

UNA VISIÓN PRÁCTICA DE LA PRÁGRAMACIÓN EN C

Raúl Alcaraz Martínez

PROGRAMACIÓN EN C**INDICE****TEORÍA**PÁGINAS**TEMA 1. DESARROLLO DE LOS LENGUAJES DE PROGRAMACIÓN.**

1.1.	Evolución de la informática.....	7
1.2.	Evolución de los sistemas operativos.	9
1.3.	Evolución de los lenguajes de programación.	11

TEMA 2. ALGORITMOS Y PROGRAMAS. ESTRUCTURA DE UN PROGRAMA.

2.1.	Introducción.	14
2.2.	Concepto de algoritmo.	15
2.3.	Pseudocódigo.	16
2.4.	Compilación.	18
2.5.	Partes de un programa.	19
2.6.	Estructura de un programa.	20

TEMA 3. TIPOS ELEMENTALES DE DATOS.

3.1.	Datos.	22
3.2.	Tipos fundamentales de datos. Palabras reservadas en C.....	23
3.3.	Identificadores.	25
3.4.	Constantes.	26
3.5.	Operadores.	28
3.6.	Conversión de tipos.	34

TEMA 4. ESTRUCTURAS DE CONTROL.

4.1.	Estructura secuencial.	37
4.2.	Estructuras de selección.	38
4.3.	Estructuras de repetición.	42
4.4.	Entrada y salida por consola.	46
4.5.	Funciones.	51
4.6.	Recursividad.	54

TEMA 5. TIPOS ESTRUCTURADOS DE DATOS.

5.1.	Matrices.	56
5.2.	Cadenas de caracteres (strings).	59
5.3.	Estructuras.	62
5.4.	Enumerados.	65
5.5.	Tipos definidos por el usuario.	67
5.6.	Punteros.	68

TEMA 6. FICHEROS.

6.1.	Introducción.	73
6.2.	Puntero a ficheros.	74
6.3.	Funciones para el tratamiento de ficheros.	75

EJERCICIOS

Ejercicios con estructuras de control.	91
Ejercicios con arrays.	113
Ejercicios con cadenas de caracteres.	123
Ejercicios con punteros.	136
Ejercicios con estructuras.	141
Ejercicios con ficheros.	147
Ejercicios avanzados y variados.	168
Ejercicios de aplicación.	192

BIBLIOGRAFÍA

Bibliografía.	227
--------------------	-----

TEORIA

TEMA 1. DESARROLLO DE LOS LENGUAJES DE PROGRAMACIÓN.

- 1.1. Evolución de la informática.
 - 1.2. Evolución de los Sistemas Operativos.
 - 1.3. Evolución de los Lenguajes de Programación.
-

1.1. EVOLUCIÓN DE LA INFORMÁTICA.

Desde el siglo XVII existen instrumentos de cálculo que son principalmente instrumentos mecánicos. Su evolución a grandes rasgos fue:

- En 1642 Pascal inventó la primera calculadora mecánica.
- En 1645 Pascal mejora su calculadora siendo capaz de sumar y restar.
- En 1671 Leibnitz desarrolla otra calculadora que es capaz de multiplicar.
- En 1694 Leibnitz mejora su calculadora añadiéndole la operación de la división.

Durante el siglo XVIII el desarrollo fue bastante escaso, apareciendo únicamente la primera máquina programable (mediante tarjetas) que era un telar.

- En 1822 Babbage desarrolló una máquina analítica.
- En 19367 aparece la primera máquina electrónica llamada Mark I. Posteriormente aparecerá el Mark II comenzando a partir de aquí el desarrollo de la electrónica.

Con el desarrollo de la electrónica podemos comenzar a hablar de generaciones que vendrán determinadas por avances e innovaciones tecnológicas. Estos son:

» Primera generación (1940 – 1955):

- Desarrollo con propósito militar.
- Primer ordenador, el Eniac.
- Utilización de válvulas de vacío.
- Voluminosos, lentos, poco fiables, programación en lenguaje máquina mediante tarjetas y cintas perforadas.
- No existen sistemas operativos.

» Segunda generación (1955 – 1964):

- Desarrollo del transistor.
- Menor tamaño, coste, mayor fiabilidad.
- Primeros lenguajes de programación.

- » Tercera generación (1965 – 1970):
 - Creación de los circuitos integrados (chips) y memorias semiconductoras.
 - Aparición de los lenguajes de alto nivel y de la multiprogramación.

- » Cuarta generación (a partir de 1970):
 - Desarrollo del microprocesador.
 - Aparición de los PC's, redes de ordenadores (Internet),...

1.2. EVOLUCIÓN DE LOS SISTEMAS OPERATIVOS.

Un **sistema operativo** es un programa que actúa como interface entre máquina y usuario, ofreciendo el entorno necesario para que el usuario pueda ejecutar programas. Por lo tanto, el sistema operativo tiene dos objetivos:

- » Facilitar el uso del sistema.
- » Emplear eficientemente el hardware del ordenador.

La evolución de los sistemas operativos fue:

- » Open – Shop:
 - Reserva del tiempo de utilización.
 - Interacción "a mano", mediante cableado e interruptores.
 - Surgen ayudas de hardware (lectores de tarjeta, impresoras,...) y de software (lenguajes de programación de alto nivel,...).
- » Operador de preparación:
 - Desaparición de la interacción manual.
 - Persona específica encargada de la preparación del ordenador.
- » Procesamiento por lotes:
 - Ejecución de trabajos de requerimiento similares seguidamente.
 - Pérdida del contacto con el ordenador por parte del usuario.
- » Monitor residente.
 - Programa almacenado en memoria que realiza el secuenciamiento automático de trabajos.
- » Operaciones Off – Line.

» Buffering.

- Sistemas de almacenamiento para mantener ocupados simultáneamente a la CPU y a los periféricos.
- Acceso secuencial.

» Spooling.

- Sistemas de almacenamiento similares a los buffers pero con acceso aleatorio.

» Multiprogramación:

- Varios trabajos activos a la vez.

1.3. EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN.

Podemos definir el concepto de programa desde dos puntos de vista:

- » El del usuario: es un fichero que realiza una función determinada.
- » El del programador: conjunto de instrucciones escritas en un lenguaje de programación y que conducen a la resolución de un problema concreto.

Las fases en el desarrollo de un programa se componen de:

1. Análisis: determinación del problema concreto a resolver.
2. Algoritmo: programa en lenguaje matemático – lógico.
3. Codificación: programa en lenguaje de programación correspondiente.
4. Prueba y depuración.
5. Mantenimiento.

