que las constantes, puede ser, según el lenguaje: numéricas, alfanuméricas, booleanas, etc.

Dado que el tipo de variable determina la cantidad de memoria necesaria para almacenar sus valores y la forma en que se accede a ellos, el lenguaje debe de conocer en todo momento el tipo de cada variable. Los lenguajes utilizan diferentes métodos para obtener esta información sobre el tipo de las variables, aunque lo más habitual es que el tipo de las variables sea declarado antes de su utilización. Por ejemplo, en C, para declarar una variable de tipo entero y nombre numero1 se escribiría: `int numero1;` En otros lenguajes, el control del tipo de la variable es realizado por el propio lenguaje de forma automática y dinámica: una variable es del tipo del que sea su valor. Este método, aunque muy práctico para el programador, suele penalizar en tiempo de acceso y en una mayor ocupación de memoria.

**Valor.-** Es su contenido, un valor concreto de su tipo. Es decir, uno de los valores posibles dentro del rango que permite el tipo.

**Ámbito.-** El ámbito de una variable está compuesto por los módulos y subprogramas en los que, existiendo la variable, ésta puede ser accedida. Esta característica tiene la virtud de hacer más seguras a las variables de un subprograma o módulo ante accesos accidentales de otros, permitiendo de esta forma “encapsular” sus variables y hacerlos independientes entre si.

**Duración.-** Una variable, una vez creada, podemos necesitar que exista durante todo el programa o temporalmente, en el subprograma o módulo que la utiliza de forma exclusiva. Además, en éste último caso, es deseable que la variable sea “destruida” (y los bytes que ocupa liberados) sin que el programador tenga que poner la instrucción oportuna. Así, si un subprograma crea una variable propia, ésta es destruida cuando el mismo finaliza; o, por el contrario si se crea una variable de uso más amplio (global), ésta tiene una duración indefinida hasta el final del programa (o hasta su destrucción explícita con la instrucción apropiada, si ésta está disponible).

**Visibilidad.-** Una variable, para que sea visible (accesible), ha de existir y debe de accederse desde dentro de su ámbito. Una variable no está visible cuando no ha sido creada o declarada, cuando no se accede desde su ámbito o porque está “oculta” (por otra del mismo nombre y de ámbito más local).

Para satisfacer los requerimientos de ámbito, duración y visibilidad, los lenguajes de programación disponen, cada uno con su terminología de variables locales, públicas o globales, estáticas, privadas, etc.

### **Variables simples y compuestas**

Las variables simples contienen un único valor. Por tanto, el nombre de la variable sólo nos proporciona un dato cuando tomamos su valor y, análogamente, solo podemos asignarle un valor. Son variables elementales.

Una variable compuesta es una asociación de variables elementales. A veces, la utilización de variables compuestas resulta muy útil, ya que permiten identificar a varios datos mediante un sólo nombre. Por ejemplo:

INSTITUTO				
NOMBRE	TIPO	DIRECCIÓN		ALUMNOS
Gran Capitán	IES	CALLE	POBLACION	1001
		Arcos de la Frontera S/N	Córdoba	

La variable INSTITUTO está compuesta de: NOMBRE (simple), TIPO (simple), DIRECCIÓN (compuesta) y ALUMNOS (simple).

La variable DIRECCIÓN está compuesta a su vez de: CALLE y POBLACIÓN (simples).

Dado que las variables compuestas están fuertemente estructuradas, se les suele llamar *estructuras*. Normalmente, para el acceso a un dato elemental de una variable compuesta se antepone el nombre de cada variable envolvente, separados por puntos. Suponiendo que todas las variables simples son alfanuméricas, salvo ALUMNOS que es numérica, y efectuando asignaciones mediante constantes a la variable del ejemplo, tendríamos:

INSTITUTO.NOMBRE ← “Gran Capitán”

INSTITUTO.TIPO ← “IES”

INSTITUTO.DIRECCION.CALLE ← “Arcos de la Frontera S/N”

INSTITUTO.DIRECCION.POBLACION ← “Córdoba”

INSTITUTO.ALUMNOS ← “1001”

No todos los lenguajes admiten variables compuestas. En general, sólo son admitidas por aquellos lenguajes en los que hay que declarar las variables.

## OPERADORES Y EXPRESIONES

Las constantes y variables de un programa pueden actuar mezcladas como operandos mediante el uso de **operadores**, formando **expresiones**. Las expresiones, junto con las palabras reservadas (o símbolos) forman **instrucciones**. Los operadores son símbolos que indican como son manipulados los datos (constantes y variables).

Dicho de otra forma, una expresión es una combinación, por medio de operadores, de constantes y/o variables que arrojan un valor determinado y en las que pueden intervenir paréntesis y nombres de funciones del lenguaje. Toda expresión tiene un valor final, que se obtiene ejecutando las operaciones indicadas por los operadores sobre los operandos.

Según lo anterior:

Una constante es una expresión. Por ejemplo: 4, 4.5, “Pedro”

El nombre de una variable es una expresión. Por ejemplo: X, NOMBRE

Una combinación de constantes y/o variables y/o funciones, relacionadas con operadores es una expresión, Por ejemplo: X+Y, X-4, (X\*Y) / (Y-2)

Según el valor que resulte, las expresiones pueden ser de 3 tipos:

**Aritméticas.**- Dan un resultado de tipo numérico. Se construyen mediante operadores aritméticos y sus operandos son de tipo numérico. Por ejemplo: 2+C, X\*3

Los operadores aritméticos suelen ser los que intervienen en las operaciones aritméticas normales:

Operador	Significado	Prioridad	Ejemplo
$\wedge$ ó **	Exponenciación	1	V**2

*	Multiplicación	2	V*2
/	División	2	V/2
+	Suma	3	V+2
-	Resta	3	V-2

Los operadores aritméticos tienen una *escala de prioridad*, es decir, en una expresión aritmética unos operadores se ejecutan antes que otros. Esta escala de prioridad puede ser modificada con el uso del paréntesis. Se ejecutarían primero las operaciones de los paréntesis más interiores). Los paréntesis tienen máxima prioridad. Siempre se aconseja que se utilicen los paréntesis si hay dudas sobre el orden en que se resuelven las operaciones de una expresión. También se aconseja su uso para hacer el código más claro y legible.

**Alfanuméricas.**- Producen resultados de tipo alfanumérico. Sus operandos son constantes y/o variables y/o funciones alfanuméricas. Por ejemplo: NOMBRE + “Álvarez”. En muchos lenguajes sólo se dispone del operador de concatenación +.

**Relacionales o Booleanas.**- Produce un resultado lógico de CIERTO o FALSO. Sus operandos puede ser de cualquier tipo, pero todos ellos del mismo. Para su construcción se utilizan los operadores relacionales. Las expresiones relacionales son muy utilizadas en la construcción de selectivas y bucles.

Operador	Significado	Ejemplo
>	Mayor a	V>2
<	Menor a	V<2
=	Igual a	V=2
<>	Distinto a	V<>2
>=	Mayor o igual a	V>=2
<=	Menor o igual a	V<=2

Para construir expresiones relacionales más complejas se pueden utilizar *operadores lógicos*. Estos operadores, como los aritméticos, tienen una escala de prioridad que igualmente puede ser alterada mediante el uso de paréntesis.

Operador	Significado	Prioridad	Ejemplo
NOT	Negación (NO)	1	NOT V**2
AND	Conjunción(Y)	2	V>2 AND X=0
OR	Disyunción (O)	3	V>2 or X=0
XOR	Disyunción exclusiva	4	V>2 XOR X=0

Dado que tanto el uso de operadores relacionales como de operadores lógicos siempre dan como resultado un valor lógico (verdadero o falso), a las expresiones relacionales se las conoce también como expresiones lógicas.

La determinación del resultado de la aplicación de los operadores lógicos se suele representar mediante las tablas de verdad de los operadores lógicos. Siendo X e Y dos expresiones relacionales:

X	Y	X AND Y	X OR Y	X XOR Y	NOT X	NOT Y
V	V	V	V	F	F	F
V	F	F	V	V	F	V

F	V	F	V	V	V	F
F	F	F	F	F	V	V

## LENGUAJE E INSTRUCCIONES

Ya se ha comentado que un programa es una secuencia finita de instrucciones que debe de ser independiente de los datos de entrada. La cantidad de instrucciones necesarias para la codificación de un programa en un lenguaje de programación determinado depende del conjunto de instrucciones disponibles en cada lenguaje. A las instrucciones disponibles por un lenguaje se les denomina también **primitivas** del lenguaje.

Un programa codificado en un lenguaje de programación es una secuencia de acciones primitivas del lenguaje. Las acciones no disponibles por el lenguaje deben de ser descompuestas en sus primitivas equivalentes.

### Evolución y clasificación de los lenguajes de programación

Se suele decir que los lenguajes de programación tienen niveles o “generaciones” según sus primitivas sean de más o menos “nivel”, en el sentido de que tengan más o menos cualidades para acercarse al lenguaje humano y de que sean capaces o no de realizar acciones más complejas con sus primitivas. Cada lenguaje, según su nivel, tiene sus ventajas e inconvenientes y han sido desarrollados para fines concretos.

**Lenguaje máquina.**- Lamentablemente el ordenador entiende un lenguaje binario que consta de 0 y 1, denominados bits. Este lenguaje es muy complicado de usar para un humano, ya que trabajar con unos y ceros resulta lento de traducir y es fácil equivocarse. Antes, los programadores introducían los datos y programas en forma de largas ristas de unos y ceros, con la ayuda de interruptores que controlaban la memoria. De esta forma el programa más simple se convertía en algo complejo y aburrido, consiguiendo que se sustituyera por otros lenguajes más fáciles de aprender y utilizar, que además reducen la posibilidad de cometer errores. La situación no tardó en cambiar con la aparición de los llamados lenguajes ensambladores.

**Lenguajes ensambladores.**- En lenguaje ensamblador todavía una instrucción primitiva se corresponde con una instrucción-máquina sólo que ahora, además de las instrucciones, se codifican simbólicamente mediante códigos mnemotécnicos fáciles de recordar los datos, direcciones de memoria, los operandos, los códigos de operación, en vez de hacerlo directamente con su representación binaria. Así se consigue que las direcciones de memoria se referencien no explícitamente, sino simbólicamente. En ensamblador, una adición o supresión de una instrucción no implica que el programador tenga que cambiar la dirección de las siguientes (el procesador hará el cálculo de direcciones), lo que facilita cierta flexibilidad en la programación. Además, la ejecución de programas es rapidísima (más rápida que en cualquier lenguaje de más nivel) y permite la explotación total de los recursos de la máquina.

Sin embargo, un lenguaje ensamblador es específico de un procesador determinado (por eso se habla de lenguajes ensambladores, en plural: para 386, 486, etc.), requiere buenos conocimientos de la forma de funcionamiento del procesador específico y la escritura de programas es larga, difícil y es muy frecuente cometer errores: los lenguajes ensambladores son de bajo nivel y se dicen que están orientados a la máquina.

**Lenguajes de alto nivel.**- Los lenguajes de alto nivel permiten programar sin conocer el funcionamiento interno de la máquina. Las instrucciones se acercan mucho al lenguaje humano (en inglés) y una instrucción en un lenguaje de alto nivel equivale a muchas otras en lenguaje máquina. Así, una instrucción de asignación como:  $X = (A+B+C) * \cos(Y)/Z$  equivale a decenas de instrucciones máquina. Cualquier aplicación mediana escrita en cualquier lenguaje de alto nivel acaba siendo , miles o centenares de miles de

instrucciones en lenguaje máquina, solo que el ingente trabajo de traducción a máquina lo realizan los programas compiladores o intérpretes, preocupándose el programador únicamente de conocer la sintaxis del lenguaje. Los lenguajes de alto nivel se les llama también **lenguajes de 3ª generación**.

Los lenguajes de programación de alto nivel han venido surgiendo desde los años 60. Cada lenguaje nace para dar respuesta a la necesidad de programar soluciones en campos concretos (aplicaciones científicas, de gestión, etc.) y la evolución de cada uno de ellos por cada fabricante ha ocasionado un gran número de versiones.

Entre la gran variedad de lenguajes de alto nivel, podemos destacar según las aplicaciones para las que han sido desarrollados:

Científicos.- Lenguaje FORTRAN. (FORMula TRANslator. IBM, 1956). Actualmente está prácticamente en desuso debido a que permite pocos tipos y estructuras de datos.

De gestión.- Lenguaje COBOL (COMmon Business Orinted Language, lenguaje común orientado a los negocios. Departamento de Defensa de EEUU, 1960). Utilizado ampliamente y de gran portabilidad. Es un lenguaje cuya sintaxis refleja excesivamente la necesidad de acercarse al lenguaje humano (excesivamente coloquial). Sin embargo, sus cualidades para el manejo de ficheros de datos y el enorme volumen de programas desarrollados en COBOL hace de él un lenguaje todavía muy utilizado.

De propósito general.- Lenguaje C (Laboratorios BELL AT&T, 1972). Polivalente y portable. El lenguaje C nació como un desarrollo del lenguaje B y se desarrolló para reescribir el sistema operativo UNIX, que en su primera fase estaba escrito en ensamblador. La principal aportación de C sobre B fue el diseño de tipos de datos y estructuras.

Posteriormente a su fin inicial, los programadores empezaron a utilizar C como una excelente herramienta para el control total de los recursos de la máquina que da como resultado ejecutables de gran rapidez, y actualmente un elevado número de aplicaciones, sistemas operativos y otros lenguajes han sido desarrollados en C.

Lenguaje Pascal (por el filósofo francés Blaise Pascal. Niklaus Wirth, 1969).

Constituye, por sus estructuras de datos y de control, una buena implementación de la programación estructurada.

Lenguaje BASIC (Beginner's All purpose Symbolic Instruction Code. Dartmouth College, 1965). Muy extendido, principalmente debido a que Microsoft e IBM lo vienen facilitando en sus sistemas operativos.

Lenguajes de 4ª Generación.- En los últimos años están surgiendo lo que se está dando en llamar metalenguajes o lenguajes de 4ª generación. Se caracterizan por que se acercan aún más al lenguaje humano. Poseen potentes instrucciones que equivalen a muchas de los lenguajes de 3ª Generación. En este grupo pueden englobarse desde las potentes herramientas suministradas por los más modernos SGBD (ORACLE) hasta las versiones visuales de los lenguajes de 3ª G, ahora ya basadas en la programación orientada a objetos (Visual Basic, Visual C++, etc.)

### **Métodos de expresión de un algoritmo**

Un algoritmo puede expresarse mediante diferentes métodos

**Pseudocódigo**.- Es una codificación simbólica, mediante lenguajes nemotécnicos que no pueden ser ejecutados por el ordenador pero que son entendibles por cualquier persona.

**Lenguajes de programación**.- Es la expresión del algoritmo para que pueda ser almacenado, leído y ejecutado por un ordenador. Sólo es entendible por una persona que conozca el lenguaje.

**Diagramas de flujo**.- Es la representación gráfica de la solución de un problema. Podemos distinguir dos tipos de diagramas de flujo:




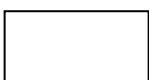
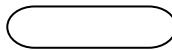
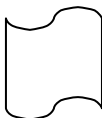
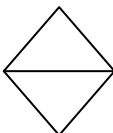


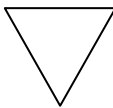


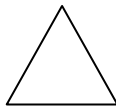
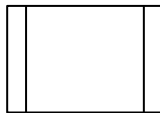
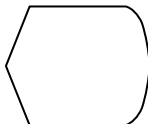
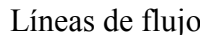

**Diagramas de flujo del sistema u ORGANIGRAMAS.**- Representan gráficamente el flujo de datos e información que maneja un programa. Muestran los dispositivos de entrada y salida que va a manejar un programa, es decir, los soportes de los datos de entrada y de salida, los nombres de los programas y el flujo de los datos. Los organigramas se desarrollan en la fase de análisis del problema, y para su confección se siguen las siguientes reglas:

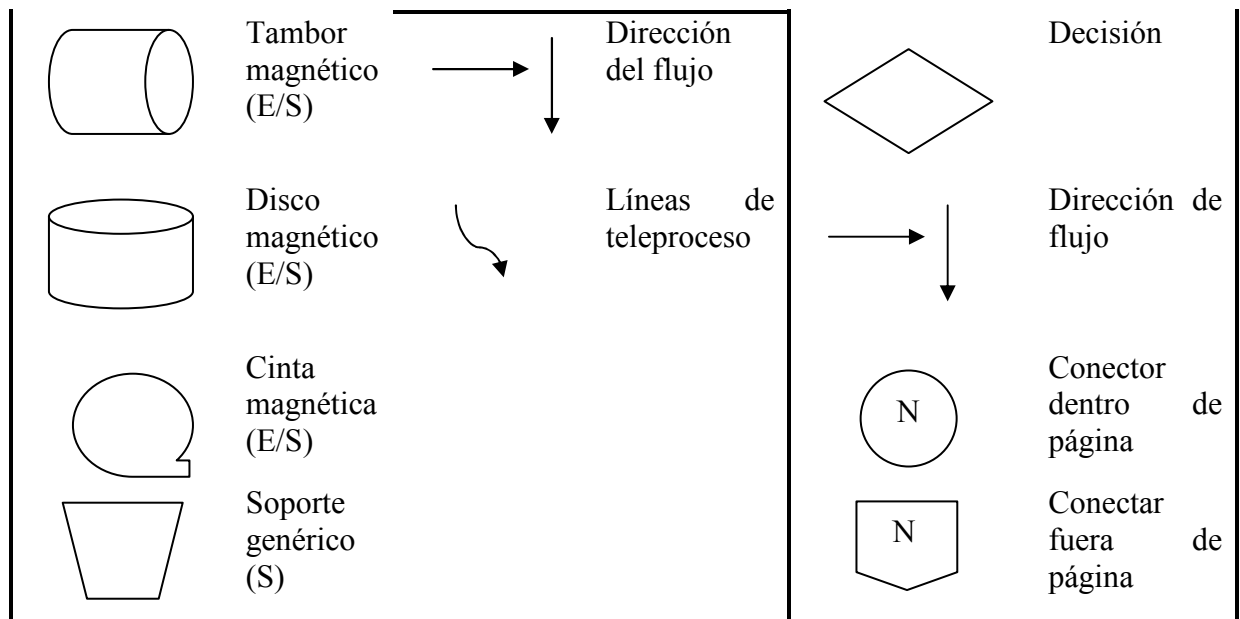
- En el centro del organigrama figura el nombre del programa.
- En la parte superior, los soportes de entrada.
- En la parte inferior, los soportes de salida.
- Al mismo nivel que el nombre de proceso, los soportes de entrada y salida.

**Diagramas de flujo del programa u ORDINOGRAMAS.**- Representan gráficamente la secuencia lógica de las operaciones que realiza el programa. Se desarrollan en la fase de programación y debe contener el comienzo y el final del programa y las operaciones en la secuencia determinada. Para confeccionar los ordinogramas se tendrán en cuenta las siguientes reglas:

- El comienzo del programa figurará en la parte superior del ordinograma.
- Los símbolos de comienzo y fin deben de aparecer una sola vez.
- El flujo de operaciones será, siempre que sea posible, de arriba abajo y de izquierda a derecha.
- Es preferible guardar una cierta simetría en la representación de selectivas y bucles, así como en el conjunto total del ordinograma.
- Dos líneas de flujo nunca deben cruzarse, utilizando cuando sea necesario conectores.

A continuación se detallan los símbolos utilizados para confeccionar los diagramas de flujo.

ORGANIGRAMAS				ORDINOGRAMAS	
Símbolos de soporte		Símbolos de proceso			
	Tarjeta perforada (E/S)		Proceso		Terminal (inicio, fin, parada)
	Cinta de papel (E/S)		Clasificación de archivos		Operación
	Impresora (S)		Fusión de archivos		Operación de E/S general
	Teclado (E)		Partición de archivos		Subprograma
	Pantalla (S)				Inicialización en declaraciones



# 2

## Programación Estructurada

Con el nacimiento de la programación se hizo necesario el desarrollo de técnicas y métodos para facilitar y mejorar el diseño y la resolución de programas. Esto llevó a la técnica de la programación estructurada, la cual se fundamenta en tres conceptos básicos:

**Estructuras básicas.**- Este concepto se basa en el denominado *Teorema de la estructura*, según el cual todo programa que tenga una sola entrada y una sola salida puede ser elaborado con 3 únicas estructuras:

Estructura secuencial.- Es aquella en la que una acción se ejecuta tras otra.

Estructura repetitiva.- Es aquella en la que se ejecutan varias veces un grupo de instrucciones, formándose un ciclo o bucle.

Estructura alternativa.- Evalúan una condición y según su resultado se toman rutas alternativas de instrucciones.

**Recursos abstractos.**- Consiste en olvidarse del ordenador y el lenguaje de programación cuando se está concibiendo la solución del problema, obteniéndose de esta forma una solución constituida por un conjunto de acciones que es necesario realizar.

La solución planteada se enfrenta con el ordenador y con el lenguaje a utilizar para comprobar si las acciones pueden ejecutarse. Si no pueden ser ejecutadas, las acciones se descomponen en subacciones más simples, continuándose este proceso de descomposición hasta que cada subacción sea susceptible de ser codificada en el lenguaje elegido y por consiguiente de ejecutarse en el ordenador.

**Razonamiento Tod-Down.**- Este razonamiento es deductivo y consiste en resolver un problema estableciendo una serie de niveles de menor a mayor complejidad (de arriba abajo). El primer nivel resuelve totalmente el problema. El segundo nivel es un refinamiento del primero, así sucesivamente siguiendo la metodología de los recursos abstractos. Si el planteamiento es correcto nunca será necesario volver atrás, puesto que situados en un nivel cualquiera, el conjunto de todos los niveles anteriores ya habrá resuelto el problema en su totalidad.

### TIPOS DE INSTRUCCIONES

Una instrucción puede ser considerada como un hecho o suceso de duración limitada que genera unos cambios previstos en la ejecución de un programa, por lo que debe ser una acción previamente estudiada y definida.

Las primitivas de un lenguaje 3G pueden clasificarse en:

1. Instrucciones de declaración de datos.
  - Instrucción declarativa y de tipificación de datos.
2. Instrucciones primitivas
  - Instrucción de entrada.
  - Instrucción de salida
  - Instrucción de asignación
3. Instrucciones compuestas
4. Instrucción de llamada a procedimientos.
5. Instrucciones de control del flujo del programa.

- Instrucción selectiva simple.
  - Instrucción selectiva compuesta.
  - Instrucción selectiva múltiple.
6. Instrucciones de control repetitiva.
- Bucle PARA
  - Bucle MIENTRAS
  - Bucle REPETIR .. HASTA
7. Instrucción de salto

Cualquier programa que se desarrolle en un lenguaje 3G tiene que ajustarse a estas primitivas que vamos a ver. Todas las instrucciones se verán en pseudocódigo y siguiendo la sintaxis establecido para ello. En primer lugar indicamos la estructura general de un programa escrito en pseudocódigo:

```
Programa: NOMBRE DE PROGRAMA
Entorno:
    Instrucciones de declaración de variables
Inicio:
    Instrucciones
Fin
```

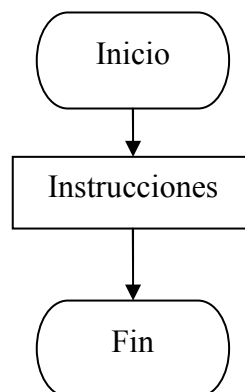
Todo programa estará rotulado con un nombre, que será breve pero descriptivo de la labor que realiza. En la sección de *Entorno* se pondrán las sentencias de declaración de variables. Como se observa, antes de escribir las instrucciones que realizan alguna operación sobre el programa hay que tener declaradas todas las variables que se van a usar en el programa, algo que exigen la mayoría de los lenguajes de programación de hoy en día. De esta forma nos acostumbraremos a ir trabajando con la rigidez natural de un lenguaje de programación y conseguiremos que el salto del pseudocódigo a cualquier lenguaje de programación sea lo menos traumático posible. La declaración de variables sigue el siguiente formato:

```
nombre_variable_1:tipo_de_datos
o también
variable_1, variable_2, ..., variable_3:tipo_de_datos
```

Se pondrán tantas variables como se necesite. Los nombres podrán tener caracteres alfabéticos, signos de subrayado o números, pero comenzarán por letra o signo de subrayado.

Entre las etiquetas *Inicio* y *Fin* se pondrán las instrucciones que componen el programa y que se verán en las siguientes secciones.

En ordinograma un programa tendría la siguiente forma:



## 1.- Instrucción de declaración de datos

Indican el tipo, las características y la identificación de los objetos que componen un programa. Son aquellas instrucciones utilizadas para informar al procesador del espacio que debe reservar en memoria con la finalidad de almacenar un dato mediante el uso de variables simples o estructuras de datos más complejas.

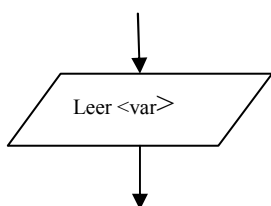
La definición consiste en indicar un nombre a través del cual haremos referencia al dato y un tipo a través del cual informaremos al procesador de las características y espacio que deberá reservar en memoria. Su sintaxis es:

`<nombre_variable> : <tipo>`

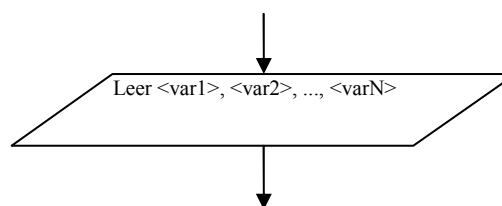
siendo <tipo> numérico, alfanumérico o booleana si se emplea pseudocódigo. Cuando se emplee un lenguaje de programación habrá que emplear los tipos propios del lenguaje y establecer su ámbito. La mayoría de los lenguajes establecen el ámbito de una variable dependiendo del lugar donde se declaren, aunque también se pueden emplear palabras claves del lenguaje.

## 2.- Instrucción de Entrada

Es aquella instrucción encargada de recoger el dato de un periférico o dispositivo de entrada (por ejemplo el teclado, un ratón, un escáner, etc.) y seguidamente almacenarlo en memoria en una variable previamente definida, para la cual se ha reservado suficiente espacio en memoria.



Leer <variable>  
<varN>



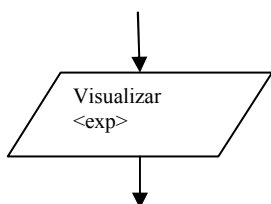
Leer <var1>, <var2>, ...

En el supuesto de leer varios valores consecutivos, con la intención de almacenarlos en variables diferentes, lo indicaremos situando uno a continuación del otro separados por comas.

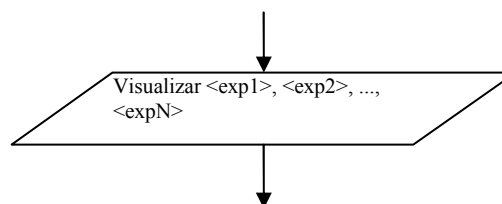
La instrucción de entrada sólo puede tener una variable, ya que el valor introducido en memoria debe de almacenarse en una variable.

## 3.- Instrucción de salida

Es aquella instrucción encargada de recoger los datos procedentes de variables o los resultados obtenidos de expresiones evaluadas y depositarlos en un periférico o dispositivo de salida (por ejemplo, la pantalla, la impresora, etc.).



Visualizar <expresión>  
... <exp.>



Visualizar <exp1>, <exp2>,  
...

Ejemplos de instrucciones de entrada y salida:

```

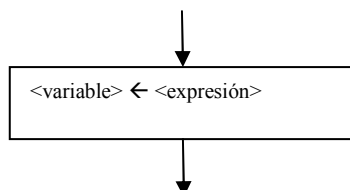
Leer EDAD
Visualizar "Nombre completo: " + NOMBRE + " " +
APELLIDOS

```

En el primer caso se introduce por teclado un número y se almacena en la variable EDAD. En el segundo se forma una expresión por la concatenación de dos valores constantes de tipo alfanumérico y dos variables del mismo tipo y se visualiza por pantalla.

#### 4.- Instrucción de asignación

Permiten dar (asignar) valores a una variable. Los valores a asignar son el resultado de una expresión. Normalmente, los lenguajes disponen de un operador de asignación, pero en pseudocódigo emplearemos el símbolo  $\leftarrow$ . La sintaxis es:



$\text{<variable>} \leftarrow \text{<expresión>}$

donde  $\text{<expresión>}$  es una expresión del mismo tipo que  $\text{<variable>}$ . Ejemplos:

$A \leftarrow 38$

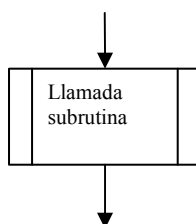
$\text{NUMERO} \leftarrow A * 2 - 1$

$\text{NOMBRECOMPLETO} \leftarrow \text{NOMBRE} + \text{" Gómez López"}$

En el primer caso se asigna el valor numérico constante a la variable A. En el segundo ejemplo el resultado de la expresión de multiplicar por 2 el valor de la variable A y restarle 1, se asigna a la variable NÚMERO. Por último se concatenan una variable alfanumérica con una constante alfanumérica, formando una expresión que se asigna a la variable NOMBRECOMPLETO.

#### 5.- Instrucción compuesta

Son aquellas instrucciones que no pueden ser ejecutadas directamente por el procesador, y están constituidas por un bloque de acciones agrupadas en subrutinas, subprogramas, funciones o módulos.



Más adelante, en este mismo capítulo veremos las funciones y procedimientos en profundidad.

#### 6.- Instrucción selectiva simple

También llamadas condicionales, debido a que siempre contienen una condición, construida a partir de una expresión relacional que arroja un resultado lógico (VERDADERO o FALSO).

En un programa, las instrucciones se ejecutan de forma secuencial, es decir, una detrás de otra. Sin embargo, en numerosas ocasiones es necesario romper ese orden secuencial y *bifurcar*, *saltar* o *transferir el control* a otras instrucciones del programa que no son consecutivas a las que en ese momento se ejecutan. Las instrucciones que siguen a la que ha bifurcado se ejecutan también secuencialmente hasta que se vuelva a bifurcar o aparezca el fin del programa.

La bifurcación consiste en la cesión del control del programa a un punto determinado del mismo (anterior o posterior) como resultado del cumplimiento (valor verdadero) de una expresión lógica o condición (bifurcación condicional), o sin necesidad de cumplirse ninguna condición previa (bifurcación incondicional).

Estas instrucciones permiten la realización de una comparación de datos con la finalidad de tomar una decisión sobre la ejecución de unas instrucciones. Si el resultado es verdadero, se ejecuta un grupo de instrucciones y si, por el contrario, resulta falso, el flujo del programa continúa en la instrucción inmediata debajo de la selectiva simple. Su sintaxis es:

```
SI <expresión lógica>
ENTONCES
    <instrucción 1>
    <instrucción 2>
    ...
    <instrucción n>
FINSI
```

Ejemplo:

```
ocupación = "ACTIVO"
SI edad = 65 ENTONCES
    ocupación = "JUBILADO"
FINSI
Visualizar ocupación
```

En el ejemplo anterior si se cumple que la variable *edad* tiene el valor 65, entonces se asigna el valor "JUBILADO" a la variable *ocupación*. Si la variable *edad* no tuviera el valor 65, entonces el flujo del programa continuaría en la siguiente instrucción al FINSI, visualizando el valor de *ocupación*.

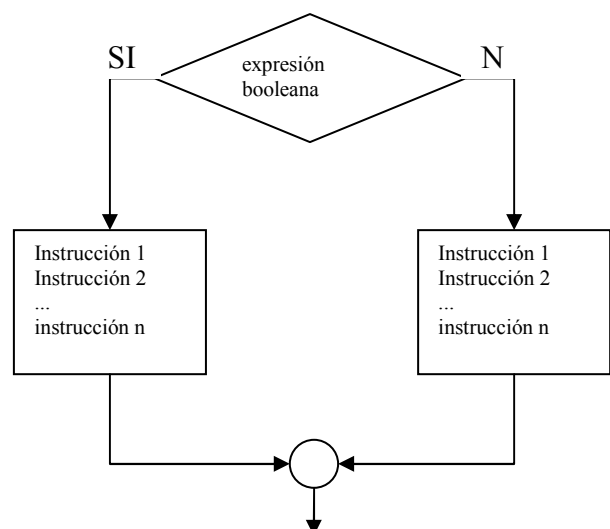
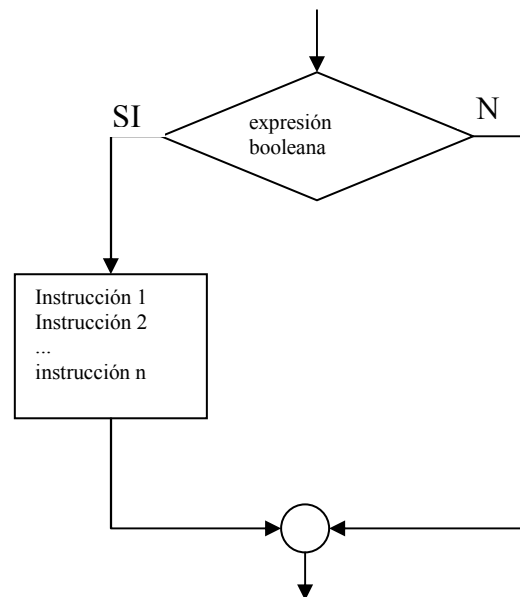
## 7.- Instrucción selectiva compuesta.

Similar a la anterior. La diferencia estriba en que si se cumple la condición se ejecutan un bloque de sentencias, pero si no se cumple, se ejecutan otro bloque de sentencias. Su sintaxis es:

```
SI <expresión lógica>
ENTONCES
    <instrucción 1>
    <instrucción 2>
    ...
    <instrucción n>
SINO
    <instrucción 1>
    <instrucción 2>
    ...
    <instrucción n>
FINSI
```

Ejemplo:

```
Leer número
SI número >= 100 ENTONCES
    Visualizar "Número excesivamente alto"
```



```

SINO
    Visualizar "Número correcto"
FINSI
Visualizar número

```

### 8.- Instrucción selectiva múltiple.

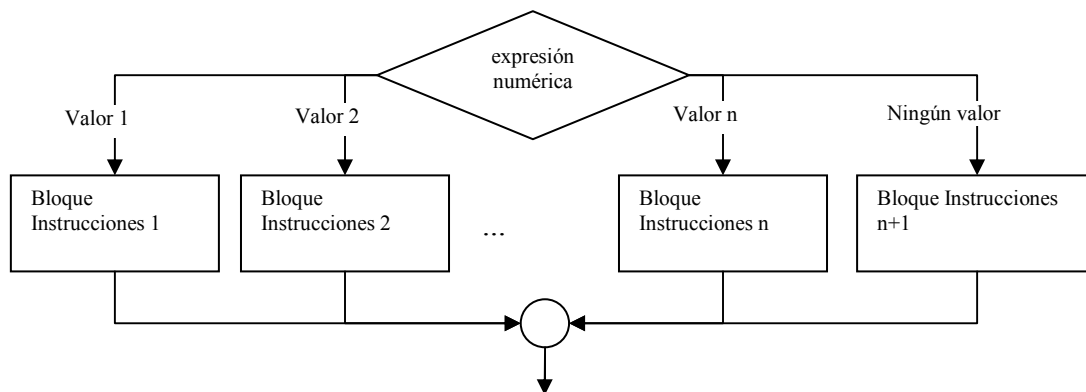
Esta instrucción evalúa el valor de una expresión, y dependiendo del valor que tome ejecuta un bloque de instrucciones. Pueden evaluarse tantos valores como se desee, asociando a cada uno de ellos un bloque de instrucciones a ejecutar en el caso de que coincida con el valor de la expresión evaluada. Su sintaxis es:

```

SEGÚN <expresión>
    CASO <expresión 1>
        Bloque de instrucciones 1
    CASO <expresión 2>
        Bloque de instrucciones 2
    CASO <expresión 3>
        Bloque de instrucciones 3
    ...
    CASO <expresión n>
        Bloque de instrucciones n
[OTRO CASO]
    Bloque de instrucciones n + 1
FINSEGÚN

```

Como se puede deducir de la sintaxis la instrucción SEGÚN evalúa el valor de <expresión> y lo compara uno a uno con cada expresión que aparece en cada CASO. Si el valor de alguna de las expresiones coincide con el valor de <expresión>, entonces se ejecuta el bloque de instrucciones asociado. Si no coincide ningún valor, entonces se ejecuta el bloque de instrucciones asociado a la cláusula OTRO CASO, que es opcional. Su representación en pseudocódigo sería:



Ejemplo:

```

SEGÚN nota
    CASO 5
        Visualizar "Suficiente"
    CASO 6
        Visualizar "Bien"
    CASO 7
        Visualizar "Notable"
    CASO 8
        Visualizar "Notable"
    CASO 9
        Visualizar "Sobresaliente"

```



CASO 10

Visualizar "Matrícula de Honor"

OTRO CASO

Visualizar "Insuficiente"

FINSEGUN

En el ejemplo, se evalúa el valor de la variable *nota*. Dependiendo del valor que tenga se visualiza la calificación correspondiente a la nota. Si la nota no coincide con ningún valor puesto, se asume que es inferior a 5 y por tanto se visualiza la constante alfanumérica "Insuficiente".

Nótese que podría hacerse una construcción parecida concatenando instrucciones selectivas simples SI..SINO..FINSI como a continuación:

```

SI nota = 5 ENTONCES
  Visualizar "Suficiente"
SINO
  SI nota = 6 ENTONCES
    Visualizar "Bien"
  SINO
    SI nota = 7 OR nota = 8 ENTONCES
      Visualizar "Notable"
    SINO
      SI nota = 9 ENTONCES
        Visualizar "Sobresaliente"
      SINO
        SI nota = 10 ENTONCES
          Visualizar "Matrícula de honor"
        SINO
          Visualizar "Insuficiente"

```

FINSI

FINSI

FINSI

FINSI

FINSI

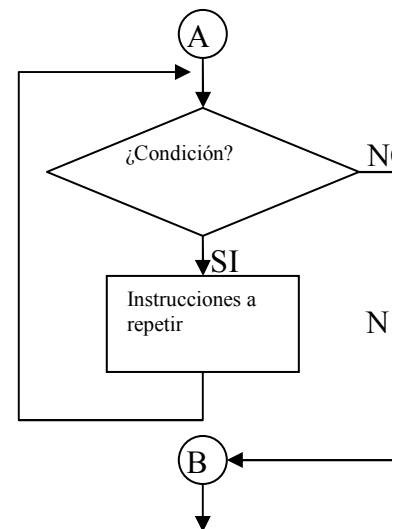
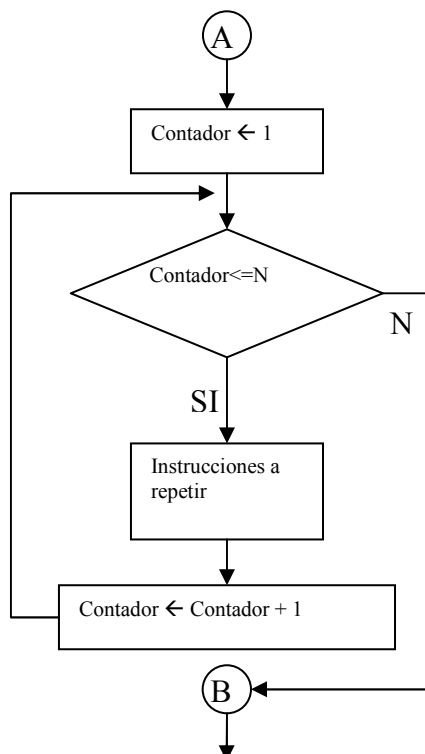
## 9.- Instrucciones de control repetitivas.

Un **bucle** es un conjunto de instrucciones sometidas a una ejecución repetitiva. Las instrucciones que se repiten son siempre las mismas, pero los datos con los que operan suelen ser diferentes. Todo bucle contiene al menos una condición que controla si el flujo del programa continúa o se sale del bucle, es decir, determina el número de repeticiones que han de ejecutarse, en caso contrario estaríamos ante un *bucle sin salida*, *bucle infinito* o *bucle sin fin*.