Un ***lenguaje de programación*** es un lenguaje definido mediante un conjunto de símbolos y reglas determinadas que permiten construir una serie de instrucciones con un significado y función concreto.

La evolución de los lenguajes de programación ha sido:

- » Lenguaje máquina (lenguaje binario):
 - Instrucciones en 0 y 1.
 - Dependen del hardware.
 - Directamente interpretables por la CPU.
- » Lenguajes ensambladores:
 - Instrucciones en notación simbólica.
 - Utilización de ensambladores o traductores.

» Lenguajes de alto nivel:

- Independientes de la máquina.
- Muchas instrucciones.
- Traducción mediante interprete (traducción línea por línea) o compilador (traducción del programa completo).
- Tres tipos: estructurados (Pascal, C,...), funcionales, lógicos.

TEMA 2. ALGORITMOS Y PROGRAMAS. ESTRUCTURA DE UN PROGRAMA.

- 2.1. Introducción.
- 2.2. Concepto de algoritmo.
- 2.3. Pseudocódigo.
- 2.4. Compilación.
- 2.5. Partes de un programa.
- 2.6. Estructura de un programa.

1.1. INTRODUCCIÓN.

Desde el nacimiento de la informática, el hombre ha buscado y desarrollado métodos y herramientas para facilitar, agilizar y mejorar el trabajo de analistas y programadores, ofreciendo nuevas vías o caminos para la búsqueda de soluciones a determinados problemas mediante el diseño de algoritmos. Por ello, los estudios realizados en este campo dieron origen a la denominada **programación estructurada y modular**, con la que se llega a demostrar que cualquier módulo de programa se puede construir utilizando tres tipos de estructuras básicas:

- » Estructura secuencial.
- » Estructura alternativa o condicional.
- » Estructura repetitiva.

Al hablar de **programación estructurada**, hacemos referencia a un conjunto de técnicas que incorporan:

- » Diseño descendente (top – down).
- » Posibilidad de descomponer una acción compuesta en términos de acciones más simples.
- » El uso de estructuras básicas de control (secuencial, alternativa y repetitiva).

Por otro lado, al hablar de **programación modular**, hacemos referencia a la división o subdivisión de un programa en módulos, de manera que cada uno de ellos tenga encomendada la ejecución de una única tarea o actividad. Cada módulo se caracteriza por ser programado y depurado individualmente, lo que lo hace totalmente independiente.

De todo ello podemos deducir tres importantes características:

- » Se minimiza la complejidad del problema y por tanto, se reducen errores en la fase de codificación.
- » Aumenta considerablemente la productividad.
- » Facilita la depuración y puesta a punto de los programas.

El objetivo de este tema es conocer el uso de una herramienta (notación pseudocodificada) que nos permita y facilite el diseño de algoritmos a partir de los cuales construir los programas.

1.2. CONCEPTO DE ALGORITMO.

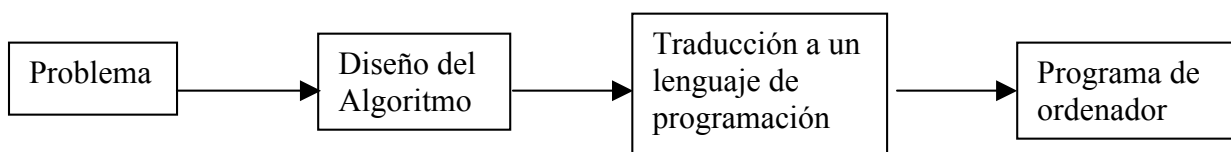
Un **algoritmo** se puede definir como la descripción abstracta de todas las acciones u operaciones que debe realizar un ordenador de forma clara y detallada, así como el orden en el que éstas deberán ejecutarse unto con la descripción de todos aquellos datos que degerán ser manipulados por dichas acciones y que nos conducen a la solución del problema, facilitando así su posterior traducción al lenguaje de programación correspondiente. El diseño de todo algoritmo debe reflejar las tres partes de un programa y que son la *entrada*, el *proceso* y la *salida*.

Es importante tener en cuenta que todo algoritmo debe ser totalmente independiente del lenguaje de programación que se utilice; es decir, que el algoritmo diseñado deberá permitir su traducción a cualquier lenguaje de programación con independendencia del ordenador en el que se vaya a ejecutar dicho programa habitualmente.

Las características que debe cumplir todo algoritmo son las siguientes:

- (a) Debe ser **conciso** y **detallado**, es decir, debe reflejar con el máximo detalle el orden de ejecución de cada acción u operación que vaya a realizar el ordenador.
- (b) Todo algoritmo se caracteriza por tener un comienzo y un final. Por ello se puede decir que es **finito** o **limitado**.
- (c) Al aplicar el mismo algoritmo “n” veces con los mismos datos de entrada, se deben obtener siempre los mismos resultados o datos de salida. Por ello se puede decir que es **exacto** o **preciso**.
- (d) Debe ser **flexible**, es decir, que debe adaptarse con facilidad a cualquier lenguaje de programación y entorno o plataforma.
- (e) Debe facilitar el entendimiento así como su traducción y las futuras modificaciones o actualizaciones que sobre él sean necesarias realizar. Por tanto, se deberá diseñar utilizando un estilo **amigable** y **entendible**.

Por lo tanto, el esquema a seguir a la hora de diseñar un programa será:



1.3. PSEUDOCÓDIGO.

El pseudocódigo, o notación pseudocodificada, se puede definir como el lenguaje intermedio entre el lenguaje natural y el lenguaje de programación seleccionado. Esta notación se encuentra sujeta a unas determinadas reglas que nos permiten y facilitan el diseño de algoritmos como fórmula de resolución a un problema. La notación pseudocodificada surge como método para la representación de instrucciones de control en una metodología estructurada y nació como un lenguaje similar al inglés, que utilizaba palabras reservadas de esta idioma para la representación de acciones concretas (start, end, stop, while, repeat, for, if, if – then – else, etc.) y que posteriormente se fue adaptando a otros lenguajes de lengua hispana.