Para controlar el número de repeticiones hay tres técnicas:

**Bucle PARA.** -Este tipo se controla mediante una variable contador que va incrementándose (normalmente en una unidad), produciéndose la salida del bucle cuando el contador alcanza cierto

valor. Básicamente su estructura es:



```

    PARA Contador DE <expresión inicio> A <expresión
    fin> [INC <expresión incremento>]
    Bloque de instrucciones
  FIN PARA

```

El bloque de instrucciones que hay dentro de la sentencia PARA .. FIN PARA se ejecuta desde <expresión inicio> hasta <expresión fin>. En cada iteración del bucle la variable *Contador* se incrementa en uno por defecto o en <expresión incremento> si se indica la cláusula *INC*. Si se indica un incremento negativo, la variable contador disminuye en <expresión incremento>.

Ejemplo: Supóngase que se desea visualizar por pantalla el cuadrado de los 10 primeros números enteros.

```

    PARA a de 1 a 10
    Visualizar a*a
  FIN PARA

```

También podría haberse visualizado en orden decreciente, tal como sigue:

```

    PARA a DE 10 A 1 INC -1
    Visualizar a*a
  FIN PARA

```

Bucle MIENTRAS. - Se ejecuta un bloque de instrucciones mientras se cumpla la condición.

La condición se evalúa al principio del bucle, por lo que la condición es de entrada/salida, pues si la primera vez no se cumple, el flujo del programa no llega a entrar en el bucle. Tanto en este esquema como en el siguiente no se conoce generalmente el número de veces que el bucle se ejecutará. Su esquema básico es:

```

    MIENTRAS <expresión lógica>
    Instrucciones a repetir
  FIN MIENTRAS

```

Mientras la <expresión lógica> sea verdadera, se ejecutarán las instrucciones del cuerpo del bucle.

Ejemplo: Visualizar por pantalla la suma de los números que se introducen por teclado hasta que la suma total supere 100.

```

    Leer número
    total ← número
    MIENTRAS total < 100
    Visualizar total
    Leer número
    total = total + número
  FIN MIENTRAS
  Visualizar total

```

Bucle REPETIR .. HASTA. - Se repite la ejecución del bucle hasta que se cumpla una condición. La condición se encuentra al final del bucle por lo que es solo de salida, pues el bucle se ejecuta al menos una vez. Su esquema es:

```

    REPETIR
    Instrucciones a repetir
  HASTA <expresión lógica>

```

Las instrucciones que hay dentro del bucle se ejecutarán hasta que <expresión lógica> sea verdadera, a diferencia que el bucle MIENTRAS en el que la condición se hace falsa para salir del mismo.

Ejemplo: Leer desde teclado números y visualizarlos por pantalla hasta que se introduzca el 0.

```

REPETIR
  Leer número
  Visualizar número
HASTA número = 0

```

Una norma muy extendida entre los programadores es utilizar el **sangrado de código**. Consiste en sangrar la escritura de las instrucciones que forman un bloque dentro de una sentencia selectiva o repetitiva. En el caso de que haya instrucciones de este tipo anidadas (dentro del cuerpo de otras sentencias), se sangrarían con respecto al bucle o selectiva en la que se encontrase. Es muy aconsejable el uso de esta técnica, ya que ayuda a clarificar el código y, por tanto, facilita la depuración de programas (búsqueda de errores en un programa).

### 10.- Instrucción de salto

Son aquellas instrucciones que alteran o rompen la secuencia normal de ejecución de un programa, perdiendo toda posibilidad de retornar el control de ejecución del programa al punto de llamada. El uso de este tipo de instrucciones no se recomienda, ya que puede complicar la legibilidad del código. Su sintaxis es:

```
SALTO <etiqueta>
```

Donde <etiqueta> es una palabra que identifica un punto en el programa.

Las instrucciones de salto se pueden clasificar en condicionales e incondicionales. Las instrucciones de salto condicional son aquellas que alteran la secuencia de ejecución de un programa sólo y exclusivamente en el caso de que la condición especificada sea cierta.

Ejemplo:

```

Programa: Salto condicional
Entorno:
  Longitud: Numérico
  Área: Numérico

```

```

Inicio:
  Leer Longitud
  SI Longitud < 0 ENTONCES
    SALTO Error
  FIN SI
  Área ← Longitud *
  Longitud
  Visualizar "El valor del
  área es: ", Área

```

#### **Error:**

```

  Visualizar "El valor introducido no es
  correcto"
Fin

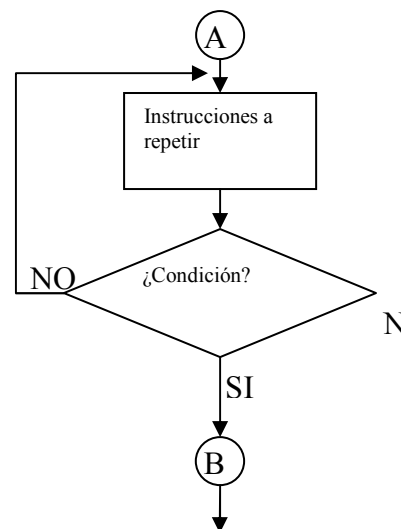
```

La instrucción de salto incondicional altera la secuencia normal de ejecución de un programa siempre, pues no van acompañadas de una condición que limite en determinadas ocasiones la realización del salto a otra parte del programa. Ejemplo

```

Programa: Salto incondicional
Entorno:
  Longitud: Numérico

```



```

Área: Numérico

Inicio:
  Comienzo:
  Leer Longitud
  Área ← Longitud * Longitud
  Visualizar "El valor del área es: ", Área
  SALTO Comienzo
Fin

```

## ELEMENTOS Y CARACTERÍSTICAS DE UN PROGRAMA

### Variables auxiliares

**Contador.-** Es una variable que se utiliza para contar. Normalmente se inicializan a 0 y cada vez que ocurre un suceso que se desea contar, se incrementan en una unidad. Contiene un valor que varía en cantidades fijas, generalmente la unidad, durante el desarrollo de un programa.

Ejemplo: Escribir un programa que lea por teclado las notas de 20 alumnos y visualice al final cuantos han aprobado.

```

Programa: Aprobados
Entorno:
  Nota: numérica, Aprob: numérica
  i: numérica
Inicio:
  Aprob ← 0
  PARA i DE 1 A 20
    Leer Nota
    SI Nota >= 5 ENTONCES
      Aprob ← Aprob + 1
    FIN SI
  FIN PARA
  Visualizar "Alumnos aprobados: ", Aprob
Fin

```

Como se puede observar al saberse de antemano los números que hay que leer desde teclado, se utiliza el bucle PARA en el que después de leer por teclado la nota se comprueba si es mayor o igual a 5 con la selectiva simple. Si la condición se cumple la variable contador *Aprob* se incrementa en uno.

**Acumulador.-** Son variables que se utilizan para realizar sumatorios o productos de distintas cantidades. Para el sumatorio se inicializan a 0 y para el producto se inicializan a 1. Es un elemento cuyo valor aumenta varias veces en cantidades variables.

Ejemplo: Escribir un programa que pida por teclado un número hasta que se introduzca el 0 y al final visualice la suma total de los números introducidos que son mayores que 100 y la suma total de los números que son menores que 100.

```

Programa: Sumatorio
Entorno:
  Total1: numérica, Total2: numérica, Número:
  numérica

Inicio:
  Total1 ← 0
  Total2 ← 0

```

```

REPETIR
  Leer Número
  SI Número >= 100 ENTONCES
    Total1 ← Total1 + Número
  SINO
    Total2 ← Total2 + Número
  FIN SI
HASTA Número = 0
Visualizar "Suma de números mayores que 100:",
Total1
Visualizar "Suma de números menores que 100:",
Total2
Fin

```

En el ejemplo anterior, se utilizan dos variables acumuladores para llevar la suma de los números que se han introducido por teclado y son mayores o iguales que 100 y las que son menores que 100. Podría haberse usado variables contadores para llevar la cuenta de los números sumados en ambos casos.

**Interruptor o Switch.**- Es una variable que puede tomar dos valores exclusivamente. Normalmente 0 y 1, cierto y falso, etc. Se utiliza para avisar de si ha ocurrido un suceso que se está esperando.

Ejemplo: Escribir un programa que lea números por teclado, hasta que se introduce el 0 y al final indique si ha habido alguno negativo.

```

Programa: Negativo
Entorno:
  Neg: numérica, Número: numérica

Inicio:
  Neg ← 0
  Leer Número
  MIENTRAS Número <> 0
    SI Número < 0 ENTONCES
      Neg ← 1
    FIN SI
  FIN MIENTRAS
  SI Neg = 0 ENTONCES
    Visualizar "Todos los números introducidos son
positivos"
  SINO
    Visualizar "Ha habido algún número negativo"
  FIN SI
Fin

```

Como se puede observar, el interruptor *Neg* solamente avisa si se ha producido el hecho de que algún número introducido sea negativo, pero no cuenta cuantos han sido. Sólo en el caso de que ninguno de ellos haya sido negativo, *Neg* tendrá el valor que se le asignó al principio, 0, ya que nunca el flujo del programa habrá pasado por la selectiva donde *Neg* cambia de valor.

### Características de un programa

Para la resolución de un problema se pueden construir diferentes algoritmos o programas. La elección del más adecuado dependerá del cumplimiento de ciertas características que se consideran “deseables” en todo programa:

**Legibilidad.**- Debe ser posible su lectura y comprensión.

**Modificabilidad.**- Debe permitir cierta facilidad para ser modificado o actualizado.

**Eficiencia.**- Debe de aprovechar al máximo posible los recursos de la máquina, minimizando el tiempo de ejecución y de ocupación de memoria.

**Corrección.**- Debe de satisfacer la resolución del problema de forma correcta.

**Modularidad.**- Ha de estar subdividido en bloques o módulos, cada uno de los cuales realizará una parte del trabajo.

**Estructuración.**- Debe de cumplir las reglas de la Programación Estructurada, para facilitar la verificación, depuración y mantenimiento.

# 3

## Programación Modular

### INTRODUCCIÓN

Con objeto de resolver los problemas que derivan del desarrollo de aplicaciones extensas y complicadas, se utiliza un diseño modular, lo que implica la descomposición del programa en partes, llamadas módulos, basándonos en un diseño descendente. La división de un programa en módulos permite su desarrollo y depuración por diversos programadores, consiguiendo una reducción del tiempo empleado en el desarrollo de la aplicación y una mejora en la calidad de cada parte del programa. La división en módulos se debe terminar cuando se llegue a unas partes del programa que realicen misiones muy específicas, sin tener grandes problemas de interconexión con otras partes del mismo. Asimismo, puede suceder que algunos de los módulos necesarios para el programa ya estén desarrollados previamente en otras aplicaciones, formando parte de librerías para desarrollo, con lo cual sólo habría que incluirlos en la nueva aplicación, realizando el montaje de los mismos, reduciendo por tanto el tiempo de desarrollo y asegurando una calidad ya contrastada anteriormente.

La programación modular consiste en dividir un problema complejo en varios más sencillos. Cada problema sencillo es un módulo, es decir, un programa independiente, de lógica simple, cuyo funcionamiento no interfiere en el de los demás. Estos módulos pueden ejecutarse uno a continuación de otro o pueden ser llamados en momentos determinados para su ejecución. Todos los programas dispondrán de un módulo principal que controle y relacione al resto. Las principales ventajas de esta filosofía de programación son:

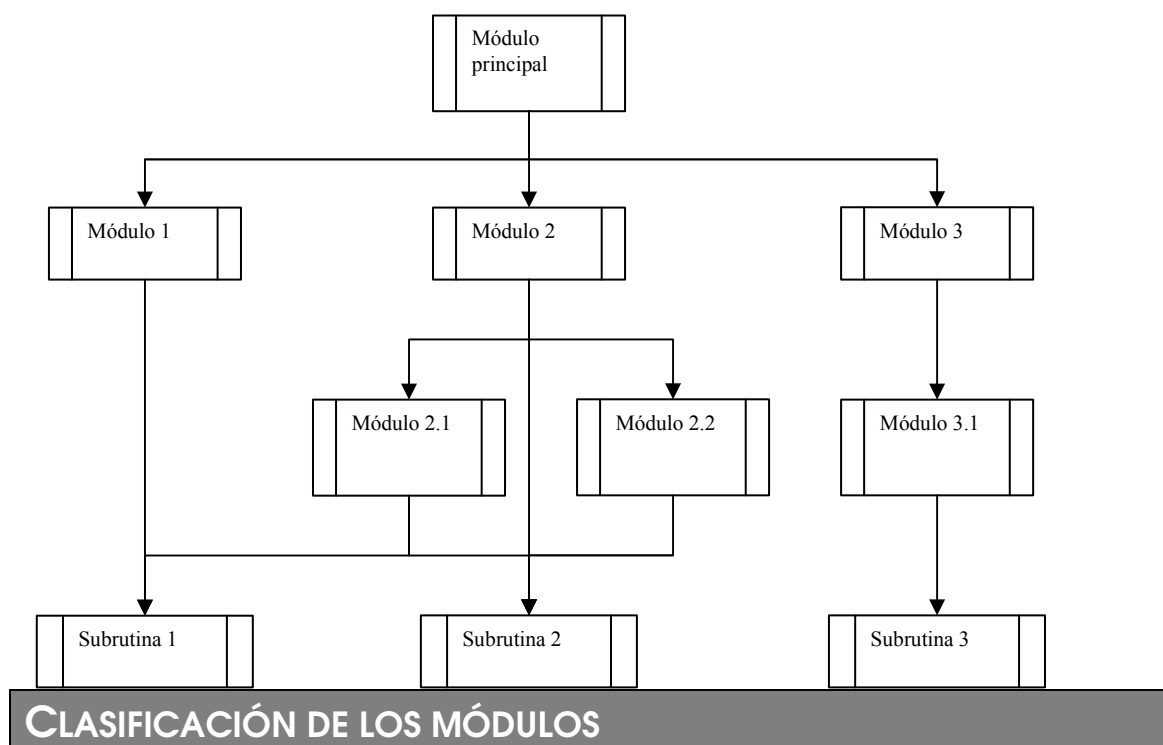
- Permite dividir la programación entre personas con pocos conocimientos sobre el problema principal.
- Cada módulo puede ser desarrollado en paralelo a otros.
- Permite la depuración de los módulos de forma independiente.
- Facilitan la elaboración de normas estándares de desarrollo.

Básicamente, la idea consiste en que cada módulo solo puede tener una entrada y una salida que lo enlaza con el módulo principal, de igual forma ocurre con los submódulos respecto a los módulos, y así sucesivamente. Hay que considerar que aunque existan estructuras repetitivas y alternativas dentro de un módulo, no debe dejar de cumplirse esta norma. No obstante, si el problema requiere que desde varios puntos del programa se ejecuten ciertas acciones comunes, se utilizarán las subrutinas o subprogramas.

Las principales ventajas de la programación modular son las siguientes:

- Facilitar la comprensión del problema y su resolución escalonada.
- Aumentar la claridad y legibilidad de los programas.
- Permitir la resolución del problema por varios programadores a la vez.
- Reducir el tiempo de desarrollo aprovechando módulos previamente desarrollados.
- Mejorar la depuración de los programas, pues se pueden depurar los módulos aisladamente de forma más sencilla e independiente.

- Facilitar un mejor y más rápido mantenimiento de la aplicación al poder realizar las modificaciones o implementaciones de una forma más sencilla tomando como base los módulos ya desarrollados.



#### En función de su situación con respecto al módulo que lo invoca

- **Interno.**- Cuando está en el mismo fichero que el módulo que lo invoca.
- **Externo.**- Cuando está en distinto fichero que el módulo que lo invoca.

#### Atendiendo al posible retorno de un valor

- **Función.**- Retorno un valor cuando devuelve el control al módulo que lo invocó. El valor retornado al módulo que hizo la llamada debe ser recogido en una expresión.
- **Procedimiento.**- No hace un retorno explícito de un valor al finalizar el módulo. Cuando se devuelve el control al módulo que hizo la llamada, la ejecución continúa en la sentencia siguiente a la que hizo la llamada.

#### En función de cuando ha sido desarrollado

- **De programa.**- Su desarrollo se ha realizado en el programa actual.
- **De librería.**- Ha sido desarrollado previamente y está contenido en ficheros de librerías.

#### En función del número de módulos distintos que realizan la llamada

- **Subprograma.**- Es invocado por un solo módulo.
- **Rutina o subrutina.**- Es invocada por diversos módulos.

### ÁMBITO DE LAS VARIABLES

Se entiende por ámbito de una variable la parte del programa donde dicha variable es accesible para ser utilizada. Las variables, según su ámbito, pueden ser:

- **Global.**- Se define en un lugar del programa que no pertenezca a ningún módulo. Normalmente se definen al principio del programa, su ámbito es todo el programa (inclusive los módulos definidos dentro de este).



- **Local.**- Se define dentro de un módulo del programa y su ámbito de validez es el módulo en el que está definida.

## FUNCIONES Y PROCEDIMIENTOS

La implementación de la descomposición modular de un programa se realiza mediante las funciones y los procedimientos.

Una función es un conjunto de sentencias que realizan una tarea determinada. Al subprograma que forma una función se le comunican unos datos de entrada (argumentos o parámetros) y produce un resultado de salida que devuelve al punto de llamada. Desde un módulo se invoca a la función para que realice una tarea. El módulo le comunica a la función los datos necesarios para poder realizar la tarea (argumentos) y cuando la función termina, devuelve al módulo que realizó la llamada el resultado de salida. Cada lenguaje de programación tiene dos tipos de funciones:

**Internas.**- Aquellas propias del lenguaje de programación y que no necesitan ser codificadas.

**Externas.**- Aquellas que no pertenecen al lenguaje y tienen que ser codificadas por el usuario. También llamadas de usuario.

Cuando las funciones internas no permiten realizar el tratamiento a los datos que se desea hay que recurrir a las funciones de usuario.

### Declaración de una función de usuario

Al ser un subprograma tiene una construcción muy similar al de los algoritmos. Consta de una cabecera de función y un cuerpo de la función.

**Cabecera.**- Tiene la siguiente sintaxis:

```
Función <Nombre_Función>(Par1:tipo1, Par2:tipo2, ...,
ParN:tipoN) :
```

Tipo\_Devolución

<Nombre\_Función>.- Nombre de la función.

Par1:tipo1, Par2:tipo2,..., ParN:TipoN.- Lista de parámetros o argumentos de entrada de la función. Cada parámetro se especifica con su nombre seguido de dos puntos y el tipo de dato. Pueden especificarse tantos parámetros como se necesiten.

Tipo\_Devolución.- Tipo de dato del valor que devuelve la función.

**Cuerpo de la función.**- Conjunto de instrucciones que forma el subprograma. La última que ejecuta es una instrucción que devuelve un valor al punto de llamada.

Un procedimiento es similar a una función, pero a diferencia de ésta, el procedimiento no devuelve ningún valor, con lo que en su definición no es necesario indicar el tipo del retorno.

Para ejecutar un procedimiento o función es necesario llamarla o invocarla desde un programa principal o desde otros subprogramas (función o procedimiento) y desde aquí proporcionarles los datos de entrada. La llamada se realiza una expresión donde aparece el nombre de la función y los parámetros encerrados entre paréntesis.

### Parámetros

Los parámetros de una función pueden ser:

- **Formales.**- Los que se declaran en la cabecera de la función. Sólo pueden ser nombres de variables y son, a efectos de ámbito, variables locales a la función.
- **Reales.**- Los que se pasan a la función en el momento de la llamada. Pueden ser variables, constantes, expresiones o valores retornados por otra función.

Tiene que haber igual número de parámetros formales que reales y ser del mismo tipo. No tienen porqué coincidir el nombre de los parámetros formales y reales.

Cada vez que se realiza la llamada a una función se establece una correspondencia entre los parámetros reales y los formales, mediante la cual el valor de los parámetros reales se copia en los parámetros formales. Así se comunican los datos desde el programa o subprograma que invoca a la función a ésta.

### Invocación de una función

La función se invoca usando en el programa o subprograma que la llama una expresión donde aparece el nombre de la función y los parámetros reales encerrados entre paréntesis. Al hacer la llamada ocurre lo siguiente:

- Se resuelven todas las expresiones que aparezcan como parámetros reales.
- Se asignan los valores de los parámetros reales a los formales.
- Se ejecutan las instrucciones que aparecen en la función.
- Se devuelve un valor al punto de llamada.

El programa o subprograma que hizo la llamada recogerá el valor devuelto por la función para su tratamiento. Lo podrá usar en una expresión o almacenar en una variable.

### Paso de parámetros a una función

Ya hemos visto antes que al pasar los parámetros a una función se establece una correspondencia entre los parámetros formales y los reales, de forma que se copia el valor del parámetro real en el formal. Esto significa que el parámetro real y el formal son dos entidades independientes, es decir, la modificación del parámetro formal dentro de la función no implica la modificación del parámetro real. En este caso se está pasando el parámetro **por valor**, ya que lo que se está transmitiendo es el valor del parámetro real al formal.

Sin embargo existe la posibilidad de que en lugar de pasar el valor del parámetro real al formal, se pase una referencia, con lo que el parámetro real y el formal son la misma cosa. Por tanto, si dentro de la función se modifica el parámetro formal, también se modifica el real. En este caso se está pasando el parámetro **por referencia**, ya que lo que se comunica a la función es una referencia (dirección de memoria) del parámetro real al formal.

Estos conceptos en el paso de parámetros a una función, confusos en apariencia, se verán con más detalle cuando se vea el lenguaje de programación C, donde podrán apreciarse con mayor nitidez.

## RECURSIVIDAD

Un concepto recursivo es aquél que se define parcialmente en términos de sí mismo. A la recursividad se le llama también *definición circular*. Se dice que un procedimiento o función es recursivo si se llama a sí mismo, es decir, si pide su propia ejecución en el curso de su desarrollo.

Como se verá más adelante, la implementación de la recursividad exige dos requisitos del lenguaje en que se vaya a codificar el algoritmo:

- Posibilidad de “apilar” las direcciones de retorno a las llamadas
- Gestión individual (local) de datos en cada invocación.

Un ejemplo clásico de función recursiva es el cálculo del factorial de un número entero. Para realizar una función que calcule el factorial de un número, en principio no hace falta necesariamente una función recursiva. Se podría resolver como:

```
Programa: Factorial
Entorno:
```

```

fact: numérico, i: numérico, número: numérico

Inicio:
  Leer número
  fact ← número
  PARA i DE número-1 A 2 INC -1
    fact ← fact * i
  FIN PARA

  Visualizar "El factorial de ", número, " es ",
  fact
Fin

```

Pero si atendemos a la definición del factorial de un número:

$\text{Factorial}(n) = n * \text{Factorial}(n-1)$

Es mucho más intuitivo realizar esta función de forma recursiva, por ejemplo:

```

Función Factorial( número:numérico ): numérico
  fact: numérico

  SI número > 1 ENTONCES
    fact ← número * Factorial( número - 1 )
  SINO
    fact ← 1
  FIN SI

  Devolver fact

```

La recursividad puede utilizarse como una alternativa a la estructura repetitiva, pero es particularmente conveniente para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos. La recursividad tienen la ventaja de que los procedimientos son transparentes y breves, pero hemos de asegurarnos que:

- La solución iterativa no recursiva equivalente sea inferior.
- La profundidad de la última invocación recursiva sea finita.
- La disponibilidad de pila y memoria sea suficiente.

### Implementación de la recursividad

Ya sabemos que una llamada a una subrutina (procedimiento o función) siempre lleva asociada un retorno a la instrucción siguiente a la que efectúa la llamada. Estas llamadas, además, pueden ser encadenadas (anidadas): se llama a un procedimiento; éste puede llamar a otro; y éste a otro; y éste a otro... Los lenguajes controlan estas situaciones anotando en una pila (*stack*) la dirección de la siguiente instrucción a la que se debe retornar, efectuando así los retornos justo en el orden inverso a la realización de las llamadas. Esta pila suele estar limitada en su tamaño, por lo que la profundidad de los anidamientos debe ajustarse a éste, aunque no suele ser problema ya que la pila suele tener un tamaño suficiente y ajustable.

Los retornos a las llamadas en un anidamiento recursivo se producen de igual manera que se ha explicado anteriormente con los anidamientos no recursivos: una llamada a un procedimiento por sí mismo genera una nueva entrada en la pila, de igual manera que si se llamase a otro procedimiento diferente de sí mismo. El nivel o profundidad de los anidamientos en la recursión tienen los mismos límites que en la no recursión, y los retornos se efectúan de la misma forma.

Sin embargo, en la no recursión, los anidamientos suelen controlarse con mayor facilidad debido a que cada llamada hay que efectuarla explícitamente con una nueva instrucción. En cambio, en un procedimiento recursivo los anidamientos a sí mismo

pueden producirse con una única llamada en su interior, lo que facilita la posibilidad de introducirnos en un anidamiento sin fin no deseado. Puede ocurrir algo análogo a la introducción en un bucle sin fin debido a que la condición de salida nunca se cumple. Por tanto es necesario extremar las precauciones para que el encadenamiento de llamadas recursivas sucesivas tenga un final cierto, y evitar así que el programa “se cuelgue” o produzca un mensaje de error y se detenga.

### **Recursividad y variables locales**

La recursividad exige al lenguaje que incluya la posibilidad de declaración de variables locales. Debido a las propiedades de estas variables, con variables locales, cada llamada recursiva a la función o procedimiento crea las mismas nuevamente, siendo las variables locales actuales salvadas para recobrar el valor una vez que se vayan produciendo los retornos. Realmente, cada invocación crea en la pila variables nuevas y propias diferentes del resto de las ejecuciones, aunque con el mismo nombre, es decir, cada nivel de profundización tiene su propio entorno por separado de variables locales que solo es accesible por él y que se almacena en posiciones de memoria distintas de los demás. En un algoritmo con recursividad infinita, esta propiedad de las variables locales hace que no solo nos podamos quedar sin espacio en la pila para los retornos, sino que se pueda producir un desbordamiento del espacio de memoria necesario para almacenar las variables locales.

Conforme se van produciendo los retornos y el procedimiento recursivo va “desenrollándose”, la memoria ocupada por las variables locales se va liberando.

### **Tipos de recursividad**

**Recursividad directa.**- Desde el interior del procedimiento se llama a sí mismo.

Procedimiento P1

...

P1

...

Retorno

**Recursividad indirecta.**- También llamada cruzada. Desde el interior de un procedimiento se llama a otro que a su vez lo llamó a él.

Procedimiento P1

...

P2

...

Retorno

Procedimiento P2

...

P1

...

Retorno

# 4

## Estructuras Estáticas Internas de Datos: Tablas

En el anterior capítulo hemos visto los tipos de datos básicos más utilizados en los lenguajes de programación (numérico, alfanumérico, lógico). En este capítulo estudiaremos formas de estructurar y manejar la información mediante el uso de **Tablas**. Estas estructuras se encuentran en prácticamente la totalidad de los lenguajes de programación, clasificadas como estructuras lineales y estáticas de datos internos.

### CONCEPTOS BÁSICOS

En este punto definiremos todos aquellos términos necesarios para el entendimiento y correcto tratamiento de tablas.

**Tabla.**- Es una estructura de datos constituida por un número fijo de elementos, todos ellos del mismo tipo y ubicados en direcciones de memoria físicamente contiguas.

**Elemento.**- Es cada uno de los datos que forma parte integrante de la tabla.

**Nombre de tabla.**- Es el identificador utilizado para referenciar a la tabla y, de forma global, los elementos que la forman.

**Tipo de tabla.**- Marca el tipo de dato básico que es común a todos y cada uno de los elementos o componentes que forman dicha estructura (numérico, alfanumérico, lógico, ...).

**Índice.**- Es un valor numérico entero y positivo a través del cual podemos acceder directa e individualmente a los distintos elementos de una tabla, pues marca la situación relativa de cada elemento.

**Tamaño de tabla.**- El tamaño o longitud de una tabla viene determinado por el número máximo de elementos que la forman, siendo el tamaño mínimo un elemento y el tamaño máximo N elementos.

**Acceso a los elementos o componentes de una tabla.**- Los elementos o componentes de una tabla tratados individualmente son datos que reciben el mismo trato que cualquier otra variable simple, con un tipo de dato que coincide con el tipo de la tabla y una denominación propia que les distingue del resto de los elementos o componentes que constituyen dicha estructura. Para acceder o referenciar un elemento en particular es suficiente con indicar el nombre de la tabla seguido del índice correspondiente entre paréntesis.

**Dimensión de tabla.**- Viene determinada por el número de índices que necesitamos para acceder a cualquiera de los elementos que forman dicha estructura.

### CLASIFICACIÓN DE TABLAS

Las tablas se clasifican según su dimensión en tablas unidimensionales, tablas bidimensionales y tablas multidimensionales.

#### Tablas Unidimensionales

Son estructuras de datos cuyos elementos son del mismo tipo, con las mismas características, y se referencian bajo un nombre o identificador común. Este tipo de estructuras también recibe el nombre de vectores.

Como los elementos de un vector se almacenan en posiciones contiguas o adyacentes en la memoria, pueden ser representados de la siguiente manera:

Direcciones de memoria contiguas y adyacentes

	d	d+1	d+2	d+3	d+4	
Primer elemento del vector	elemento 1	elemento 2	elemento 3	elemento 4	elemento 5	Último elemento del vector

Los vectores o tablas unidimensionales son estructuras de datos lineales de una dimensión, es decir, que para acceder o referenciar un elemento es suficiente con indicar su nombre seguido del índice (un solo índice) entre paréntesis. La nomenclatura para referenciar un elemento individual o componente de un vector es la siguiente:

`Nombre_Tabla(índice)`

Siendo *Nombre\_Tabla* el identificador de la tabla e *índice* una expresión numérica, cuyo resultado es un número entero mayor que cero y menor que el tamaño de la tabla.

Por ejemplo: Vector que contiene los primeros seis números pares.

**Vector:** Números

(1)	(2)	(3)	(4)	(5)	(6)	Índices
2	4	6	8	10	12	Elementos

Nombre de tabla: Números

Tipo: Numérico

Tamaño: 6, pues hay seis elementos

Elementos de la tabla: Números(1) con el valor 2, Números(2) con el valor 4, Números(3) con el valor 6, Números(4) con el valor 8, Números(5) con el valor 10, Números(6) con el valor 12.

Índices: 1, 2, 3, 4, 5 y 6

Dimensión: Una, pues para acceder a cualquiera de los seis elementos que forman la tabla únicamente necesitamos utilizar un índice (del 1 al 6).

Ejercicio: Basándonos en la tabla anterior, diseñar un algoritmo que recorra la tabla elemento por elemento y muestre en pantalla el contenido de todas las posiciones impares.

### Tablas bidimensionales

Al igual que las tablas unidimensionales, una tabla bidimensional es un conjunto de elementos del mismo tipo y características, que se referencian bajo un mismo nombre.

Este tipo de estructuras también reciben el nombre de matrices.

Una matriz puede representarse de la siguiente forma:

Índices	1	2	3	...	C
1	elemento 1,1	elemento 1,2	elemento 1,3	...	elemento 1,C
2	elemento 2,1	elemento 2,2	elemento 2,3	...	elemento 2,C
...					
F	elemento F,1	elemento F,2	elemento F,3	...	elemento F,C

Este tipo de estructuras se caracteriza porque para acceder a cada uno de los elementos de la tabla es necesario utilizar dos índices, donde el primero marca la fila y el segundo marca la columna. Por ello se dice que una matriz es una estructura de datos de dos dimensiones. La nomenclatura utilizada para acceder o referenciar a un elemento de la matriz es la siguiente:

`Nombre_Tabla(índice_fila, índice_columna)`

Siendo *Nombre\_Tabla* el identificador de la tabla, *índice\_fila* e *índice\_columna* son una expresión numérica, cuyo resultado es un número entero mayor que cero y menor que el tamaño de la primera dimensión y segunda dimensión respectivamente.

Ejemplo: Considérese una tabla de dos dimensiones de nombre *Notas* que contiene las calificaciones obtenidas por los diez alumnos de una clase en cuatro asignaturas. La matriz *Notas* tendrá una columna por alumno y una fila por asignatura; por ello podemos decir que el tamaño de dicha matriz es 4\*10 (filas\*columnas).

**Matriz:** Notas

	Alumnos de 1..10									
Índices	1	2	3	4	5	6	7	8	9	10

1	9	4	8.25	6	10	2.25	1.75	7.25	3	5.5
2	7.75	7	9.25	2.5	5	6	3.5	6.75	1	4
3	5.25	4.75	8.5	7	6.5	10	0.25	8.25	8	7.5
4	5.5	0.25	7	8.5	8	4.75	9	8	6.5	6

Nombre de tabla: Notas

Tipo: Numérico

Tamaño: 4\*10, pues hay 4 filas y 10 columnas.

Para acceder directamente a los elementos resaltados lo haríamos de la siguiente manera: Notas(1,3), Notas(1,9), Notas(3,5), Notas(3,8), Notas(4,1)

Índices: De 1 a 4 en la primera dimensión y de 1 a 10 en la segunda.

Dimensión: Dos, pues para acceder a cualquiera de los elementos que forman la matriz necesitamos utilizar dos índices.

Ejercicio: Basándonos en la tabla anterior, diseñar un algoritmo que calcule la nota media de cada alumno y muestre en pantalla el resultado.

### Tablas multidimensionales

También reciben el nombre de poliedros y son aquellas tablas de tres o más dimensiones que al igual que las tablas unidimensionales y bidimensionales están constituidas por elementos del mismo tipo.

El acceso a cada elemento se realizará mediante el empleo de tres o más índices en función del número de dimensiones de la tabla. La nomenclatura utilizada para acceder o referenciar a un elemento individual es la siguiente:

Nombre\_Tabla(índice1, índice2, ..., índice\_n)

Siendo *Nombre\_Tabla* el identificador de la tabla, *índice*, *índice2*,..., *índice\_n* son una expresión numérica, cuyo resultado es un número entero mayor que cero y menor que el número de elementos de cada dimensión.

## OPERACIONES CON TABLAS

Como hemos visto al principio del capítulo, las tablas son estructuras de datos formadas por un conjunto de elementos individuales que se caracterizan por recibir el mismo tratamiento que las variables o datos simples. Las operaciones que se pueden realizar son las mismas independientemente de la dimensión de la tabla, pero variando el número de índices.

### Declaración de una tabla

La instrucción que declara una tabla tiene la siguiente sintaxis en pseudocódigo:

```
nombre_tabla(nº_elementos1 [,nº_elementos2, ...,
nº_elementos_n]): tipo
```

donde *nombre\_tabla* es el identificador de la variable tabla, *nº\_elementos1*, *nº\_elementos2*, ..., *nº\_elementos\_n* es el número de elementos por dimensión, siendo obligatorio indicar al menos el tamaño de una dimensión, y *tipo* es el tipo de datos básico que tendrán todos los elementos de una tabla. Por ejemplo, para declarar un vector llamado *numeros* de 10 elementos de tipo numérico se escribiría:

```
numeros(10): numérico
```

Para declarar una matriz de 3 filas y 4 columnas, llamada *nombres*, con elementos de tipo alfanumérico, se escribiría:

```
Nombres(3,4): alfanumérico
```

### Recorrido o acceso secuencial

La manera más habitual de recorrer un vector es de izquierda a derecha partiendo del primero elemento de la tabla, *elemento(1)*, hasta alcanzar el último elemento de la misma, *elemento(N)*. Para agilizar y optimizar el recorrido o tratamiento secuencial de un vector se utilizan las denominadas instrucciones de control repetitivas, de las cuales la más utilizada en el tratamiento de tablas es la instrucción *PARA*, aunque ello no

implica que se puedan utilizar las otras dos instrucciones de control repetitivas para realizar dicho cometido.

Por ejemplo, si tuviéramos un vector de 10 elementos y quisiéramos visualizar cada elemento por pantalla se podrían usar el siguiente fragmento de código:

```
PARA i de 1 a 10
  Visualizar elemento(i)
FIN PARA
```

Lo más habitual es utilizar la instrucción *PARA*, ya que al ser conocido el número de elementos que hay en la tabla se sabe de antemano el número de iteraciones a realizar. Un fragmento de código que obtendría los mismos resultados utilizando la instrucción *MIENTRAS* y *REPETIR – HASTA* sería:

```
i ← 1
MIENTRAS i <= 10
  Visualizar elemento(i)
  i ← i + 1
FIN MIENTRAS
```

Con la instrucción *REPETIR – HASTA* sería:

```
i ← 1
REPETIR
  Visualizar elemento(i)
  i ← i + 1
HASTA i > 10
```

Como se podrá haber observado, los elementos individuales de la tabla pueden ser tratados como variables simples. También se puede recorrer la tabla en sentido inverso, de derecha a izquierda. Solamente hay que comenzar por el último elemento e ir disminuyendo el índice hasta alcanzar el primero.

```
PARA i DE 10 A 1 INC -1
  Visualizar elemento(i)
FIN PARA
```

Con instrucción *MIENTRAS*:

```
i ← 10
MIENTRAS i >= 1
  Visualizar elemento(i)
  i ← i - 1
FIN MIENTRAS
```

Con la instrucción *REPETIR – HASTA* sería:

```
i ← 10
REPETIR
  Visualizar elemento(i)
  i ← i - 1
HASTA i = 1
```

## Carga de datos

El objetivo de esta operación es dar valores a todos o parte de los elementos de una tabla previamente definida en memoria. La carga de los datos se puede realizar de dos formas:

**Inicialización.** - Consiste en asignar una lista de valores al vector en el mismo momento de ser definido. Por ejemplo, para declarar un vector de 5 elementos numéricos, llamado *pares*, con los cinco primeros números pares, se escribiría:

```
Pares(5): numérico ← (2,4,6,8,10)
```

**Asignación.** - A cada elemento del vector referenciado por su índice se le aplica una sentencia de asignación, cuyo formato es el siguiente:



```
nombre_tabla(índice) ← <expresión>
```

Por ejemplo, si se quisiera asignar el dato “José” al elemento de la 2ª columna y la 4ª fila de una matriz llamada *nombres*, se escribiría:

```
nombres(2,4) ← “José”
```

La asignación de valores individuales se utiliza normalmente para modificar el valor almacenado en un elemento. En aquellos casos en los que se quiera asignar un mismo valor a todos los elementos de un vector, emplearemos un bucle de la siguiente manera:

```
PARA i DE 1 A 10
  Pares(i) ← 0
FINPARA
```

Por otro lado, en aquellos casos en los que se pueda indicar una expresión para asignar a cada elemento que permita obtener un valor diferente a cada elemento, también se podría utilizar un bucle PARA.

```
PARA i DE 1 A 10
  Pares(i) ← i * 2
FINPARA
```

### Lectura

La operación de lectura permite almacenar valores en los distintos elementos de una tabla, valores que son introducidos a través de un dispositivo externo o periférico de entrada, como por ejemplo, el teclado. La sintaxis de la instrucción utilizada es la siguiente:

```
Leer nombre_tabla(índice)
```

Como se podrá observar es similar a la lectura por teclado de una variable simple.

Cuando se desee cargar todos los elementos del vector se puede utilizar un bucle PARA, tal y como sigue:

```
PARA i DE 1 A 10
  Leer numeros(i)
FIN PARA
```

Para realizar una carga parcial, solamente hay que indicar el índice del elemento al que se desea cargar el dato:

```
Leer numeros(3)
```

### Escritura

La operación de escritura trata de mostrar sobre un dispositivo de salida, por ejemplo la pantalla, los valores contenidos en una tabla. Su sintaxis es:

```
Visualizar nombre_tabla(índice)
```

Al igual que antes se puede mostrar toda la tabla o elementos individuales.

```
PARA i DE 1 A 10
  Visualizar numeros(i)
FIN PARA
```

Para realizar una escritura parcial, solamente hay que indicar el índice del elemento al que se desea visualizar el dato:

```
Visualizar numeros(8)
```

## OPERACIONES DE BÚSQUEDA Y ORDENACIÓN EN TABLAS

La operación de búsqueda consiste en averiguar si un elemento determinado pertenece o no al conjunto de elementos que forman parte integrante de una tabla y, en caso afirmativo, indicar la posición que dicho elemento ocupa. Los métodos de búsqueda más usuales son los que se describen a continuación.

### Búsqueda secuencial o lineal un vector desordenado

Esta operación consiste en recorrer el vector secuencialmente de izquierda a derecha hasta encontrar el elemento buscado o hasta alcanzar el final del vector, en cuyo caso finalizaría la operación de búsqueda sin haber localizado el elemento en cuestión. Este método también puede aplicarse si el vector está desordenado. El segmento de código que realiza esto es:

```
Leer elemento
i ← 1
MIENTRAS i <= N AND vector(i) <> elemento
    i ← i + 1
FINMIENTRAS
SI i > N ENTONCES
    Visualizar "Elemento no encontrado"
SINO
    Visualizar "El elemento está en la posición ", i
FINSI
```

### Búsqueda secuencial o lineal en un vector ordenado

Esta operación consiste en recorrer el vector secuencialmente de izquierda a derecha hasta encontrar el elemento buscado o hasta sobrepasar el valor de dicho elemento, lo que indicaría que no se encuentra entre los valores almacenados dentro del vector, no siendo necesario llegar al final para determinar que el valor buscado no existe. Este método es mucho más eficiente que el anterior en el manejo de vectores ordenados. El segmento de código que realiza esto es:

```
Leer elemento
i ← 1
MIENTRAS i <= N AND vector(i) < elemento
    i ← i + 1
FINMIENTRAS
SI i <= N AND vector(i) = elemento ENTONCES
    Visualizar "Elemento no encontrado"
SINO
    Visualizar "El elemento está en la posición ", i
FINSI
```

### Búsqueda binaria o dicotómica en un vector ordenado

Este método consiste en acotar el tramo de búsqueda entre dos extremos (extremo izquierdo y extremo derecho), calculando el punto o elemento central de dicho tramo.

$\text{Centro} = (\text{Ext\_izq} + \text{Ext\_dch}) / 2$

Seguidamente se compara el elemento central obtenido con el valor buscado. En caso de no ser iguales, nos centramos en buscar en el tramo derecho ( $\text{Centro} + 1$  y  $\text{Ext\_dch}$ ) o en el tramo izquierdo ( $\text{Ext\_izq}$  y  $\text{Centro} - 1$ ), volviendo a repetir esta operación tantas veces como sea necesario.

La búsqueda finaliza cuando se encuentre el valor buscado o si el tramo de búsqueda se reduce hasta el punto de quedar anulado, es decir, cuando  $\text{Ext\_izq} > \text{Ext\_dch}$ .

Este método sólo es válido para vectores ordenados. El fragmento de código que realiza esto es:

```
Leer elemento
izq ← 1
dch ← N
encontrado ← Falso
MIENTRAS izq <= dch AND NOT encontrado
    centro ← (izq + dch) / 2
```

```

    SI vector(centro) = elemento ENTONCES
        encontrado ← Verdadero
        Visualizar elemento, "encontrado en la
posición", centro
    SINO
        SI vector(centro) < elemento ENTONCES
            izq ← centro + 1
        SINO
            dch ← centro - 1
        FINSI
    FINSI
FINMIENTRAS
SI NOT encontrado ENTONCES
    Visualizar "El elemento buscado no existe en el
vector"
FINSI

```

### Ordenación de un vector

La ordenación consiste en reorganizar un conjunto de elementos de acuerdo con una serie de criterios previamente establecidos; estos criterios permiten agrupar dichos elementos siguiendo una secuencia específica, permitiéndonos así establecer una ordenación creciente o decreciente.

Existen multitud de métodos de ordenación, aunque solamente veremos la ordenación por intercambio directo. Este método consiste en recorrer el vector de izquierda a derecha o viceversa según la variante del método utilizada, comparando los elementos situados en posiciones adyacentes e intercambiándolos entre sí cuando se comprueba si el vector queda o no totalmente ordenado mediante la utilización de un interruptor, indicándonos el momento en el que debe finalizar la ordenación. El fragmento de código que realiza esta ordenación es:

```

j ← 1
REPETIR
    j ← j + 1
    sw ← 0
    PARA k DE N A j INC -1
        SI vector(k-1) > vector(k) ENTONCES
            aux ← vector(k-1)
            vector(k-1) ← vector(k)
            vector(k) ← aux
            sw ← 1
        FINSI
    FINPARA
HASTA sw = 0 AND j < N

```

Esta variante recorre el vector de derecha a izquierda, existe otra para recorrerlo en sentido contrario, es decir, de izquierda a derecha.

# 5

## Estructuras estáticas externas de datos: Ficheros

### CONCEPTOS Y DEFINICIONES

**Fichero.-** Conjunto de información relacionada entre si y estructurada en unidades mas pequeñas, denominadas registros, que forman un bloque que puede ser manipulado de forma unitaria.

**Registro lógico.-** Estructuras de datos homogéneas referentes a una misma entidad o cosa, dividida a su vez en elementos mas pequeños, denominados campos, que pueden ser del mismo o diferente tipo. El registro es considerado en si mismo como una unidad de tratamiento dentro del fichero.

**Campo.-** Es cada uno de los elementos que constituyen un registro lógico, siendo considerado como una unidad de tratamiento dentro del mismo registro y caracterizado por el tipo de dato que tiene asociado (numérico, alfanumérico, lógico, de tipo fecha, etc.) y por un identificador o nombre a través del cual podemos referenciarle y acceder a su contenido.

**Campo clave:** Para diferenciar un registro de otro dentro de un fichero, se recurre a una información que sea única y esta información solo la puede proporcionar el campo clave. Dicho campo puede ser creado con independencia del resto de información (campos) del registro o se puede utilizar como tal uno de los ya disponibles en la estructura de dicho registro, con el objetivo de poder realizar a través de el operaciones de búsqueda y clasificación. La definición de un campo clave en un fichero no es obligatoria, es decir, que puede carecer de ella o puede estar formada por varias definiciones de campos claves, en cuyo caso se establecen diferentes categorías entre ellas, existiendo lo que se denomina **clave principal y claves secundarias**.

**Registro físico:** También llamado bloque, es la cantidad de información que el sistema puede transferir como unidad, en una sola operación de E/S, entre la memoria principal del ordenador y los periféricos o dispositivos de almacenamiento. El tamaño del bloque o registro físico dependerá de las características del ordenador.

**Bloqueo de registro o registro bloqueado:** Un registro físico puede constar de un numero variable de registros lógicos. Por tanto, suponiendo que utilizáramos como soporte de almacenamiento el disco, se podrían transferir varios registros lógicos de la memoria al disco y del disco a la memoria en una sola operación de E/S. Este fenómeno recibe el nombre de bloqueo y el registro físico así formado se llama bloque.

**Factor de bloqueo:** Se conoce con esta denominación al numero de registros lógicos contenidos en un bloque o registro físico. Por ejemplo: imaginémonos que definimos un registro de nombre "**Alumnos**" con los siguientes campos de datos:

Identificador	Tipo de datos	Tamaño del dato
NOMBRE	Cadena (20)	1 byte * 20 = 20 bytes
APELLIDOS	Cadena(30)	1 byte * 30 = 30 bytes
DIRECCIÓN	Cadena(45)	1 byte * 45 = 45 bytes
EDAD	Numérico entero	2 bytes

De todo ello podemos deducir que el tamaño de cada uno de los registros del fichero es de  $20 + 30 + 45 + 2 = 97$  bytes. Ahora bien, suponiendo que el bloque es de 210 bytes, ¿cuál será el factor de bloqueo?  $210/97 = 2$ , sobran 16 bytes.

**Registro expandido:** Es justamente el concepto contrario de registro bloqueado, es decir, cuando el registro lógico ocupa varios bloques se le da la denominación de registro expandido.

Los conceptos y definiciones vistos hasta ahora demuestran que los ficheros son auténticas estructuras de datos jerarquizadas.

## CLASIFICACIÓN DE REGISTROS

Como hemos visto, un registro está constituido por elementos más pequeños denominados campos. Estos elementos, considerados como unidades de tratamiento dentro de los registros, pueden ser de longitud variable e incluso existir un número distinto de campos en cada uno de los registros que conforman un fichero.

Atendiendo, por tanto, a la longitud de los campos y al número de campos por registro, estos se pueden clasificar en:

### Registros de longitud fija

Atendiendo a la estructura interna de sus campos, se pueden dar las siguientes posibilidades:

1. Mismo número de campos por registro e igual longitud de los campos en el mismo y distintos registros.
2. Igual número de campos por registro, distinta longitud de cada campo del mismo registro e idéntica longitud del mismo campo en distintos registros.
3. Igual número de campos por registro, distinta longitud de campo en el mismo y diferentes registros.
4. Diferente número de campos por registro y distinta longitud de campo en el mismo y diferentes registros.

En todas las posibilidades vistas en esta primera clasificación hay que tener presente que la suma de las longitudes de los campos de cada registro es siempre la misma para todos los registros del mismo fichero.

### Registros de longitud variable

Son aquellos cuya longitud varía de un registro a otro. En este tipo de ficheros es necesario establecer desde el programa un tratamiento para diferenciar el comienzo y fin de cada campo y cada registro.

## OPERACIONES CON REGISTROS

- **Altas:** Consiste en la adición o inserción de uno o varios registros en el fichero. Esta operación solo será posible si el fichero ya existe.
- **Bajas:** Consiste en eliminar uno o varios registros del fichero. Esta operación requiere un primer proceso de lectura para la localización del registro que se pretende borrar.
- **Modificaciones:** Consiste en realizar un cambio total o parcial de uno o varios campos de los registros de un fichero. Esta operación requiere un primer proceso de lectura para la localización del registro que se desea modificar y un segundo proceso de escritura para la actualización de todo o parte del registro.
- **Consultas:** Esta operación permite acceder a uno o varios registros, con la intención de visualizar el contenido total o parcial de sus campos en pantalla o impresora en forma de listados ordenados, siguiendo ciertos criterios de clasificación establecidos por el usuario.

## CLASIFICACIÓN DE FICHEROS

Según la función y el uso que se hace de ellos, los ficheros se clasifican en dos grandes grupos denominados *permanentes* y *temporales*.

### Permanentes

Son aquellos cuyos registros sufren pocas alteraciones o variaciones a lo largo del tiempo y contienen información muy valiosa para el buen funcionamiento de la aplicación.

- **Maestros:** También reciben el nombre de ficheros de *Situación*. Son los encargados de mantener constantemente actualizados los campos cuya información es variable. Por ejemplo, un fichero de inventario con información sobre la cantidad de piezas existente en el almacén (situación de las existencias).
- **Constantes:** Contiene información fija y necesaria para el funcionamiento de la aplicación e información con un bajo índice de variación en el tiempo. Por ejemplo, un fichero de códigos postales en el que se relacionan códigos postales de diversas poblaciones y distritos.
- **Históricos:** Contienen información acumulada a lo largo del tiempo sobre las actualizaciones sufridas en los ficheros maestros y constantes. Por ejemplo, el inventario a final de año de las existencias de un almacén.

### Temporales

- **De movimiento o transacción:** Este tipo de ficheros suele contener la información necesaria para la actualización de los ficheros maestros. Información que se obtiene de los resultados obtenidos en operaciones realizadas y que posteriormente es utilizada para la actualización de los campos que tienen en común el fichero maestro y el de movimiento. El periodo de vida de este tipo de ficheros es muy corto, debido a que su función o utilidad finaliza al efectuarse la modificación o actualización de dichos datos (campos) en el fichero maestro. Una vez realizada dicha operación, el fichero de movimientos puede ser destruido o mantenido durante un tiempo limitado. Por ejemplo, los movimientos de una cuenta bancaria.
- **De maniobra o transitorios:** Son verdaderos ficheros auxiliares, creados durante la ejecución de programas o aplicaciones, con el fin de obtener cierta información, que posteriormente será procesada para conseguir unos resultados. Su periodo de vida es inferior al de los ficheros de movimiento o transacción, pues son destruidos antes de que el programa o aplicación finalice su ejecución, y no son visibles para el usuario. Por ejemplo, un fichero auxiliar que contiene la clasificación de un fichero de inventario para hacer un listado estadístico anual. Una vez finalizado el proceso de listado, el fichero que contiene dicha clasificación es destruido.

## OPERACIONES CON FICHEROS

Seguidamente estudiaremos las diferentes operaciones que podemos realizar sobre ficheros con independencia del tipo de organización o acceso que hayamos seleccionado.

- **Creación:** Para poder realizar cualquier operación sobre un fichero es necesario que haya sido creado previamente, almacenando sobre el soporte seleccionado la información requerida para su posterior tratamiento. Esto requiere que inicialmente se establezca la forma en la que la información almacenada en el fichero

será procesada en un futuro, así como el tipo de organización y acceso que emplearemos para el manejo de dichos datos.

- **Apertura:** Para poder trabajar con un fichero, este debe estar abierto, permitiendo así el acceso a los datos, dando la posibilidad de realizar sobre ellos las operaciones de lectura y escritura necesarias. Como norma general, un fichero nunca deberá permanecer abierto mas tiempo del estrictamente necesario, entendiendo como tal el periodo de tiempo empleado para la realización de cualquier operación de lectura o escritura sobre el.
- **Cierre:** Una vez finalizadas las operaciones efectuadas sobre el fichero, este debe permanecer cerrado para limitar el acceso a los datos y evitar así un posible deterioro o pérdida de información.
- **Actualización:** Esta operación permite tener actualizado el fichero mediante la escritura de nuevos registros y la eliminación o modificación de los ya existentes. Esta operación puede afectar a parte o la totalidad de los registros.
- **Ordenación o clasificación:** Esta operación permite establecer un orden entre los registros almacenados dentro del fichero. La ordenación puede ser ascendente o descendente.
- **Copiado o duplicado:** Esta operación parte de un fichero origen y crea un nuevo fichero destino con la misma estructura y contenido que el primero. Dicha operación deja intacto el fichero original.
- **Concatenación:** Se parte de la existencia de dos ficheros con la misma estructura, de manera que la concatenación de ambos crea un tercer fichero de igual estructura, cuya información es la suma del contenido del primer fichero mas el contenido del segundo. Dicha operación no afecta a los ficheros originales.
- **Fusión o mezcla:** Esta operación permite obtener de dos o mas ficheros, con la misma clasificación y estructura interna de sus datos, un nuevo fichero que contenga los registros de todos los anteriores sin alterar la ordenación que estos tenían establecida. Dicha operación no afecta a los ficheros que intervienen en el proceso de fusión.
- **Intersección:** Consiste en crear un nuevo fichero partiendo de los registros comunes de dos o mas ficheros con la misma estructura.
- **Partición o rotura:** Esta operación permite obtener varios ficheros a partir de uno inicial en función de alguna de las características internas de sus campos.
- **Compactación o empaquetamiento:** Esta operación permite la reorganización de los registros de un fichero eliminando los huecos libres intermedios existentes entre ellos, normalmente ocasionados por la eliminación de registros.
- **Consulta:** A través de las consultas es posible acceder a los registros del fichero y conocer el contenido de sus campos.
- **Borrado o destrucción:** Es la operación inversa a la creación de un fichero y en consecuencia, una vez efectuada dicha operación, se pierde toda posibilidad de acceder a los datos previamente almacenados.

## ORGANIZACIÓN Y ACCESO

Al tratar los ficheros como estructuras fundamentales de información es imprescindible hablar de dos conceptos íntimamente relacionados, *organización y acceso*

- **Organización:** Puede ser definida como la forma en la que los datos son estructurados y almacenados internamente en el fichero y sobre el soporte de almacenamiento. El tipo de organización de un fichero se establece durante la

fase de creación del mismo, siendo invariable durante su periodo de vida, por lo que a la hora de determinar el tipo de organización hay que tener en cuenta ciertos requisitos o requerimientos que son los que realmente condicionan el tipo de organización que se debe establecer, como por ejemplo, el tamaño del fichero, frecuencia de utilización o uso, etc.

- **Acceso:** Es el procedimiento necesario que debemos seguir para situarnos sobre un registro concreto con la intención de realizar una operación de lectura o escritura sobre el. Los tipos de acceso son:
  - Acceso secuencial. Es utilizado para realizar consultas totales o parciales del fichero. Este tipo de acceso puede ser utilizado tanto en dispositivos secuenciales como en dispositivos direccionables.
  - Acceso directo. Este tipo de acceso permite situarnos sobre un registro directamente a través de su clave, sin necesidad de leer o escribir los registros que le preceden en el fichero.
  - Acceso por índice. Permite acceder indirectamente a un registro previa consulta secuencial a una tabla que contiene la clave y dirección relativa de cada uno de los registros del fichero. Posteriormente se realiza un acceso directo al registro buscado.
  - Acceso dinámico. Son aquellos que permiten cualquiera de los tipos de organización conocida. Permite inicialmente un acceso directo o por índice a un registro y, a partir de ese registro, el acceso se realiza de forma secuencial.

### **Ficheros de organización secuencial**

Son aquellos ficheros caracterizados por:

- Una vez creados, los registros que lo forman se escriben o graban sobre el soporte de almacenamiento en posiciones de memoria físicamente contiguas, en la misma secuencia u orden en el que han sido introducidos sin dejar huecos o espacios libres intermedios entre ellos.
- Una vez grabados los datos en el fichero no tienen porque quedar necesariamente ordenados.
- Generalmente, debemos elegir un campo que forme parte de la estructura del registro para indicar la secuencia de clasificación u ordenación; dicho campo es el que utilizaremos como campo clave para realizar operaciones tales como búsquedas o clasificaciones, aunque no es obligatorio la definición de dichos campos claves, quedando bajo criterio del programador la selección de aquellos campos que considere los mas adecuados para ello.
- El acceso a los datos almacenados en este tipo de ficheros siempre es secuencial, independientemente del tipo de soporte utilizado. Por ello, si dispongo de un fichero con 700 registros y en un momento determinado deseamos acceder al registro 433, para poder hacerlo obligatoriamente tenemos que pasar por los 432 anteriores.

Las ventajas mas destacables en este tipo de organización son:

- Rapidez en el acceso a un bloque de registros que se encuentran almacenados en posiciones de memoria físicamente contiguas.
- No deja espacios vacíos intermedios entre registro y registro, optimizando al máximo la memoria ocupada.

Los inconvenientes mas destacables en este tipo de organización son:

- En consultas de registros individuales, este tipo de organización deja de ser la mas idónea, ya que habría que consultar secuencialmente todos los registros situados en posiciones anteriores al registro buscado.



- En consultas individuales o de un solo registro debemos realizar un proceso en el que se compare el valor del campo del registro que se pretende localizar con el valor del mismo campo correspondiente a cada registro leído del fichero.
- No permite la inserción de nuevos registros.
- No permite la eliminación de registros.

### Ficheros de organización relativa

Son aquellos ficheros caracterizados porque:

- El almacenamiento de los registros sobre el soporte seleccionado (memoria auxiliar) se realiza a través de un identificador o clave que indica por una parte la posición del registro dentro del fichero y por otra la posición de memoria donde está ubicado.
- La dirección de almacenamiento del registro dentro del soporte utilizado se obtiene siempre del identificador o clave del propio registro, sabiendo que:
- Si la clave no es numérica, es decir, alfabética o alfanumérica, se aplican algoritmos o fórmulas de transformación (matemáticas general mente) para obtener valores enteros positivos que facilitan su posterior manejo y tratamiento.
- Si la clave es numérica se aplica un algoritmo de transformación para obtener un rango de valores comprendidos entre el intervalo de valores de las direcciones de memoria disponibles, estableciendo así una relación directa entre dirección lógica y dirección física.

El algoritmo de transformación que es aplicado a la clave debe cumplir tres condiciones:

- Aprovechar al máximo el espacio disponible en memoria.
- Establecer una correspondencia directa entre dirección lógica (clave) y dirección física (memoria).
- Producir el menor número de registros que con diferentes claves generan idénticas direcciones de almacenamiento tras aplicar el algoritmo de transformación.
- Los algoritmos de transformación utilizados en la conversión de claves reciben la denominación de **Hashing**. Existen dos variantes de la organización relativa denominados *directa* y *aleatoria o indirecta*.

### Organización directa

Este tipo de organización se da en aquellos casos en los que las claves son numéricas, estableciéndose una correspondencia directa entre dirección lógica y dirección física. Al ser la clave de tipo numérico no surgen problemas, pues los registros se ubican en direcciones de memoria de tipo numérico entero, lo que facilita establecer una correspondencia directa entre la clave y la dirección de memoria; de todo ello, podemos deducir que la secuencia lógica de almacenamiento de los registros en el fichero coincide con la secuencia física de almacenamiento de los registros sobre el soporte utilizado.

El valor de la clave siempre está en relación con la capacidad máxima del soporte físico utilizado para el almacenamiento, por lo que nunca podemos almacenar un registro cuya clave esté por encima de los límites máximos del fichero.

Utilizando este tipo de organización, cada dirección solo puede ser ocupada por un registro. El hecho de existir más de un registro con la misma clave es causa de error ya que ello supondría la posibilidad de almacenar en el fichero un registro repetido, lo cual no es posible en este tipo de organización. Este hecho recibe el nombre de ***sinónimo o colisión***.

Las ventajas más destacables en este tipo de organización son:

- Permite acceder a los datos de dos formas diferentes.
  - Directamente, mediante el identificador o clave del fichero.
  - Secuencialmente, partiendo siempre del primer registro almacenado en el fichero.
- Permite realizar operaciones de escritura/lectura simultáneamente.
- Son muy rápidos en el tratamiento de registros individuales.

Los inconvenientes mas destacables en este tipo de organización son:

- Una consulta total del fichero puede suponer un gran inconveniente, pues al realizar un acceso secuencial sobre los datos del fichero estamos obligados a analizar posición por posición desde la primera hasta la ultima pudiendo ocurrir que algunas posiciones estén vacías, lo cual implica una considerable perdida de tiempo, por lo que es necesario el empleo de técnicas avanzadas de programación para realizar lecturas secuenciales.
- Deja gran cantidad de huecos (posiciones libres de memoria) dentro del fichero. Estos huecos surgen debido a que los identificadores de los registros que se desean almacenar después de haber sido introducidos en el fichero indican posiciones de almacenamiento probablemente no contiguas, lo que implica un desaprovechamiento del soporte de almacenamiento o memoria auxiliar, respecto al numero real de registros almacenados.

Se deben dar soluciones al problema de los sinónimos o colisiones.

### **Organización aleatoria o indirecta**

Esta variante de la organización relativa se da en aquellos casos en los que la clave debe sufrir un proceso de conversión que permita obtener un valor numérico entero, facilitando así la correspondencia directa que se debe establecer entre la clave y la dirección de memoria. En este caso, la secuencia lógica de almacenamiento no coincide con la secuencia física.

El valor de la clave siempre debe estar en relación con la capacidad máxima del soporte físico utilizado para el almacenamiento de la información; por este motivo nunca podremos almacenar registros cuya dirección de almacenamiento este por encima de los límites máximos del fichero y, si así ocurriese, resultaría que el algoritmo de almacenamiento no seria correcto.

En este tipo de organización, cada dirección puede ser ocupada por mas de un registro, pues este hecho no es causa de error, ya que ello se debe a que el algoritmo de transformación que hemos aplicado al identificador o clave (en este caso alfabética o alfanumérica) ha generado con distintos identificadores la misma posición de almacenamiento en memoria, lo cual si es posible en esta variante de organización relativa. Este hecho recibe el nombre de *sinónimo o colisión*.

Las ventajas mas destacables en este tipo de organización son:

- Permite un acceso inmediato a los registros haciendo únicamente referencia a su clave.
- No requieren procesos u operaciones de ordenación.
- Este tipo de organización permite realizar operaciones de escritura/lectura a la vez.
- Son muy rápidos en el tratamiento individual de registros.
- Permite acceder secuencialmente a los datos siempre que se desee.

Los inconvenientes mas destacables en este tipo de organización son:

- Las consultas completas del fichero pueden resultar excesivamente lentas.
- De la misma forma que ocurría con la organización directa, este tipo de organización deja gran cantidad de huecos o espacios libres dentro del fichero.

- Tanto el algoritmo para la transformación o conversión de las claves como el algoritmo necesario para el almacenamiento y tratamiento de sinónimos corren a cuenta del programador, siendo responsabilidad de este aplicar un método que deje el menor número de huecos libres y genere el menor número de sinónimos.

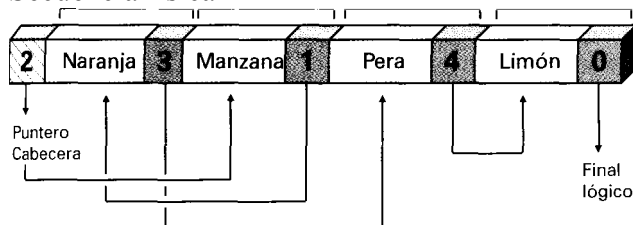
### Variantes de la organización secuencial

Existen tres variantes de la organización secuencial, que surgen como posible alternativa a las desventajas e inconvenientes que presentan los tipos de organización anteriormente estudiados; es decir, por una parte para mejorar la mala gestión de los ficheros que presenta la organización secuencial y por otro lado para evitar la separación entre secuencia lógica y secuencia física de almacenamiento de los registros sobre el soporte de almacenamiento que presenta la organización relativa.

### Organización secuencial encadenada

Son ficheros de organización secuencial gestionados con la ayuda de punteros, es decir, campos que contienen la dirección donde se encuentra ubicado cada uno de los registros del fichero, siendo considerado como un campo adicional que forma parte integrante de la estructura del registro y que tiene la capacidad de indicar cual es el siguiente o el anterior registro en secuencia lógica y no en secuencia física.

Secuencia física



En este tipo de ficheros las inserciones de nuevos registros siempre se realizan al final, pues los registros se almacenan secuencialmente según el orden de llegada. Por este motivo la secuencia física y la secuencia lógica no coinciden en este tipo de organización y puede ocurrir que el último registro en secuencia física sea el primero en secuencia lógica y viceversa, mientras que las eliminaciones de registros deterioran progresivamente el fichero ya que el espacio de los registros borrados no puede ser ocupado por otros nuevos, pues dicho espacio no es liberado físicamente, ya que el borrado se efectúa marcando el registro de manera que sea ignorado, aunque siga existiendo.

### Organización secuencial indexada

Este tipo de organización surge con el propósito de subsanar los inconvenientes que presenta la organización secuencial y relativa, pero aprovechando al mismo tiempo las ventajas de ambas (tratamiento de grandes volúmenes de información en la organización secuencial y acceso rápido y directo a los registros en la organización relativa).

Se caracteriza principalmente porque la información almacenada en un fichero con este tipo de organización se gestiona mediante la utilización de tablas cuyos elementos contienen las direcciones de los datos a los que se puede acceder secuencialmente.

Este tipo de ficheros está formado por tres zonas o áreas denominadas área primaria, área de índices y área de excedentes.

- Área primaria. Es la zona donde están contenidos los registros ordenados ascendentemente por el valor de su clave. Esta zona o área del fichero se encuentra segmentada, es decir, dividida en segmentos, donde cada segmento almacena un bloque de N registros, todos ellos consecutivos y almacenados en posiciones de memoria físicamente contiguas.

Se caracteriza por ser un área de organización secuencial, donde el acceso a cada registro se realiza en una doble operación que consiste en:

1. Acceder directamente al segmento donde se haya ubicado el registro buscado.
  2. Una vez localizado el segmento, accedemos secuencialmente a los registros en el contenidos hasta localizar el registro buscado, o hasta alcanzar el final del segmento en caso de no hallarse el registro deseado.
- **Área de índices.** Este área se caracteriza principalmente porque tiene la misma estructura y cualidades que un fichero de organización secuencial, pero con la característica de que los registros que la forman están constituidos por solo dos campos. El primer campo contiene la clave del ultimo registro de cada segmento mientras que el segundo campo contiene la dirección de entrada a cada uno de los segmentos.
  - **Área de excedentes.** De la misma forma que en los ficheros de organización relativa aleatoria existía una zona del fichero, normalmente situada al final del mismo, denominada **zona de sinónimos**, en ficheros con organización secuencial indexada existe una zona o área de similares características denominada **área de excedentes u overflow**, destinada a albergar todos aquellos registros que no han tenido cabida en el área primaria, por ser sus claves intermedias a la de los registros previamente almacenados en dicho área.

En este tipo de organización, el acceso a los registros se hará previa consulta a la tabla o área de índices, para determinar el segmento donde se encuentra el registro buscado dentro del fichero.

Para consultar un fichero secuencial indexado los pasos serian los siguientes:

1. Consultamos el área de índices secuencialmente para localizar el segmento donde se halla el registro que queremos buscar.
2. Una vez localizada la dirección de entrada al segmento, vamos al área primaria y accedemos directamente, situándonos en el primer registro de dicho segmento.
3. Realizamos una consulta secuencial del segmento hasta localizar el registro deseado o alcanzar el final del mismo.
4. En el supuesto de que el registro buscado no se hallase en el área primaria, una vez recorrido todo el segmento accederíamos al área de excedentes u overflow, para así determinar su posible localización en dicho área. Los ficheros de organización secuencial indexada son eficaces tanto en consultas esporádicas como en consultas completas o totales del fichero.

### Organización secuencial indexada-encadenada

Este tipo de organización aprovecha lo mejor de las dos variantes de organización secuencial anteriormente vistas (encadenada e indexada).

Se caracteriza principalmente por la utilización de punteros e índices de forma simultanea, lo que implica un considerable aumento del espacio ocupado en memoria para la implementación de índices y campos puntero, pero proporciona una gran rapidez en la búsqueda de registros.

En la eliminación de registros se generan huecos que realmente son posiciones de memoria ocupadas por registros marcados, pero que no han sido eliminados físicamente del fichero. La única posibilidad de eliminar dichos huecos es en futuras operaciones en las que necesitemos reorganizar el fichero.

En el caso de inserciones de nuevos registros, estas se hacen directamente sobre la zona de desbordamiento, pues al ser variantes de la organización secuencial mantienen las

mismas características y cualidades, no permitiendo la inserción de nuevos registros en el área primaria después de la creación del fichero.

Estos ficheros deben ser reorganizados periódicamente por dos motivos:

1. La eliminación de registros aumenta considerablemente el tamaño de los ficheros, ya que no se eliminan físicamente del dispositivo de almacenamiento impidiendo así liberar el espacio de memoria ocupado.
2. Las inserciones generan un área de excedentes excesivamente grande.

## TRATAMIENTO DE FICHEROS SECUENCIALES

Seguidamente estudiaremos la nomenclatura utilizada en el diseño de algoritmos correspondiente a cada una de las principales operaciones que se pueden realizar sobre un fichero.

### Definición del fichero

Consiste en definir una variable mediante un identificador (que será utilizado en el programa como nombre lógico del fichero) y un tipo de dato *Fichero* (que indica su organización). Se deberá definir una estructura de datos de tipo *Registro* en la que se especificaran los campos que componen dicha estructura y que será utilizada para almacenar en memoria la información contenida en cada registro del fichero. La sintaxis es:

```
<Nombre_fichero> Fichero <Tipo de organización>
<Nombre_Registro> Registro
    <Nombre_Campo1> Tipo de Dato
    <Nombre_Campo2> Tipo de Dato
    ...
    < Nombre_CampoN> Tipo de Dato
Fin Registro
```

Por ejemplo, si se tiene un fichero de empleado llamado *F\_empleado* de organización secuencial y con los siguientes campos en el registro: *Cod\_emp*, *Nomb\_emp*, *Dir\_emp* y *Sueldo*, se definiría así:

```
F_empleado Fichero Secuencial
R_empleado Registro
    Cod emp Numérico
    Nomb emp Alfanumérico(20)
    Dir emp Alfanumérico(30)
    Sueldo emp Numérico
FinRegistro
```

### Asignación física

Consiste en asociar el nombre lógico del fichero (definido en el programa) con un nombre físico empleado para almacenar la información del fichero en un determinado soporte.

```
<Nombre_fichero> = "Nombre físico"
```

Por ejemplo, para asociar el fichero definido en el apartado anterior al archivo *Emple.dat*, se escribiría la siguiente instrucción:

```
F empleado = "Emple.dat"
```

En la cadena de caracteres utilizada para el nombre físico del fichero se puede especificar la trayectoria o camino donde se encuentra ubicado dicho fichero.

### Apertura del fichero

Consiste en dejar disponible para el programa la información contenida en un fichero físico previamente asignado a un nombre lógico de fichero. Esta operación requiere que se especifique el modo de apertura (tipo de acceso) del fichero, que no es más que la

forma en la que pretendemos acceder a los datos. Existen los siguientes modos de apertura:

Lectura.- Permite pasar información del fichero a la memoria. Este modo de apertura requiere que el fichero exista.

Escritura.- Permite pasar información de la memoria al fichero. Si el fichero no existe, se crea con el nombre físico asignado y, si existe, la información contenida en él se destruye.

Añadir.- Permite pasar información de la memoria al fichero a partir del último registro almacenado en el caso de que el fichero exista. Si el fichero no existe lo crea con el nombre físico asignado.

Lectura/Escritura.- Permite pasar información desde el fichero a la memoria y viceversa. El fichero debe existir.

La sintaxis de la instrucción de apertura de un fichero es:

```
Abrir <Nombre_fichero> Modo <apertura>
```

Por ejemplo, para abrir el fichero de empleados anterior:

```
Abrir F_empleado lectura
```

### **Cierre del fichero**

Consiste en dejar inaccesible la información contenida en un fichero, quedando el nombre lógico desconectado del nombre físico del fichero. La sintaxis es:

```
Cerrar <Nombre_fichero>
```

Por ejemplo, para cerrar el fichero de empleados:

```
Cerrar F_empleado
```

### **Escritura en el fichero**

Consiste en traspasar la información previamente almacenada en memoria a un fichero de datos. Su sintaxis es:

```
Escribir <Nombre_registro> en <Nombre_Fichero>
```

Por ejemplo, para escribir el registro de empleado en el fichero:

```
Escribir R_empleado en <F_empleado>
```

### **Lectura del fichero**

Consiste en traspasar la información contenida en un fichero a la memoria en una estructura de datos tipo registro, donde será tratada posteriormente. La lectura se inicia al principio del fichero y, cada vez que se ejecuta dicha sentencia, la cabeza de lectura del soporte se sitúa al principio del siguiente registro. Su sintaxis es:

```
Leer <Nombre_registro> de <Nombre_fichero>
```

Por ejemplo, para leer un registro del fichero de empleados

```
Leer R_empleado de F_empleado
```

La lectura del fichero se puede realizar hasta llegar al final del mismo, siendo posible detectar dicho final por medio de una función  $FF(Nombre\ fichero)$ , que retorna el valor lógico *verdadero* cuando se alcanza el final del fichero.

### **Consultas**

Para la realización de una consulta, hay que tener en cuenta si el fichero está ordenado o desordenado y si la búsqueda es de un solo registro o de todos los registros que reúnan una determinada condición. En los algoritmos seguidamente desarrollados, se supone que tenemos definido un fichero cuyo nombre lógico es *Fich*, utilizando un registro de nombre *R Fich*, siendo la condición establecida que el contenido del *campo n* sea igual a un valor introducido por teclado. El fichero *Fich* está asignado a un fichero físico y abierto para lectura.

#### **Consulta de un fichero secuencial desordenado**

Cuando se quiera encontrar la primera ocurrencia de un registro.

```

Leer valor
Encontrado <- falso
Mientras NOT FF(Fich) y NOT encontrado
  Leer R_Fich de Fich
  Si campo_n = valor
    Encontrado <- verdadero
  FinSi
FinMientras
Si encontrado
  Visualizar R_Fich
Sino
  Visualizar "no encontrado"
FinSi

```

Si lo que se quiere es encontrar todas las ocurrencias de un registro

```

Leer valor
Encontrado <- falso
Mientras NOT FF(Fich)
  Leer R_Rich de Fich
  Si campo_n = valor
    Visualizar R_Fich
    Encontrado <- verdadero
  FinSi
FinMientras
Si NOT encontrado
  Visualizar "no encontrado"
FinSi

```

### **Consulta en un fichero secuencial ordenado**

Para buscar el primer registro

```

Leer valor
salir <- falso
Mientras NOT FF(Fich) AND NOT salir
  Leer R_Fich de Fich
  Si campo_n >= valor
    salir <- verdadero
  FinSi
FinMientras
Si salir AND campo_n = valor
  Visualizar R_Fich
Sino
  Visualizar "no encontrado"
FinSi

```

Si lo que se quiere es encontrar todas las ocurrencias de un registro

```

Leer valor
salir <- falso
encontrado <- falso
Mientras NOT FF(Fich) AND NOT salir
  Leer R_Fich de Fich
  Si campo_n > valor
    salir <- verdadero
  Sino
    Si campo_n = valor

```

```

        Visualizar R_Fich
    encontrado <- verdadero
FinSi
FinSi
FinMientras
Si NOT encontrado
    Visualizar "No encontrado"
FinSi

```

### Partición de un fichero

Consiste en obtener varios ficheros partiendo de un fichero origen. Esta operación se puede realizar de dos formas:

Supongamos que disponemos de un fichero de nombre: *fichero\_origen*, que deseamos partir en dos ficheros de nombre 'Fich-1 ' (conteniendo los registros que en su campo\_n tengan un valor determinado) y Fich-2 (conteniendo el resto de los registros).

```

Abrir fichero_origen Lectura
Abrir Fich-1 Escritura
Abrir Fich-2 Escritura
Leer valor
Mientras NOT FF(fichero_origen)
    Leer registro_origen de fichero_origen
    Si campo_n = valor
        Escribir registro_origen en Fich-1
    Sino
        Escribir registro_origen en Fich-2
    FinSi
FinMientras
Cerrar fichero_origen
Cerrar Fich-1
Cerrar Fich-2

```

### Rupturas de control

Consiste en realizar un tratamiento del fichero por grupos homogéneos, produciéndose una ruptura de control cuando se lee un campo de control de un registro con un contenido distinto del registro anterior. El fichero debe estar ordenado en secuencia según los campos de control establecidos para realizar las rupturas.

### Actualización

Consiste en realizar altas, bajas y modificaciones de los registros de un fichero. Se pueden considerar esencialmente dos métodos para la actualización de ficheros secuenciales:

#### Actualización por lotes

Este método consiste en actualizar un fichero origen ordenado por un campo clave con un lote de transacciones contenidas en un fichero de movimientos con la misma ordenación que el fichero origen y que contiene un campo que indica el tipo de movimientos (alta, baja o modificación). La actualización por lotes implica que la información que no sea baja o errónea pasara a un fichero destino con los registros ordenados y actualizados. Los errores producidos en la actualización pueden ser enviados a una impresora o escritos en un soporte de almacenamiento para su posterior tratamiento.

#### Actualización interactiva

Consiste en actualizar un fichero con las transacciones introducidas por teclado, produciéndose las operaciones de altas, bajas o modificaciones de los registros según



una selección del tipo de movimiento presentado en pantalla. Se debe utilizar un fichero temporal para reorganizar el fichero cuando se produce una baja o se pretenden borrar registros marcados. Si el fichero esta ordenado, las altas producen una inserción en el lugar que le corresponde, debiéndose utilizar un fichero temporal para reorganizar el fichero y, si el fichero esta desordenado, las altas se realizan añadiendo registros al fichero. Los errores que se efectúan en la actualización se presentan en pantalla a medida que se producen.

# 6

## Estructuras dinámicas internas de datos

Las estructuras estáticas son válidas cuando se conoce previamente la cantidad de datos que se habrán de utilizar durante la ejecución del programa. Sin embargo, cuando no es así, es necesario disponer de métodos para ocupar posiciones adicionales de memoria a medida que se vayan necesitando y de liberarlas cuando no se necesiten. Estas variables que se crean y se destruyen durante la ejecución de un programa se denominan variables dinámicas y se utilizan para crear estructuras dinámicas de datos que se pueden ampliar y comprimir durante la ejecución de un programa.

Al no conocerse de antemano el tamaño que tendrán durante la ejecución, no es posible reservar una cantidad fija de memoria para su ubicación. Como para utilizar una variable en un programa es necesario que ésta ocupe posiciones de memoria principal, existen técnicas que tratan de solucionar este problema, la más extendida de las cuales es la asignación dinámica de memoria, que consiste en reservar durante la traducción del programa a una cantidad de memoria fija para almacenar la dirección inicial de cada variable, realizándose una asignación dinámica de memoria conforme las variables se crean o modifican su tamaño durante la ejecución del programa.

Una estructura dinámica de datos es una colección de elementos denominados nodos de la estructura (normalmente de tipo compuesto) que son enlazados juntos mediante un tipo de dato específico denominado *puntero* y que pueden crecer y contraerse durante la ejecución del programa.

### PUNTEROS

Un puntero es básicamente una variable estática de tipo entero utilizada para contener la dirección de memoria de otra variable.

Los punteros se utilizan principalmente para realizar operaciones con estructuras dinámicas de datos, es decir, estructuras creadas en tiempo de ejecución, siendo su objetivo el de permitir el manejo o procesamiento (creación, acceso, eliminación, etc.) de estas estructuras de datos.