La notación pseudocodificada o pseudocódigo se caracteriza por:

- (a) No puede ser ejecutado directamente por un ordenador, por lo que tampoco es considerado como un lenguaje de programación propiamente dicho.
- (b) Permite el diseño y desarrollo de algoritmos totalmente independientes del lenguaje de programación posteriormente utilizado en la fase de codificación, pues no está sujeto a las reglas sintácticas de ningún lenguaje excepto las del suyo propio.
- (c) Es extremadamente sencillo de aprender y utilizar.
- (d) Facilita al programador enormemente el paso del algoritmo al correspondiente lenguaje de programación.
- (e) Esta forma de representación permite una gran flexibilidad en el diseño del algoritmo a la hora de expresar acciones concretas.
- (f) Permite con cierta facilidad la realización de futuras correcciones o actualizaciones gracias a que no es un sistema de representación rígido.
- (g) La escritura o diseño de un algoritmo mediante el uso de esta herramienta, exige la "indentación" o "sangría" del texto en el margen izquierdo de las diferentes líneas.
- (h) Permite obtener soluciones mediante aproximaciones sucesivas, es decir, lo que se conoce comúnmente como diseño descendente o **top down** y que consiste en la descomposición sucesiva del problema en niveles o subproblemas más pequeños, lo que nos permite la simplificación del problema general.

Toda notación pseudocodificada debe permitir la descripción de:

- » Instrucciones primitivas (entrada, salida, y asignación).
- » Instrucciones de proceso o cálculo.
- » Instrucciones de control.
- » Instrucciones compuestas.
- » La descripción de todos aquellos elementos de trabajo y estructuras de datos que se vayan a manipular en el programa (variables, constantes, tablas, registros, ficheros, etc.)-

Todo algoritmo representado en notación pseudocodificada deberá reflejar las siguientes partes:

- » **Cabecera:** es el área o bloque informativo donde quedará reflejado el nombre del algoritmo y el nombre del programa al que pertenece dicho diseño, debiéndose especificar el primero en el apartado denominado *Módulo* y el segundo en el apartado denominado *Programa*.
- » **Cuerpo:** se denomina así al resto del diseño, el cual queda dividido en otros dos bloques denominados *Bloque de datos* y *Bloque de acciones o instrucciones*.

2.4. COMPILACIÓN.

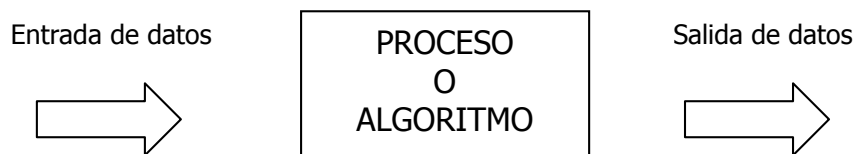
Lo primero es destacar que un interprete no es lo mismo que un compilador, ya que un interprete lee y ejecuta línea a línea del programa, mientras que el compilador lee completamente el programa y posteriormente lo ejecuta, por lo tanto, la ejecución del programa con un compilador es mucho más rápida que con un interprete.

Las fases de ejecución de un programa en C serán:

1. Escritura del programa fuente con un editor, como puede ser el que tiene turbo C 3.1 o el de Borland C ++ 5.0 y almacenamiento del mismo en el disco.
2. Introducir el programa fuente en memoria.
3. Compilar el programa.
4. Verificar y corregir errores de compilación.
5. Obtención del programa objeto.
6. Obtención del programa ejecutable mediante el enlazador.
7. Ejecución del mismo.

2.5. PARTES DE UN PROGRAMA.

Todo programa está constituido por un conjunto de órdenes o instrucciones capaces de manipular un conjunto de datos. Estas órdenes o instrucciones pueden ser divididas en tres grandes bloques claramente diferenciados, correspondientes cada uno de ellos a una parte del diseño de un programa.



» **Entrada de datos.**

En este bloque se engloban todas aquellas instrucciones que toman datos de un dispositivo o periférico externo, depositándolos posteriormente en la memoria central o principal para su tratamiento.

» **Proceso o algoritmo.**

Engloba todas aquellas instrucciones encargadas de procesar la información o aquellos datos pendientes de elaborar y que previamente habían sido depositados en memoria principal para su posterior tratamiento. Finalmente, todos los resultados obtenidos en el tratamiento de dicha información son depositados nuevamente en la memoria principal, quedando de esta manera disponibles.

» **Salida de datos o resultados.**

Este bloque está formado por todas aquellas instrucciones que toman los resultados depositados en memoria principal una vez procesados los datos de entrada, enviándolos seguidamente a un dispositivo o periférico externo.

2.6. ESTRUCTURA DE UN PROGRAMA.

Todo programa en C consta de una o más funciones, una de las cuales es "main".

El programa siempre comenzará por la ejecución de la función *main*. Las definiciones de las funciones adicionales pueden preceder o seguir a *main*.

Cada función debe contener:

- » Una cabecera de la función (nombre más argumentos / parámetros).
- » Una lista de declaraciones de los argumentos de la cabecera.
- » Una sentencia compuesta que contiene el resto de la función.

Las sentencias compuestas se encierran entre llaves.

Estas llaves pueden contener otras sentencias compuestas o combinaciones de sentencias elementales (llamadas sentencias de expresión).

Cada sentencia de expresión termina con punto y coma (;).

Los comentarios pueden aparecer en cualquier parte del código, y han de estar encerrados por marcas especiales:

`/* Comentarios */`

`// Comentarios (hasta el final de línea)`

Cuando utilicemos alguna función propia de alguna librería, será necesario que añadamos delante del *main* la declaración de inclusión de la librería.

TEMA 3. TIPOS ELEMENTALES DE DATOS.

- 3.1. Datos.
 - 3.2. Tipos fundamentales de datos. Palabras reservadas en C.
 - 3.3. Identificadores.
 - 3.4. Constantes.
 - 3.5. Operadores.
 - 3.6. Conversión de tipos.
-

3.1. DATOS.

Un **dato** es un elemento de información que puede tomar un valor entre varios posibles.

Si tienes valores fijos durante todo el programa son **constantes**.

Si pueden modificar su valor durante la ejecución del programa son **variables**.

Una variable puede ser considerada como la abstracción de una posición de memoria.

Podemos clasificar los datos en:

- » simples: son los que están predefinidos en C y podemos utilizar directamente.
- » Estructurados.

Los datos se caracterizan por:

- » **Nombre o identificador:** nombre con el que se hace referencia a una función o al contenido de una zona de memoria (deberá describir su contenido o su función).
- » **Tipo:** identifica el rango posible de valores así como posibles operaciones sobre ellos.
- » **Valores:** elementos determinados, dentro del rango indicado por el tipo y contenido en el espacio de memoria reservado (se interpreta en función del tipo).