#### Operaciones básicas con punteros

##### Definición

Consiste en reservar el suficiente espacio en memoria principal para almacenar un dato de tipo numérico entero, es decir, una dirección de memoria. Su sintaxis es:

```
<nombre_puntero> puntero a Tipo_Dato
```

donde <nombre\_puntero> es el nombre de la variable de tipo puntero y Tipo\_Dato es el tipo de dato de la variable que apunta.

Inicialmente la variable de tipo puntero no contendrá ningún valor, es decir, su valor será indeterminado si previamente no ha sido inicializada con un valor concreto.

##### Inicialización

Con esta instrucción de asignación, indicamos que la variable puntero no apunta a ninguna dirección de memoria. Si sintaxis es:

```
<nombre_puntero> <- nulo
```

##### Creación de una variable dinámica

Consiste en hacer que la variable puntero apunte a una dirección de memoria libre a la que posteriormente se le asignara un dato o valor y que deberá ser del mismo tipo que el definido para la variable puntero. Para realizar esta operación requerimos de un

procedimiento que reserve la memoria libre necesaria para dicho puntero. Su sintaxis es:

```
Reservar (<nombre_puntero>)
```

donde <nombre\_puntero> es la variable tipo puntero a la que apunta la zona de memoria reservada.

### **Destrucción de una variable dinámica**

Consiste en hacer que la variable puntero deje de apuntar a la dirección de memoria asignada a la variable dinámica, liberando el espacio ocupado por esta en memoria principal. Su sintaxis es:

```
Liberar (<nombre_puntero>)
```

donde <nombre\_puntero> es la variable tipo puntero a la que apunta la zona de memoria que se va a liberar.

### **Acceso a una variable dinámica**

Con el nombre de la variable puntero accederemos a un valor *numérico entero*, correspondiente a la dirección de memoria de la variable dinámica referenciada.

Con el nombre de la variable puntero precedido del prefijo "Val-" (valor) accedemos o referenciamos indirectamente al dato contenido en una variable dinámica. Su sintaxis es:

```
Val-<nombre_puntero>
```

Para almacenar un dato en una variable dinámica referenciada por una variable puntero debemos efectuar la siguiente operación de asignación:

```
Val-ptr1 <- 45
```

Se ha asignado el valor 45 a la variable que es apuntada por el puntero ptr1.

Hasta este momento hemos visto todas aquellas operaciones básicas que podemos realizar en el manejo y tratamiento de una variable puntero pero, cuando operamos con dos o mas variables de este tipo, podemos realizar otras operaciones de especial relevancia y que son las mostradas a continuación:

```
ptr1 <- ptr2
```

Esta instrucción de asignación, hace que la variable puntero ptr1 apunte a la misma variable dinámica que la variable puntero ptr2.

```
Val-ptr1 <- Val-ptr2
```

Esta instrucción de asignación hace que la variable dinámica apuntada por ptr1 tenga el mismo valor que la variable dinámica apuntada por ptr2.

## **LISTAS**

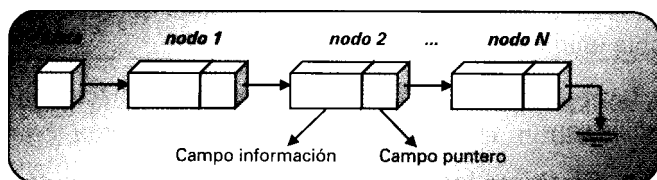
Una lista es una estructura de datos dinámica formada por un conjunto de elementos, denominados **nodos**, del mismo tipo y almacenados en memoria principal siguiendo una secuencia lógica.

### **Listas enlazadas o encadenadas**

Son aquellas listas cuyos elementos se encuentran almacenados en posiciones de memoria no contiguas. En este tipo de listas, los nodos tienen la estructura de auténticos registros divididos en unidades de tratamiento más pequeñas denominadas campos.

Cada nodo esta formado por un mínimo de dos campos:

- **Campo información**, que es el campo que contiene el dato.
- **Campo puntero siguiente**, que es el campo que actúa de enlace con el siguiente nodo de la lista en secuencia lógica.



Por convenio:

- Cuando se crea una lista, esta debe estar inicialmente vacía.
- El campo puntero correspondiente al ultimo nodo de la lista apunta a un valor Nulo.
- Para acceder a un nodo de la lista, se parte del nodo inmediatamente anterior, a excepción del primer nodo, al cual se accede a través de un enlace especial o puntero externo a la lista.
- Las operaciones que podemos realizar sobre una lista enlazada o encadenada son las mismas que podemos realizar sobre una lista contigua, con la diferencia de que al no encontrarse los nodos situados en posiciones adyacentes de memoria, las operaciones de inserción o eliminación requieren un tratamiento especial, pero al mismo tiempo, mas rápido y sencillo.

Seguidamente se muestran los algoritmos correspondientes a las principales operaciones que se pueden realizar sobre una lista enlazada, dadas las siguientes definiciones:

#### **Definición de la estructura de los nodos de la lista.**

```
Nodo Registro
    <variable_información>:Tipo Dato
    <siguiente> puntero a Nodo
FinRegistro
```

#### **Enlace o puntero externo a la lista (primer nodo).**

```
<nombre_lista> puntero a Nodo
```

#### **Crear una lista**

Consiste en declarar la variable puntero que apuntará al primer nodo de la lista. Como la lista estará vacía inicialmente, se inicializa la variable. Su sintaxis es:

```
<nombre_lista> <- nulo
```

#### **Búsqueda de un elemento en una lista ordenada**

Las búsquedas se realizan partiendo del primer nodo o nodo inicial hasta alcanzar el nodo buscado o un nodo que tiene un valor del campo información mayor que el buscado.

#### **Añadir o insertar un nodo a la lista**

Consiste en crear un nuevo nodo e insertarlo en la lista en el lugar que le corresponda según el orden establecido en la misma. Existen varias posibilidades en función del lugar que ocupe el nodo nuevo insertado:

1. Insertar el primer elemento
  - a. Poner el puntero inicial con la dirección del elemento que se inserta
  - b. Poner el puntero del elemento insertado apuntando al que era el primero
2. Insertar el último elemento
  - a. Poner el puntero del elemento insertado con valor nulo
  - b. Poner el puntero del elemento anterior (que era el último) apuntando al insertado
3. Insertar un elemento intermedio
  - a. Poner el puntero del elemento insertado apuntando al siguiente
  - b. Poner el puntero del elemento anterior apuntando al insertado

### Eliminar o borrar un elemento de la lista

Dependiendo del lugar que ocupe el nodo a borrar en la lista, existen varias posibilidades:

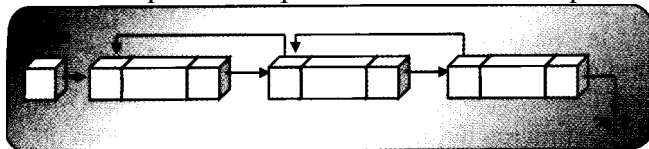
1. Supresión del primer elemento.- Basta con modificar el valor del puntero inicial, que tomará el valor del puntero del elemento siguiente al primero.
2. Supresión del último elemento.- Basta con poner a nulo el puntero siguiente del penúltimo nodo.
3. Supresión de un nodo intermedio.- Basta con poner el puntero del elemento anterior con el valor del puntero del elemento siguiente al eliminado.

### Recorrido secuencial de una lista

Consiste en recorrer uno a uno y en orden los elementos de la lista y realizar algún tratamiento con el campo información de cada uno de ellos.

### Listas doblemente enlazadas

Hemos visto que las listas simplemente enlazadas se recorren de izquierda a derecha partiendo del primer elemento o nodo de la lista sin existir la posibilidad de retroceder, es decir que únicamente podemos recorrerla en un solo sentido (de izquierda a derecha). Este inconveniente es solventado mediante el use de listas doblemente enlazadas, que son aquellas que pueden recorrerse en ambas direcciones gracias a que los nodos que la componen están formados por tres campos:



- **Campo información**, que contiene el dato del nodo.
- **Campo puntero anterior**, un campo puntero o enlace que apunta al nodo anterior. Este puntero permite retroceder o recorrer la lista hacia atrás (de derecha a izquierda).
- **Campo puntero siguiente**, un segundo campo puntero o enlace que apunta al nodo siguiente de la lista.

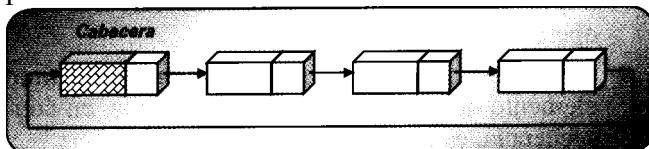
La desventaja que presentan este tipo de listas, es que, aun conteniendo la misma información que por ejemplo una lista simplemente enlazada, cada nodo ocupa mas espacio en memoria al estar constituido por un segundo campo puntero.

Las operaciones que podemos realizar sobre una lista doblemente enlazada son las mismas que podemos realizar sobre una lista contigua o una lista simplemente enlazada, con la diferencia de que las operaciones de inserción o eliminación requieren un tratamiento especial y algo mas complejo que estas ultimas, al disponer cada nodo de dos punteros.

### Listas circulares

Las listas circulares se caracterizan porque el campo puntero del ultimo nodo, en lugar de apuntar a un valor nulo, apunta al primer nodo o elemento de la lista, convirtiéndose así en una estructura de datos circular.

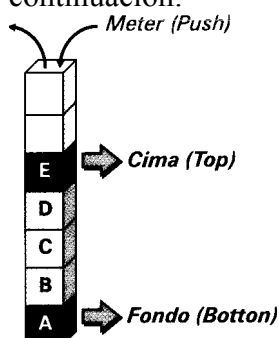
La ventaja que este tipo de listas ofrece es la de permitir el acceso a un nodo a partir de cualquier otro nodo perteneciente también a la lista. Por el contrario, el inconveniente que presentan este tipo de listas es el de mantener un nodo que se diferencie del resto y que sea identificado como un nodo cabecera, para evitar que se produzcan bucles infinitos en el tratamiento de dicha estructura de datos.



## Pilas

Una pila es un grupo ordenado de elementos homogéneos (todos del mismo tipo) en la cual los elementos solo pueden ser añadidos o eliminados por un extremo llamado cabecera o cima (top) de la pila. Esto significa que los elementos se sacan en orden inverso al seguido en el proceso de inserción. Por ello, una pila es considerada una estructura de datos LIFO (Last In First Out), es decir que el ultimo elemento que entra es el primero que sale.

Por definición, sobre una pila podemos realizar dos tipos de operaciones básicas que son **almacenamiento** (push) y **recuperación** (pop). Para manejar una estructura de datos tipo pila, un programador debe definir un conjunto de operaciones que permitan al usuario acceder y manipular los elementos en ella almacenados. Generalmente cada una de estas operaciones suele recibir un nombre fijo e identificativo de la operación que realiza, siendo la terminología utilizada mas común la mostrada a continuación:



- **Meter o Apilar:** Se denomina así a la operación que añade un elemento a la pila.
- **Sacar o Desapilar:** Es la operación que saca un elemento de la pila.
- **Inicializar pila:** Una vez creada la pila y antes de utilizarla, esta debe quedar inicialmente vacía.
- **Pila vacía:** Esta operación de tipo lógico es sumamente importante ya que antes de sacar un elemento debemos comprobar si la pila se encuentra vacía.
- **Pila llena:** Solo en aquellos casos en los que sea necesario determinar si la pila se encuentra llena antes de añadir un nuevo elemento debido a la implementación utilizada (solo en el caso de implementación con tablas), emplearemos esta operación de tipo lógico.
- **Cima:** Esta operación permite obtener el ultimo elemento que fue añadido a la pila y que, en consecuencia, es el primero en salir de ella.

Una pila puede ser implementada de dos formas diferentes:

1. Utilizando una tabla unidimensional.
2. Mediante el use de punteros.

### Implementación con punteros

La implementación con punteros se representa por medio de una lista enlazada en la cual cualquier inserción o eliminación de un elemento se realiza mediante un puntero (stack pointer), que señala continuamente la *cima* de la pila.

Cuando la pila se crea, el puntero o stack pointer tiene valor Nulo, pues inicialmente después de ser creada la pila se encuentra vacía. Conforme vayamos añadiendo elementos a la pila, *Cima* ira tomando el valor de la dirección de memoria donde se encuentra ubicado el ultimo nodo o elemento de la pila.

## Colas

Una cola es un grupo ordenado de elementos del mismo tipo, en la cual los elementos se añaden por un extremo (Final) y se quitan por el otro extremo (Frente).

Esto significa que los elementos se sacan en el mismo orden en el que fueron insertados o introducidos en la cola, siendo por ello considerada como una estructura de datos FIFO (First In First Out), es decir que el primer elemento en entrar es el primer elemento en salir. Un ejemplo ilustrativo de cola sería una cola de impresión donde todos los trabajos con igual prioridad lanzados a la impresora serán atendidos en un riguroso orden de llegada. Para manejar una estructura de datos de tipo cola, un programador debe definir un conjunto de operaciones que permitan al usuario acceder y manipular los elementos almacenados en ella. La terminología más común utilizada para referenciar este conjunto de operaciones es la mostrada a continuación:

- **Inicializar cola:** Esta operación deja inicialmente vacía la cola una vez creada.
- **Encolar:** Es la operación que añade un elemento al final de la cola.
- **Desencolar:** Es la operación que saca un elemento del frente de la cola.
- **Cola vacía:** Esta operación de tipo lógico determina si la cola se encuentra o no vacía antes de sacar un elemento de la misma.
- **Cola llena:** Solo en aquellos casos en los que sea necesario determinar si la cola se encuentra llena antes de añadir un nuevo elemento debido a la implementación utilizada (solo en el caso de implementación con tablas), emplearemos esta operación de tipo lógico.

Una cola, al igual que una pila, no está incorporada en la mayoría de los lenguajes de programación y puede ser implementada de dos formas diferentes:

1. Utilizando una tabla unidimensional.
2. Mediante el uso de punteros.

En la presente obra únicamente estudiaremos la implementación de colas mediante la utilización de punteros (listas enlazadas) dejando la implementación con tablas como ejercicio práctico para el alumno.

### Implementación con punteros

La implementación con punteros se representa por medio de una lista enlazada, y para controlar la inserción o eliminación de elementos utilizamos dos punteros externos denominados *Frente* (que apunta al primer elemento introducido en la cola) y *Final* (que apunta al último elemento introducido en la cola).

Cuando la cola se crea, los punteros *Frente* y *Final* tienen valor Nulo, ya que inicialmente la cola debe encontrarse vacía.

Conforme vayamos añadiendo elementos a la cola, *Frente* tomará el valor correspondiente a la dirección del primer elemento de la cola y *Final* tomará el valor correspondiente a la dirección del último elemento de la cola. De esta manera podemos determinar en todo momento el comienzo y al final de la misma.

## ESTRUCTURAS DINÁMICAS NO LINEALES

### Árboles

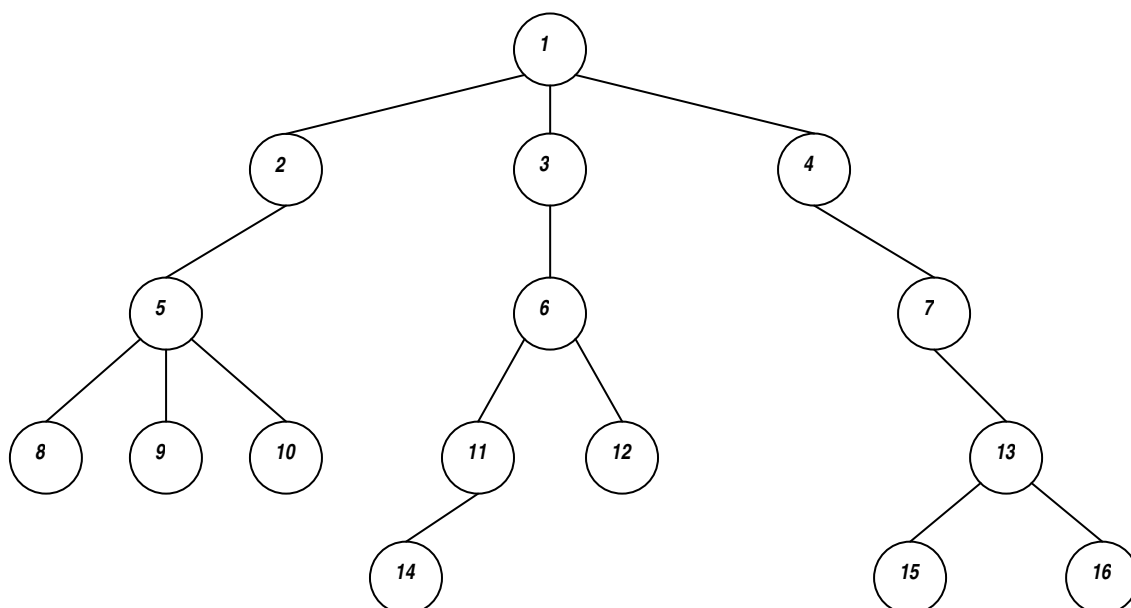
Hasta ahora todo lo que hemos visto han sido estructuras de datos lineales, pero en ocasiones se precisan estructuras de datos jerarquizadas (como, por ejemplo, árboles) en el desarrollo de determinados procesos informáticos.

Un árbol es una estructura de datos jerarquizada y no lineal constituida por un conjunto de elementos homogéneos, es decir del mismo tipo, que se caracterizan porque establecen una jerarquía entre los elementos que las forman.

La terminología utilizada en este tipo de estructuras de datos es la siguiente:

- **Raíz:** Es el primer elemento del árbol, caracterizado por ser el único que no dispone de elemento antecesor.
- **Nodo:** Es cada uno de los elementos que constituyen la estructura arborescente.

- **Nodo terminal:** también recibe el nombre de hoja y es aquel nodo que no dispone de descendencia o elemento sucesor.
- **Altura o profundidad:** Es el número de capas o niveles que tiene el árbol. El nodo raíz tiene el nivel 0 y los nodos originarios del nodo raíz tienen nivel I y así sucesivamente.
- **Antecesor:** también denominado ascendiente, es el nodo del cual son originarios otros nodos o elementos.
- **Sucesor:** también denominado descendiente, es el nodo o elemento dependiente de un nodo ascendiente.
- **Rama:** Es el camino que finaliza con un nodo terminal.
- **Recorrido o camino:** Son los enlaces establecidos en nodos consecutivos
- Por ejemplo, consideremos el árbol de la figura:



- Nodo raíz: 1
- Nodos o elementos: 1, 2, 3, 4, 5, 6,..., 16
- Nodos terminales: 8, 9, 10, 14, 12, 15, 16
- Altura o profundidad: 5
- Niveles:
  - Nivel 0: nodo 1
  - Nivel 1: nodos 2, 3 y 4
  - Nivel 2: nodos 5, 6 y 7
  - Nivel 3: nodos 8, 9, 10, 11, 12 y 13
  - Nivel 4: nodos 14, 15 y 16
- Un ejemplo de rama será la formada por los nodos 1, 3, 6, 11 y 14
- Por ejemplo, el antecesor de 5 es 2 y los sucesores son 8, 9 y 10

Los árboles se clasifican en función del número máximo de sucesores o descendientes de un nodo, de forma que un árbol donde cada nodo tiene un máximo de dos sucesores se denomina árbol binario; si tiene un máximo de tres sucesores se denomina árbol ternario y así sucesivamente.

### Árbol binario

Un árbol binario es una estructura de datos arborescente caracterizada porque cada uno de los nodos que forma parte integrante del árbol puede tener ninguno, uno o un



máximo de dos sucesores o descendientes, denominados descendiente izquierdo y descendiente derecho.

Se dice que un árbol binario es o está **equilibrado** si para cada uno de los nodos que constituyen dicha estructura, la altura de sus dos subárboles son iguales o se diferencian en uno y se dice que un árbol binario es **completo** cuando cada nodo tiene dos sucesores a excepción de los nodos terminales u hojas.

Los árboles binarios, del mismo modo que las pilas y las colas, pueden ser implementados mediante *listas enlazadas*, donde cada nodo estaría constituido por tres campos, uno de información y dos campos puntero, uno que haría referencia al descendiente izquierdo y otro al descendiente derecho, o bien, mediante *tablas*, para lo cual necesitaríamos de una matriz de tres columnas y  $n$  filas, donde cada fila almacenaría un nodo o elemento de dicha estructura.

### Recorrido de un árbol binario

Esta operación consiste en pasar por todos y cada uno de los nodos que forman parte del árbol una sola vez.

Las principales formas que existen de recorrer un árbol binario son:

1. Recorrido en **preorden**.
  - a. Consultar nodo raíz.
  - b. Recorrer subárbol izquierdo en preorden (nodo raíz-nodo izquierdo-nodo derecho).
  - c. Recorrer subárbol derecho en preorden (nodo raíz-nodo izquierdo-nodo derecho).
2. Recorrido en **inorden**.
  - a. Recorrido del subárbol izquierdo en inorden (nodo izquierdo-nodo raíz-nodo derecho).
  - b. Acceso al nodo raíz.
  - c. Recorrido del subárbol derecho en inorden (nodo izquierdo-nodo raíz-nodo derecho).
3. Recorrido en **postorden**.
  - a. Recorrido del subárbol izquierdo en postorden (nodo izquierdo-nodo derecho-nodo raíz).
  - b. Recorrido del subárbol derecho en postorden (nodo izquierdo-nodo derecho-nodo raíz).
  - c. Acceso al nodo raíz.

### Árbol binario de búsqueda

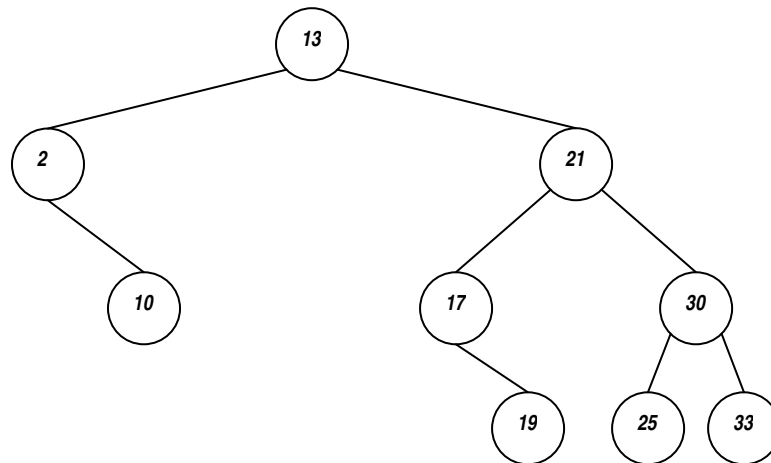
Es considerado como una variante especial del árbol binario y se caracteriza porque el valor de cualquier nodo debe ser superior al valor de cualquiera de los nodos del subárbol izquierdo e inferior que el valor de cualquiera de los nodos del subárbol derecho.

Este tipo de árboles es sumamente eficaz en la búsqueda o localización de un nodo concreto. Su uso permite obtener algoritmos de ordenación muy rápidos, facilitando operaciones tales como recorridos, inserciones o eliminaciones.

Los pasos que hay que seguir en la búsqueda de un nodo serían los siguientes:

1. Comparar el valor que deseamos buscar con el valor del nodo raíz.
2. Si el valor del nodo es mayor, continuamos la búsqueda por el subárbol izquierdo.
3. Si el valor del nodo es menor, continuamos la búsqueda por el subárbol derecho.

4. Repetir el paso anterior, hasta localizar el nodo con el valor deseado o localizar un nodo terminal, en cuyo caso se da por terminada la búsqueda al no existir el valor buscado.



Supongamos que en el árbol de la figura queremos buscar el valor 25. Los pasos que se deben seguir serían los siguientes:

Comparar valor buscado con el valor del nodo raíz  $25 > 13$ .

Como es mayor delimitamos la búsqueda al subárbol derecho y comparamos de nuevo  $25 > 21$ .

Al ser mayor delimitamos la búsqueda de nuevo al subárbol derecho y volvemos a comparar  $25 > 30$ .

Como en este caso la condición no es cierta, pues el valor buscado es menor, delimitamos la búsqueda al subárbol izquierdo.

Elemento encontrado.

# 7

## Lenguaje C

### INTRODUCCIÓN

El lenguaje C fue desarrollado por Dennis Ritchie en los Laboratorios Bell, en 1972. La idea inicial era crear un lenguaje de propósito general que facilitara la programación y la realización de muchas de las tareas anteriormente reservadas al lenguaje ensamblador. La evolución de los ordenadores y la creciente popularidad del lenguaje C en todo tipo de ordenadores dio lugar a la aparición de varias implementaciones de C con un alto grado de incompatibilidad entre ellas. Para resolver este problema, en 1983, el American National Standard Institute (ANSI) creó un comité para obtener una definición no ambigua del lenguaje C que fuera independiente de la arquitectura de cualquier máquina. De esta forma quedó definido en 1989 el estándar ANSI para el lenguaje C, conocido comúnmente como ANSI C.

### CARACTERÍSTICAS DEL LENGUAJE C

El lenguaje de programación C ha sido utilizado para el desarrollo de infinidad de herramientas de trabajo (sistemas operativos, compiladores, procesadores de texto, bases de datos, etc.). Mientras que otros lenguajes de programación se caracterizan por ser utilizados en áreas más concretas, el lenguaje C se caracteriza por no tener ninguna connotación sectorial, es un lenguaje de propósito general.

La ventaja más destacable es la portabilidad, es decir, la posibilidad de utilizarlo tanto en macroordenadores como en mini y microordenadores, lo que le sitúa entre uno de los lenguajes más portables.

Otra característica estrechamente relacionada con la anterior es la compatibilidad, que se da cuando el código escrito para una máquina concreta es fácilmente transferible a otro compilador y a otra máquina.

Las características más destacables del lenguaje C son:

- Lenguaje muy flexible.
- Muy apropiado para controlar rutinas hechas en lenguaje ensamblador.
- Permite generar programas de fácil modificación.
- Lenguaje predominante bajo cualquier máquina UNIX.
- Muy veloz y potente, lo que le permite la creación de software efectivo.
- Posibilita una programación estructurada y modular.
- Produce programas de código compacto y eficiente.
- Aunque es considerado un lenguaje de alto nivel, mantiene muchas características de los lenguajes de bajo nivel, por lo que puede ser clasificado como un lenguaje de nivel bajo-medio.
- Es un lenguaje compilado.

### IDENTIFICADORES

Los identificadores en C son nombres constituidos por una secuencia de letras, dígitos y el carácter subrayado que permiten hacer referencia a funciones, variables, constantes y otros elementos dentro de un programa.

Las reglas que hay que seguir para la construcción de un identificador en C son las siguientes:

- Deben comenzar obligatoriamente por un carácter alfabético (a-z, A-Z) o el signo de subrayado (\_).
- Siguiendo al primer carácter, pueden ser intercalados caracteres alfabéticos, el signo de subrayado y caracteres numéricos (0-9).
- El número de caracteres utilizados en la construcción del identificador va en función del compilador utilizado.
- No está permitido el uso de blancos intermedios.
- Las letras mayúsculas y minúsculas son interpretadas como caracteres diferentes.

El lenguaje C proporciona un listado de palabras claves o reservadas que no debemos emplear para la construcción de identificadores. Estas son:

auto	double	int	struct
break	else	long	
switch			
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

## TIPOS DE DATOS BÁSICOS

Los tipos de datos básicos en C son:

- Entero **int**
- Real de simple precisión **float**
- Real de doble precisión **double**
- Carácter **char**
- Vacío (sin valor) **void**

### Tipo entero

Se define con la palabra clave **int** y se utiliza para representar números enteros con signo. El rango de representación para este tipo de dato depende del tamaño de la palabra del procesador de la máquina donde se va a ejecutar el programa, así como del modificador empleado.

- Para procesadores de 16 bits el rango es [-32768, 32767]
- Para procesadores de 32 bits el rango es [-2147483648, 2147483647]

Una constante entera se compone de una serie de dígitos precedidos por un signo + o un signo -. Si la constante es positiva se puede omitir el signo.

Normalmente los dígitos se expresan en el sistema de numeración decimal, pero pueden expresarse utilizando el sistema de numeración octal y hexadecimal. Para expresar una constante en octal se debe utilizar como primer dígito el cero y para expresarla en hexadecimal se deben preceder los dígitos y letras representativos de la cantidad correspondiente con un cero y una equis mayúscula o minúscula.

### Tipo de dato real

Se define con la palabra clave **float** para punto flotante de simple precisión y **double** para punto flotante en doble precisión. Se utiliza para representar los números reales. Su rango de representación es:

- Para float [3.4E-38, 3.4E38]

- Para double [1.7E-308, 1.7E308]

Para definir una constante real se emplean dos tipos de notaciones:

- Notación decimal, con el siguiente formato: [signo]  
Parte\_entera.Parte\_decimal
- Notación científica o exponencial, con el siguiente formato: [signo]  
Mantisa E exponente. La E puede ser minúscula. El número sería la Mantisa multiplicada por la base exponencial (en este caso 10) elevada al exponente.

### Tipo de dato carácter

Se define la palabra clave **char** y se utiliza para representar un carácter perteneciente a un determinado código utilizado por el ordenador. Los códigos utilizados son el ASCII de 8 bits. Internamente un dato tipo char es un número entero que se corresponde con el código utilizado y que externamente tiene una representación en forma de carácter.

Una constante de tipo char se expresa encerrando el carácter en cuestión entre comillas simples. La cadena de caracteres es una serie de caracteres encerrada entre comillas dobles.

Existen unos caracteres especiales, denominados **secuencias de escape**, que tienen un significado especial. Estos son:

Código	Significado
\a	pitido
\b	retroceso
\n	nueva línea
\r	retorno de carro
\t	tabulador horizontal
\'	comilla simple
\"	comillas dobles
\0	carácter nulo
\\	barra invertida

Cuando una secuencia de escape va dentro de una cadena de caracteres no va encerrada entre comillas simples.

### Tipo de dato vacío

Se define con la palabra clave **void** que significa que el dato es nulo, por lo que no ocupa espacio en memoria y por tanto su tamaño en bytes es cero. Este tipo de datos no es aplicable a variables que no sean de tipo puntero. Se utiliza principalmente para:

- Definir un puntero genérico (no se conoce el tipo de dato de la variable que hay que apuntar).
- Especificar que una función no retorna de forma explícita ningún valor.
- Declarar que una función no utiliza parámetros.

## MODIFICADORES DE TIPO BÁSICO

Existen unas palabras clave usadas para modificar el significado los tipos básicos anteriores, en cuanto al signo y al rango de representación de datos. Estas son:

### Modificador short

Se emplea para la representación de números enteros pequeños con signo, y puede ocupar menos espacio de memoria que un *int*. No existen constantes de tipo *short int* ya que explícitamente las constantes enteras son de tipo *int*.

### Modificador long

Se emplea para números enteros más grandes de lo permitido con un *int* y puede ocupar más espacio en memoria, pero solo con procesadores de 16 bits. En procesadores de 32 bits es similar a un *int*.

Para especificar una constante entera de tipo *long*, cuando su valor está dentro del rango de un *int*, se le debe posponer la letra *L* o *l*.

En algunos compiladores se puede usar el modificador *long* para el tipo de datos *double*, lo que permite una precisión más alta.

### Modificadores signed

Se emplea para números enteros con signo, siendo el modificador de tipo por defecto para el tipo de dato *int*, por lo que normalmente no se usa.

### Modificador unsigned

Se emplea para números enteros sin signo, por lo que al eliminar el signo se gana un bit aumentando el rango de valores de [0,65535] en procesadores de 16 bits y de [0,4294967295] en procesadores de 32 bits.

Se puede especificar que una constante es de tipo *unsigned* posponiéndole la letra *U*.

Se pueden combinar el modificador *unsigned* con los modificadores *short* y *long*.

Para datos de tipo *char*, se puede utilizar los modificadores *signed* y *unsigned* para expresar que el valor almacenado es un número entero con signo o sin signo. El rango de valores representables es [-128,127] para el *signed char* (ASCII de 7 bits) y de [0,255] para el *unsigned char* (ASCII de 8 bits).

## DECLARACIÓN DE VARIABLES

Declarar una variable es reservar el suficiente espacio en la memoria para poder almacenar un valor. Solamente con la declaración no se tiene el valor deseado en memoria. Dicho valor se puede asignar por inicialización de la variable o durante la ejecución del programa. Los sitios donde pueden declararse las variables son:

- Dentro de una función (variable local).
- Fuera de toda función (variable global).
- En la lista de parámetros de una función (parámetro formal).

La sintaxis para declarar una variable es:

```
[Almacenamiento][Modificador] Tipo_dato_base  
Nombre_Variable;
```

Los elementos encerrados entre corchetes indican que son opcionales, los que no, son obligatorios. Estos son:

- **Clase de Almacenamiento.**- Especifica la forma en que se almacenará la variable. Existen cuatro especificadores para determinar la clase de almacenamiento: *auto*, *extern*, *static* y *register*. Por defecto, si no se especifica, se considera *auto* (variable local). Cuando se estudien próximamente las funciones en C, se verán con detalle las demás.
- **Modificador de tipo.**- Cambia el significado de un tipo básico, afectando esencialmente al signo y al tamaño reservado en memoria (rango de representación) para dicha variable. Existen cuatro modificadores de tipo: *signed*, *unsigned*, *short* y *long*. Por defecto, si no se especifica, se considera *signed* (con signo).

- **Tipo de dato básico.**- Es uno de los tipos de datos básicos vistos anteriormente.
- **Nombre de la variable.**- Es un identificador ajustado a las normas especificadas anteriormente y debe ser lo más significativo posible de acuerdo al contenido de la propia variable.

En una sentencia de declaración se pueden declarar más de una variable en forma de lista, separadas por comas, siendo todas ellas del mismo tipo.

Las variables pueden ser inicializadas, lo que significa que pueden asignarse un valor en el momento de su declaración.

## OPERADORES

Los operadores en C se pueden clasificar de la siguiente forma:

### Indicadores de expresión

Se utilizan para indicar una prioridad o una determinada misión en las expresiones. Estos son:

1. Paréntesis (), que pueden tener dos significados:
  - a. Para determinar la máxima prioridad en una expresión, comenzando por los más internos.
  - b. Para encerrar los parámetros de una función.
2. Corchetes [], que sirven para encerrar los índices de los vectores.

### Operadores aritméticos

Se utilizan para efectuar operaciones aritméticas. Se clasifican en:

	Operador	Símbolo
Monarios	Signo negativo	-
	Incremento	++
	Decremento	--
Binarios	Suma	+
	Resta	-
	Multiplicación	*
	División	/
	Módulo	%

1. Operador de signo negativo.- Se utiliza para indicar el signo menos de un operando numérico. Con este símbolo se puede cambiar de signo a una variable, pues equivale a multiplicar por -1.
2. Operador Incremento.- Se utiliza para aumentar el valor de su operando en 1. La expresión ++c equivale a  $c = c + 1$ . El operador ++ puede ir como prefijo ++c o como sufijo c++. La diferencia estriba en el momento en que se efectúa la operación donde está incluido el operando. Este operador es muy útil cuando se emplea para el incremento de contadores.
3. Operador decremento.- Se utiliza para disminuir el valor de un operando en 1. La operación --c equivale a  $c = c - 1$ . Puede ir como prefijo o como sufijo, actuando de forma similar al operador ++.
4. Operadores suma y resta.- Se utiliza para realizar la suma o la resta entre dos operandos.

5. Operadores multiplicación y división.- El operador \* realiza el producto entre dos operandos. El operador / realiza la división obteniendo el cociente de la misma. En el lenguaje C no existe ningún símbolo para la división entera, obteniéndose el cociente de una división entera cuando los dos operandos son valores numéricos enteros (en esta división el cociente no se redondea, sino que se trunca). Si un operando es numérico real la división será real.
6. El operador módulo.- El operador % permite obtener el resto de una división entera. Por tanto, los dos operandos deben ser enteros.

La prioridad entre los operadores aritméticos es:

- ++ --	Signo menos, incremento, decremento
* / %	Multiplicación, división y módulo
+ -	Suma, resta

### Operadores relacionales y lógicos

Se utilizan para realizar operaciones lógicas entre operandos, obteniendo como resultado un valor verdadero o falso. En el lenguaje C el resultado falso es 0 y el resultado verdadero es 1. En un sentido más amplio, cuando se evalúa una expresión lógica, se considera verdadero todo valor que no sea cero.

Los operadores relacionales son los siguientes:

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual
!=	Distinto

Es frecuente confundir el operador = de igualdad con el operador = de asignación. No es lo mismo `c = 5` (asigna a la variable c el valor 5) que `c == 5` (compara c con 5).

Los operadores lógicos son los siguientes:

Operador	Significado
&&	Conjunción Y
	Disyunción O
!	Negación, no (monario)

Los operadores conjunción && y disyunción || utilizan dos símbolos iguales, y se debe tener la precaución de no confundirlos con los operadores de tratamiento de bits, que emplean estos símbolos de forma única & y |.

El operador O exclusivo no existe en C, pero puede componerse a través de la siguiente expresión: `A || B && !( A && B )`.

La prioridad de los operadores relacionales y lógicos es:

!	negación lógica
> >= < <=	mayor, mayor o igual, menor, menor o igual
== !=	igual, distinto



&&	Conjunción logica
	Disyunción lógica

### Operadores para el tratamiento de bits

Se utilizan para realizar operaciones a nivel de bit, y los operandos deben ser de tipo *char* o *int*. Estos operadores se clasifican en:

Operador	Significado
&	Y (AND binario)
	O (OR binario)
^	O exclusivo (XOR binario)
~	Complemento a 1 (NOT binario)
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Los operadores de desplazamiento de bits mueven los bits de un operando hacia la derecha, caso de >>, o hacia la izquierda, caso de <<, tantas posiciones como indica otro operando que debe ser un número entero. Cada desplazamiento de un bit a la izquierda realiza una multiplicación del operando por 2, y cada desplazamiento a la derecha realiza una división del operando por 2.

### Operadores de asignación

Se utilizan para cargar una variable con el valor de una expresión. Pueden ser de dos tipos, simple o compuesto.

El operador de **asignación simple** utiliza el símbolo = indicando que la variable situada a su izquierda se cargará con el valor resultante de la expresión situada a la derecha.

El operador de **asignación compuesto** utiliza dos símbolos, uno es el de asignación simple y el otro es un símbolo de operación.

Los símbolos que se pueden utilizar son: + - / % << >> & ^ |. El formato es: *Variable Símbolo\_operación Expresión;*. Es equivalente a *Variable = Variable Símbolo\_operación Expresión;*

### Operador coma

Se utiliza para separar dos expresiones dentro de una expresión total, evaluándose primero la expresión de la izquierda y convirtiendo la de la derecha en el valor de la expresión total. Lo que se realiza en realidad es una secuencia de operaciones, no debiéndose confundir este operador con la coma, que se utiliza para separar los parámetros de una función o con la coma que se utiliza para separar las variables en una definición múltiple. Se emplea esencialmente en dos casos:

1. Para realizar una asignación de una expresión, que previamente requiere otra expresión.
2. Para realizar varias operaciones dentro de la condición de un bucle.

### Operador de tamaño

Se utiliza para obtener la longitud en bytes de una variable o de un especificador de tipo de dato. Emplea la palabra reservada *sizeof*. Cuando se emplea para un especificador de dato, éste debe ir entre paréntesis y cuando se emplea para una variable, puede ir o no con paréntesis.

Empleo del operador *sizeof* facilita la portabilidad del código a otros ordenadores.

El operador *sizeof* permite expresar la longitud de una variable compuesta, sin necesidad de sumar las longitudes de cada uno de sus componentes, facilitando así la portabilidad.

## Operador de molde

Se utiliza para convertir el tipo de dato de un operando utilizado en una expresión. Se expresa precediendo al operando con el tipo de dato deseado encerrado entre paréntesis. El tipo de dato del operando no varía en su definición, únicamente se convierte para la expresión en la que se utiliza el moldeado. El formato es: *(Tipo)operando*.

## Operador condicional

Se utiliza para realizar una operación alternativa mediante una condición. Es un operador ternario, o sea, que requiere tres operandos que pueden ser tres expresiones. Se emplean los símbolos *?* y *:*. El formato es: *Expresión1 ? Expresión2 : Expresión3*; La operación se realiza de la siguiente forma: Se evalúa la *Expresión1*, si el resultado es verdadero se evalúa la *Expresión2* y su resultado se toma como resultado de la expresión total, por el contrario si el resultado de la *Expresión1* es falso, se toma el resultado de la *Expresión3* como resultado de la expresión total.