3.2. TIPOS FUNDAMENTALES DE DATOS. PALABRAS RESERVADAS.

Los tipos de datos fundamentales en el lenguaje de programación C son los cinco que se muestran en la siguiente tabla:

TIPO	TAMAÑO (Bits)	SIGNIFICADO
char	8	Tipo de dato carácter
int	16	Tipo de dato entero
float	32	Tipo de dato real de simple precisión
double	64	Tipo de dato real de doble precisión
void	-	Vacío (sin valor)

El tipo *void* no tiene tamaño y por tanto, no ocupa espacio en memoria. Este tipo de datos se aplica:

- » Definir un puntero genérico.
- » Especificar que una función no retorna de forma explícita ningún valor.
- » Declarar que una función no utiliza parámetros.

Los principales **modificadores** que se pueden aplicar a los datos básicos, entendiendo por modificador todo elemento que se utiliza para alterar el significado del tipo base de forma que se ajuste más precisamente a las necesidades de cada momento, se encuentran recogidos en la siguiente tabla:

TIPO	TAMAÑO (Bits)	RANGO MÍNIMO
char	8	-127 a 127
Unsigned char	8	0 a 255
Signed char	8	-127 a 127
Int	16	-32.767 a 32.767
Unsigned int	16	0 a 65.535
Signed int	16	Igual que int
Short int	16	Igual que int
Unsigned short int	16	0 a 65.535
Signed short int	16	Igual que short int
Long int	32	-2.147.483.647 a 2.147.483.647
Signed long int	32	Igual que long int

Unsigned long int	32	0 a 4.294.967.295
Float	32	Seis dígitos de precisión
Double	64	Diez dígitos de precisión
Long double	80	Diez dígitos de precisión

Las palabras reservadas son las que encontramos en la siguiente lista:

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

3.3. IDENTIFICADORES.

Los identificadores en C son nombres constituidos por una secuencia de letras, dígitos y el carácter subrayado que nos permite 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:

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

3.4. CONSTANTES.

Las constantes en C se refieren a valores fijos que no pueden ser modificados por el programa. Pueden ser de cualquier tipo de dato básico. La forma en que se representa cada constante depende de su tipo, así:

- ☞ Las constantes de carácter van encerradas en comillas simples.
- ☞ Las constantes enteras se especifican como números sin parte fraccionaria.
- ☞ Existen dos tipos en coma flotante: *float* y *double*, de los cuales existen distintas variaciones dependiendo de los modificadores vistos en apartados anteriores.
- ☞ Las constantes de cadenas de carácter van encerradas entre comillas dobles.
- ☞ Las constantes simbólicas tienen un identificador y se definen mediante la palabra clave *const*.
- ☞ Las constantes de tipo enumeración, son aquellos valores que puede tomar un tipo enumerado.

Otro tipo de constantes son:

- » Constantes hexadecimales y octales.

A veces es más cómodo utilizar un número en base 8 o en base 16, denominándose a estos sistemas de numeración sistema octal y hexadecimal respectivamente. C permite especificar constantes enteras en hexadecimal o en octal en lugar de decimal. Una constante hexadecimal consiste en 0x seguido de la constante en forma hexadecimal. Una constante octal comienza por 0. Algunos ejemplos:

```
int hex = 0x80    /*128 en decimal*/  
int oct = 012     /*10 en decimal*/
```

- » Constantes de carácter con barra invertida.

El incluir entre comillas simples las constantes de carácter es suficiente para la mayoría de los caracteres imprimibles. Pero unos pocos, como el retorno de carro, son imposibles de introducir por el teclado. Por esta razón, C incluye las constantes especiales de carácter con barra invertida.

C admite varios códigos especiales de barra invertida, que se muestran en la siguiente tabla, para que se pueda introducir fácilmente esos caracteres especiales como constantes.

CÓDIGO	SIGNIFICADO
\b	Espacio atrás.
\f	Salto de página.
\n	Salto de línea.
\r	Salto de carro.
\t	Tabulación horizontal.
\"	Comillas dobles.
\'	Comillas simples.
\0	Nulo.
\\	Barra invertida.
\v	Tabulador vertical.
\a	Alerta.
\N	Constante octal (donde N cte octal).
\xN	Constante hexadecimal.

Por ejemplo, el siguiente programa manda al dispositivo de salida un salto de línea y una tabulación y luego imprime *esto es una prueba*:

```
# include <stdio.h>
void main (void)
{
printf ("\n\t Esto es una prueba");
}
```

3.5. OPERADORES.

Los operadores son signos que indican operaciones a realizar con las variables y/o constantes sobre las que actúan. C tiene cuatro clases de operadores: aritméticos, relacionales, lógicos y a nivel de bits, además de otros operadores especiales para determinadas tareas.

Operadores aritméticos.

Los principales operadores aritméticos se presentan en la siguiente tabla:

OPERADOR	ACCIÓN
-	Resta, además de menos monario
+	Suma
*	Multiplicación
/	División
%	Módulo
--	Decremento
++	Incremento

En C el operador % de módulo actúa igual que en otros lenguajes, obteniendo el resto de una división entera. Por ello, % no puede aplicarse a los tipos de coma flotante.

El operador ++ añade 1 a su operando y -- le resta 1, es decir:

```
x = x+1    /* es equivalente o igual a ++x */
x = x-1    /* es equivalente o igual a --x */
```

Los operadores de incremento y de decremento pueden proceder (prefija) o seguir (postfija) al operando. Por ejemplo: ++x o x++.

Sin embargo, existe una diferencia entre la forma prefija y la postfija cuando se utilizan estos operadores en una expresión. Cuando un operador de incremento o de decremento precede a su operando, C lleva a cabo la operación de incremento o de decremento antes de utilizar el valor del operando. Si el operador sigue al operando, C utiliza su valor antes de incrementarlo o decrementarlo. Por ejemplo:

```
x = 10;
y = ++x;    /* pone y a 11 */
```



```
x = 10;
y = x++;    /*y toma el valor 10 */
```

Operadores relacionales y lógicos.

En el término *operador relacional* la palabra relacional se refiere a la relación entre unos valores y otros. En el término *operador lógico* la palabra lógico se refiere a las formas en que esas relaciones pueden conectarse entre sí.

La clave de los conceptos de operadores relacionales y lógicos es la idea cierto (*true*) y falso (*false*). En C, cierto es cualquier valor distinto de 0 y falso es 0. Las expresiones que utilizan los operadores relacionales y lógicos devuelven el valor 1 en caso de cierto y 0 en caso de falso.