## Orden de prioridad de los operadores

Como resumen, vamos a ver una tabla que refleja todos los operadores, donde se establece su prioridad o precedencia de mayor a menor, así como su asociatividad o comienzo de uso en el caso de estar con el mismo nivel de prioridad.

Operador	Significado	Asociatividad
() [] . ->	Paréntesis y llamada a función Subíndice de tabla Miembro de una estructura. Punto Miembro de una estructura. Flecha	Izquierda a derecha
! ~ - ++ -- * & (tipo) sizeof	Negación lógica Complemento a 1 Signo menos Incremento Decremento Indirección para punteros Dirección de una variable Molde Tamaño	Derecha a izquierda
* / %	Multiplicación División Módulo	Izquierda a derecha
+ -	Suma Resta	Izquierda a derecha
>> <<	Desplazamiento a la derecha Desplazamiento a la izquierda	Izquierda a derecha
< <= > >=	Menor que Menor o igual que Mayor que Mayor o igual que	Izquierda a derecha
== !=	Igual Distinto	Izquierda a derecha
& ^   &&	Y a nivel de bits O exclusivo a nivel de bits O a nivel de bits Y lógico	Izquierda a derecha

	O lógico	Izquierda a derecha
?:	Condicional	Derecha a izquierda
= += -= *= /= %= <<= >>= &= ^=  =	Asignación	Derecha a izquierda
,	Coma	Izquierda a derecha

## Expresiones

Las expresiones son un conjunto de operandos (constantes, variables y valores retornados por funciones) y de operadores. Después de efectuar en la expresión las operaciones indicadas por los operadores se obtiene un determinado valor que se corresponderá con uno de los tipos básicos de los datos en C. Se debe tener en cuenta el orden de prioridad y la asociatividad de los operadores para obtener un correcto resultado de la expresión, recordando que los paréntesis tienen la máxima prioridad empezando por los más internos.

En las expresiones se pueden considerar dos clases de conversiones de tipo:

1. Conversión de tipo en la evaluación de una expresión.- Se aplican unas normas de promoción del tipo de datos cuando se relaciona con otro tipo de dato de mayor nivel teniendo en cuenta que es una promoción temporal, de forma que se mantiene el tipo de dato con el que se ha definido la variable. Las reglas de conversión son las siguientes:
  - a. Si existe un molde de tipo para un determinado operando, éste se convierte en el tipo de dato especificado por el operador.
  - b. Se realiza una conversión automática para los tipos de datos *float* que se convierten en *double* y para los tipos de datos *char* y *short int* que se convierten en *int*.
  - c. Para los demás tipos de datos se realiza la promoción siguiente:
    - Si un operando es *long double*, el otro se convierte en *long double*.
    - Si un operando es *double*, el otro se convierte en *double*.
    - Si un operando es *long*, el otro se convierte en *long*.
    - Si un operando es *unsigned*, el otro operando se convierte en *unsigned*.
    - En los demás casos se convierte en *int*.
2. Conversión de tipo en las asignaciones.- La norma que se sigue es que el resultado de la evaluación de la expresión situada a la derecha de la asignación se convierte al tipo de la variable situada a la izquierda de la misma. Puede existir por tanto promoción (no se pierde el valor de la expresión) o pérdida de rango (puede existir transformaciones que produzcan pérdida de información).

# 8

## Estructura de un programa en C

### INTRODUCCIÓN

En C un programa tiene la siguiente estructura básica:

```
#include
```

```
#define
```

*Declaraciones globales*

- Prototipos de funciones
- Variables

*Función principal main*

```
main() {
```

*declaraciones locales*

*sentencias*

```
}
```

*Definiciones de otras funciones*

```
tipol func1( ... ) {
```

...

```
}
```

...

Vamos a ver cada parte de forma más detallada.

## Directivas del preprocesador

El preprocesador en un programa en C se puede considerar como un editor de texto inteligente que consta de directivas (instrucciones al compilador antes de que se compile el programa principal). Las dos directivas más usuales son `#include` y `#define`.

Todas las directivas del preprocesador comienzan con el símbolo almohadilla (`#`), que indica al compilador que lea la directivas antes de compilar la parte (función) principal del programa. Las *directivas* son instrucciones al compilador. No son sentencias, no acaban en punto en coma, sino instrucciones que se dan al compilador antes de que el programa se compile. Aunque las directivas pueden definir macros, nombres de constantes, archivos fuente adicionales, etc., su uso más frecuente en C es la inclusión de archivos de cabecera.

La directiva `#include` indica al compilador que lea el archivo fuente que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva. Estos archivos se denominan *archivos de cabecera*. Estos tienen extensión `.h` y contienen código fuente C y se sitúan en un programa C mediante la directiva del preprocesador `#include` con el siguiente formato:

```
#include <nombrearchivo.h>
```

O también

```
#include "nombrearchivo.h"
```

nombre de archivo.h debe ser un archivo texto ASCII que resida en disco. En realidad la directiva del preprocesador mezcla un archivo de disco en el programa fuente.

Existen unos archivos especiales denominados archivos de cabecera almacenados en el directorio INCLUDE. Estos archivos tienen la extensión `.h` El más frecuente es `stdio.h` que proporciona al compilador C la información necesaria sobre las funciones de biblioteca que realizan operaciones de entrada y salida. Como casi todos los programas necesitarán visualizar información en pantalla y leerla desde teclado, necesitarán incluir `scanf()` y `printf()` en los mismos. Para ello será necesario que el programa contenga la línea siguiente:

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

Cuando el nombre del archivo se encierra entre ángulos significa que los archivos se encuentran en el directorio por defecto INCLUDE. Si el nombre del archivo se encierra entre comillas dobles significa que el archivo se encuentra en el directorio actual. Los dos métodos no son excluyentes.

## Declaraciones globales

Indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones de su programa. Las declaraciones globales se sitúan antes de la función `main()`. Si se declara global una variable, cualquier función del programa puede acceder a dicha variable.

Las declaraciones de funciones se denominan *prototipos* y se verá con más detalle en el capítulo dedicado a las funciones.

## Función main()

Todo programa en C tiene una, y sólo una, función `main()` que es el punto de entrada al programa. Su estructura es:

```
main ( ) {  
  
    instrucciones  
  
}
```

Las instrucciones son las sentencias que se ejecutarán. Existen dos tipos de instrucciones ejecutables en C:

- 1.- Instrucción simple. Implica la ejecución de una acción y acaba en punto y coma.
- 2.- Bloque de instrucciones. Es un conjunto de instrucciones simples encerradas entre llaves. Dentro de un bloque se pueden declarar variables que son locales a ese bloque.

### Funciones definidas por el usuario

Un programa en C es una colección de funciones. Todos los programas se construyen a partir de una o más funciones que se integran para crear una aplicación. Todas las funciones contienen una o más sentencias C y se crean generalmente para realizar una única tarea, tales como imprimir la pantalla, escribir un archivo, etc. Se pueden declarar y ejecutar un número de funciones casi ilimitado en un programa C.

En el capítulo dedicado a las funciones se profundizará sobre las funciones de usuario.

### Comentarios

Un comentario es cualquier información que se añade a su archivo fuente para proporcionar documentación de cualquier tipo. El compilador ignora los comentarios, no realiza ninguna tarea concreta. El uso de comentarios es totalmente opcional, aunque muy recomendable.

Generalmente, se considera buena práctica de programación comentar su archivo fuente tanto como sea posible, al objeto de que tanto el autor del programa como otros programadores puedan leer fácilmente el programa con el paso del tiempo. Es una buena práctica de programación comentar el programa en la parte superior de cada archivo fuente. La información que se suele incluir es el nombre del archivo, el nombre del programador, una breve descripción, la fecha en que creo e información de la revisión.

Los comentarios en C son de dos tipos:

1. Una sola línea. En este caso el comentario comienza con el símbolo // y acaba al final de la línea.
2. Multilínea.- En este caso el comentario consiste en todo texto encerrado entre los símbolos /\* y \*/

### Creación de un programa

Los pasos a realizar son:

1. Escribir un programa con un editor de texto ASCII y grabarlo en un archivo con extensión .c. Este archivo es el *código fuente* del programa.
2. Compilar el código fuente. Se traduce el código fuente en un *código objeto* (extensión .obj, lenguaje máquina entendible por el ordenador). Un archivo objeto contiene instrucciones en lenguaje máquina que se pueden ejecutar por el ordenador.
3. Enlazar el código objeto con las bibliotecas correspondientes. Una biblioteca C contiene código objeto de una colección de funciones que realizan tareas. El enlace del código objeto del programa con el objeto de las funciones producirán código ejecutable.

### Depuración de un programa en C

Rara vez los programas funcionan bien la primera vez que se ejecutan. Los errores que se producen en los programas han de ser detectados, aislados (fijados) y corregidos. El proceso de encontrar errores se denomina **depuración** del programa. La corrección del error es probablemente la etapa más fácil, siendo la detección y aislamiento del error las tareas más difíciles.

Existen diferentes situaciones en las cuales se suelen introducir errores en un programa. Dos de las más frecuentes son:

- Violación de las reglas gramaticales del lenguaje de alto nivel en el que se escribe el programa.
- Los errores de diseño del algoritmo en el que está basado el programa.

Cuando el compilador detecta un error, visualiza un *mensaje de error* indicando que se ha cometido un error y posible causa del error. Desgraciadamente algunos mensajes de error son difíciles de interpretar y a veces se llegan a conclusiones erróneas. También varían de un compilador a otro compilador. A medida que se gana experiencia, el proceso de puesta a punto de un programa se mejora considerablemente. Desde el punto de vista conceptual existen tres tipos de errores:

### **Errores de sintaxis**

Son aquellos que se producen cuando el programa viola la sintaxis, es decir, las reglas de gramática del lenguaje. Errores de sintaxis típicos son: escritura incorrecta de palabras reservadas, omisión de signos de puntuación (comillas, punto y coma, ...). Los errores de sintaxis son los más fáciles de fijar, ya que la mayoría de ellos son detectados y aislados por el compilador.

Estos errores se suelen detectar por el compilador durante el proceso de compilación. A medida que se produce el proceso de traducción del código fuente a lenguaje máquina del ordenador, el compilador verifica si el programa que se está traduciendo cumple las reglas de sintaxis del lenguaje. Si el programa viola alguna de estas reglas, el compilador genera un mensaje de error que explica el problema aparente.

### **Errores lógicos**

Representa errores del programador en el diseño del algoritmo y posterior programa.

Los errores lógicos son más difíciles de encontrar y aislar ya que no suelen ser detectados por el compilador. El programa se compilará y ejecutará bien, aunque producirá resultados incorrectos.

Una vez se ha determinado que un programa contiene un error lógico, si es que se encuentra en la primera ejecución y no pasa desapercibida al programador, encontrar el error es una de las tareas más difíciles de la programación. El **depurador** (debugger) es un programa software diseñado para detectar, verificar y corregir errores, ayudando en las tareas de depuración.

Se pueden detectar errores lógicos comprobando el programa en su totalidad, comprobando su salida con los resultados previstos. Se pueden prevenir errores lógicos con un estudio minucioso y detallado del algoritmo antes de que el programa se ejecute, pero resultará fácil cometer errores lógicos y es la experiencia y dominio del lenguaje lo que permitirá detectarlos.

### **Errores de regresión**

Son aquellos que se crean accidentalmente cuando se intenta corregir un error lógico.

Siempre que se corrige un error se debe comprobar totalmente la exactitud para asegurarse que se fija el error que se está tratando y no produce otro error. Los errores de regresión son comunes, pero son fáciles de leer y corregir. Una ley no escrita es que: *un error se ha producido, probablemente, por el último código modificado.*

### **Mensajes de Error**

Los compiladores emiten mensajes de error o de advertencia durante las fases de compilación, de enlace o de ejecución de un programa. Los mensajes de error producidos durante la compilación se suelen producir, normalmente, por errores de sintaxis y suele variar según los compiladores; pero, en general, se agrupan en tres grandes bloques:

- Errores fatales. - Son raros. Algunos de ellos indican un error interno del compilador. Cuando ocurre un error fatal, la compilación se detiene

inmediatamente, se debe tomar la acción apropiada y a continuación se vuelve a iniciar la compilación.

- Errores de sintaxis.- Son los errores típicos de sintaxis, errores de línea de órdenes y errores de acceso a memoria o disco. El compilador terminará la fase actual de compilación y se detiene.
- Advertencias (warning).- No impide la compilación. Indican condiciones que son sospechosas, pero legítimas como parte del lenguaje.

### Errores en tiempo de ejecución

Un error en tiempo de ejecución puede ocurrir como resultado de que el programa obliga al ordenador a realizar una operación ilegal, tal como dividir un número por cero, raíz cuadrada de un número negativo o manipular datos no válidos o indefinidos.

Cuando ocurre este tipo de error, el ordenador detiene la ejecución del programa y visualizará un mensaje de diagnóstico.

### Pruebas

Para determinar si un programa contiene un error lógico, se debe ejecutar utilizando datos de muestra y comprobar la salida verificando su exactitud. Esta prueba se debe hacer varias veces utilizando diferentes entradas. Si cualquier combinación de datos de entrada produce una salida incorrecta, entonces el programa contiene un error lógico. Una vez se ha determinado que un programa contiene un error lógico, la localización del error es una de las partes más difíciles de la programación. La ejecución se debe realizar paso a paso (seguir la traza) hasta el punto en que se observe que un valor calculado difiere del valor esperado. Para simplificar la traza de un programa, la mayoría de los compiladores de C proporcionan un depurador integrado incorporado con el editor, y todos ellos en un mismo paquete software, que permiten al programador ejecutar realmente un programa, línea a línea, observando los efectos de la ejecución de cada línea en los valores de los objetos del programa. Una vez que se ha localizado el error, se utilizará el editor de texto para corregirlo.

## INSTRUCCIONES DE DECLARACIÓN

Este tipo de instrucciones se utiliza para informar al compilador del espacio que debe reservar en memoria para almacenar un dato simple o estructurado, teniendo siempre presente que todas las variables, sean del tipo que sean, deberán ser definidas antes de ser utilizadas o referenciadas.

La declaración consiste en enunciar el nombre de la variable, asociarle un tipo y opcionalmente darle un valor inicial. Los pasos necesarios para definir una variable simple son:

1. Seleccionar el tipo de dato necesario.
2. Seleccionar el nombre para la variable, procurando que sea siempre lo más significativo posible de acuerdo con el contenido de la misma.
3. Utilizar el siguiente formato para construir la instrucción de definición correspondiente, donde *<Modificador\_tipo>* es opcional  
`<Modificador_tipo>Tipo_básico_dato Nombre_Variable;`
4. Se puede definir más de una variable del mismo tipo en la misma sentencia o instrucción, separándolas por comas.  
`<Modificador_tipo>Tipo_básico_dato Var1, Var2, ... , VarN;`
5. Opcionalmente, se pueden inicializar las variables en las instrucciones de declaración.  
`<Modificador_tipo>Tipo_básico_dato Nombre_Variable  
= <Valor inicial>;`



En el caso de definir más de una variable del mismo tipo a las que opcionalmente se les desea dar un valor inicial (a todas o alguna de ellas), es necesario utilizar el siguiente formato de definición.

```
<Modificador_tipo>Tipo_básico_dato Var1 = Val1,
Var2 = Val2, ... , VarN = ValN;
```

## INSTRUCCIONES DE ASIGNACIÓN

Es el hecho o suceso a través del cual podemos calcular el resultado de una expresión de mayor o menor complejidad y almacenar el valor simple obtenido en una variable. El operador de asignación utilizado en el lenguaje de programación C es el signo igual (=). El formato de una instrucción de asignación es el siguiente:

```
Nombre_Variable = Expresión;
```

donde el tipo de la variable utilizada y el tipo del valor obtenido al efectuar el cálculo de la expresión debe coincidir, ya que el resultado obtenido en la parte derecha de la instrucción es almacenado en la variable especificada en la parte izquierda de la misma.

## INSTRUCCIONES DE ENTRADA Y SALIDA

Son aquellas instrucciones que permiten introducir datos desde un periférico a la memoria del ordenador y viceversa.

Todo este conjunto de herramientas se encuentra en la librería estándar “stdio.h”, que es aquella que proporcionará, mediante funciones, acceso a las vías normales de entrada y salida de los distintos dispositivos o periféricos conectados a nuestro ordenador. Por ello es imprescindible acostumbrarse a incluir en todos los programas la librería stdio.h. Cuando se ejecuta un programa en C, se abren automáticamente y de forma simultánea cinco vías de acceso estándar, asociadas cada una de ellas a un dispositivo periférico.

Vía de acceso	Periférico asociado
stdin	Entrada estándar de datos (teclado)
stdout	Salida estándar de datos (pantalla)
stderr	Salida estándar de errores (pantalla)
stdprn	Interfaz serie de comunicaciones estándar (RS232)
stdaux	Salida estándar de datos (impresora)

### Función de entrada (teclado)

**int getch( void );**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	No tiene argumentos
<b>Valor devuelto</b>	Carácter leído o EOF
<b>Finalidad</b>	Lee un carácter del dispositivo estándar de entrada (teclado)

Guarda en un buffer todos aquellos caracteres introducidos a través del dispositivo estándar de entrada (stdin) hasta la pulsación de la tecla INTRO o hasta alcanzar un número máximo de caracteres, que varía en función de la implementación del lenguaje. El carácter es leído como *unsigned char* y convertido automáticamente a *int*. En el caso de alcanzar el carácter de fin de fichero o producirse un error, la función devuelve EOF (End Of File).

**int scanf( const char \*formato, lista de argumentos);**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	Un puntero a la cadena de control con los formatos de tipo y lista de direcciones de memoria de las variables.
<b>Valor devuelto</b>	Número de campos procesados con éxito
<b>Finalidad</b>	Almacena en variables de memoria los datos introducidos a través del dispositivo estándar de entrada.

Lee cualquier tipo de dato del dispositivo estándar de entrada, convirtiéndolos automáticamente en el formato interno apropiado, donde **formato**, apunta a una cadena de control constituida por tres tipos de caracteres:

- 1.- Especificadores de formato para la entrada de datos, que nos indican el tipo de dato que se deberá leer.

Código	Significado
%c	Lee un solo carácter
%h	Lee un entero corto
%d	Lee un número entero decimal
%o	Lee un número octal
%i	Lee un número entero decimal
%s	Lee una cadena de caracteres
%e	Lee un valor número real
%x	Lee un número hexadecimal
%f	Lee un valor número real
%p	Lee un puntero

- 2.- Caracteres de espacio en blanco.
- 3.- Caracteres que no sean espacios en blanco.

La cadena de control apuntada por **formato** se lee de izquierda a derecha asociando cada código de formato con el correspondiente argumento en la lista de argumentos siguiendo el mismo orden de izquierda a derecha.

En aquellos casos en los que la cadena de control, apuntada por **formato**, esté constituida por caracteres que no sean espacios en blanco o especificadores de formato para la entrada de datos, se pueden dar las siguientes posibilidades:

- 1.- La función `scanf()` utiliza el carácter especificado para detectar y descartar dicho carácter de la secuencia de entrada. Por ejemplo, “%d:%d”, hace que `scanf()` lea un valor entero, seguidamente lee y descarta los dos puntos y a continuación vuelve a leer un segundo valor numérico entero.
- 2.- Si se especifica un asterisco entre el símbolo % y el código de formato, `scanf()` lee el dato del tipo especificado pero no lo asigna. Por ejemplo la entrada 8+2 con `scanf(“%d%*c%d”, &sum1, &sum2);` almacena el valor 8 en `sum1`, lee y descarta el signo de suma y a continuación coloca el valor 2 en `sum2`.
- 3.- La última posibilidad es definir un conjunto de caracteres que serían los aceptados en la secuencia de entrada y almacenados en un array de caracteres, de manera que la primera forma consistiría en definir los

caracteres que se van a llenar entre corchetes, por ejemplo, “%[ABCDE]” y la segunda forma sería especificando un rango de valores, por ejemplo “%[A-E]”. En ambos casos, cuando se introduce un carácter que no corresponde con ninguno de los especificados, scanf(), finaliza la secuencia de entrada añadiendo el carácter nulo a la cadena y pasando al siguiente campo.

### Funciones de salida (pantalla)

**int putchar( int carácter );**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	Variable carácter o constante carácter.
<b>Valor devuelto</b>	Variable carácter, constante carácter o EOF
<b>Finalidad</b>	Visualiza por pantalla un carácter.

Escribe el carácter especificado como argumento en el dispositivo estándar de salida en la posición en la que se encuentra actualmente el cursor. Si la llamada a la función putchar() se ejecuta correctamente devuelve el carácter mostrado, y en caso de error, devuelve EOF.

**int printf( const char \*formato, lista de argumentos );**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	Puede recibir un número indeterminado de argumentos, de cualquier tipo y cualquier formato
<b>Valor devuelto</b>	El número de caracteres escritos o EOF
<b>Finalidad</b>	Escribe una cadena en el dispositivo de salida estándar.

Envía una serie de argumentos especificados en **lista de argumentos** al dispositivo estándar de salida, que son controlados mediante los formatos de impresión especificados en la cadena **formato**.

Las reglas a tener en cuenta al invocar la función printf() son:

- 1.- Que el formato de impresión coincida con el tipo de dato que queremos imprimir o sea equivalente siguiendo las reglas de conversión de tipos.
- 2.- Que el número de formatos de impresión sea el mismo que el número de argumentos especificados en **lista de argumentos**.
- 3.- Respetar siempre el orden de aparición de los formatos de impresión y los respectivos valores a imprimir (siempre de izquierda a derecha).

Esta función se caracteriza por la enorme flexibilidad a la hora de mostrar información en pantalla, ya que permite la impresión de constantes, variables, cadenas de caracteres, resultados de expresiones, etc.

Si todo fue correctamente en la ejecución de la función, ésta devuelve el número exacto de caracteres que han sido escritos en el dispositivo estándar de salida. En caso de ocurrir un error, devuelve EOF.

<b>Formato de Impresión</b>	<b>Significado</b>
%h	Imprime un entero corto de 1 byte (short int)
%u	Imprime enteros decimales sin signo de 2 bytes (unsigned int)
%d	Imprime enteros decimales con signo de 2 bytes (int)

%ld	Imprime un entero largo de 2 bytes (long int)
%f	Imprime valores con punto decimal de 4 bytes (float)
%lf	Imprime valores con punto decimal de 8 bytes (double)
%c	Imprime un carácter de 1 byte (char)
%s	Imprime un cadena de caracteres o string.
%o	Imprime un entero octal sin signo.
%x	Imprime un entero hexadecimal sin signo
%X	Imprime un entero hexadecimal sin signo
%e	Imprime valores reales en notación científica.
%E	Imprime valores reales en notación científica.
%g	Usa %e o %f según el tamaño del valor a imprimir
%%	Visualiza el signo %
%p	Visualiza un puntero
%i	Visualiza enteros decimales con signo

Los formatos de impresión disponen de modificadores que permiten especificar la longitud del campo, el número de decimales y la justificación a derecha o izquierda de la información que deseamos mostrar en pantalla. Dichos modificadores pueden, por tanto ser definidos como apéndices que unidos a los formatos de impresión nos permiten dar el formato deseado a la información presentada en pantalla.

Modificador de formato	Significado
%-Número letra	Ajusta la información al extremo izquierdo del campo
%Número letra	Indica la anchura del campo en tipos int, char o cadenas.
%EnteroDecimal letra	Igual al anterior pero con valores reales.
%L letra	El dato a imprimir es de tipo long

Para especificar el número de posiciones decimales que se han de imprimir para un número en coma flotante, se coloca un punto tras el especificador de longitud del campo, seguido del número de decimales que se desea que aparezca. Por ejemplo, %10.3f imprime un número de al menos 10 caracteres con 3 posiciones decimales. Si ponemos %5.7s imprime una cadena de al menos 5 caracteres y no más de 7. Si la cadena es más larga, se trunca los caracteres finales de la derecha.

Por otro lado, el modificador asterisco sobre una función printf() tiene un efecto diferente al producido sobre la función scanf().

En aquellos casos en los que no deseamos declarar con antelación el tamaño de un campo y queramos que el programa lo ajuste automáticamente, utilizaremos un modificador muy especial que es el modificador asterisco (\*). Para ello debemos tener en cuenta que es necesario seguir informando al programa de la anchura del campo, siendo posible proporcionarle esta información durante la ejecución del mismo. Por

ejemplo, con `printf("El número es: %*d \n", ancho_campo, numero);` se visualiza la variable `numero` con un ancho de campo especificado con `ancho_campo`.

Esta técnica también puede aplicarse a variables de punto flotante. Por ejemplo:

`printf("Peso = %*.f \n", ancho, precision, peso);`

Existen otros dos modificadores de órdenes de formato que permiten a `printf()` imprimir enteros largos (`long`) y cortos (`short`). Estos modificadores se pueden aplicar a los especificadores de tipo `d`, `i`, `o`, `u`, `x` y son los siguientes:

- 1.- El modificador "`l`" indica a `printf()` que lo que sigue es un tipo de dato "`long`", por ejemplo `%ld`, para imprimir un *long int*. Este modificador también puede preceder a los especificadores de coma flotante `e`, `f` y `g` indicando así que se desea imprimir un número de doble precisión.
- 2.- El modificador "`h`" indica a `printf()` que lo que sigue es un tipo de dato "`short`". Por ejemplo, `%u` imprime un *short unsigned int*.

## ESTRUCTURAS DE CONTROL

Las estructuras de control se basan en expresiones condicionales que provocan la ejecución controlada de determinadas acciones en función del resultado obtenido por dicha expresión.

El lenguaje C, a diferencia de otros lenguajes de programación, carece de los denominados tipos de datos lógicos. Por ello la evaluación de expresiones condicionales originan un valor 0 para falso y valor 1 para verdadero; es más, podemos decir que cualquier valor distinto de cero es cierto incluidos los valores negativos, considerando el cero como único valor falso.

### Alternativa simple

```
if( expresión_condicional )
    sentencia;
```

La sentencia puede estar formado por una sola instrucción o ser un bloque de instrucciones en cuyo caso van encerradas entre llaves.

```
if( expresión_condicional ) {
    instrucción 1;
    instrucción 2;
    ...
    instrucción n;
}
```

Primero se evalúa la expresión condicional. Si es cierta se ejecuta la instrucción o instrucciones, de lo contrario la ejecución del programa continua por la siguiente instrucción al *if*.

### Alternativa doble

Este tipo de sentencias condicionales ofrece dos posibilidades o alternativas para elegir según se cumpla o no la condición establecida, de manera que, si la expresión condicional es cierta, se ejecutará la primera sentencia y en caso contrario la segunda.

```
if( expresión_condicional )
    sentencia 1;
else
    sentencia 2;
```

Si se van a ejecutar un bloque de instrucciones:

```
if( expresión_condicional ) {
    instrucción1_1;
    instrucción1_2;
    ...
}
```

```

        instrucción1_n;
    }
    else {
        instrucción2_1;
        instrucción2_2;
        instrucción2_m;
    }

```

Cada instrucción a su vez, puede ser otra alternativa simple o doble, es decir, se pueden anidar las instrucciones alternativas *if*, debiendo tener cuidado donde se pone el *else*.

**Una sentencia *else* siempre quedará asociada con el *if* precedente más próximo.**

### **Bloque de sentencias if ... else-if**

Esta construcción de sentencias condicionales es muy común en C, donde las condiciones se establecen escalonadamente. Su formato es:

```

if( expresión_condicional_1 ) {
    bloque_instrucciones_1;
}
else if( expresión_condicional_2 ) {
    bloque_instrucciones_2;
}
...
else if( expresión_condicional_n ) {
    bloque_instrucciones_n;
}
else {
    bloque_instrucciones_por_defecto;
}

```

Las condiciones siempre son evaluadas secuencialmente de arriba abajo y de la condición más externa a la más interna. En el momento en que se encuentra una condición cierta, se ejecuta el bloque de sentencias asociado a ella, pasando por alto el resto de las condiciones establecidas escalonadamente; por el contrario, en el caso de no ser cierta ninguna de las condiciones evaluadas, se ejecuta el bloque de sentencias del último “else” donde, al igual que en los casos anteriormente vistos, puede ser omitida.

### **Sentencia if ... else abreviada con el operador ? :**

Su formato es el siguiente:

```
expresión_condicional ? sentencia_1 : sentencia_2;
```

Se evalúa la expresión condicional. Si es cierta se ejecuta la *sentencia\_1* y si es falsa se ejecuta la *sentencia\_2*. Generalmente se utiliza cuando la ejecución de ambas sentencias produce un valor que se asigna a una variable. Por ejemplo:

```
c = ( a == 4 ) ? 2 * a : 3 * a;
```

Si *a* es igual a 4 se ejecuta  $2 * a$  y su valor se devuelve, asignándose a la variable *c*. Si *a* es distinto de 4 se ejecuta  $3 * a$  y su valor también se devuelve, asignándose a *c*.

### **Alternativa múltiple. Sentencia switch()**

La ventaja de este tipo de sentencias frente al bloque de condicionales *if ... else if*, es que en ocasiones son de más fácil entendimiento y desarrollo, además de generar un código más elegante. Esta sentencia compara el valor resultante de una expresión o variable con una lista de constantes de tipo entero o carácter, de manera que cuando se establece una asociación o correspondencia entre ambos se ejecuta una o más sentencias. Su formato es:

```

switch( expresión ) {
    case constante_1: {
        bloque_instrucciones_1;
    }
}

```

```

        break;
    }
    case constante_2: {
        bloque_instrucciones_2;
        break;
    }
    ...
    case constante_n: {
        bloque_instrucciones_n;
        break;
    }
    default: {
        bloque_instrucciones_por_defecto;
    }
}

```

*Expresión* es cualquier expresión que devuelva un valor de tipo entero (*int*) o carácter (*char*). *constante\_1*, *constante\_2*, ..., *constante\_n* solo pueden ser una constante de tipo numérico entero o de tipo carácter. *bloque\_instrucciones* puede ser una instrucción o un conjunto de ellas que serán ejecutadas cuando el resultado de *expresión* coincida con el valor de la constante especificada.

Con *break* se sale del bloque de código. Su uso es opcional, aunque si se omite genera una ejecución continuada que parte del bloque de sentencias asociado a la constante *case* evaluada como cierta, pasando por el resto de los bloques de sentencias asociados a otras constantes *case* hasta la localización de una sentencia *break* o la finalización de la sentencia *switch*.

En aquellos casos en los que no se establece ninguna correspondencia entre el valor obtenido como resultado en la expresión y las constantes *case* especificadas se ejecuta siempre por defecto la sentencia *default*. El uso de esta sentencia no es estrictamente obligatorio siendo perfectamente omitible, aunque siempre es recomendable mantenerla en un buen estilo de programación.

A la hora de construir una alternativa múltiple, es conveniente tener presente que:

- 1.- En una sentencia *switch* no puede haber dos constantes *case* con el mismo valor, salvo que una sentencia *switch* se encuentre anidada dentro de otra sentencia *switch*, en cuyo caso sería posible que existieran constantes *case* iguales.
- 2.- A diferencia de la sentencia *if*, la sentencia *switch* no puede evaluar expresiones relacionales o lógicas, únicamente puede realizar operaciones de comparación de igualdad entre el resultado de la expresión y las constantes especificadas en cada sentencia *case*.
- 3.- Cualquier constante de tipo carácter utilizada en una sentencia *switch* es convertida automáticamente a su equivalente valor entero.

### Estructura repetitiva while

Su formato es:

```

while( expresión_condicional )
    instrucción;

```

Si se desea ejecutar un bloque de instrucciones:

```

while( expresión_condicional ) {
    bloque_instrucciones
}

```

La instrucción e el bloque de instrucciones se ejecutará mientras *expresión condicional* sea cierta. El número de iteraciones del bucle es de 0 a  $n$  veces, ya que la condición se evalúa al principio.

### Estructura repetitiva **do-while**

A diferencia de la sentencia *while*, una sentencia *do-while* evalúa la *expresión condicional* establecida para el control del bucle al final del mismo, lo que significa que las instrucciones contenidas dentro de éste se ejecutarán como mínimo una vez y como máximo  $n$  veces. Su formato es:

```
do {
    bloque_instrucciones;
}while( expresión_condicional );
```

### Estructura repetitiva **for**

En C, esta estructura o instrucción repetitiva ofrece una gran potencia y enorme flexibilidad, pues no presenta un formato fijo, dando posibilidad a un amplio número de variaciones en su utilización o tratamiento.

Su formato más general es:

```
for( exp_inicial; exp_condicional; exp_incremento )
    instrucción;
```

Para un bloque de instrucciones:

```
for( exp_inicial; exp_condicional; exp_incremento )
{
    bloque_instrucciones;
}
```

Toda instrucción *for* consta de tres partes:

- 1.- Inicialización.- Normalmente suele ser una instrucción de asignación donde la variable de control del bucle toma un valor inicial.
- 2.- Expresión condicional.- Esta segunda parte está constituida por una condición que determina el momento en el que debe finalizar la ejecución del bucle.
- 3.- Expresión incremento.- Normalmente se actualiza la variable de control del bucle mediante el incremento o decremento de la misma, siempre partiendo del valor que ésta haya tomado inicialmente en la primera parte del bucle, así como en sucesivas iteraciones.

Estas tres partes son opcionales, siendo posible la ausencia de alguna o todas ellas, debiendo estar siempre separadas por punto y coma. La secuencia de ejecución es:

- 1.- Se inicializa la variable de control del bucle.
- 2.- Evaluar la expresión condicional. Si es verdadera se ejecuta la instrucción o el bloque de instrucciones. Si no, se continua la ejecución del programa en la siguiente instrucción al bucle *for*.
- 3.- Actualizar la variable de control del bucle. Regresar al punto 2.

### Sentencia **break**

Este tipo de sentencia se utiliza con una doble finalidad. Para marcar el final de un *case* en una sentencia *switch* y para forzar de manera inmediata la terminación de un bucle, rompiendo su secuencia interna de ejecución. Su uso permite establecer ciertas salidas desde dentro del bucle ignorando la evaluación de la expresión condicional del ciclo.

En aquellos casos en los que existan estructuras de control repetitivas anidadas, un *break* produce la salida inmediata de aquel bucle en el que se encuentre comprendida.

### Sentencia **continue**

Es utilizada en estructuras de control repetitivas siendo su modo de funcionamiento similar a la sentencia *break*, salvo que en lugar de forzar la terminación, fuerza una



nueva iteración del bucle, saltando cualquier porción de código existente entre el lugar donde se halla ubicada dicha sentencia y el final del bucle. Al forzar una nueva iteración, vuelve a evaluarse la expresión condicional que controla la ejecución del bucle.

En una estructura *for* las acciones a realizar son:

- 1.- Saltar todas las sentencias o instrucciones que haya entre dicha sentencia y el final del bucle.
- 2.- Ejecutar la parte de incremento o decremento del bucle
- 3.- Evaluar la expresión condicional del *for*.
- 4.- Si la condición es cierta proseguir con la ejecución del bucle, si no, salir a la siguiente instrucción.

En una estructura de control *while* o *do-while* las acciones a realizar son:

- 1.- Saltar todas las sentencias o instrucciones que haya entre dicha sentencia y el final del bucle.
- 2.- Evaluar la expresión condicional de la estructura de control.
- 3.- En caso de ser cierta la condición, proseguir con la ejecución del bucle.

### **Sentencia goto**

Su uso está restringido en una programación estructurada, pues tiende a generar programas ilegibles de difícil entendimiento y comprensión, lo que dificulta, y en ocasiones impide, cualquier tipo de actualización o modificación sobre los mismos, obligando al programador a desarrollar un nuevo programa o aplicación.

En C, la sentencia *goto* requiere de una etiqueta o identificativo seguido de dos puntos, teniendo que estar necesariamente ubicados los dos en la misma función o módulo.

Su formato es:

```
goto etiqueta;
```

# 9

## Funciones en C

### ESTRUCTURA DE UNA FUNCIÓN

Una función es un conjunto de sentencias que se pueden invocar su ejecución desde cualquier parte del programa. Las funciones permiten al programador un grado de abstracción en la resolución de un problema. Las funciones en C no se pueden anidar, es decir, una función no puede declararse dentro de otra función. En C cualquier función puede ser invocada desde cualquier otra función, incluso a si misma (función recursiva). En una función en C se definen tres conceptos: definición, llamada y declaración (prototipo).

#### Definición de una función

Una función se define en un programa C con la siguiente estructura:

```
tipo_retorno nombre_función( lista de parámetros )
{
    Cuerpo de la función

    return expresión;
}
```

En la definición de la función se distinguen los siguientes elementos:

- Tipo de retorno. - Es el tipo de datos que devuelve la función al punto de llamada y aparece antes del nombre de la función. Si la función no devuelve nada (procedimiento) el tipo de retorno es *void*. Si se omite el tipo de retorno, por defecto C devuelve un valor tipo **int**. Por legibilidad, aunque la función retorne un valor tipo *int*, se debe poner específicamente.
- Nombre de la función. - Es el identificador de la función. Las reglas para nombrar funciones son las mismas que para los identificadores.
- Lista de parámetros. - Son los parámetros de la función separados por comas y encerrados entre paréntesis. Cada uno tiene su tipo de datos y nombre. Si la función no tiene parámetros no se pone nada o la palabra clave *void*.
- Cuerpo de la función. - Son las sentencias que se ejecutarán cuando se invoque la función. Van encerradas entre llaves.
- return expresión. - Es el valor que se devuelve al punto de llamada. Solamente se puede devolver un valor, aunque la sentencia *return* puede aparecer tantas veces como se desee. Cuando el flujo del programa llegue a una sentencia *return* se devuelve el control al punto de invocación y todas las sentencias debajo del *return* no se ejecutan. Si no hubiera ninguna, la función termina después de la última sentencia. El valor de retorno sigue las mismas reglas que se aplican a un operador de asignación. Si la función no devuelve nada (tipo de retorno *void*), entonces puede omitirse la sentencia *return*, pero si se pone no lleva ninguna expresión.

Por ejemplo, una función que recibe dos números enteros y devuelve el mayor de los dos:

```
int Mayor( int a, int b ) {
    int mayor;

    if( a > b ) mayor = a;
    else mayor = b;

    return mayor;
}
```

### Llamada a una función

Las funciones, para poder ser ejecutadas, han de ser llamadas o *invocadas*. Cualquier expresión puede contener una *llamada a una función* que redirigirá el control del programa a la función nombrada. Normalmente la llamada a una función se realizará desde la función principal **main()**, aunque también podrá ser desde otra función.

La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina (alcanza la sentencia *return*, o la llave de cierre {}) si se omite *return*) el control del programa vuelve a la función que realizó la llamada.

Si la función devuelve un valor, en la función de llamada se tiene que utilizar este valor asignándolo a una variable o utilizarlo en alguna expresión. Se puede llamar a una función y no utilizar el valor que se devuelve.

Por ejemplo, supongamos que la función anterior se invoca desde el *main()*.

```
#include <stdio.h>
int main() {
    int a, b;

    printf("\nIntroduce a: ");
    scanf("%d",&a);
    printf("\nIntroduce b: ");
    scanf("%d",&b);

    printf("\nEl mayor es %d", Mayor(a,b) );
}
```

Como se puede observar, para llamar a una función se pone el nombre de la función y entre paréntesis los parámetros que se pasan a la función separados por comas.

### Declaración de una función. Prototipo

La declaración de una función se denomina prototipo. Los prototipos de una función contienen la cabecera de la función, con la diferencia de que los prototipos terminan en punto y coma. Específicamente un prototipo consta de:

1. Tipo de retorno.
2. Nombre de la función.
3. Tipos de los parámetros separados por comas y encerrados entre paréntesis.

En C no es estrictamente necesario que una función se declare o defina antes de su uso, no es necesario incluir su prototipo aunque si es recomendable para que el compilador pueda hacer chequeos en las llamadas a las funciones. Los prototipos suelen ir antes de la definición de la función *main()* para que sean reconocidas en todo el programa. Por ejemplo, el prototipo de la función anterior sería:

```
int Mayor( int, int );
```

Si la función no tiene parámetros, se pone los paréntesis vacíos o la palabra clave *void*.

## PARÁMETROS DE UNA FUNCIÓN

Esta sección examina el mecanismo que C utiliza para pasar parámetros a funciones y cómo optimizar el paso de parámetros, dependiendo del tipo de dato que se utiliza.