Los operadores relacionales son los que se muestran en la siguiente tabla:

OPERADOR	ACCIÓN
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
!=	Distinto

Los operadores lógicos se muestran en la siguiente tabla:

OPERADOR	ACCIÓN
&&	Y
	O
!	NO

La tabla de verdad que regula las operaciones lógicas es la siguiente:

P	q	p&q	p q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Operadores a nivel de bits.

Las operaciones a nivel de bits se refieren a la comprobación, asignación o desplazamiento de los bits reales que componen un byte o una palabra, que corresponden a los tipos estándar de C *char* e *int* con sus variantes. Los operadores a nivel de bits no se pueden usar sobre *float*, *double*, *long double*, *void* u otros tipos más complejos.

Los operadores a nivel de bits se muestran en la siguiente tabla:

OPERADOR	ACCIÓN
&	Y
	O
^	O exclusiva (XOR)
~	Complemento a uno
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

La tabla de verdad que nos regula estas operaciones es:

A	B	~A	A&B	A B	A^B
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Operador de asignación.

Se utiliza para cargar una variable con el valor de una expresión. Puede 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 su derecha.

Variable = Expresión

- » El operador de asignación compuesto (asignación recursiva) utiliza dos símbolos, uno es el de asignación simple y el otro es un símbolo de operación. Los símbolos que se pueden utilizar, son: + - / % << >> & ^ |

Variable Símbolo_operación = Expresión

Equivalente a:

Variable = Variable Símbolo_operación Expresión

C permite la asignación múltiple de variables tal y como se muestra en el siguiente ejemplo:

```
x = y = z = 0;
```

Operador de tamaño.

Se utiliza para obtener la longitud en bytes de una variable o de un especificador de tipo de dato. El operador es *sizeof*.

Operador ?

C contiene un operador muy potente y aconsejable que puede usarse para sustituir ciertas sentencias de la forma *if – then – else*. El operador ? toma la forma general:

```
Exp1 ? Exp2 : Exp3;
```

Donde Exp1, Exp2 y Exp3 son expresiones.

El operador ? actúa de la siguiente forma: evalúa Exp1. Si es cierta, evalúa Exp2 y toma ese valor para la expresión. Si Exp1 es falsa, evalúa Exp3 tomando su valor para la expresión. Por ejemplo:

```
x = 10;  
y = x>9 ? 100 : 200; /* si x es mayor que 9 y valdrá 100 si no valdrá  
200 */
```

Operadores de puntero & y *.

Un puntero es la dirección de memoria de una variable. Una *variable puntero* es una variable específicamente declarada para contener un puntero a su tipo específico. El conocer la dirección de una variable puede ser una gran ayuda en ciertos tipos de rutinas.

El primer operador del puntero es &, que devuelve la dirección de memoria a la que apunta el puntero.

El segundo operador del puntero es *, que devuelve el valor de la posición contenido en la posición de memoria apuntada por el puntero.

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 estructura. Punto Miembro de estructura. Flecha	Izquierda a derecha
! ~ - ++ -- * & (tipo) sizeof	'no' lógico Complemento a uno Signo menos Incremento Decremento Indirección para punteros Dirección para punteros 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 a Distinto de	Izquierda a derecha
& ^ && 	'y' a nivel de bit 'o' exclusivo a nivel de bit 'o' a nivel de bit 'y' lógico 'o' lógico	Izquierda a derecha
?:	Condicional	Derecha a izquierda
= += -= *= /= %= >>= <<= &= ^= =	Asignación	Derecha a izquierda
,	Coma	Izquierda a derecha

3.6. CONVERSIÓN DE TIPOS.

La conversión de tipos tiene lugar cuando en una expresión se mezclan variables de distintos tipos (conversión implícita) o se puede forzar explícitamente a una conversión por el programador (conversión explícita). Por lo tanto, existen dos tipos de conversión de tipos:

Conversión explícita (casting).

Este mecanismo permite convertir una variable a un determinado tipo., para expresar esta operación, se debe anteponer el nombre del tipo de datos deseado al nombre de la variable. Por ejemplo, si `valor` estaba declarado como tipo `int` convertirá a tipo `float` de la siguiente forma:

```
Resultado = (float) valor;
```

La operación de conversión debería usarse cuidadosamente cuando se pasa de un rango superior a uno inferior, ya que podría producirse una pérdida de datos al utilizar un tipo de datos con menos capacidad de representación.

No se puede realizar conversiones del tipo `void` a cualquier otro tipo, pero si de cualquier otro tipo a `void`.

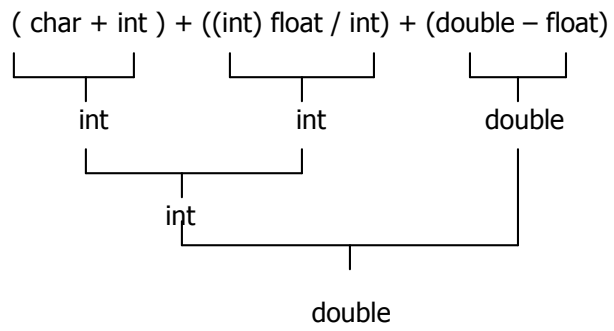
Conversión implícita.

Es el nombre que recibe las operaciones que realiza el compilador convirtiendo el tipo del valor del lado izquierdo de una expresión al mismo tipo del valor del lado derecho de la misma o viceversa.

Esta conversión la realiza teniendo en cuenta el tipo de mayor precedencia existente en la expresión, siendo el orden de precedencia de los tipos el siguiente:

```
long double  
double  
float  
unsigned long  
long  
unsigned int  
int  
char
```

Un ejemplo de esta conversión lo podemos observar en la siguiente figura:



TEMA 4. ESTRUCTURAS DE CONTROL

- 4.1. Estructura secuencial.
- 4.2. Estructuras de selección.
- 4.3. Estructuras de repetición.
- 4.4. Entrada y salida por consola.
- 4.5. Funciones.
- 4.6. Recursividad.

Las **estructuras de control** son utilizadas para controlar la secuencia de ejecución de un programa y, en consecuencia, determinados bloques de instrucciones.

4.1. ESTRUCTURA SECUENCIAL.

El orden de ejecución de las instrucciones de un programa, es de arriba – abajo y de izquierda – derecha, una detrás de otra, respetando siempre el orden inicialmente establecido entre ellas, salvo que dicho orden se altere mediante el uso de estructuras alternativas o repetitivas.

Ejemplo:

El siguiente ejemplo, muestra el algoritmo (en notación pseudocodificada) correspondiente a un programa que lee dos valores numéricos y calcula primero la suma, después la resta, a continuación el producto y seguidamente, la división de ambos valores, escribiendo finalmente el resultado obteniendo en cada una de estas operaciones.

VARIABLES

A	Numérico real
B	Numérico real
S	Numérico real
R	Numérico real
P	Numérico real
D	Numérico real

ALGORITMO

Leer A,B

$S = A + B$

$R = A - B$

$P = A * B$

$D = A / B$

Escribir S, R, P, D

4.2. ESTRUCTURAS DE SELECCIÓN.

Las sentencias de selección son aquellas que controlan la ejecución o la no ejecución de una o más instrucciones, en función de que se cumpla o no una condición previamente establecida.

La estructura de selección más simple es la **alternativa simple** cuya forma es la siguiente:

```
if (expresión)
{
    bloque de sentencias
}
```

Un ejemplo con esta estructura es:

Algoritmo que lee dos valores numéricos y los almacena en dos variables de nombre 'x' y 'z' mostrando en aquellos casos en los que 'x' es mayor 'z' un mensaje que diga "verdadero".

VARIABLES

x numérico entero
z numérico entero

ALGORITMO

Leer x, z
Si (x>z)
 Escribir "verdadero"
Fin si

Una modificación de la sentencia anterior es la **alternativa doble**, que nos permite realizar una serie de operaciones en caso de que no se cumpla la condición inicial. La forma de esta sentencia es:

```

if (expresión)
{
    bloque de sentencias
}
else
{
    bloque de sentencias
}

```

Un ejemplo de estas sentencias es el siguiente:

Algoritmo que lee dos valores numéricos distintos 'x' e 'y' y determina cuál es el mayor dejando el resultado en una tercera variable de nombre 'z'.

VARIABLES

x	numérico real
y	numérico real
z	numérico real

ALGORITMO

Leer x, y

Si (x>y)

z=x

Sino

z=y

Fin si

Escribir " el mayor es:",z

Otra posibilidad que nos ofrecen estas sentencias de selección es los **if anidados**. Un if anidado es un if que es el objeto de otro if o else. En C una sentencia *else* siempre se refiere al *if* más próximo que esté en el mismo bloque el *else* y que no esté ya asociado con un *if*. La forma de este tipo de sentencias es:

```
if (expresión)
{
    bloque de sentencias
}
else if (expresión)
{
    bloque de sentencias
}
else
{
    bloque de sentencias
}
```

Como puede observarse los *if* anidados tienden a ser estructuras muy grandes y complejas, para simplificar este aspecto C incorpora una sentencia de selección múltiple, **switch**, que compara sucesivamente el valor de una expresión con una lista de constantes enteras o de caracteres. Cuando se encuentra una correspondencia, se ejecutan las sentencias asociadas con la constante. La forma general de la sentencia *switch* es:

```
Switch (expresión)
{
    case constante1: sentencias1; [break;]
    case constante2: sentencias2; [break;]
    case constante3: sentencias3; [break;]
    ...
    default: sentencias;
}
```

La sentencia *switch* no puede evaluar expresiones relacionales o lógicos. Solo puede realizar comparaciones entre el resultado de la expresión y las constantes de cada *case*. Se evalúa la expresión del primer *case* y se considera el resultado de dicha evaluación. Si coincide con el valor se ejecutan las sentencias1, las sentencias2,...; si coincide con el valor del segundo *case* se ejecutan las sentencias2, las sentencias3,...; y así sucesivamente se irían comprobando todos los *case*.

Si se desea ejecutar únicamente la sentencia correspondiente a un *case* hay que poner la sentencia *break* después de las sentencias. El efecto del *break* es dar por terminada la ejecución de la sentencia *switch*.

Un ejemplo en el que se emplea esta estructura es el siguiente:

Programa que dado un carácter imprime:

- "es una vocal mayúscula" para A, E, I, O, U.
- "es una vocal minúscula" para a, e, i, o, u.
- "no es una vocal" para el resto.

ALGORITMO

```
Switch (caracter)
{
    case 'A':
        ...
    case 'U': printf("es una vocal mayúscula"); break;
    case 'a':
        ...
    case 'u': printf("es una vocal minúscula");
    default: print("no es una vocal");
}
```

Tres aspectos importantes que se deben destacar de esta sentencia son:

- » La sentencia *switch* se diferencia de la sentencia *if* en que *switch* solo puede comprobar la igualdad, mientras que *if* puede evaluar expresiones relacionales o lógicas.
- » No puede haber dos constantes *case* en el mismo *switch* que tengan los mismos valores. Por supuesto, una sentencia *switch* contenida en otra sentencia *switch* puede tener constantes *case* que sean iguales.
- » Si se utilizan constantes de tipo carácter en la sentencia *switch*, se convierten automáticamente a sus valores enteros.

4.3. ESTRUCTURAS DE REPETICIÓN.

Son aquellas que nos permiten variar o alterar la secuencia normal de ejecución de un programa haciendo posible que un bloque de instrucciones se ejecute más de una vez de forma consecutiva.

Existen principalmente tres tipos de estructuras de repetición, que son las siguientes:

Estructura *Mientras*.

La estructura *Mientras* se caracteriza porque su diseño permite repetir un bloque de instrucciones de 0 – n veces, es decir, que en aquellos casos en los que la condición establecida sea verdadera, el número de veces que se ejecutará dicho bloque de instrucciones será una vez como mínimo y n veces como máximo, mientras que en el caso de que la condición establecida sea falsa dicho bloque de instrucciones no se ejecutará ninguna vez.

La forma general de esta estructura es:

```
While (expresión condicional)
{
    Sentencia1
    ...
    sentenciaN
}
```

Un ejemplo con esta estructura es el siguiente:

Diseño del algoritmo correspondiente a un programa que lee un número entero positivo y determina el número de dígitos decimales necesarios para la representación de dicho valor.

VARIABLES.

Ndigitos	Numérico entero
Pot	Numérico entero
N	Numérico entero

ALGORITMO

```
Ndigitos = 1
Pot = 10
Leer N
```

```

Mientras Pot <= N
    Ndigitos ++
    Pot*=10
Fin mientras
Escribir "Se necesitan ", Ndigitos

```

Estructura *Repetir – Mientras*.