Suponiendo que se tenga la declaración de una función *circulo* con tres parámetros:

```
void circulo( int x, int y, int diámetro );
```

Cuando se llama a *circulo* se deben pasar los tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión.

```
circulo(25, 40, vueltas * 4 );
```

### Paso de parámetros por valor

Significa que cuando C compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro variable local, el cambio sólo afecta a la función y no tiene efecto fuera de ella. Si el parámetro real es una variable su valor no se altera al cambiar el valor del parámetro formal. Por ejemplo

```
#include <stdio.h>

void func( int );

int main() {
    int i = 6;
    func(i);

    //La variable i sigue valiendo 6
    printf("\nEl valor de i es %d");
}

void func( int a ) {
    printf("El valor de a es %d");

    //Se modifica el parámetro formal.
    //No afecta a la variable i en main.
    a++;
}
```

### Paso de parámetro por referencia

Cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por *referencia* o *dirección*.

En este método el compilador pasa la dirección de memoria del valor del parámetro a la función cuando se modifica el valor del parámetro (variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado. Para pasar una variable por referencia, el símbolo & debe preceder al nombre de la variable y el parámetro variable correspondiente de la función debe declararse como puntero. Por ejemplo, la siguiente función intercambia los valores de dos variables

```
#include <stdio.h>

void cambia( int *, int * );
```

```

int main() {
    int a = 6;
    int b = 9;
    printf("\nLos valores de a y b son %d y %d", a, b
);
    cambia(&a, &b);

    //Ahora han intercambiado sus valores
    //Se han modificado en la función
    printf("\nLos valores de a y b son %d y %d", a, b
);
}

void cambia( int * x, int * y ) {
    int aux;

    printf("\nLos valores de x e y son %d y %d", x, y
);

    //Se intercambia los valores.
    aux = *x;
    *x = *y;
    *y = aux;
}

```

C permite utilizar punteros para implementar parámetros por referencia, ya que por defecto el paso de parámetros es por valor. Los parámetros reales que se pasan por referencia solamente pueden ser variables, ya que son las únicas que tienen dirección de memoria.

### Parámetros *const* de una función

Con el objeto de añadir seguridad adicional a las funciones, se puede añadir a una descripción de un parámetro el especificador *const*, que indica al compilador que sólo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación.

```

void f1( const int x, int y ) {
    x = 10;    //Error por cambiar un objeto
    constante.
}

```

## ÁMBITO

El *ámbito* o *alcance* de una variable determina cuáles son las funciones que reconocen ciertas variables. Si una función reconoce una variable, la variable es *visible* en esa función. El ámbito es la zona de un programa en la que es visible una variable. Existen cuatro tipos de ámbitos: programa, archivo fuente, función y bloque. Se puede designar una variable para que esté asociada a uno de estos ámbitos. Tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito.

Normalmente el lugar de declaración en el programa determina el ámbito. Los especificadores *static*, *extern*, *auto* y *register*, pueden afectar al ámbito.

### Ámbito del programa

Las variables que tienen *ámbito de programa* pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman *variables globales*. Para

hacer una variable global se declara al principio del programa, fuera de cualquier función.

Una variable global es visible desde su punto de declaración en el archivo fuente. Es decir, si se define una variable global, cualquier línea del resto del programa podrán utilizar esa variable.

### **Ámbito de archivo fuente**

Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada *static* tiene ámbito de archivo fuente. Las variables con este tipo de ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente. Si un archivo fuente tiene más de una función, todas las funciones que siguen a la declaración de la variable pueden referenciarla.

### **Ámbito de una función**

Una variable con ámbito de función puede referenciarse desde cualquier parte de la función. Las variables declaradas dentro del cuerpo de la función se dicen que son *locales* a la función. Las variables locales no se pueden utilizar fuera del ámbito de la función en que están definidas.

### **Ámbito de bloque**

Una variable declarada en un bloque tiene *ámbito de bloque* y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declaradas dentro de una función tienen ámbito de bloque de la función; no son visibles fuera del bloque. Una variable local declarada en un bloque anidado, sólo es visible en el interior de ese bloque.

### **Funciones locales**

Además de tener un ámbito restringido, las variables locales son especiales por otra razón, existen en memoria sólo cuando la función está activa (es decir, mientras se ejecutan las sentencias de la función). Cuando la función no se está ejecutando, sus variables locales no ocupan espacio en memoria, ya que no existen. Algunas reglas que siguen las variables locales son:

- Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir una variable con el mismo nombre. Cada variable es distinta y pertenece a su función específica.
- Las variables locales de las funciones no existen en memoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden compartir la misma memoria para sus variables locales (pero no al mismo tiempo).

## **CLASES DE ALMACENAMIENTO**

Los especificadores de clases de almacenamiento permiten modificar el ámbito de una variable.

### **Variables automáticas**

Las variables que se declaran dentro de una función se dice que son automáticas (*auto*), significando que se les asigna en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada *auto* es opcional y normalmente no se especifica.

### **Variables externas**

A veces se presenta el problema de que una función necesita utilizar una variable que otra función inicializa. Como las variables locales sólo existen temporalmente mientras se está ejecutando su función, no pueden resolver el problema. De lo que se trata es de que una función de un archivo de código fuente utilice una variable definida en otro archivo. Una solución es declarar la variable local con la palabra reservada *extern*.

Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro lugar.

<p><b>Archivo externa1.c</b></p> <pre>#include &lt;stdio.h&gt;  extern void leerReal(void);  float f;  int main() {     leerReal();     printf("\nEl valor de f=%f", f);     return 0; }</pre>	<p><b>Archivo externa2.c</b></p> <pre>#include &lt;stdio.h&gt;  void leerReal(){     extern float f;     printf("\nIntroduce f:");     scanf("%f",&amp;f); }</pre>
--	--

### Variables registro

Otro tipo de variable en C es la variable *registro*. Precediendo a la declaración de una variable con la palabra reservada *register*, se sugiere al compilador que la variable se almacene en uno de los registros hardware del microprocesador. La palabra *register* es una sugerencia al compilador y no una orden. La familia de microprocesadores 80x86 no tiene muchos registros hardware de reserva, por lo que el compilador puede decidir ignorar sus sugerencias.

Una variable registro debe ser local a una función, nunca puede ser global al programa completo. El uso de la variable *register* no garantiza que un valor se almacene en un registro. Esto sólo sucederá si existe un registro disponible. Si no existen registros suficientes, C ignora la palabra reservada *register* y crea la variable localmente como ya se conoce.

Una aplicación típica de una variable registro es como variable de control de un bucle. Guardando la variable de control de un bucle en un registro, se reduce el tiempo que la CPU requiere para buscar el valor de la variable de la memoria.

### Variables estáticas

Las variables estáticas son opuestas, en su significado, a las variables automáticas. Las variables estáticas no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable *static* se inicializa sólo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada *static*. Normalmente se utilizan para mantener valores entre llamadas a funciones.

```
#include <stdio.h>

void func();

int main() {
    int i;

    //Se llama a la función func() 100 veces.
    for( i = 0; i < 100; i++ )
        func();
}
```

```
}  
  
void func() {  
    //Se declara e inicializa una vez  
    static int a = 1;  
    //Cada invocación habrá aumentado en 1.  
    printf("La función se ha invocado %d veces", a);  
    a++;  
}
```

## FUNCIONES DE BIBLIOTECA

Todas las versiones del lenguaje C ofrecen una biblioteca estándar de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las funciones estándar o predefinidas, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo de cabecera. En los módulos de programa se pueden incluir líneas `#include` con los archivos correspondientes en cualquier orden.



# 10

## Arrays en C

### INTRODUCCIÓN

Las tablas son conocidas en C comúnmente como *arrays*, son colecciones de datos del mismo tipo que se referencian por un nombre y son almacenadas en posiciones de memoria físicamente contiguas donde la dirección de memoria más baja corresponde al primer elemento y la dirección de memoria más alta corresponde al último elemento del array.

### ARRAYS UNIDIMENSIONALES

#### Declaración

Al igual que ocurre con la definición de datos simples, para la definición de estructuras de datos más complejas, como los vectores, matrices, cadenas de caracteres, etc. es necesario especificar un tipo, un nombre o identificador, darle opcionalmente un valor inicial y además establecer un tamaño o dimensión.

El formato general para la definición de un array unidimensional es:

```
tipo_básico nombre_array[tamaño];
```

- Tipo básico. - Indica el tipo de dato del array o vector, que será común a todos los elementos que lo forman. Para ello utilizaremos los tipos básicos proporcionados por el lenguaje de programación y que en este caso son *char*, *int*, *float* y *double*. Su uso es obligatorio en la definición, pues de ella depende el espacio que posteriormente se reservará en memoria para el almacenamiento de los datos que la forman.
- Nombre\_array. - Es un identificador que deberá adaptarse a las reglas vistas anteriormente.
- Tamaño. - Indica la longitud o el número de elementos que contendrá el array.

#### Acceso a los datos

El acceso a un elemento específico de un array unidimensional se realiza a través de su nombre seguido del índice (posición relativa que ocupa dicho elemento dentro del array) entre corchetes.

```
nombre_array[expresión_índice];
```

La *expresión\_índice* es cualquier expresión de tipo entero. Teniendo en cuenta que en C los arrays tienen el 0 como índice de su primer elemento, *expresión\_índice* tomará valores entre 0 y N-1, siendo N el tamaño máximo del array definido.

#### Carga de datos

La carga de datos la podemos realizar de dos formas distintas:

1. Inicialización. Esta operación consiste en asignar una lista de valores a los diferentes elementos de un array, ya que C permite la inicialización de arrays tanto globales como locales en el momento de su definición. El formato general para la inicialización de un array de cualquier dimensión es el siguiente:

```
tipo_básico nombre_array[tamaño] = { lista  
valores };
```

Donde *lista valores* es una lista de constantes separadas por comas, cuyo tipo es compatible con el tipo básico especificado en la definición del array. Tiene que haber tantos valores como elementos tenga el array o menos. Si se ponen menos el resto hasta el final del array se rellenan con 0. Si se ponen más valores que elementos tiene el array ocurre un error.

2. Asignación.-Esta operación consiste en asignar de manera individual a cada elemento del array un valor.

```
int vector[4];
...
vector[0] = 35;
vector[1] = 22;
vector[2] = 1002;
vector[3] = 98;
```

Antes de utilizar un array, es conveniente darle un valor inicial. Para inicializar un array en tiempo de ejecución, la forma más cómoda y sencilla es recorrerlo secuencialmente desde el primer elemento hasta el último elemento al mismo tiempo que se asigna a cada componente o elemento un valor específico que puede ser el mismo o distinto para cada uno de ellos.

```
#include <stdio.h>
main() {

    int vector[7];
    int i;

    //Carga del vector a 0
    for( i = 0; i < 7; i++ )
        vector[i] = 0;
}
```

## ARRAYS BIDIMENSIONALES

### Declaración

El formato para la declaración de un array bidimensional es el siguiente:

```
tipo_básico nombre_array [tamaño dim 1][tamaño dim
2];
```

Donde *tipo básico* y *nombre\_array* tienen el mismo significado que si el array es unidimensional. *Tamaño dim 1* indica el tamaño de la primera dimensión del array, en este caso el número de filas, y *tamaño dim 2* indica el tamaño de la segunda dimensión del array, en este caso el número de columnas.

### Acceso a los datos

A diferencia de los arrays unidimensionales, en los arrays bidimensionales el acceso a cada elemento se realiza a través de dos índices, siendo el formato que se debe utilizar el siguiente:

```
nombre_array[expresión índice 1][expresión índice
2];
```

Donde *expresión índice 1* y *expresión índice 2*, indican la posición relativa que ocupa cada elemento dentro del array. El primer índice tomará valores entre 0 y N-1, siendo N el número de filas, y el segundo índice tomará valores entre 0 y M-1, siendo M el número de columnas.

## Carga de datos

La carga de datos la podemos realizar de las dos siguientes formas:

1. Inicialización.- Esta operación, al igual que con los arrays unidimensionales, sólo la podemos realizar en el momento de la definición. Su formato es el descrito anteriormente:  

```
tipo_básico nombre_array[T1][T2] = { lista
valores };
y además también podemos utilizar este otro:
tipo_básico nombre_array[T1][T2] = {      {
lista val 1 },
                                           { lista
val 2 },
                                           ...
                                           { lista val N }
                                           };
```

Como se puede observar consiste en introducir tantas listas de valores como filas tenga el array. Cada lista tendrá tantos valores como columnas tenga el array.

2. Asignación.-Esta operación consiste en asignar de manera individual a cada elemento del array un valor.  
Al igual que ocurre con los arrays unidimensionales, antes de utilizar una estructura de datos de este tipo, es conveniente darle un valor inicial. Para inicializar un array bidimensional en tiempo de ejecución, se recorre secuencialmente desde el primer elemento hasta el último elemento, al mismo tiempo que se asigna a cada componente o elemento del array un valor específico que puede ser el mismo o distinto para cada uno de ellos. Al ser el array bidimensional se necesitan dos bucles *for*, uno que recorra las filas y otro que recorra las columnas.

```
#include <stdio.h>
main() {

    int vector[7][5];
    int i, j;

    //Carga del array a 0
    for( i = 0; i < 7; i++ )
        for( j = 0; j < 5; j++ )
            vector[i][j] = 0;
}
```

## ARRAYS MULTIDIMENSIONALES

El formato para declarar un array multidimensional (tres o más dimensiones) es el siguiente:

```
tipo_básico nombre_array[Tamaño 1][Tamaño
2]...[Tamaño N];
```

Para acceder o referenciar a un elemento individual utilizaremos el siguiente formato:

```
nombre_array[exp_ind1][exp_ind2]...[exp_indN];
```

C permite utilizar arrays de cualquier dimensión, aunque no suelen ser utilizados arrays de más de tres dimensiones, debido a la gran cantidad de memoria necesaria para

almacenarlos y a la dificultad que supone su tratamiento, pues se debería manejar un número elevado de índices.

## ARRAYS DE CARACTERES

Las cadenas de caracteres son arrays unidimensionales de tipo carácter (*char*) que requieren un tratamiento especial frente a los arrays unidimensionales con otros tipos de datos (*int*, *float* o *double*).

### Declaración

El formato para declarar un array de caracteres es el siguiente:

```
char nombre_cadena[tamaño + 1];
```

Toda cadena de caracteres debe finalizar con el carácter nulo, lo que indica que, para una cadena de longitud N, solamente podemos utilizar los N-1 primeros elementos. Esto obliga a crear un array cuya longitud deberá ser superior en uno al número de caracteres que comprende la tira. Por ejemplo, la cadena:

```
char nombre[7];
```

tiene la siguiente representación gráfica

p	e	p	i	t	o	\0
---	---	---	---	---	---	----

### Acceso a datos

La forma de acceder o referenciar un elemento de un array de caracteres es la misma que para arrays unidimensionales. Otra característica de los arrays de tipo carácter es que el nombre del array es un puntero que siempre apunta al primer elemento del array.

### Carga de datos

La carga de datos la podemos realizar de la siguiente manera:

1. Inicialización.- El formato para la inicialización de un array de caracteres es el siguiente:

```
char nombre_cadena[Tamaño] = "cadena";
```

Utilizando este método de inicialización, el carácter nulo es añadido automáticamente al final de la cadena. Existe otro formato para la inicialización de array de caracteres que es el siguiente:

```
char nombre_cadena[Tamaño] = { 'c1', 'c2',  
..., 'cN', '\0' };
```

El carácter nulo no debe ocupar obligatoriamente el último elemento o componente del array, sino el último elemento o componente después del último valor añadido en la tira de caracteres, ya que la longitud de la cadena (número de caracteres que la forman), puede ser inferior al espacio reservado para ella, quedando posiciones del array sin ocupar a la derecha del carácter nulo.

Siempre es conveniente inicializar una cadena antes de utilizarla y limpiarla antes de volver a utilizarla. Para realizar dicha operación podemos emplear cualquiera de las dos sentencias de asignación:

```
nombre_cadena[0] = '\0';
```

```
*nombre_cadena = '\0';
```

La inicialización de un array de cadenas se puede realizar de la siguiente forma:

```
char nombres[4][7] =  
{ "JUAN", "PEDRO", "LUIS", "MARIA" };
```

2. Asignación.-Esta operación consiste en asignar de manera individual a cada elemento del array un carácter delimitado por comillas simples y, una vez que asignamos todos aquellos caracteres que queremos que formen

parte de la tira de caracteres, asignamos el terminador de cadena o carácter nulo.

## FUNCIONES DE ENTRADA/SALIDA PARA CADENAS

**char \* gets( char \*cadena );**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	Nombre de la cadena o array donde deseamos almacenar los caracteres leídos.
<b>Valor devuelto</b>	Puntero al primer carácter de la cadena leída.
<b>Finalidad</b>	Lee una cadena de caracteres desde teclado y la almacena en la variable cadena especificada como parámetro.

Esta función es más adecuada para leer cadenas de caracteres que scanf(), ya que está última leería una cadena hasta el primer espacio en blanco, mientras que gets() lee hasta la pulsación de INTRO, que posteriormente convierte a carácter nulo (con el que todas las cadenas terminan).

Los caracteres leídos son almacenados en el array *cadena*. Se leen todos aquellos caracteres que se encuentran antes del carácter de nueva línea o de final de fichero. El carácter de retorno de carro no formará parte integrante de la cadena de caracteres, siendo sustituido por el carácter nulo. Si se ejecuta correctamente, gets() devuelve un puntero al primer carácter de la cadena leída. En caso de producirse algún error devolverá un puntero nulo quedando indeterminado el contenido del array *cadena*. La función gets() no tiene un límite en la lectura consecutiva de caracteres.

**int puts( char \*cadena );**

<b>Prototipo</b>	stdio.h
<b>Argumentos</b>	Cadena de caracteres a visualizar
<b>Valor devuelto</b>	El último carácter escrito o EOF
<b>Finalidad</b>	Escribe una cadena en el dispositivo de salida estándar.

El carácter nulo que marca el final de la cadena de caracteres es convertido al carácter de nueva línea. La función puts() sólo puede imprimir cadenas de caracteres, lo que la hace más limitada que printf(), pero su ejecución es más rápida.

Si se ejecuta correctamente, devuelve el último carácter escrito en el dispositivo estándar de salida, siendo siempre en este caso el carácter de salto de línea; en caso contrario devuelve EOF.

## ARRAYS INDETERMINADOS

Son arrays, que se caracterizan porque pueden ser declarados e inicializados al mismo tiempo sin necesidad de especificar su tamaño, pues C calcula automáticamente las dimensiones del array definido, reservando suficiente espacio en memoria para contener todos los caracteres presentes más el delimitador de cadena o carácter nulo.

El formato de declaración de un array indeterminado es el siguiente:

```
tipo_básico nombre_array[] = "Lista de valores"
```

Por ejemplo:

```
char cadena1[]="Acceso denegado.";
```

También es posible la utilización de arrays indeterminados con arrays de dos o más dimensiones. En estos casos, es necesario especificar el tamaño de todas las dimensiones a excepción de la situada más a la izquierda.

```
tipo_básico nombre_array[][Tam2]...[TamN] = lista  
valores;
```

# 11

## Punteros y gestión dinámica de memoria

### INTRODUCCIÓN

Un **puntero** es una variable que contiene como valor una dirección de memoria, que suele ser la dirección de memoria de otra variable; por tanto, podemos decir que el puntero *apunta a dicha variable*. La variable apuntada o referenciada suele ser de cualquier tipo básico (*char, int, float, double*), derivado (*puntero*) o estructurado (*tabla, estructura, etc.*).

Los punteros tienen una gran utilidad en C, y constituyen una de sus principales características y elementos de mayor potencia, pues permiten una programación eficaz a nivel de máquina (bajo nivel) cuando se maneja directamente la memoria del ordenador. La utilización de punteros en el lenguaje C proporciona las siguientes ventajas:

- En el caso de paso de variables a una función por dirección o referencia, es necesario emplear parámetros formales como punteros.
- Permiten realizar operaciones de asignación dinámica de memoria.
- Permiten efectuar operaciones con estructuras de datos dinámicas.

Los punteros se deben usar con gran precaución, pues al manejar posiciones de memoria pueden provocar fallos en el programa, siendo difíciles de localizar y solucionar.

Los principales errores que se suelen cometer con el uso de punteros son los siguientes:

- Asignación incorrecta de direcciones, por lo que se puede llegar a escribir en zonas de memoria que contengan código o datos del programa.
- Operaciones incorrectas con los punteros, cuando no se tiene en cuenta el tipo y el valor de las variables apuntadas

### CONCEPTOS BÁSICOS

#### Definición de punteros y asignación de direcciones

Para la definición de punteros y asignación de direcciones a los mismos se utilizan los operadores de puntero **\*** y **&**.

Para definir un puntero se utiliza el operador **\*** de tipo monario, que afecta directamente al nombre de la variable a la cual precede. El formato es:

```
tipo_básico *nombre_puntero;  
tipo_básico *puntero1, *puntero2, ..., * punteroN
```

Cuando se declaran varios punteros no hay que cometer el error de utilizar un operador **\*** para declararlos todos. Por ejemplo

```
int *pnum, num;
```

En esta definición la primera variable *pnum* es un puntero a un entero y la segunda es un entero.

El tipo de dato empleado en la definición del puntero debe ser del mismo tipo de dato que las posibles variables a las que dicho puntero puede apuntar. En el caso de que el tipo de dato sea *void*, se obtiene una definición de puntero de tipo genérico de tal forma que su tipo de dato implícito será el de la variable cuya dirección se le asigne.

Cuando se define un puntero, su contenido o valor no está definido, pues en ese momento se considera como basura el contenido de su posición de memoria. Para evitar

que un puntero se utilice indebidamente cuando no se le ha asignado todavía la dirección de una variable se le puede inicializar a un valor nulo.

```
int *pnum = NULL;
```

Para indicar la dirección de una variable se utiliza el operador **&** de tipo monario precediendo al nombre de la variable; por tanto, en una variable hay que diferenciar cuándo se está empleando su valor (referenciando por su nombre) o cuando se está empleando su dirección (al preceder su nombre con el símbolo **&**).

```
int cant;
//Se utiliza el nombre para asignar a la variable
el valor 15
cant = 15;
```

```
&cant; //Indica la dirección de la variable cant
```

Para que una variable puntero apunte a otra variable es necesario asignar la dirección de dicha variable al puntero. El tipo de dato del puntero debe coincidir con el tipo de dato de la variable apuntada, excepto en el caso de un puntero genérico (con el tipo de dato void). La asignación de la dirección se puede realizar de dos formas:

1. Por inicialización en la definición del puntero.

```
float num;
float *pnum = &num;
```

2. Por una sentencia de asignación después de haber definido el puntero

```
float *pnum;
pnum = &num;
```

Para visualizar la dirección contenida en un puntero se puede utilizar el formato **%p** con la función printf().

Después de tener asignada una dirección a un puntero, durante la ejecución del programa se le puede volver a asignar la dirección de otra variable, apuntando por tanto a esa variable y abandonando el apuntamiento anterior. Debe mantenerse la condición de que el tipo de dato de la variable que se va a apuntar sea del mismo tipo que el del puntero.

```
float nume, contador;
float *pnum = &nume;
...
pnum = &contador;
```

## Indirección

Se entiende por indirección la forma de referenciar el valor de una variable a través de un puntero que apunta a dicha variable. Para realizar la indirección se utiliza el operador monario **\*** que, aplicado al nombrel del puntero correspondiente, indica el valor de la variable cuya dirección está contenida en dicho puntero.

```
int a = 40, b;
int *pnum = &a;
//pnum apunta a a. Ahora cambio el valor de a
accediendo con //el puntero
*pnum = 568;

//Asigno a b el valor de a. Accedo a a con el
puntero
b = *pnum;
```

Hay que tener en cuenta cuándo se utiliza el valor de la variable apuntada utilizando la indirección y cuándo se utiliza la dirección contenida en el puntero.



## OPERACIONES CON PUNTEROS

En el lenguaje C se pueden realizar una serie de operaciones con punteros que son: asignación de punteros, aritmética de punteros y comparación de punteros.

### Asignación de punteros

Es posible asignar un puntero a otro puntero, con la condición de que sean del mismo tipo de dato. Cuando esta operación se efectúa, los dos punteros están apuntando a la misma variable, pues contienen la misma dirección de memoria.

```
int a = 5, b, c;
int *p1 = &a, *p2;

p2 = p1; //p2 se carga con la dirección contenida
        en p1

b = *p1;
c = *p2;
//b y c tienen el mismo valor
```

### Aritmética de punteros

Las operaciones aritméticas que se pueden realizar con punteros son las siguientes:

#### Suma o resta un número entero a un puntero

Si se suma o se resta un número entero a un puntero, lo que se produce es un incremento o decremento de la dirección contenida en dicho puntero. El número de posiciones de memoria incrementadas o decrementadas dependerá del número empleado y del tipo de dato del puntero.

$$\text{nombre\_puntero} + n = \text{dirección} + n * \text{tamaño tipo de datos}$$

Ejemplo:

```
float num, *punt, *pnum;
punt = &num;
pnum = punt + 2;
/*
Si la dirección contenida en punt es 4274, la
dirección de
pnum es 4274 + 2 * 4 = 4282
*/
```

El avance por las posiciones de memoria depende del tipo de dato apuntado. Con los punteros a variables de tipo *char*, la suma de un número entero es siempre de un byte, mientras que para los demás tipos el avance depende del tamaño en bytes del tipo de dato correspondiente, es decir, del espacio que en memoria se reserva para ellos.

### Incremento y decremento de punteros

El incremento o decremento de punteros es similar a las operaciones vistas anteriormente, es decir, la suma de un número entero a un puntero, con la particularidad de que en este caso el valor incrementado siempre es 1 y, por tanto, la dirección contenida en el puntero se incrementa o decrementa en las posiciones indicadas por el tamaño del tipo de dato del puntero. Se pueden emplear los operadores ++ y --.

```
float *pnum;
...
pnum++;
```

La variable puntero *pnum*, después del incremento contiene una dirección incrementada en el número de bytes correspondientes al tamaño del tipo de dato puntero, que al ser de tipo *float*, será de 4 bytes.

El decremento en los punteros es similar al incremento, disminuyendo la dirección contenida en el puntero en tantas posiciones de memoria como indique el tipo de dato del puntero. Hay que tener cuidado con las prioridades de los operadores \*, ++, -- para no producir confusiones.

```
//Primero incrementa pnum y luego obtiene su valor
*pnum++

//Primero obtiene el valor apuntado por pnum y
luego lo //incrementa.
(*pnum)++
```

### Resta de dos punteros

Es una operación que se realiza para conocer el número de elementos (del tipo definido) que separan las direcciones apuntadas por los punteros. Se suele utilizar con estructuras lineales de datos como las tablas.

```
int tabula[10], d;
int *ptab1 = &tabula[1];
int *ptab2 = &tabula[4];
d = ptab1 - ptab2;
//d contiene el número de elementos separados por
los //punteros.
```

### Comparación de punteros

Se utiliza normalmente la comparación de punteros para conocer las posiciones relativas en memoria que ocupan las variables apuntadas por punteros. Generalmente, la operación más habitual es la de igualdad, para saber si dos punteros están apuntando a la misma variable. Ejemplo

```
int a;
int *ptab1 = &a;
int *ptab2 = &a;
if( ptab1 == ptab2 ) printf("Los dos punteros
apuntan a a");
```

## PUNTEROS Y ARRAYS

En el lenguaje C, los punteros y los arrays están estrechamente relacionados. Para facilitar su estudio vamos a diferenciar entre punteros y arrays unidimensionales (vectores), punteros y cadenas de caracteres y, por último, punteros y arrays bidimensionales (tablas).

### Punteros y arrays unidimensionales

El nombre del array (sin índice) es un puntero que contiene la dirección del primer elemento del array. El nombre del array es una constante de tipo puntero y por tanto el compilador no permitirá que se pueda cambiar, en las instrucciones del programa, la dirección contenida en el nombre del array.

```
int tabula[20];
//tabula es equivalente a &tabula[0]
//tabula++ es errónea, ya que no puede cambiar su
valor
```

Se pueden utilizar variables puntero que contengan la dirección de un array, lo que posibilita la realización de todas aquellas operaciones permitidas con punteros.

```
//carga la dirección del primer elemento
int *ptab = tabula; //Igual a int *ptab =
&tabula[0]
//Incrementa puntero ptab (apunta al 2º elemento)
ptab++;
//ptab avanza cuatro elementos, apunta al sexto
elemento
ptab += 4;
```

La referencia al valor e un elemento del array se puede hacer por medio de la indirección de un puntero.

```
int tabula[30], num;
int *ptab = tabula; //Puntero apunta al principio
del vector
num = *(ptab + 4); //num tiene el valor del 4º
elemento
```

Para leer un elemento de un array

```
scanf("%d", ptab + 4 );
```

Para escribir un elemento del array

```
printf("%d", *(ptab + 4) );
```

Una vez definido un puntero se puede indexar como se hace con el nombre del array.

```
int *ptab1 = tabula;
printf("%d", ptab[0]) // equivale a printf("%d",
*ptab);
printf("%d", ptab[8]) // equivale a printf("%d",
*(ptab+8));
```

Esta indexación de punteros sólo es válida cuando se utiliza para apuntar a array, siendo errónea en los demás casos.

```
int a,b;
int *punt = &a;
b = punt[5]; //Error
```

También es posible utilizar el nombre de un array como puntero para referenciar a los elementos del array.

```
int vector[10];
...
printf("%d", *(vector+3) );
```

La diferencia entre el uso del nombre del array como puntero y el uso de un puntero definido específicamente para contener la dirección del array es que el primero es una constante y siempre debe apuntar a la misma dirección de memoria (posición del primer elemento del array), mientras que el segundo es una variable que puede cambiar su contenido, o sea que puede contener la dirección de cualquier elemento del array; incluso se puede volver a utilizar la variable puntero para apuntar a otro array distinto, siempre que sus elementos sean del mismo tipo que el definido para el puntero.

```
int num[20], cant[8];
int *punt = num; //punt apunta al array num
...
punt = cant; //punt apunta al array cant
```

Como resumen se puede decir que un array se puede manejar con índices o con puntros (utilizando aritmética de punteros). Dentro de un mismo módulo de un programa es más cómodo para el programador el uso de la indexación del array, sobre todo para acceder a determinados elementos del array, pero cuando hay que utilizar parámetros formales (ver capítulo de funciones) en una función que recibe la dirección de un array es

necesario utilizar y definir variables puntero. Estas variables puntero, dentro del módulo donde están definidas, se pueden manejar bien como puntero, o bien con la indexación correspondiente a un array. El uso de punteros, a pesar de ser más difícil, presenta la siguientes ventajas:

- Mayor rapidez en la ejecución del programa.
- Utilizan menos memoria.

### Punteros y cadenas de caracteres

Una cadena de caracteres queda definida como un array unidimensional terminada en el carácter nulo.

```
char saludo[] = "Hola, como estás";
```

La cadena de caracteres ocupa en memoria 17 bytes y está referenciada por el nombre de la cadena *saludo* que se corresponde con la dirección del primer carácter. Asimismo, se puede utilizar un puntero para definir la cadena de caracteres.

```
char *psal = "Hola, como estás";
```

En el primer caso, *saludo* es un puntero constante, pudiendo modificar caracteres individuales dentro de la cadena, pero el puntero *saludo* siempre se refiere a la misma posición de memoria. En el segundo caso, *psal* es una variable puntero inicializada para apuntar a una cadena constante, no debiéndose modificar el contenido de la cadena y teniendo la ventaja de que este puntero se puede volver a utilizar para apuntar a otra cadena. Son muy útiles los punteros a cadenas para definir mensajes de cierta longitud, que deban ser utilizados repetidas veces en un programa.

```
char *pmens = "El número de caracteres contenido
              en la línea es superior a 80
              caracteres";
...
printf(pmens);
```

### Punteros y arrays bidimensionales

El uso de punteros en arrays multidimensionales es similar al caso de arrays unidimensionales. El nombre del array es la dirección del primer elemento del array.

```
int matriz[10][7];
int *pmat = matriz; //Equivale a int *pmat =
&matriz[0][0]
```

Para referenciar un elemento mediante un puntero habrá que utilizar fórmulas de transformación. En el caso de un array bidimensional definido con *f* filas y *c* columnas, el elemento que ocupa las posiciones de la fila *i* y la columna *j* se referencia por medio de un puntero con

```
*(nombre_puntero + i * c + j)
```

Ejemplo

```
int temp[15][30];
printf( "%d", *(temp + 3 * 30 + 6) );
//Equivale a printf("%d", temp[3][6]);
```

## ARRAYS DE PUNTEROS

Podemos definir arrays de punteros de la misma manera que definimos arrays de cualquier otro tipo de dato, con la peculiaridad de que los elementos del array serán direcciones de memoria. Su formato es

```
tipo_básico *nombre_puntero[num_elementos];
```

Ejemplo:

```
int *apunt[15]; //Vector de 15 punteros a enteros
int cant;
```

```

apunt[6] = &cant;    //El elemento 6 apunta a cant
*apunt[6] = 37; //cant vale 37, que se ha cargado
                con el //puntero
printf("El valor de cant es: %d", *apunt[6]);

```

El uso más generalizado de arrays de punteros es para establecer arrays de punteros a cadenas de caracteres, especialmente para manejo y tratamiento de arrays de mensajes.

```

char *mensajes[] = { "Error de lectura",
                    "Error de escritura",
                    "Error de compilación" };

```

Se puede referenciar un determinado mensaje en función del índice.

```

printf("%s" mensajes[1]);

```

Se podía haber utilizado un array bidimensional para definir el array de mensajes, realizando la siguiente definición.

```

char mensajes[3][21];

```

Esta definición ocupa más memoria, mientras que el array de punteros ocupa la memoria estrictamente necesaria para almacenar las cadenas. El único problema es que en el array de punteros no se puede expandir ninguna cadena, debiendo redefinir la cadena correspondiente.

## INDIRECCIÓN MÚLTIPLE

La indirección múltiple consiste en que un puntero contiene la dirección de otro puntero que a su vez apunta a una variable. Su formato de definición es el siguiente:

```

tipo_básico **nombre_puntero;

```

Ejemplo

```

int cant;
int *pcan = &cant;
int **ppcan = &pcan;
**pcan = 57;
print("%d", cant); //visualiza 57

```

En realidad se está utilizando un formato de un array bidimensional de punteros, tal como se describía en el apartado anterior para arrays de punteros a cadenas, pues cada elemento del array maneja direcciones de memoria.

```

char nom[35];
char *pnom = nom;
char **ppnom = &pnom;
gets(*ppnom);
printf("%s", nom);

```

La indirección múltiple se puede extender a más niveles, sin embargo, no es conveniente pasar del segundo nivel de puntero a puntero, pues se presentan dificultades en el seguimiento del programa así como en el tratamiento y manejo de los punteros.

## FUNCIONES DE ASIGNACIÓN DINÁMICA DE MEMORIA

Hasta ahora sólo hemos manejado estructuras de datos que se caracterizaban porque obligaban a definir de antemano un tamaño y estructura para que en tiempo de compilación se reservase en memoria el suficiente espacio para ellas (estructuras de datos estáticas), permaneciendo invariables a lo largo de la ejecución del programa. Pero existe una segunda manera de almacenar y tratar la información sobre la memoria principal y que es mediante la asignación dinámica de memoria. Este segundo método

nos da opción a reservar el espacio que necesitemos en memoria durante la ejecución del programa permitiéndonos posteriormente su liberación, dando la posibilidad de que en otra parte del programa se use, en otro momento, la misma zona de memoria con un cometido diferente. Esta manera de utilizar la memoria la odemos repetir tantas veces como sea necesario durante la ejecución de un programa, teniendo siempre la precaución de ir liberando aquellas porciones o parcelas de memoria reservadas anteriormente y que ya no vamos a volver a utilizar.

En este capítulo trataremos aquellas herramientas necesarias para la asignación dinámica de memoria y que C proporciona por medio de funciones de librería.

Las funciones de librería que el ANSI C define para las operaciones de asignación dinámica de memoria son las descritas a continuación:

**void \*calloc( size\_t num, size\_t tam);**

<b>Prototipo</b>	stdlib.h
<b>Argumentos</b>	Número de elementos a reservar y tamaño de cada uno de ellos
<b>Valor devuelto</b>	Devuelve un puntero al primer byte del bloque de memoria reservado o un puntero NULL en el caso de no haberse podido reservar el bloque de memoria solicitado.
<b>Finalidad</b>	Reserva un bloque de memoria para almacenar <i>num</i> elementos de <i>tam</i> bytes cada uno de ellos.

Los argumentos necesarios para utilizar la función *calloc()* son:

- **num:** Indica el número de elementos, con la misma estructura, que ocuparán el bloque de memoria reservado.
- **tam:** Indica el tamaño en bytes de cada uno de los elementos que van a ocupar del bloque de memoria reservado.

La cantidad de memoria reservada viene determinada por el resultado que se obtiene al multiplicar el número de elementos a almacenar en el bloque por el tamaño en bytes de cada uno de esos elementos, es decir *num \* tam*. Ejemplo:

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

int *reserva_memoria( int num_elementos ) {

    int *pt;
    pt = (int *)calloc( num_elementos, sizeof(int) );
    if( !pt )
        printf("Imposible reservar memoria");

    return pt;
}
```

**void \*malloc(size\_t tam);**

<b>Prototipo</b>	stdlib.h
<b>Argumentos</b>	Tamaño en bytes del bloque de memoria a reservar.
<b>Valor devuelto</b>	Devuelve un puntero al primer byte del bloque de memoria reservado o un puntero NULL en el caso de no haberse podido reservar el bloque de memoria

	solicitado.
<b>Finalidad</b>	Reserva un bloque de memoria de <i>tam</i> bytes para almacenar información.

Los argumentos necesarios para utilizar la función *malloc()* son:

- **tam:** Indica el tamaño en bytes del bloque de memoria que se desea reservar.

Es muy importante comprobar que el puntero devuelto por *malloc()* no es un puntero nulo antes de hacer uso de él.

**void free( void \*puntero );**

<b>Prototipo</b>	stdlib.h
<b>Argumentos</b>	Puntero que apunta a la zona de memoria que se desea liberar.
<b>Valor devuelto</b>	Ninguno
<b>Finalidad</b>	Libera el bloque de memoria apuntado por <i>puntero</i> y que previamente había sido asignado mediante <i>malloc()</i> o <i>calloc()</i> , dando la posibilidad a que dicho bloque de memoria se pueda volver a asignar dinámicamente.

Los argumentos necesarios para utilizar la función *free()* son:

- **puntero:** Variable puntero que debe contener la dirección de memoria del primer byte del bloque de memoria que deseamos liberar. En el caso de que no sea un puntero previamente utilizado con *malloc()* o con *calloc()*, la llamada a la función *free()* puede ocasionar una parada o interrupción del sistema.

Ejemplo:

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

main() {
    char *nombres[50];
    int j;

    //Se reserva espacio para 50 nombres
    for( j = 0; j < 50; j++ ) {
        if( (nombres[j] = (char*)malloc(45) ) == NULL )
        {
            printf("Memoria insuficiente");
            exit(1);
        }
        gets( nombres[j] );
    }
    //Se libera la memoria ocupada
    for( j = 0; j < 50; j++ )
        free( nombres[j] );
}
```

**void \*realloc( void \*ptr, size\_t nuevotam);**

<b>Prototipo</b>	stdlib.h
<b>Argumentos</b>	Puntero que apunta al bloque de memoria reservado y

	tamaño en bytes del bloque de memoria a reservar.
<b>Valor devuelto</b>	Devuelve un puntero al primer byte del nuevo bloque de memoria reservado o un puntero NULL en el caso de no haberse podido reservar el bloque de memoria solicitado.
<b>Finalidad</b>	Cambia el tamaño del bloque de memoria apuntada por <i>ptr</i> al nuevo tamaño indicado por <i>nuevotam</i> .

Los argumentos necesarios para utilizar la función *realloc()* son:

- **ptr**: Puntero que apunta al bloque de memoria reservado.
- **nuevotam**: Valor en bytes que indica el nuevo tamaño del bloque de memoria apuntado por *ptr* y que puede ser mayor o menor que el original.