La estructura *repetir – mientras* se caracteriza porque su diseño permite repetir un bloque de instrucciones de 1 – n veces, es decir, ya sea verdadera o falsa la condición establecida, el número de veces que se ejecutará el bloque de instrucciones será de una vez como mínimo y de n veces como máximo.

La forma general de esta estructura es:

```

do
{
    sentencial
    ...
    sentenciaN
}

```

Un ejemplo con este tipo de estructuras es el siguiente:

Algoritmo correspondiente a un programa que lee un número entero positivo y seguidamente escribe el carácter asterisco (*) un número de veces igual al valor numérico leído. En aquellos casos en los que el valor leído no sea entero, numérico y positivo se deberá escribir solamente un asterisco.

VARIABLES

```

ast          numérico entero
numast       numérico entero

```

ALGORITMO

```

ast =0
Leer numast
Repetir
    ast++
    escribir '*'
Mientras (ast<numast)

```


Existen instrucciones repetitivas que utilizan estructuras *Hasta* y *repetir – hasta* cuyo funcionamiento es muy similar a las estructuras *Mientras* y *Repetir – Mientras*, pero modificando la condición y las salidas de la condición.

Estructura *Para*.

Este tipo de instrucciones repetitivas se caracteriza porque el número de veces que se repetirá el bloque de instrucciones generalmente está fijado de antemano.

La estructura general de esta sentencia es la siguiente:

```
for (inicialización; condición; incremento/decremento)
{
    sentencia1;
    ...
    sentenciaN;
}
```

La estructura *para* es una forma compacta de representar un bucle *Mientras* específico. Para comprobar esto un ejemplo es el que se muestra en la siguiente tabla:

Estructura <i>para</i>	Estructura <i>Mientras</i>
VARIABLES P Numérico entero C Numérico entero N Numérico entero ALGORITMO P=1 Para C de 1 a 5 con inc=1 Leer N P=P+N Fin para Escribir " producto =", P	VARIABLES P Numérico entero C Numérico entero N Numérico entero ALGORITMO P=1 C=1 Mientras C<=5 Leer N P=P+N C=C+1 Fin mientras Escribir " producto =", P

El control del bucle en una estructura *para* se realiza de antemano el número de veces que se van a efectuar la operaciones del bucle. Este número de veces puede ser establecido por

una constante o por una variable que almacena el valor introducido por teclado, o bien calculado en función de los valores inicial, final e incremento.

Un ejemplo con este tipo de estructuras es el siguiente:

Algoritmo correspondiente a un programa que escribe el producto de una serie de números introducidos por teclado.

VARIABLES

Producto	Numérico entero
nv	Numérico entero
Vcont	Numérico entero
N	Numérico entero

ALGORITMO

```
Producto = 1
Leer nv
Para Vcont de 1 a nv con inc=1
    Leer N
    Producto*=N
Fin para
Escribir "El producto es:", Producto
```

4.4. ENTRADA Y SALIDA POR CONSOLA.

C es prácticamente único en su tratamiento de las operaciones de entrada y salida. Esto se debe a que el lenguaje C no define ninguna palabra clave para realizar la E/S. Por el contrario, la entrada y la salida se realizan a través de funciones de la biblioteca. El sistema de E/S de C es una pieza de ingeniería elegante que ofrece un mecanismo flexible a la vez que consistente para transferir datos entre dispositivos. Sin embargo, el sistema de E/S de C es bastante amplio e incluye muchas funciones diferentes.

En C existe E/S por consola y E/S por archivo. Técnicamente C distingue poco entre la E/S por consola y la E/S por archivo. Sin embargo, son mundos conceptualmente distintos. En este apartado examinaremos la E/S por consola y en el tema correspondiente examinaremos la E/S por archivo.

Por lo tanto, en C se consideran funciones de entrada aquellas que nos permiten introducir datos desde un periférico a la memoria del ordenador, por el contrario, se consideran funciones de salida, aquellas que nos permiten sacar datos desde la memoria del ordenador a un periférico o dispositivo externo. Estas funciones se encuentran en una librería estándar de nombre "stdio.h".

De todas las funciones definidas en la función de librería "stdio.h" vamos a comentar las consideradas como funciones básicas: `getchar()`, `putchar()`, `scanf()` y `printf()`.

Getchar ().

La función completa es: *int getchar(void)*.

Como podemos observar no lleva argumentos y lee un carácter del dispositivo estándar de entrada.

Un ejemplo en el que utilizamos esta función es el siguiente. Se trata de un programa que contabiliza el número de caracteres de una línea de texto introducida a través del dispositivo estándar de entrada (teclado).

```
# include <stdio.h>

main ( )
{
    int j=0;
    while (getchar( )!= '\n')
        j++;
    printf( " La línea tiene %d caracteres. \n",j);
}
```

Putchar ().

La definición completa de esta función es: *int putchar (int ch)*.

Como puede observarse tiene como argumento una variable o una constante carácter y escribe o muestra en pantalla (dispositivo estándar de salida) un carácter.

Un ejemplo empleando esta función es el mostrado a continuación. Se trata de un programa que muestra en pantalla todos aquellos caracteres introducidos por teclado hasta la pulsación del carácter salto de línea.

```
# include <stdio.h>
main ( )
{
    char ch;
    do
    {
        ch = getchar( );
        putchar (ch);
    }
    while( ch != '\n');
}
```

Scanf ().

La definición de esta función es la siguiente: *int scanf (const char *formato, lista de argumentos)*.

Como puede observarse los argumentos que admite esta función es un puntero a la cadena de control con los formatos de tipo y lista de variables. La finalidad de la función es almacenar en variables de memoria los datos introducidos a través del dispositivo estándar de entrada.

En el siguiente cuadro se muestra los especificadores de formato para la entrada de datos que nos indican el tipo de salto leído mediante el uso de la función *scanf ()*.

CÓDIGO	SIGNIFICADO
%c	Lee un solo carácter
%d	Lee un número entero decimal
%I	Lee un número entero decimal
%e	Lee un valor numérico real
%f	Lee un valor numérico real
%h	Lee un entero corto
%o	Lee un número octal
%s	Lee una cadena de caracteres
%x	Lee un número hexadecimal
%p	Lee un puntero

En el siguiente programa se muestra como se utiliza esta función. Se trata de un programa que muestra la versatilidad y flexibilidad de la función *scanf* (), que puede ser utilizada para leer cualquier tipo de dato del dispositivo estándar de entrada (teclado).

```
#include <stdio.h>
main ( )
{
    int edad, fecha_act;
    char nombre [25];
    puts ("Introduce tu edad: ");
    scanf ("%d",&edad);
    puts ("Introduce el año actual: ");
    scanf ("%d",&fecha_act);
    puts ("Introduce tu nombre: ");
    scanf ("%s",noimbre);
    printf ("%s, naciste en %d.\n",nombre,fecha_act - edad);
}
```

Printf ().