Ejemplo:

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

main() {
    char *saludo = "Hola";
    char nombre[] = "Juan";
    char *pt;
    int j;

    //Se reserva bloque de memoria
    pt = (char *) malloc(5);
    if( !pt ) {
        printf("Memoria insuficiente");
        exit(1);
    }
    pt = (char *)saludo;

    //Muestra el contenido de pt
    for( j = 0; j < 5; j++ )
        printf("%c", *(pt+j) );

    //Modificar el tamaño de bloque reservado
    pt = (char *)realloc( pt, 10 );
    if( !pt ) {
        printf("Memoria insuficiente");
        exit(1);
    }
    strcat( pt, nombre );
    printf("\n%s", pt);
    free(pt);
}
```



# 12

## Estructuras compuestas

### INTRODUCCIÓN

Las estructuras compuestas son estructuras de datos basadas en los tipos de datos ya vistos en capítulos anteriormente y que el usuario puede construir de la forma que crea más conveniente. En C, las estructuras compuestas se pueden clasificar en :

- Estructuras
- Campos de bits
- Uniones
- Enumeraciones

Existen también tipos de datos que el usuario puede definir en base a otros tipos de datos básicos o estructurados, empleando la palabra clave *typedef*.

### ESTRUCTURAS

Una estructura es una variable definida por el programador que está compuesta por un conjunto de variables que se referencian bajo un mismo nombre. Las variables o elementos de la estructura están relacionadas de una forma lógica desde el punto de vista de la información que tratan.

Para poder emplear una estructura es necesario hacer una declaración de la misma mediante la palabra clave *struct*, estableciendo así un patrón o plantilla de la estructura que se va a emplear. Formato de la declaración:

```
struct nombre_estructura {  
    tipo elemento1,  
    tipo elemento2,  
    ...  
    tipo elementoN;  
};
```

Ejemplo

```
struct cliente {  
    int dni;  
    char nom[35];  
    char direc[45];  
    float cuenta;  
};
```

Con las sentencias del ejemplo se ha realizado una declaración de plantilla tipo *cliente*, pero todavía no se ha definido en memoria ninguna variable. La declaración suele hacerse de forma global para permitir la definición de las variables del tipo estructura en los diversos módulos o funciones, sin tener que volver a declarar la plantilla en cada función.

#### Definición de variables

Existen dos formas para realizar la definición de las variables. La primera es en la misma declaración de la plantilla. El formato sería:

```
struct nombre_estructura {  
    tipo elemento1,
```

```

        tipo elemento2,
        ...
        tip elementoN;
    } lista_variables;

```

**Ejemplo**

```

struct cliente {
    int dni;
    char nom[35];
    char direc[45];
    float cuenta;
} client1, client2, cliente3;

```

Con esta sentencia se ha declarado una plantilla del tipo *cliente* y a la vez se han definido en memoria tres variables *client1*, *client2*, *cliente3* que contienen los elementos del tipo declarado.

Existe un caso particular en el que se puede suprimir el nombre de la estructura cuando se definen las variables en la declaración:

```

struct {
    tipo elemento1,
    tipo elemento2,
    ...
    tip elementoN;
} lista_variables;

```

**Ejemplo**

```

struct {
    int dni;
    char nom[35];
    char direc[45];
    float cuenta;
} client1, client2, client3;

```

Con esta sentencia, en el ejemplo anterior, se han definido tres variables, pero más adelante no se puede volver a definir otras variables, pues no se tiene el nombre de la plantilla. Es conveniente por tanto especificar el nombre de la plantilla para poder utilizarlo en cualquier función.

La segunda forma de declarar variables es después de haber realizado la declaración de la estructura:

```

struct nombre_estructura lista_variables;

```

**Ejemplo**

```

struct cliente client4, client5;

```

Con esta sentencia se han definido en memoria dos variables *client4* y *client5* del tipo *cliente*.

**Referencia a elementos**

Para hacer referencia a cada uno de los elementos de la estructura se emplea el operador *.* (punto). El formato es:

```

nombre_variable_estructura.nombre_elemento

```

Siguiendo el ejemplo anterior, para referenciar los elementos de la variable *client1*.

```

client1.dni, client1.nom, client1.direc,
client1.cuenta

```

Para acceder a un carácter de un elemento del tipo cadena de caracteres:

```

nombre_variable_estructura.nombre_elemento[índice]

```

**Ejemplo**

```

client1.nom[3]

```

Para leer los elementos se utilizarán las funciones de lectura dependiendo del tipo de datos de cada elemento. Por ejemplo

```
gets(client1.nom);
scanf("%d",&client1.dni);
```

Para escribir los elementos se utilizarán las funciones de escritura según el tipo de datos de cada elemento. Por ejemplo:

```
puts(client1.nom);
printf("%d", client1.dni);
printf("%c", client1.direc[1]);
```

### Carga de datos

Para asinar valores a los datos de la estructura se puede realizar de dos formas. La primera es especificando en el programa los valores de los datos, al definir las variables mediante una asignación de valores a los elementos expresados por una lista entre llaves. Ejemplo:

```
struct cliente client1 = { 34567,
                           "Antonio López Gómez",
                           "C/ Mayor 25 - MADRID",
                           12500.5 };
```

Se puede también hacer la inicialización de la misma forma cuando se hace la definición a la vez que la declaración.

Posteriormente se pueden asignar valores directamente en operaciones de asignación como se si trataran de variables simples. Ejemplo:

```
strcpy(client1.nom, "Antonio López Gómez");
strcpy(client1.direc, "C/ Mayor 25 - MADRID");
client1.dni = 34567;
client1.cuenta = 125000.5;
```

También es posible asignar todos los valores de los elementos de una variable estructura a otra variable estructura del mismo tipo. Ejemplo

```
client1 = client2;
```

Además, se pueden introducir los valores directamente desde teclado a los elementos de la estructura, utilizando para ello las funciones de entrada de datos correspondientes al tipo de datos de cada elemento.

```
scanf("%d", &client1.dni);
gets(client1.nom);
gest(client1.direc);
scanf("%f", &client1.cuenta);
```

## ESTRUCTURAS ANIDADAS

Existe la posibilidad de que un elemento de una estructura sea a su vez otra estructura. En una plantilla de estructuras se pueden anidar otras estructuras que deben estar previamente declaradas. Su formato de declaración es:

```
struct nombre_estructura_1 {
    tipo elemento1,
    tipo elemento2,
    ...
    tipo elementoN;
};

struct nombre_estructura_2 {
    tipo elemento_a,
```

```

    struct nombre_estructura_1 elemento_b;
    tipo elemento_c,
    ...
    tip elementoN;
};

struct nombre_estructura_2 nombre_variable;

```

Ejemplo

```

struct dir {
    char calle[25];
    int codpos;
    char poblacion[20];
};

```

```

struct cliente {
    int dni;
    char nom[35];
    struct dir direc;
    float cuenta;
};

```

```

struct cliente client1;

```

Para referenciar elementos de la estructura anidada se emplea sucesivamente el operador punto.

```

nombre_variable.elemento_b.elemento_1

```

Ejemplo:

```

client1.direc.calle;

```

## ARRAY DE ESTRUCTURAS

Las estructuras, igual que los demás tipos de datos, pueden formar parte de un array. El formato de declaración del array sería:

```

struct nombre_estructura nombre_variable[tamaño];

```

Ejemplo:

```

struct cliente clientetab[50];

```

Con esta definición se crea una tabla de 150 elementos, todos de tipo cliente. La referencia a un elemento de la tabla se hará usando el índice correspondiente. Ejemplo:

```

clientetab[5].nom

```

El tratamiento de los arrays de estructuras es similar al de los arrays de elementos de tipo básico, teniendo en cuenta la forma de referenciar sus elementos.

## PUNTEROS Y ESTRUCTURAS

Los punteros a estructuras funcionan del mismo modo que los punteros a otros tipos de variable pero se emplea un operador específico para referenciar los valores de sus elementos. Para definir un puntero a un tipo de estructura:

```

struct nombre_estructura *nombre_puntero;

```

Ejemplo:

```

struct cliente *pclient;

```

Para asignarle luego una dirección

```

pclient = &client;

```

```

pcliien = clientetab;

```

Para referenciar un elemento de una estructura apuntada por un puntero se debe emplear el operador flecha ->, que está compuesto por dos símbolos, el guión y el mayor que.

Ejemplo:

```
pclient->nom
pclient->direc
```

Si se referencia a un elemento de una estructura anidada:

```
pclient->direc.calle
```

De igual forma se podrán leer sus elementos, accediendo a ellos con el operador flecha.

```
gets(pclient->nom);
gets( (pclient + 2)->direc.calle);
scanf("%f", &((pclient + 1)->cuenta) );
```

De forma análogo, para escribir un elemento

```
puts(pclient->nom);
puts( (pclient + 2)->direc.calle);
printf("%f", (pclient + 1)->cuenta );
```

## FUNCIONES Y ESTRUCTURAS

El paso de estructuras a funciones se puede realizar de las dos formas que se describen a continuación

### Paso de elementos individuales

Cuando los elementos se pasan por valor, se tratan igual que variables simples. En la llamada a la función:

```
func1( client1.cuenta, clien1.dni );
```

En la declaración de la función

```
void func1( float cuen, int iden ) { ... }
```

Si los elementos se pasan por referencia hay que pasar la dirección de memoria de cada elemento. En la llamada a la función:

```
func1( &client1.cuenta, clien1.dni );
```

En la declaración de la función:

```
void func1( float *cuen, int iden ) { ... }
```

### Paso de la estructura completa

Es conveniente que la plantilla de la estructura esté declarada a nivel global. Cuando se pasa por valor se está pasando a la función todos los elementos de la estructura. En la llamada a la función:

```
func1( client1 );
```

En la declaración de la función:

```
void func1( struct cliente cli ) { ... }
```

Por otro lado, si la estructura se pasa por referencia hay que pasar la dirección de la estructura. En la llamada a la función:

```
func1( &client1 );
```

En la declaración de la función:

```
void func1( struct cliente * cli ) { ... }
```

Las funciones pueden retornar una estructura o un puntero a una estructura. En la llamada a la función:

```
client = func1();
```

En la declaración de la función:

```
struct cliente func1() {
    struct cliente client;
    ...
    return client;
```

}

## CAMPOS DE BITS

Los campos de bits son estructuras compuestas de elementos cuyo tamaño se expresa con uno o más bits. El compilador guarda la información de los campos de bits en la menor unidad de memoria posible. Con este tipo de estructura es posible acceder a los bits de un byte o de una palabra, almacenar variables lógicas en el menor espacio posible y realizar tratamientos sobre los bits de un dispositivo.

La definición de una estructura con campos de bits se realiza de forma similar a las estructuras, pero expresando la longitud de cada campo con un número de bits. Su formato es:

```
struct nombre_estructura {
    tipo elemento1: longitud1,
    tipo elemento2: longitud2,
    ...
    tipo elementoN: longitudN;
} lista_variables;
```

El tipo de dato de cada campo de bits debe ser *int* o *unsigned int* y la longitud es el tamaño en bits que tiene cada campo (en pun PC es de 1 a 16 bits). Si la longitud es 1 el tipo del campo debe ser necesariamente *unsigned int*, pues no puede contener un valor y un signo al mismo tiempo. Ejemplo

```
struct func {
    unsigned depart:4; //15 departamentos
    unsigned sexo: 1; //1 varón, 0 mujer
    unsigned estado:2; //01 soltero 02 casado ...
    unsigned situacion:1; //1 activo 0 excedencia
} funcionario;
```

Con la variable *funcionario* (utilizando campos de bits) se ha reunido información en un solo byte, mientras que si se hubieran utilizado campos de tipo *int* se habrían necesitado por lo menos 8 bytes. Para referenciar un campo de bits se emplea el mismo procedimiento que en las estructuras. No es obligatorio expresar todos los bits de un byte o una palabra. Tampoco es necesario dar nombre a los campos de bits que no se van a utilizar y que facilitan el acceso a los campos de bits que interesan.

```
struct func {
    unsigned departamento:4;
    unsigned: 3;
    unsigned situacion:1;
} funcionario;
```

En una estructura se pueden combinar variables normales con campos de bits. Ejemplo

```
struct func {
    long dni;
    char nom[35];
    char direc[45];
    float sueldo;
    unsigned estado: 2;
    unsigned situacion: 1;
} funcionario;
```

El uso de los campos de bits tiene una serie de limitaciones:

1. No se puede obtener la dirección de un campo de bits, pues la menor unidad direccionable es el byte.

2. Se pueden producir problemas con la portabilidad, pues en cada máquina puede ser distinto el orden de funcionamiento de los campos, de izquierda a derecha o viceversa.
3. En algunos compiladores, los campos de bits no puede formarse como arrays.

## UNIONES

Una unión es una estructura en la que sus elementos ocupan la misma posición de memoria. El tamaño de la unión es el correspondiente al elemento mayor de la unión y los demás elementos comparten dicha memoria. En cada momento de la ejecución solamente puede haber en la unión un valor, correspondiente al tipo de cada elemento, aunque haya espacio suficiente para contener a la vez varios valores de distinto tipo, debiendo saber el programador qué clase de información está contenida en la unión en cada caso.

La declaración de plantilla, definición de variables y referencia a los elementos es similar a las estructuras con la diferencia de emplear la palabra reservada **union**.

Declaración de la union:

```
unión nombre_union {
    tipo elemento1;
    tipo elemento2;
    ...
    tipo elementoN;
};
```

Ejemplo:

```
union tipos_diversos{
    int num;
    char letra;
    float importe;
};
```

La definición de variables de tipo unión puede hacerse en la declaración de la plantilla o después de haberla declarado.

```
unión nombre_union {
    tipo elemento1;
    tipo elemento2;
    ...
    tipo elementoN;
} lista_variables;
```

Ejemplo:

```
union tipos_diversos{
    int num;
    char letra;
    float importe;
} grupo;
```

Si se realiza después de la declaración:

```
union nombre_union lista_variables;
```

Ejemplo:

```
union tipos_diversos grupo;
```

Cuando se quiera hacer referencia a los elementos individuales de la unión se hará igual que en las estructuras.

```
grupo.num = 2;
```

```
union tipos_diversos *pu = &grupo;
pu->letra;
```

Las uniones se pueden utilizar para las conversiones de tipo y para que el código del programa sea más portable.

## ENUMERACIONES

Una enumeración es una estructura de constantes numéricas enteras cuyos nombres especifican todos los valores deseados que dicha estructura puede tener. La declaración de la plantilla y la definición de variables es similar a las estructuras, con la diferencia de utilizar la palabra reservada **enum** y que sus elementos son una lista de constantes.

```
enum nombre_enum{
    constante1,
    constante2,
    ...
    constanteN
};
```

Ejemplo:

```
enum colores {
    rojo,
    amarillo,
    verde,
    azul
};
```

Para definir variables en la declaración

```
enum nombre_enum{
    constante1,
    constante2,
    ...
    constanteN
} lista_variables;
```

Ejemplo:

```
enum colores {
    rojo,
    amarillo,
    verde,
    azul
} color;
```

Si las variables se definen después de la declaración

```
enum nombre_enum lista_variables;
```

Ejemplo:

```
enum colores color;
```

Internamente, los elementos que forman parte de una enumeración son constantes, cuyos valores (numéricos enteros) son asignados por el compilador siguiendo una progresión de izquierda a derecha, es decir, cero para la constante situada más a la izquierda, uno para la situada una posición a la derecha y así sucesivamente.

Internamente, en el ejemplo anterior la enumeración *color* tendría los siguientes valores: *rojo*=0, *amarillo*=1, *verde*=2 y *azul*=3.

Se pueden redefinir los valores que el compilador puede asignar, es decir, inicializando las constantes con los valores que se deseen, recordando que cada variable no inicializada es superior en una unidad a la constante anterior. Ejemplo:



```
enum colores {
    rojo,
    amarillo =4,
    verde,
    azul
} color;
```

Internamente, los valores son: *rojo=0, amarillo=4, verde=5, azul=6*. Se pueden redefinir todos los valores de las constantes, llegando a dar el mismo valor a constantes con distinto nombre.

```
enum colores {
    rojo = 1,
    granate = 1,
    amarillo = 5,
    verde = 10,
    azul = 15
} color;
```

Para referenciar a los elementos se utiliza el nombre de la constante correspondiente.

Ejemplo:

```
if( color == verde )
    printf("El color es verde \n");
```

Una variable enumeración es un número entero y no se puede mostrar como una cadena de caracteres en una sentencia de escritura. Ejemplo

```
printf("El color de la pared es %s", azul );
```

En la sentencia anterior, se visualizaría: *El color de la pared es 3*.

Si se quieren visualizar las cadenas equivalentes a las constantes que se incluyen en las enumeraciones, se puede hacer de dos formas. La primera es con una sentencia *switch*.

```
enum arq { romanico, gotico, plateresco, neoclasico
} estilo;
switch( estilo ) {
    case romanico: {
        printf("romanico");
        break;
    }
    case gotico: {
        printf("gotico");
        break;
    }
    case plateresco: {
        printf("plateresco");
        break;
    }
    case neoclasico: {
        printf("neoclasico");
    }
}
```

La otra forma es con una tabla de cadenas. Esta forma está condicionada a respetar el valor que el compilador asigna a cada constante.

```
char estilos[][11] = {"romanico", "gotico",
    "plateresco", "neoclasico" };
printf("El estilo es %s", estilos[estilo]);
```

## TIPOS DE DATOS DEFINIDOS POR EL USUARIO

El usuario o programador puede definir en el lenguaje C nuevos nombres para los tipos de datos existentes, mediante la palabra clave **typedef**. El formato es:

```
typedef tipo_dato nombre_nuevo;
```

Ejemplo

```
typedef short float real;
```

Con esta definición no se ha creado un tipo de dato nuevo, sino que se emplea otro nombre para un tipo de dato existente que sigue estando activo. Se pueden utilizar diversos nombres para un mismo tipo de dato. Ejemplo:

```
typedef long mayor_entero;
```

```
typedef long grande;
```

En el programa, para una variable de tipo *long int* se pueden usar los siguientes nombres para definir su tipo:

```
long longitud;
```

```
mayor_entero longitud;
```

```
grande longitud;
```

Las ventajas del uso de *typedef* son principalmente dos: La primera es hacer más portable el código de una máquina a otra, pues si manejan tamaños distintos para un tipo de dato determinado sería suficiente modificar la sentencia typedef para ajustarse al nuevo tamaño.

```
typedef int entero; //En máquina de 32 bits
```

```
typedef long int entero; //En máquina de 16 bits
```

La otra ventaja es hacer que el tipo de dato sea más explícito para el usuario.

```
typedef float Moneda;
```

```
//Para usuario con variables que representan un  
valor moneda
```

```
Importe precio;
```

```
Importe total;
```

También se puede utilizar *typedef* para tipos de datos complejos y compuestos.

```
typedef client {  
    long dni;  
    char nom[35];  
    char direc[45];  
}Cliente;
```

En el programa se usaría:

```
Cliente amigo;
```

```
Cliente num_clientes[120];
```

# 13

## Estructuras dinámicas internas de datos en C

### INTRODUCCIÓN

Este capítulo está dedicado a la implementación en lenguaje C de las estructuras dinámicas de datos (**edd**) vistas en el capítulo 6. En concreto nos ocuparemos de las listas enlazadas simples, listas enlazadas dobles, listas circulares, pilas, colas. Primero comenzaremos haciendo una introducción de las operaciones necesarias para crear cada una de estas estructuras y posteriormente veremos las funciones necesarias para la gestión completa de cada una de ellas.

Es muy conveniente que se tengan muy claros los conceptos vistos en el capítulo anterior, ya que los punteros serán de aplicación exhaustiva.

Como se habrá podido observar a través de los ejercicios propuestos, las estructuras dinámicas de datos se aplican en multitud de campos o áreas. Por tanto, cada nodo en alguna de estas edd puede tener un valor del campo información muy variado. Debido a esto en un programa C que gestione este tipo de edd tendrá que definir primero la estructura del nodo, es decir, indicar que información contendrá cada nodo. Para ello se deberá de definir una plantilla mediante la palabra clave *struct* que permita indicar esto. A partir de ahí, se podrán declarar variables de este tipo de estructura. Para todos los ejemplos que se verán en el presente capítulo vamos a suponer el caso más simple: el campo información de cada nodo es un número entero. Según esto, la estructura que representa un nodo sería la siguiente:

```
struct nodo {  
    int inf;  
    struct nodo *siguiente;  
}
```

Como se puede observar, la estructura tiene dos campos:

1. **inf.**- Campo información del nodo, de tipo entero.
2. **siguiente.**- Puntero al siguiente nodo.

Si nos fijamos el puntero *siguiente* apunta a un tipo *struct nodo*, que es lo que se está definiendo. C permite este tipo de referencias *circulares*.

### LISTAS ENLAZADAS SIMPLES

#### Creación

Para crear una lista enlazada simple, solamente hay que declarar un puntero a la estructura nodo e inicializarlo a valor NULL, que será la forma de saber que la lista está vacía.

```
struct nodo *lista = NULL;
```

#### Insertar un nodo en la lista

En esta operación hay que crear un nuevo nodo y posteriormente insertarlo en la lista, teniendo en cuenta que los nodos siguen una secuencia ordenada en función del valor de algún campo de la información que contienen. Para ello se expone a continuación la función *InsertarLista* con el siguiente prototipo.

```
void InsertarLista( struct nodo **);
```

Como se puede observar el parámetro que recibe es el puntero al principio de la lista, pero teniendo en cuenta que su valor puede cambiar, es necesario pasarlo por referencia. De ahí que el tipo sea un doble puntero a *struct nodo*.

Independientemente del lugar que ocupe el nodo nuevo en la lista, es necesario crear el nodo reservando memoria para el mismo y rellenando su campo información. El puntero *siguiente* en principio lo establecemos a valor NULL. Posteriormente se cambiará su valor, si es necesario.

```
nuevo = (struct nodo *)malloc( sizeof( struct
nodo ) );
printf("\nIntroduce el número del nuevo nodo: ");
scanf("%d", &nuevo->inf);
getchar();
nuevo->siguiente = NULO;
```

En la inserción de un nuevo nodo en la lista, hay que tener en cuenta varias posibilidades.

#### La lista está vacía.

```
if( *lista == NULO ) {
    //Lista vacia
    *lista = nuevo;
}
```

#### El nuevo nodo va en primer lugar

```
else if( (*lista)->inf > nuevo->inf ) {
    nuevo->siguiente = *lista;
    *lista = nuevo;
}
```

#### El nuevo nodo va en medio de la lista o al final

```
else {
    anterior = *lista;
    siguiente = anterior->siguiente;

    while( siguiente != NULO && siguiente->inf <
nuevo->inf ) {
        anterior = siguiente;
        siguiente = siguiente->siguiente;
    }
    anterior->siguiente = nuevo;
    nuevo->siguiente = siguiente;
}
```

#### Borrar un nodo de la lista

En este caso, habrá que liberar la memoria del nodo eliminado, previamente se habrá localizado en la lista para poder realizar la reasignación de punteros necesaria. El prototipo de la función es:

```
void BorrarLista( struct nodo ** );
```

Al igual que antes, habrá que tener en cuenta donde se encuentra el nodo para realizar unas operaciones u otras. En primer lugar hay que leer de teclado el valor del campo información del nodo a borrar para poder buscarlo. Al igual que antes, hay que enviar como parámetro por referencia el nodo al principio de la lista, ya que puede ser modificado.

```
printf("\nIntroduce el nodo a borrar: ");
scanf("%d",&inf);
getchar();
```

**La lista está vacía**

```

if( !(*lista) ) {
    printf("\n La lista está vacia");
    getchar();
}

```

**El nodo está en primer lugar**

```

else {
    if( (*lista)->inf == inf ) {
        anterior = *lista;
        *lista = (*lista)->siguiente;
        free( anterior );
    }
}

```

**El nodo está en medio o al final de la lista**

```

else {
    anterior = *lista;
    siguiente = anterior->siguiente;

    while( siguiente && siguiente->inf < inf ) {
        anterior = siguiente;
        siguiente = siguiente->siguiente;
    }

    if( siguiente == NULO || siguiente->inf > inf
) {
        printf("\nEl nodo no está en la lista");
        getchar();
    }
    else {
        anterior->siguiente = siguiente->siguiente;
        free( siguiente );
    }
}

```

**Recorrido de la lista en sentido ascendente**

Para recorrer la lista se utilizará una función recursiva. No es necesario enviar el puntero al principio de la lista por referencia, ya que no se va a modificar.

```

void VisualizarListaA( struct nodo *lista ) {

    if( lista ) {
        printf("%d -> ", lista->inf );
        VisualizarListaA( lista->siguiente );
    }
    else {
        printf("NULO");
    }
}

```

**Recorrido de la lista en sentido descendente**

Si en la función anterior se invierte el orden de las sentencias que visualizan el nodo información y avanza al siguiente nodo, nos encontramos con que la lista se recorre en orden inverso. La función sería:

```

void VisualizarListaD( struct nodo *lista ) {

    if( lista ) {

```

```

        VisualizarListaD( lista->siguiente );
        printf("%d -> ", lista->inf );
    }
}

```

### Búsqueda de un nodo

Para saber si existe un nodo con determinada información se puede emplear una función que recorra la lista hasta encontrarlo, si existe. Devolvería la posición donde se encuentra, que podría ser utilizada como valor lógico. Si no está se devuelve 0.

```

int BuscarNodo( struct nodo *lista, int inf ) {
    int pos = 1;
    while( lista && lista->inf < inf ) {
        lista = lista->siguiente;
        pos++;
    }
    if( !(lista && lista->inf == inf) ) pos = 0;
    return pos;
}

```

## LISTAS ENLAZADAS DOBLES

En este caso, cada nodo tiene un puntero que apunta al siguiente en la lista y otro que apunta al anterior. Debido a esto la plantilla para cada nodo sería:

```

struct nodo {
    int inf;
    struct nodo *siguiente; //Puntero al siguiente
    struct nodo *anterior; //Puntero al nodo anterior
};

```

### Creación

La creación de una lista enlazada doble es similar a la creación de una lista enlazada simple. Se declara una variable puntero a nodo y se inicializa a null.

```

struct nodo *lista = NULO;

```

### Insertar un nodo en la lista

Las operaciones a realizar y los condicionantes son similares a los que existen en las listas enlazadas simples. Sin embargo, ahora habrá que tener en cuenta la existencia de un segundo puntero que apunta al nodo anterior. El prototipo de la función es.

```

void InsertarLista( struct nodo ** );

```

Las primeras operaciones son la de reserva de memoria del nuevo nodo y rellenado de sus campos información.

```

//Se reserva memoria para el nuevo nodo
nuevo = (struct nodo *)malloc( sizeof( struct nodo ) );

```

```

//Leemos el campo información
printf("\nIntroduce el número del nuevo nodo:");
scanf("%d", &nuevo->inf);
getchar();

```

```

//Por si acaso, asignamos nulo a ambos punteros
nuevo->anterior = NULO;

```

```
nuevo->siguiente = NULO;
```

A continuación, el nodo se inserta en la lista. Según el lugar que ocupe existen varias posibilidades.

**La lista está vacía.**

```
if( !(*lista) ) {      //Equivalente a if( *lista ==
    NULO )
    //La lista está vacía
    *lista = nuevo;
}
```

**El nuevo nodo va en primer lugar**

```
else if( (*lista)->inf > nuevo->inf ) {
    //Ahora comprobamos que vaya en primer lugar
    nuevo->siguiente = *lista;
    (*lista)->anterior = nuevo;
    *lista = nuevo;
}
```

**El nuevo nodo va en medio o al final de la lista**

```
else {
    //El nuevo nodo está en medio o al final
    anterior = *lista;
    siguiente = (*lista)->siguiente; //También
anterior->siguiente
    printf("primero");

    while( siguiente && siguiente->inf < nuevo->inf
) {
        anterior = anterior->siguiente;
        siguiente = siguiente->siguiente; //También
anterior->siguiente
    }

    nuevo->siguiente = siguiente;
    printf("segundo");
    nuevo->anterior = anterior;
    printf("tercero");
    anterior->siguiente = nuevo;
    printf("cuarto");
    if( siguiente ) {
        siguiente->anterior = nuevo; // también if(
siguiente != NULO )
    }
}
```

**Borrar un nodo de la lista**

Muy similar a la eliminación de nodos en las listas enlazadas simples, salvo la dificultad añadida del segundo puntero apuntando al nodo anterior. El prototipo de la función es:

```
void BorrarLista( struct nodo ** );
```

Inicialmente, hay que recoger del teclado la información relativa al nodo que se pretende borrar para poder buscarlo.

```
printf("\nIntroduce el numero del nodo a borrar:
");
scanf("%d", &inf );
```

```
getchar();
```

#### La lista está vacía

```
if( !(*lista) ) { // if( *lista != NULO )
    printf("\nLa lista está vacía");
    getchar();
}
```

#### El nodo está en primer lugar

```
else if( (*lista)->inf == inf ) {
    anterior = *lista;
    *lista = (*lista)->siguiente;

    if( *lista ) (*lista)->anterior = anterior-
>anterior; //Será NULO
    free(anterior);
}
```

#### El nodo está en medio o al final

```
else {
    anterior = *lista;
    siguiente = anterior->siguiente;

    while( siguiente && siguiente->inf < inf ) {
        anterior = siguiente;
        siguiente = siguiente->siguiente;
    }

    if( siguiente && siguiente->inf == inf ) {
        anterior->siguiente = siguiente->siguiente;
        if( siguiente->siguiente ) siguiente-
>siguiente->anterior = anterior;
        free( siguiente );
    }
    else {
        printf("\nEl nodo no está en la lista");
    }
}
```

#### Recorrido de la lista en sentido ascendente

Similar a la función de recorrido en listas enlazadas simples.

```
void VisualizarListaA( struct nodo *lista ) {
    if( lista ) {
        printf("%d -> ", lista->inf );
        VisualizarListaA( lista->siguiente ) ;
    }
    else
        printf("NULO");
}
```

#### Recorrido de la lista en sentido descendente

Similar a la función de recorrido descendente en listas enlazadas simples.

```
void VisualizarListaD( struct nodo *lista ) {
    if( lista ) {
        VisualizarListaD( lista->siguiente ) ;
        printf("%d -> ", lista->inf );
    }
}
```



```
    }
}
```

### Recorrido en ambos sentidos

La mayor utilidad de las listas enlazadas dobles es la posibilidad de recorrerla en ambos sentidos o invertir el sentido del recorrido en un momento determinado. Para ello se ha escrito la siguiente función en la que cada vez que se presenta la información de un nodo, se pregunta al usuario que nodo quiere ver, si el siguiente o el anterior. La función es:

```
void VisualizarAmbosSentidos( struct nodo *lista,
char sentido ) {

    char opc;

    if( lista ) {
        //El nodo no es nulo. Presenta la información
        printf("%d ", lista->inf );

        //Determinar si se quiere ir al siguiente o al
        anterior
        printf("\n¿Quiere ver el [a]nterior o el
[s]iguiente? ");
        scanf("%c", &opc);
        getchar();

        switch( opc ) {
            case 'A':
            case 'a': {
                VisualizarAmbosSentidos( lista->anterior,
'a' );
                break;
            }
            case 'S':
            case 's': {
                VisualizarAmbosSentidos( lista->siguiente,
's' );
            }
        }
    }
    else if( sentido == 'a' )
        printf("Principio de la lista");
    else printf("Final de la lista");
}
```

## LISTAS CIRCULARES

La estructura del nodo es similar a la de la lista enlazada simple, con un campo información y un puntero al nodo siguiente.

### Creación

Al igual que en las listas vistas anteriormente, la creación de una lista circular solamente necesita de la declaración de una variable apuntero de tipo *struct nodo* e inicializarla a NULL.

```
struct nodo *lista = NULO;
```

### Insertar un nodo en la lista

Esta operación es muy similar a la inserción en una lista enlazada simple, la única diferencia es la forma de reconocer el final de la lista. En las listas enlazadas simples se ha llegado al final de la lista cuando el puntero siguiente del nodo que se está visitando apunta a NULL. En las listas circulares, este nodo apunta al primer nodo de la lista o a NULL si está vacía. El prototipo de la función es:

```
void InsertarLista( struct nodo ** );
```

Aquí también hay que pasar como parámetro el puntero que apunta al principio de la lista y por referencia ya que su valor puede cambiar.

Primero hay que reservar memoria para el nuevo nodo que se pretende insertar, rellenar su campo información y establecer el puntero siguiente a NULL.

```
nuevo = (struct nodo *)malloc( sizeof( struct nodo
) );
nuevo->siguiente = NULO;
```

```
printf("\nIntroduce el valor del nuevo nodo: ");
scanf("%d", &nuevo->inf);
getchar();
```

Ahora se insertará el nodo nuevo en la lista. Para ello se ejecutarán unas sentencias u otras en función del lugar que le corresponde.

#### La lista está vacía

```
if( !(*lista) ) { //if( *lista == NULO )
    //La lista está vacía
    *lista = nuevo;
    nuevo->siguiente = nuevo;
}
```

#### El nuevo nodo va en primer lugar

```
else if( (*lista)->inf > nuevo->inf ) {
    //Va en primer lugar
    nuevo->siguiente = *lista;
    (*lista)->siguiente = nuevo;
    *lista = nuevo;
}
```

#### El nuevo nodo va en medio de la lista o al final

```
else {
    //Va en medio o al final
    anterior = *lista;
    siguiente = (*lista)->siguiente;

    while( siguiente != *lista && siguiente->inf <
nuevo->inf ) {
        anterior = siguiente;
        siguiente = siguiente->siguiente;
    }

    anterior->siguiente = nuevo;
    nuevo->siguiente = siguiente;
}
```

### Borrar un nodo en la lista

En este caso, al igual que en los demás tipos de estructuras dinámicas, habrá que liberar la memoria que estaba utilizando el nodo eliminado. Antes habrá que localizarlo y si existe, reasignar los punteros. El prototipo de la función es:

```
void BorrarLista( struct nodo ** );
```

En este caso habrá que tener especial cuidado cuando se borre el último nodo, ya que habría que redirigir el puntero del penúltimo para que apunte al primero.

Inicialmente hay que leer desde teclado el valor del campo información del nodo a borrar para poder buscarlo.

```
printf("\nIntroduce el valor del nodo a borrar:
");
scanf("%d",&inf);
getchar();
```

#### La lista está vacía

```
if( !(*lista) ) { //if( *lista == NULL )
    printf("\nLista vacia. Pulse INTRO");
    getchar();
}
```

#### El nodo está en primer lugar

```
else if( (*lista)->inf == inf ) {
    //Se va a borrar el primer nodo de la lista
    //Busca el final de la lista a partir del 2º
nodo
    anterior = (*lista)->siguiente;
    //Parará cuando el puntero anterior apunte al
último nodo de la lista
    while( anterior->siguiente != *lista ) {
        anterior = anterior->siguiente;
    }

    siguiente = *lista;
    //Asigno el puntero del último nodo para que
apunte al principio
    anterior->siguiente = (*lista)->siguiente;
    //El nuevo principio es el 2º nodo
    if( *lista == (*lista)->siguiente ) *lista =
NULO;
    else *lista = (*lista)->siguiente;

    free(siguiete);
}
```

#### El nodo está en medio o al final de la lista

```
else {
    anterior = *lista;
    siguiente = (*lista)->siguiente;

    while( siguiente != *lista && siguiente->inf <
inf ) {
        anterior = siguiente;
        siguiente = siguiente->siguiente;
    }
}
```

```

        if( siguiente->inf == inf ) {
            anterior->siguiente = siguiente->siguiente;
            free( siguiente );
            printf( "\nNodo borrado" );
        }
        else {
            printf( "\nEl nodo no está en la lista" );
        }
    }
}

```

### Recorrido de la lista en sentido ascendente

Se ofrecen funciones para el recorrido de una lista ascendente. La primera es iterativa y la segunda es una versión recursiva. En este caso y para la versión recursiva hay que pasar como parámetro el puntero al inicio de la lista, ya que necesita reconocer de alguna forma cuando ha recorrido la lista completa.

```
void VisualizarLista( struct nodo *lista ) {
```

```

    struct nodo *principio = lista;

    if( lista ) {
        printf( "%d ->", lista->inf );
        lista = lista->siguiente;
        while( lista != principio ) {
            printf( "%d ->", lista->inf );
            lista = lista->siguiente;
        }
    }
    printf( "FINAL" );
}

```

```
void VisualizarListaA( struct nodo *lista, struct
nodo *principio ) {
```

```

    if( lista != principio ) {
        printf( "%d -> ", lista->inf );
        VisualizarListaA( lista->siguiente, principio
    );
    }
    else {
        printf( "FINAL" );
    }
}

```

### Búsqueda de un nodo

Tal y como se ha visto anteriormente, esta función servirá para localizar un nodo en la lista. Devolverá la posición donde se encuentra o 0 si no está.

```
int BuscarNodo( struct nodo *lista, int inf ) {
```

```

    int pos = 1;
    struct nodo *principio = lista;

    if( lista ) {

```

```

        while( lista->inf < inf && principio != lista-
>siguiente ) {
            lista = lista->siguiente;
            pos++;
        }
        if( lista != principio || lista->inf == inf )
return pos;
        else return 0;
    }
    else {
        return 0;
    }
}

```

## PILAS

Una pila es un caso especial de una lista enlazada simple, aquél en el que las inserciones y eliminaciones se realizan por un extremo, en ambos casos por el mismo. Este extremo es conocido como la **cima de la pila**. Esta particularidad simplifica enormemente el código de las funciones de gestión de una pila.

La estructura de cada nodo es similar al de las listas enlazadas simples. Para los ejemplos que se verán a continuación, supondremos que la pila no tiene límite. Queda como ejercicio, realizar una gestión completa de una pila con un límite  $n$ .

### Creación

Para crear una pila, basta con declarar una variable puntero a nodo y asignarle el valor NULL.

```
struct nodo *pila = NULL;
```

### Apilar

Para apilar un nodo, hay que reservar memoria para el nodo, rellenar el campo información y posteriormente introducirlo en la cima de la pila.

```

void Apilar( struct nodo **pila, int inf ) {
    struct nodo *nuevo;
    nuevo = (struct nodo *)malloc( sizeof( struct
nodo ) );
    printf("\nIntroduce el número del nuevo nodo: ");
    &nuevo->inf = inf;

    nuevo->siguiente = *pila;
    *pila = nuevo;
}

```

### Desapilar

Eliminación del nodo que hay en la cima de la pila. Es muy útil y frecuente que el valor del campo información del nodo desapilado se devuelva.

```

int Desapilar( struct nodo **pila ) {
    struct nodo *auxiliar;
    int inf = 0;

    if( *pila ) {
        auxiliar = *pila;
        inf = auxiliar->inf;
        *pila = (*pila)->siguiente;
    }
}

```

```

        free( auxiliar );
    }
    else {
        printf("\nPila vacía");
    }
    return inf;
}

```

### Cima de la pila

Consiste en obtener el valor del campo información del nodo que se encuentra en la cima de la pila, pero sin eliminarlo.

```

int Cima( struct nodo *pila ) {
    if( pila ) {
        return pila->inf;
    }
    else {
        printf("\nPila vacía");
        return 0;
    }
}

```

### Búsqueda de un nodo

Consiste en recorrer la pila encontrar un nodo con un determinado valor del campo información. Teniendo en cuenta que la pila es una lista simple, podría usarse una función que la recorriera. Sin embargo, aquí emplearemos otro método para ajustarnos a una característica fundamental de la pila, por la cual solamente se pueden realizar tratamiento a los nodos que se extraen de la pila. Consiste en sacar nodos hasta dejar en la cima el que se pretende buscar, para posteriormente volver a apilar los nodos que se sacaron en orden inverso y dejar la pila tal y como estaba.

Aunque en principio parece enrevesado, utilizando las funciones de *Apilar* y *Desapilar* vistas anteriormente se simplifica bastante.

```

int BuscarNodo( struct nodo *pila, int inf ) {

    struct nodo *pilaux = NULO;
    int pos = 1;

    while( pila && pila->inf != inf ) {
        Apilar( &pilaux, Desapilar( &pila ) );
        pos++;
    }

    if( pila ) {
        while( pilaux ) Apilar( &pila, Desapilar(
&pilaux ) );
        return pos;
    }
    else
        return 0;
}

```

## COLAS

Una cola es un caso especial de una lista enlazada simple, aquél en el que las inserciones se producen por un extremo (el principio de la cola) y las eliminaciones por el otro (el final de la cola). La estructura de cada nodo es similar al de las listas enlazadas simples.

Para facilitar su implementación, son necesarios dos punteros: uno apunta al final de la cola y el otro al principio. Para hacerlo se puede optar por una de dos posibles maneras:

1. Una estructura con dos campos del mismo tipo: un puntero al tipo estructura nodo.
2. Dos variables puntero del mismo tipo: un puntero al tipo estructura nodo.

Para el código desarrollado a continuación, se ha optado por la primera posibilidad, con lo que hay que escribir la estructura que contiene los dos punteros de la cola.

```
//Estructura con el principio y el final de la cola
struct cola {
    struct nodo *principio;
    struct nodo *final;
};
```

### Creación

Una vez declarada una variable del tipo estructura con los dos punteros al principio y fin de la cola, hay que inicializarlos a valor NULL, que indica que la cola está creada.

```
struct cola iniciofin;
iniciofin.principio = NULL;
iniciofin.final = NULL;
```

### Insertar un nodo en la cola

La inserción de un nodo nuevo en la cola es una operación muy simple debido a que siempre se realizan por un extremo, por el principio. Primero se reserva memoria para el nuevo nodo y se rellena el campo información.

```
nuevo = (struct nodo *)malloc( sizeof( struct
nodo ) );
printf("\nIntroduce el valor del nuevo nodo: ");
scanf("%d", &nuevo->inf );
getchar();
nuevo->siguiente = NULL;
```

Ahora, se inserta en la cola

```
if( !iniciofin->principio ) { //if( iniciofin-
>principio != NULL )
    iniciofin->principio = nuevo;
    iniciofin->final = nuevo;
}
else {
    iniciofin->principio->siguiente = nuevo;
    iniciofin->principio = nuevo;
}
```

Como se puede observar se envía como parámetro por referencia la estructura con los dos punteros de inicio y fin de la cola. Solamente es necesario un puntero simple, ya que lo que se pasa por referencia es una estructura, y no un puntero a nodo, como ocurría con las listas.

### Borrado de un nodo de la cola

Las eliminaciones también son operaciones muy simples, al hacerlo por el final de la cola.

```
void BorrarCola( struct cola *iniciofin ) {

    struct nodo *auxiliar;

    if( iniciofin->principio ) { //if( iniciofin-
>principio != NULO )
        auxiliar = iniciofin->final;
        iniciofin->final = iniciofin->final->siguiente;
//auxiliar->siguiente;
        free( auxiliar );

        if( !iniciofin->final ) iniciofin->principio =
NULO;
    }
}
```

Como se podrá observar, no es necesario solicitar el campo información del nodo a borrar, ya que siempre se borra por el final de la cola.

### Recorrido de una cola

Consiste en comenzar por el principio y visitar los nodos, haciendo algún tipo de tratamiento, hasta llegar al final de la cola. En este caso, como en todos los anteriores, el tratamiento a realizar será la visualización por pantalla del campo información.

```
void VisualizarCola( struct cola iniciofin ) {
    struct nodo *auxiliar;
    auxiliar = iniciofin.final;
    printf("Cola: ");
    while( auxiliar ) {                //while( auxiliar !=
NULO )
        printf("%d -> ", auxiliar->inf );
        auxiliar = auxiliar->siguiente;
    }
    printf("Final");
}
```

### Búsqueda de un nodo en la cola

Consiste en recorrer la cola, comenzando por el principio, para buscar un nodo. Habrá que rellenar por teclado el campo información del nodo que se pretende buscar. Si lo encuentra devuelve la posición en la que se encuentra, y si no, devuelve 0.

```
int BuscarNodo( struct cola iniciofin, int inf ) {

    struct nodo *auxiliar;
    int pos = 1;

    auxiliar = iniciofin.final;
    while( auxiliar && auxiliar->inf != inf ) {
        auxiliar = auxiliar->siguiente;
        pos++;
    }

    if( auxiliar ) return pos;
```



```
    else return 0;  
}
```

## 14

## Ficheros en C

## FLUJOS

Un **flujo** (*stream*) es una abstracción que se refiere a un *flujo o corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal (*«pipe»*) por la que circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo, por el canal que comunica el archivo con el programa van a fluir las secuencias de datos. Hay tres flujos o canales abiertos automáticamente:

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Estas tres variables se inicializan al comenzar la ejecución del programa para admitir secuencias de caracteres, en modo texto. Su cometido es el siguiente:

stdin.- Asocia la entrada estándar (teclado) con el programa.

stdout.-Asocia la salida estándar (pantalla) con el programa.

stderr.- Asocia la salida de mensajes de error (pantalla) con el programa.

Así cuando se ejecuta `printf("Calle Mayor 2.")`; se escribe en stdout, en pantalla; si se desea leer una variable entera con `scanf("%d", &x)`; se captan los dígitos de la secuencia de entrada stdin.

El acceso a los archivos se hace con un buffer intermedio. Se puede pensar en el buffer como un array donde se van almacenando los datos dirigidos al archivo, o desde el archivo; el buffer se vuelca cuando de una forma u otra se da la orden de vaciarlo. Por ejemplo, cuando se llama a una función para leer del archivo una cadena, la función lee tantos caracteres como quepan en el buffer. A continuación se obtiene la cadena del buffer; una posterior llamada a la función obtendrá la siguiente cadena del buffer y así sucesivamente hasta que se quede vacío y se llene con una llamada posterior a la función de lectura.

El lenguaje C trabaja con archivos con buffer, y está diseñado para acceder a una amplia gama de dispositivos, de tal forma que trata cada dispositivo como una secuencia, pudiendo haber secuencias de caracteres y secuencias binarias. Con las secuencias se simplifica el manejo de archivo en C.

## PUNTERO FILE

Los archivos se ubican en dispositivos externos como cintas, cartuchos, discos, disco compactos, etc. y tienen un nombre y unas características. En el programa el archivo tiene un nombre interno que es un

puntero a una estructura predefinida (puntero a archivo). Esta estructura contiene información sobre el archivo, tal como la dirección del buffer que utiliza, el modo de apertura del archivo, el último carácter

leído del buffer y otros detalles que generalmente el usuario no necesita saber. El identificador del tipo de la estructura es `FILE` y está declarada en el archivo de cabecera `stdio.h`:

```
typedef struct{
```

```

short level;
unsigned flags; /*estado del archivo: lectura, binario ... */
char fd;
unsigned char hold; short bsize;
unsigned char *buffer, *curp; unsigned istemp; short token;
}FILE;

```

El detalle de los campos del tipo FILE puede cambiar de un compilador a otro. Al programador le interesa saber que existe el tipo FILE y que es necesario definir un puntero a FILE por cada archivo a procesar. Muchas de las funciones para procesar archivos son del tipo FILE \*, y tienen argumento(s) de ese tipo.

#### Ejemplo

Se declara un puntero a FILE; se escribe el prototipo de una función de tipo puntero a FILE y con un argumento del mismo tipo.

```

FILE* pf;
FILE* mostrar(FILE*); /* Prototipo de una función definida por
el programador* /

```

Cabe recordar que la entrada estándar al igual que la salida están asociadas a variables puntero a FILE:

```
FILE *stdin, *stdout;
```

## APERTURA DE UN ARCHIVO

Para procesar un archivo en C (y en todos los lenguajes de programación) la primera operación que hay que realizar es abrir el archivo. La apertura del archivo supone conectar el archivo externo con el programa, e indicar cómo va a ser tratado el archivo: binario, de caracteres, etc. El programa accede a los archivos a través de un puntero a la estructura FILE, la función de apertura devuelve dicho puntero. La función para abrir un archivo es f open () y el formato de llamada es:

```
fopen(char * nombre archivo, char *modo);
```

- nombre archivo.- Cadena con el identificador externo del archivo.
- modo.- Cadena con el modo en que se va a tratar el archivo.

La función devuelve un puntero a FILE, a través de dicho puntero el programa hace referencia al archivo. La llamada a fopen () se debe de hacer de tal forma que el valor que devuelve se asigne a una variable puntero a FILE, para así después referirse a dicha variable.

#### Ejemplo

Declara una variable de tipo puntero a FILE. A continuación escribir una sentencia de apertura de un archivo.

```

FILE* pf;
pf = fopen(nombre_archivo, modo);

```

La función puede detectar un error al abrir el archivo, por ejemplo que el archivo no exista y se quiera leer, entonces devuelve NULL.

#### Ejemplo

Se desea abrir un archivo de nombre LICENCIA. EST para obtener ciertos datos.

```

#include <stdio.h>
#include <stdlib.h>

FILE *pf;
char nm[] = "C:\\LICENCIA.EST";
pf = fopen(nm, "r");
if( pf == NULL ) {
    puts("Error al abrir el archivo");
    exit(1);
}

```

### Ejemplo

En este ejemplo se abre el archivo de texto JARDINES.DAT para escribir en él los datos de un programa.

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

FILE *ff;
char nm[] = "C:\\LICENCIA.EXT";
pf = fopen(nm, "r");
if( pf == NULL ) {
    puts("Error al abrir el archivo");
    exit(1);
}
```

### Ejemplo

En este ejemplo se abre el archivo de texto JARDINES. DAT para escribir en él los datos de un programa. En la misma línea en que se ejecuta f open () se comprueba que la operación ha sido correcta, en caso contrario termina la ejecución.

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

FILE *ff;
char* arch = "C:\\AMBIENTE\\JARDINES.DAT";

if ((ff = fopen(nm, "w"))==NULL) {
    puts("Error al abrir el archivo para escribir.");
    exit(-1);
}
```

El prototipo de fopen () se encuentra en el archivo stdio.h, es el siguiente:

```
FILE* fopen (const char* nombre-archivo, const char* modo);
```

### Modos de apertura de un archivo

Al abrir el archivo fopen() se espera como segundo argumento el modo de tratar el archivo. Fundamentalmente se establece si el archivo es de lectura, escritura o añadido; y si es de texto o binario. Los modos básicos se expresan en esta tabla:

Modo	Significado
r	Abre para lectura
w	Abre para crear un nuevo archivo( si ya existe se pierden sus datos)
a	Abre para añadir al final
r+	Abre archivo ya existente para modificar (leer/escribir).
w+	Crea un archivo para escribir/leer (si ya existe se pierden los datos).
a+	Abre el archivo para modificar (escribir/leer) al final. Si no existe es como w+.

En estos modos no se ha establecido el tipo del archivo, de texto o binario. Siempre hay una opción por defecto y aunque depende del compilador utilizado, suele ser modo texto. Para no depender del entorno es mejor indicar si es de texto o binario. Se utiliza la

letra t para modo texto, la b para modo binario como último carácter de la cadena modo (también se puede escribir como carácter intermedio). Por consiguiente, los modos de abrir un archivo de texto:

"rt", "wt", "at", "r+t", "w+t", "a+t".

Y los modos de abrir un archivo binario:

"rb)", "wb", "ab", "r+b", "w+b", "a+b".

Ejemplo

Se dispone archivo de texto LICENCIA. EST, se quiere leerlo para realizar un cierto proceso y escribir datos resultantes en al archivo binario RESUMEN. REC. Las operaciones de apertura son:

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

FILE *pfl, *pf2;
char org[] = "C:\\LICENCIA.EST"; char dst[] = "C:\\RESUMEN.REC";

pfl = fopen (org, "rt");
pf2 = fopen(dst,"wb");
if (pfl == NULL || pf2 == NULL){
    puts("Error al abrir los archivos.");
    exit (1);
}
```

## NULL Y EOF

Las funciones de biblioteca que devuelven un puntero (strcpy () , fopen () ...) especifican que si no pueden realizar la operación (generalmente si hay un error) devuelven NULL. Ésta es una macro definida en varios archivos de cabecera, entre los que se encuentran stdio.h y stdlib. h .

Las funciones de librería de E/S de archivos, generalmente empiezan por f de file, tienen especificado que son de tipo entero de tal forma que si la operación falla devuelven EOF, también devuelven EOF para indicar que se ha leído el fin de archivo. Esta macro está definida en stdio . h .

Ejemplo

El siguiente segmento de código lee del flujo estándar de entrada hasta fin de archivo:

```
int c;
while ((c=getchar()) != EOF)
```

## Cierre de archivos

Los archivos en C trabajan con una memoria intermedia, son con buffer. La entrada y salida de datos se almacena en ese buffer, volcándose cuando está lleno. Al terminar la ejecución del programa podrá ocurrir que haya datos en el buffer, si no se volcasen en el archivo quedaría este sin las últimas actualizaciones. Siempre que se termina de procesar un archivo y siempre que se termine la ejecución del programa los archivos abiertos hay que cerrarlos para que entre otras acciones se vuelque el buffer.

La función fclose (puntero\_file) cierra el archivo asociado al puntero\_file, devuelve EOF si ha habido un error al cerrar. El prototipo de la función se encuentra en stdio.h y es:

```
int fclose(FILE* pf);
```

Ejemplo

Abrir dos archivos de texto, después se cierra cada uno de ellos.

```
#include <stdio.h>

FILE *pfl, *pf2;
pfl = fopen("C:\\DATOS.DAT", "a+");
pf2 = fopen("C:\\TEMPS.RET", "b+");
fclose(pfl);
```

```
fclose(pf2);
```

## CREACIÓN DE UN ARCHIVO SECUENCIAL

Una vez abierto un archivo para escribir datos hay que grabar los datos en el archivo. La biblioteca C proporciona diversas funciones para escribir datos en el archivo a través del puntero a FILE asociado.

Las funciones de entrada y de salida de archivos tienen mucho parecido con las funciones utilizadas para entrada y salida para los flujos stdin (teclado) y stdout (pantalla): printf (), scanf (), getchar (), putchar (), gets () y puts (). Todas tienen una versión para archivos que empieza por la letra f, así se tiene fprintf (), fscanf (), fputs (), fgets (); la mayoría de las funciones específicas de archivos empiezan por f.

### Funciones putc () y fputc ()

Ambas funciones son idénticas, putc () está definida como macro. Escriben un carácter en el archivo asociado con el puntero a FILE. Devuelven el carácter escrito, o bien EOF si no puede ser escrito. El formato de llamada:

```
putc(c, puntero-archivo);
fputc(c, puntero-archivo);
```

siendo c el carácter a escribir.

#### Ejemplo

Se desea crear un archivo SALIDA. PTA con los caracteres introducidos por teclado.

Una vez abierto el archivo, un bucle mientras (while) no sea fin de archivo (macro EOF) lee carácter a carácter y se escribe en el archivo asociado al puntero FILE.

```
#include <stdio.h>

int main() {
    int c;
    FILE* pf;
    char *salida = "SALIDA.TXT" ;
    if ((pf = fopen (salida,"wt") == NULL) {
        puts("ERROR EN LA OPERACION DE APERTURA");
        return 1;
    }
    while ((c=getchar())!=EOF)
        putc(c,pf);

    fclose(pf);
    return 0;
}
```

En el ejemplo anterior en vez de putc (c, pf) se puede utilizar fputc (c, pf) . El prototipo de ambas funciones se encuentra en stdio.h , es el siguiente:

```
int putc(int c, FILE* pf);
int fputc(int c, FILE* pf);
```

### Funciones getc () y fgetc ()

Estas dos funciones son iguales, igual formato e igual funcionalidad; pueden considerarse que son recíprocas de putc() y fputc(). Éstas, getc() y fgetc(), leen un carácter (el siguiente carácter) del archivo asociado al puntero a FILE. Devuelven el carácter leído o EOF si es fin de archivo (o si ha habido un error). El formato de llamada es:

```
getc(puntero_archivo);
fgetc(puntero_archivo);
```

#### Ejemplo

El archivo SALIDA. PTA, se desea leer para mostrarlo por pantalla y contar las líneas que tiene.

Una vez abierto el archivo de texto en modo lectura, un bucle mientras no sea fin de archivo (macro EOF) lee carácter a carácter y se escribe en pantalla. En el caso de leer el carácter de fin de línea se debe saltar a la línea siguiente y contabilizar una línea más.

```
#include <stdio.h>

int main() {
    int c, n=0;
    FILE* pf;
    char *nombre = "\\SALIDA.TXT";
    if ((pf = fopen(nombre,"rt")) == NULL) {
        puts("ERROR EN LA OPERACION DE APERTURA");
        return 1;
    }
    while ((c=getc(pf))!=EOF) {
        if (c == '\n')
            n++; printf("\n");
        else
            putchar(c);
    }
    printf("\nNúmero de líneas del archivo: %d",n);
    fclose(pf);
    return 0;
}
```

El prototipo de ambas funciones se encuentra en stdio.h y es el siguiente:

```
int getc(FILE* pf);
int fgetc(FILE* pf);
```

### Funciones fputs () y fgets ()

Estas funciones escriben/leen una cadena de caracteres en el archivo asociado. La función fputs() escribe una cadena de caracteres. La función devuelve EOF si no ha podido escribir la cadena, un valor no negativo si la escritura es correcta; el formato de llamada es:

```
fputs(cadena, puntero_archivo);
```

La función fgets() lee una cadena de caracteres del archivo. Termina la captación de la cadena cuando lee el carácter de fin de línea, o bien cuando ha leído n-1 caracteres, siendo n un argumento entero de la función. La función devuelve un puntero a la cadena devuelta, o NULL si ha habido un error. El formato de llamada es:

```
fgets(cadena, n, puntero-archivo);
```

### Ejemplo

Lectura de un máximo de 80 caracteres de un archivo:

```
#define T 81
char cad[T];
FILE *f;
fgets(cad, T, f);
```

### Ejemplo

El archivo CARTAS.DAT contiene un texto al que se le desea añadir nuevas líneas, de longitud mínima 30 caracteres desde el archivo PRIMERO.DAT.

El problema se resuelve abriendo el primer archivo en modo añadir ("a") , el segundo archivo en modo lectura ("r") . Las líneas se leen con fgets(), si cumplen la condición de longitud se escriben en el archivo CARTAS. Al tener que realizar un proceso completo del archivo, se realizan iteraciones mientras no fin de archivo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MX 121
#define MN 30
int main() {
    FILE *in, *out;
```

```

char nom1[] = "\\CARTAS.DAT"; char nom2[] = "\\PRIMERO.DAT";
char cad[MN];
in = fopen(nom2,"rt"); out= fopen(nom1,"at");
if (in==NULL || out==NULL) {
    puts("Error al abrir archivos. ");
    exit(-1);
}
while (fgets(cad,MN,in)) { /*itera hasta que devuelve puntero
NULL*/
    if (strlen(cad) >= MN)
        fputs(cad,out);
    else
        puts (cad);
}
fclose(in);
fclose(out);
return 0;
}

```

El prototipo de ambas funciones está en stdio.h, es el siguiente:

```

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

```

### Funciones fprintf () y fscanf ()

Las funciones printf() y scanf () permiten escribir o leer variables cualquier tipo de dato estándar, los códigos de formato (%d, %f...) indican a C la transformación que debe de realizar con la secuencia de caracteres (conversión a entero...). La misma funcionalidad tiene fprintf () y fscanf () con los flujos (archivos asociados) a que se aplican. Estas dos funciones tienen como primer argumento el puntero a f ¡le asociado al archivo de texto.

#### Ejemplo

Se desea crear el archivo de texto PERSONAS.DAT de tal forma que cada línea contenga un registro con los datos de una persona que contenga los campos nombre, fecha de nacimiento (dia(nn), mes(nn), año(nnnn) y mes en ASCII).

En la estructura persona se declaran los campos correspondientes. Se define una función que devuelve una estructura persona leída del teclado. El mes en ASCII se obtiene de una función que tiene como entrada el número de mes y devuelve una cadena con el mes en ASCII. Los campos de la estructura son escritos en el archivo con fprintf ().

```

#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* declaración de tipo global estructura */
typedef struct {
    char* nm;
    int dia;
    int ms;
    int aa;
    char mes[11];
}PERSONA;

void entrada( PERSONA* p );
char* mes_asci(short n);

int main() {

    FILE *pff;
    char nf[] = "\\PERSONAS.DAT"; char r = 'S';
    if ((pff = fopen(nf,"wt"))==NULL) {
        puts("Error al abrir archivos. ");
    }
}

```



```

        exit(-1);
    }
    while (toupper(r) == 'S') {
        PERSONA pt;
        entrada(&pt);
        printf("%s %d-%d-%d %s\n",pt.nm,pt.dia,pt.ms,pt.aa,pt.mes);
        fprintf(pff,"%s-%d-%d-%d %s\n", pt.nm, pt.dia, pt.ms, pt.aa,
        pt.mes);
        printf("Otro registro?: ");
        scanf("%c%c",&r);
    }
    fclose(pff);
    return 0;
}

void entrada(PERSONA* p) {
    char bf[81];
    printf("Nombre: ");
    gets(bf);
    p->nm =(char*)malloc((strlen(bf)+1)*sizeof(char));
    strcpy(p->nm,bf);
    printf("Fecha de nacimiento(dd mm aaaa): ");
    scanf("%d %d %d%c",&p->dia,&p->ms,&p->aa);
    printf("\n %s\n",mes_asci(p->ms));
    strcpy(p->mes,mes_asci(p->ms));
}

char* mes_asci(short n) {
    static char *mes[12]= {
        "Enero", "Febrero", "Marzo",
        "Abril", "Mayo", "Junio",
        "Julio", "Agosto",
        "Septiembre", "Octubre",
        "Noviembre", "Diciembre"
    };

    if (n >=1 && n <= 12)
        return mes[n-1];
    else
        return "Error mes"
}

```

El prototipo de ambas funciones está en stdio.h, y es el siguiente:

```

int fprintf(FILE* pf,const char* formato,.. .);
int fscanf(FILE* pf,const char* formato,.. .);

```

### **Función feof ( )**

Diversas funciones de lectura de caracteres devuelven EOF cuando leen el carácter de fin de archivo. Con dicho valor, que es una macro definida en stdio. h, ha sido posible formar bucles para leer un archivo completo. La función feof() realiza el cometido anterior, devuelve un valor distinto de 0 (true) cuando se lee el carácter de fin de archivo, en caso contrario devuelve 0 (false).

#### **Ejemplo**

El siguiente ejemplo transforma el ejemplo visto en la página 70, utilizando la función feof ( )

```

int c, n=0;
FILE* pf;
char *nombre = "SALIDA.TXT";
...

while (!feof(pf)) {
    c=getc(pf);

```

```

    if (c == '\n')
        n++; printf("\n");
    ...

```

El prototipo de la función está en `stdio.h`, es el siguiente:

```
int feof(FILE* pf);
```

### Función `rewind()`

Una vez que se alcanza el fin de un archivo, nuevas llamadas a `feof()` siguen devolviendo un valor distinto de cero (true). Con la función `rewind()` se sitúa el puntero del archivo al inicio de éste. El formato de llamada es

```
rewind(puntero_archivo)
```

El prototipo de la función se encuentra en `stdio.h`:

```
void rewind(FILE* pf);
```

### Ejemplo

Este ejemplo lee un archivo de texto, cuenta el número de líneas que contiene y a continuación sitúa el puntero del archivo al inicio para una lectura posterior.

```

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

FILE* pg;
char nom[]="PLUVIO.DAT";
char buf[121];
int nl = 0;

if ((pg = fopen(nom,"rt")) ==NULL) {
    puts("Error al abrir el archivo.");
    exit(-1);
}
while (!feof (pg) ) {
    fgets(buf,121,pg);
    nl++;
}
rewind(pg);
/* De nuevo puede procesarse el archivo
while (!feof (pg) ) {
    ...
}

```

## ARCHIVOS BINARIOS EN C

Para abrir un archivo en modo binario hay que especificar la opción `b` en el modo. Los archivos binarios son secuencias de 0,s y 1,s. Una de las características de los archivos binarios es que optimizan la memoria ocupada por un archivo, sobre todo con campos numéricos. Así, almacenar en modo binario un entero supone una ocupación de 2 bytes o 4 bytes (depende del sistema), y un número real 4 bytes o 8 bytes; en modo texto primero se convierte el valor numérico en una cadena de dígitos (`%6d`, `%8.2f` ...) y después se escribe en el archivo. La mayor eficiencia de los archivos binarios se contrapone con el hecho de que su lectura se tiene que hacer en modo binario y que sólo se pueden visualizar desde el entorno de un programa C. Los modos para abrir un archivo binario son los mismos que para abrir un archivo de texto, sustituyendo la `t` por `b`:

"rb", "wb", "ab", "r+b", "w+b", "a+b"

### Ejemplo

En este ejemplo se declaran 3 punteros a `FILE`. A continuación se abren tres archivos en modo binario.

```
FILE *pf1, *pf2, *pf3;
```

```

pf1 = fopen("gorjal.arr", "rb");
pf2 = fopen("tempes.feb", "w+b");
pf3 = fopen("telcon.fff", "ab");

```

La biblioteca de C proporciona dos funciones especialmente dirigidas al proceso de entrada y salida de archivos binarios con buffer, son `fread()` y `fwrite()`.

### Función de salida `fwrite()`

La función `fwrite()` escribe un buffer de cualquier tipo de dato en un archivo binario. El formato de llamada es:

```
fwrite(direccion_buffer, tamaño, num elementos, puntero archivo);
```

#### Ejemplo

En el ejemplo se abre un archivo en modo binario para escritura. Se escriben números reales en doble precisión en el bucle `for`. El buffer es la variable `x`, el tamaño lo devuelve el operador `sizeof`.

```

FILE *fd;
double x;
fd = fopen("real es.num", "wb");
for (x=0.5; x > 0.01; ) {
    fwrite(&x, sizeof(double), 1, fd);
    x = pow(x, 2.);
}

```

El prototipo de la función está en `stdio.h`:

```
size_t fwrite(const void *ptr, size_t tam, size_t n, FILE *pf);
```

El tipo `size_t` está definido en `stdio.h` y es un tipo `int`.

#### Ejemplo

Se dispone de una muestra de las coordenadas de puntos de un plano representada por pares de números enteros (`x`, `y`), tales que  $1 \leq x \leq 100$  e  $1 \leq y \leq 100$ . Se desea guardar en un archivo binario todos los puntos disponibles.

El nombre del archivo es `PUNTOS.DAT`. Según se lee un punto se comprueba la validez del punto y se escribe en el archivo con una llamada a la función `fwrite()`. La condición de terminación del bucle es la lectura del punto (0,0)

```

#include <stdio.h>
struct punto {
    int x, y;
};
typedef struct punto PUNTO;
int main() {
    PUNTO p;
    char *nom = "C:\\PUNTOS.DAT";
    FILE *pp;
    if ((pp = fopen(nom, "wb")) == NULL) {
        puts("\nError en la operación de abrir archivo.");
        return -1;
    }
    puts("\nIntroduce coordenadas de puntos, fin: (0,0)");
    do {
        printf("Coordenadas deben ser >=0 :");
        scanf("%d %d", &p.x, &p.y);
        if (p.x > 0 || p.y > 0)
            fwrite(&p, sizeof(PUNTO), 1, pp);
    } while (p.x < 0 || p.y < 0)

    fclose(pp);
    return 0;
}

```

Los archivos binarios están indicados especialmente para guardar registros, estructuras en C. El método habitual es la escritura sucesiva de estructuras en el archivo asociado al puntero, la lectura de estos archivos es similar.

## Función de lectura f read ( )

Esta función lee de un archivo n bloques de bytes y los almacena en un buffer. El número de bytes de cada bloque (tamaño) se pasa como parámetro, al igual que el número n de bloques y la dirección del buffer (o variable) donde se almacena. El formato de llamada:

```
fread(direccion buffer,tamaño,n,puntero_archivo);
```

La función devuelve el número de bloques que lee y debe de coincidir con n. El prototipo de la función está en stdio . h:

```
size_t fread(void *ptr,size_t tam,size_t n,FILE *pf);
```

### Ejemplo

En el ejemplo se abre un archivo en modo binario para lectura. El archivo se lee hasta el final del archivo; cada lectura de un número real se acumula en la variable s.

```
FILE *fd;
double x, s=0.0;
if((fd = fopen("real es.num","rb"))==NULL)
    exit(-1);

while (!eof (fd) ) {
    fread(&x, sizeof(double), 1, fd);
    s+= x;
}
```

### Ejercicio

En un ejemplo anterior se ha creado un archivo binario de puntos en el plano. Se desea escribir un programa para determinar los siguientes valores:

$n_{ij}$  número de veces que aparece un punto dado (i,j) en el archivo.

Dado un valor de j, obtener la media de i para los puntos que contienen a j.

La primera instrucción es abrir el archivo binario para lectura. A continuación se solicita el punto donde se cuentan las ocurrencias en el archivo. En la función cuenta\_pto() se determina dicho número; para lo cual hay que leer todo el archivo. Para ejecutar el segundo apartado, se solicita el valor de j. Con un bucle desde i=1 hasta 100 se cuenta las ocurrencias de cada punto (i,j) llamando a la función cuenta\_pto(); antes de cada llamada hay que situar el puntero del archivo al inicio, llamando para ello a la función rewind() .

```
#include <stdio.h>
struct punto {
    int i,j;
}
typedef struct punto PUNTO;
FILE *pp;
int cuenta_pto(PUNTO w);
int main() {
    PUNTO p;
    char *nom ="C:\PUNTOS.DAT";
    float media,nmd,dnm;
    if ((pp = f open (nom, "rb")) ==NULL) {
        puts("\nError al abrir archivo para lectura.");
        return -.1;
    }
    printf("\nIntroduce coordenadas de punto a buscar: ");
    scanf("%d %d",&p.i,&p.j);
    printf("\nRepeticiones del punto (%d,%d): %d\n",
        p.i,p.j,cuenta_pto(p));

    /* Cálculo de la media i para un valor j */
    printf ("Valor de j: ");
    scanf ("%d",&p.j);
```

```

media=nmd/dnm= 0.0;

for (p.i=1; p.i<= 10; p.i++) {
    int st;
    rewind(pp);
    st = cuenta_pto(p);
    nmd += (float)st*p.i;
    dnm += (float)st;
}

if (dnm >0.0)
    media = nmd/dnm;
printf("\nMedia de los valores de i para %d  %.2f",p.j,media);
return 0;
}

int cuenta_pto(PUNTO w) {
    PUNTO p;
    int r; r = 0;
    while (!feof(pp)) {
        fread(&p,sizeof(PUNTO),1,pp);
        if (p.i==w.i && p.j==w.j) r++;
    }
    return r;
}

```

## FUNCIONES PARA ACCESO ALEATORIO

El acceso directo -aleatorio- a los datos de un archivo se hace mediante su posición, es decir, el lugar relativo que ocupan. Tiene la ventaja de que se pueden leer y escribir registros en cualquier orden y posición. Son muy rápidos de acceso a la información que contienen. El principal inconveniente que tiene la organización directa es que necesita programar la relación existente entre el contenido de un registro y la posición que ocupan.

Las funciones `fseek()` y `ftell()` se usan principalmente para el acceso directo a archivos en C. Estas consideran el archivo como una secuencia de bytes; el número de byte es el índice del archivo. Según se va leyendo o escribiendo registros o datos en el archivo, el programa mantiene a través de un puntero la posición actual. Con la llamada a la función `ftell()` se obtiene el valor de dicha posición. La llamada a `fseek()` permite cambiar la posición del puntero al archivo a una dirección determinada.

### Función `fseek()`

Con la función `fseek()` se puede tratar un archivo en C como un array que es una estructura de datos de acceso aleatorio. `fseek()` sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos. En el ejemplo siguiente se supone que existe un archivo de productos, se pide el número de producto y se sitúa el puntero del archivo para leer el registro en una operación de lectura posterior.

#### Ejemplo

Declarar una estructura (registro) `PRODUCTO`, y abrir un archivo para lectura. Se desea leer un registro cuyo número (posición) se pide por teclado.

```

typedef struct {
    char nombre[41];
    int unidades;
    float precio;
    int pedidos;
} PRODUCTO;

```

```

PRODUCTO uno;
int n, stat;
FILE* pfp;

if ((pfp = fopen("conservas.dat","r"))==NULL) {
    puts("No se puede abrir el archivo.");
    exit(-1);
}
/* Se pide el número de registro */
printf("Número de registro: ");
scanf("%d",&n);

/* Sitúa el puntero del archivo */
stat = fseek(pfp, n*sizeof(PRODUCTO),0);

/* Comprueba que no ha habido error */
if (stat != 0) {
    puts("Error, puntero del archivo movido fuera de este");
    exit(-1);
}

/* Lee el registro */
fread(&uno, sizeof(PRODUCTO), 1, pfp);
...

```

El segundo argumento de `fseek()` es el desplazamiento, el tercero es el origen del desplazamiento, el 0 indica que empiece a contar desde el principio del archivo. El formato para llamar a `fseek ()`

```
fseek(puntero_archivo, desplazamiento, origen);
```

desplazamiento: es el número de bytes a mover; tienen que ser de tipo `long`.

origen : es la posición desde la que se cuenta el número de bytes a mover. Puede tener tres valores, que son:

0 : Cuenta desde el inicio del archivo.

1 : Cuenta desde la posición actual del puntero al archivo.

2 : Cuenta desde el final del archivo.

Estos tres valores están representados por tres identificadores (macros):

0 : `SEEK_SET`

1 : `SEEK_CUR`

2 : `SEEK_END`

La función `fseek()` devuelve un valor entero, distinto de cero si se comete un error en su ejecución; cero si no hay error. El prototipo se encuentra en `stdio.h`

```
int fseek(FILE *pf, long dsplz, int origen);
```

### Ejemplo

Para celebrar las fiestas patronales de un pueblo se celebra una carrera popular de 9 Km. Se establecen las categorías masculina (M) y femenina (F), y por cada una de ellas, senior y veterano. Los nacidos antes de 1954 son veteranos (tanto para hombres como para mujeres) y el resto seniors. Según se realizan inscripciones se crea el archivo binario `CARRERA.POP`, de tal forma que el número de dorsal es la posición que ocupa el registro en el archivo. La carrera se celebra; según llegan los corredores se toman los tiempos realizados y los números de dorsales.

Se desea escribir un programa para crear el archivo `CARRERA.POP` y un segundo programa que actualice cada registro, según el número de dorsal, con el tiempo realizado en la carrera.

En una estructura se agrupan los campos necesarios para cada participante: nombre, año de nacimiento, sexo, categoría, tiempo empleado (minutos, segundos), número de dorsal

y puesto ocupado. El primer programa abre el archivo en modo binario para escribir los registros correspondientes a los participantes en la posición del número de dorsal. Los números de dorsal se asignan según la categoría, para las mujeres veteranas del 51 al 100; para mujeres senior de 101 al 200. Para hombres veteranos de 251 al 500, y para senior del 501 al 1000.

El programa, en primer lugar inicializa los nombres de los registros del archivo a blancos. Los dorsales se asignan aleatoriamente, comprobando que no estén previamente asignados. El segundo programa abre el archivo en modo modificación, accede a un registro, según dorsal y escribe el tiempo y puesto. Los tipos de datos que se crean para la aplicación, estructura fecha, estructura tiempo, estructura atleta, se incluyen en el archivo atleta.h.

```
/* Archivo atleta.h */
typedef struct fecha {
    int d, m, a;
}FECHA;

typedef struct tiempo {
    int h, m, s;
}TIEMPO;

struct atleta {
    char nombre[28];
    FECHA f;
    char sx; /* Sexo */
    char cat; /* Categoría */ TIEMPO t;
    unsigned int dorsal;
    unsigned short puesto;
};

typedef struct atleta ADTA;

#define desplz(n) (n-1)*sizeof(ADTA)

/* Programa para dar entrada en el archivo de atletas. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include "atleta.h"

void inicializar(FILE*);
void unatleta(ADTA* at,FILE*);
unsigned numdorsal(char s, char cat, FILE* pf);

int main() {
    FILE *pf;
    ADTA a;
    char *archivo= "C:\\CARRERA.POL";
    randomize();
    if ((pf=fopen(archivo, "wb+"))==NULL) {
        printf("\nError al abrir el archivo %s, fin del
proceso.\n");
        return -1;
    }
    inicializar(pf);
    //Se introducen registros hasta teclear como nombre: FIN
    unatleta(&a,pf);
    do {
```

```

        fseek(pf,desplz(a.dorsal),SEEK_SET);
        fwrite(&a,sizeof(ADTA),1,pf);
        unatleta(&a,pf);
    }while (strcmpi(a.nombre,"FIN"));
    fclose(pf);
    return 0;
}

void unatleta(ADTA* at, FILE*pf) {
    printf("Nombre: ");

    gets(at->nombre);
    if (strcmpi(at->nombre,"fin")) {
        printf("Fecha de nacimiento: ");
        scanf("%d %d %d%c",&at->f.d,&at->f.m,&at->f.a);
        if (at->f.a<1954)
            at->cat = 'V';
        else
            at->cat = 'S';
        printf("Sexo: ");
        scanf("%c%c",&at->sx);
        at->sx=(char)toupper(at->sx);
        at->t.h = 0;
        at->t.m = 0;
        at->t.s = 0;
        at->dorsal = numdorsal(at->sx,at->cat,pf);
        printf("Dorsal asignado: %u\n",at->dorsal);
    }
}

unsigned numdorsal(char s, char cat, FILE* pf) {
    unsigned base, tope, d; ADTA a;

    if (s=='M' && cat=='V') {
        base = 251;
        tope = 500;
    }
    else if (s=='M' && cat=='S') {
        base = 501;
        tope = 1000;
    }
    else if (s=='F' && cat=='V') {
        base = 51;
        tope = 100;
    }
    else if (s=='F' && cat=='S') {
        base = 101;
        tope = 200;
    }
    d = (unsigned) random(tope+1-base)+base;

    fseek(pf,desplz(d),SEEK SET);
    fread(&a,sizeof(ADTA),1,pf);

    if (!(*a.nombre)) /* Cadena nula: está vacío */
        return d;
    else
        return numdorsal(s,cat,pf);
}

void inicializar(FILE*pf) {

```



```

    int k;
    ADTA a;

    a.nombre[0] = '\0';
    for (k=1; k<=1000; k++)
        fwrite(&a,sizeof(ADTA),1,pf);
}

/* Programa para dar entrada a los tiempos de los atletas.
Primeramente, dado un numero de dorsal se visualiza el registro del
atleta, a continuación se introduce los minutos y segundos
realizados por el atleta.*/

#include <stdio.h>
#include <string.h>
#include "atleta.h"

void datosatleta(ADTA at);

int main() {
    FILE *pf;
    ADTA a;
    TIEMPO h={0,0,0};
    char *archivo= "C:\CARRERA.POL";
    unsigned dorsal=1;

    if ((pf=fopen(archivo, "rb+"))==NULL) {
        printf("\nError al abrir el archivo %s, fin del
proceso.\n");
        return -1;
    }

    /* El proceso iterativo termina con el dorsal 0 */
    printf("\n Dorsal del atleta: ");
    scanf("%u",&dorsal);
    for ( ; dorsal ; ) {
        // Se situa el puntero en el registro
        fseek(pf,desplz(dorsal),SEEK_SET);
        fread(&a,sizeof(ADTA),1,pf);
        if (*a.nombre) {
            datosatleta(a);
            printf("\n Tiempo realizado en minutos y segundos: ");
            scanf("%d %d",&h.m,&h.s);
            a.t = h;
            fseek(pf,desplz(dorsal),SEEK_SET);
            fwrite(&a,sizeof(ADTA),1,pf);
        }
        else
            printf("Este dorsal no está registrado.\n");

        printf("\n Dorsal del atleta: "); scanf("%u",&dorsal);
    }
    fclose(pf);
    return 0;
}

void datosatleta(ADTA at) {
    printf("Nombre      :%s\n",at.nombre);
    printf("Fecha de nacimiento:%d-%d-%d:\n", at.f.d, at.f.m,
at.f.a );
    printf("Categoria          :%c\tDorsal:  %u\n",at.cat,at.dorsal);
}

```

```
        if (at.t.m>0)
            printf("Tiempo de carrera :%d min %d seg\n",at.t.m,at.t.s);
    }
```

### **Función ftell ( )**

La posición actual del archivo se puede obtener llamando a la función `ftell()` y pasando un puntero al archivo como argumento. La función devuelve la posición como número de bytes (en entero largo: `long int`) desde el inicio del archivo (byte 0).

#### **Ejemplo**

En este ejemplo se puede observar cómo se desplaza el puntero del archivo según se escriben datos en él.

```
#include <stdio.h>

int main(void) {
    FILE *pf;
    float x = 123.5;
    pf = fopen("CARTAS.TXT", "w");
    printf("Posición inicial: old\n",ftell(pf)); //muestra 0
    fprintf(pf,"Caracteres de prueba");
    printf("Posición actual: %ld\n",ftell(pf)); //muestra 20
    fwrite(&x,sizeof(float),1,pf);
    printf("Posición actual:  %ld\n", ftell(pf)); //muestra 24
    fclose(pf);
    return 0;
}
```

Para llamar a la función se pasa como argumento el puntero a `FILE`. El prototipo se encuentra en `stdio.h`

```
long int ftell( FILE *pf );
```