La función completa (definición) es la siguiente: *int printf (const char *formato, lista de argumentos)*.

Los argumentos que puede recibir es un número indeterminado de cualquier tipo y de cualquier formato. Su finalidad es escribir una cadena en el dispositivo estándar de salida.

Seguidamente se muestra un cuadro con los distintos formatos de impresión que utiliza la función *printf* () para la salida de los datos:

FORMATOS DE IMPRESIÓN	EXPLICACIÓN
%h	Imprime un entero corto de 1 byte
%u	Imprime enteros decimales sin signo de 2 bytes
%d	Imprime enteros decimales con signo de 2 bytes
%ld	Imprime un entero largo de 2 bytes
%f	Imprime valores con punto decimal de 4 bytes
%lf	Imprime valores con punto decimal de 8 bytes
%c	Imprime un carácter de 1 bytes
%s	Imprime una 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 (notación científica)
%E	Imprime valores reales (notación científica)
%g	Una %e o %f según el tamaño del valor a imprimir
%%	Imprime el signo %
%p	Muestra un puntero
%í	Imprime 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	EXPLICACIÓN
% - Numero Letra	Ajusta la información al extremo izquierdo del campo
%Numero 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'

En el siguiente programa se pone de manifiesto la flexibilidad de la función *printf* () para mostrar mensajes donde parte de la información que los constituye puede variar a lo largo del programa.

```
#include <stdio.h>
main ( )
{
    printf ("A los %d %stos apagaron las luce%c ", 2+2+1,
        "minu", 's');
    printf ("y todo quedó %s durante %d %c%c.\n", "oscuro", 4+6,
        'g');
    printf ("El %s me costó 1%d%c000 %s.\n", 6, '.', "pts");
}
```

4.5. FUNCIONES.

Es necesario dividir los programas en subprogramas o funciones más pequeñas que serán llamadas por la principal.

Las ventajas que esto conlleva son:

- » Modularización.
- » Ahorro de memoria y tiempo de desarrollo.
- » Independencia de datos y ocultación de la información.

Una función en C es una porción de código o programa que realiza una determinada tarea.

Una función está asociada a un nombre o un identificador que se utiliza para referirse a ella desde el resto del programa. En toda función C hay que distinguir entre su:

- » Definición (cabecera, argumentos, sentencias).
- » Declaración (especificación de tipos).
- » Llamada (con su nombre y los argumentos instanciados).

La función en C se llama incluyendo el nombre, seguido de los argumentos en una sentencia del programa principal o de otra función.

Los argumentos son los datos que se envían a la función incluyéndolos entre paréntesis a continuación del nombre, separados por comas.

El resultado es el valor de retorno de la función, que aparece sustituyendo el nombre de la función en el mismo lugar donde se ha hecho la llamada.

Un ejemplo de función es la que se muestra en el siguiente código, el cual nos permite calcular la potencia de un número positivo dado, dando también el exponente entero positivo al que se desea elevar el número.


```

int potencia (int x, int n)
{
    int c=0,pot=1;
    While (n>c)
    {
        pot*=x;
        c++;
    }
    return pot;
}

```

Por lo tanto, una función es un módulo o parte de una aplicación informática que forma un bloque, una unidad de código. El uso de funciones es una característica peculiar y muy importante del lenguaje C, pues permite un desarrollo modular dando sencillez a éste y precisión en la depuración de los programas, como ya hemos comentado. El formato para la definición de una función es el que se muestra a continuación:

```

Tipo nombre_función (<Param_1>, <Param_2>, ..., <Param_n>)
{
    <Cuerpo de la función>
}

```

Los parámetros se pueden clasificar en:

- » **Parámetros actuales:** Son variables locales en el módulo que llama cuyo valor o dirección es enviado al módulo invocado.
- » **Parámetros formales:** Son variables locales en el módulo invocado, que reciben el valor o la dirección de los parámetros actuales del módulo que lo invoca en el momento de ser ejecutada la llamada.

Seguidamente se muestran dos ejemplos muy similares en los que se pone de manifiesto las dos formas que existen de pasar valores a una función C, es decir, un primer ejemplo donde se pasan parámetros por **valor** y un segundo ejemplo donde se pasan los mismos parámetros pero por **referencia**.

Paso de parámetros por dirección o referencia.

Los parámetros formales que deberán ser de tipo puntero reciben la dirección de los parámetros actuales y, por tanto, las modificaciones que se realicen en el valor de las variables apuntadas por los parámetros formales afectarán al valor de los parámetros actuales. Con este

paso de parámetros se permite de una forma implícita que una función retorne más de un valor a la función que llama.

Paso de parámetros por valor.

Los parámetros formales reciben el valor de los parámetros actuales y por tanto, las modificaciones que se realicen en los parámetros formales no afectan al valor de los parámetros actuales.

En el siguiente ejemplo se realiza el paso por valor a la función:

```
void permutamal (double x, double y)
{
    double temp;
    temp=x;
    x=y;
    y=temp;
}
```

Esta función no realiza bien su tarea debido a que solo permuta los valores de las copias que se le pasa en cada llamada.

En la función siguiente se realiza el paso por referencia a la función:

```
void permutabien (double *x, double *y)
{
    double temp;
    temp=*x;
    *x=*y;
    *y=temp;
}
```

Esta función realiza correctamente su función.

Por último acerca de las funciones podemos decir que en cualquiera de sus definiciones pueden aparecer variables de tres tipos:

- » *Auto* (por defecto), solo visible para la propia función, se crean cada vez que se llama a la función.
- » *Static*, solo visibles para la propia función, conservan su valor entre distintas llamadas a la función.
- » *Extern*, son variables definidas fuera de la función, son visibles para la propia función.

4.6. RECURSIVIDAD.

En C, las funciones pueden llamarse a sí mismas. Si una expresión en el cuerpo de una función llama a la propia función, se dice que ésta es *recursiva*. La recursividad es el proceso de definir algo en términos de sí mismo y a veces se llama *definición circular*.

Un sencillo ejemplo de función recursiva es *factorial ()*, que calcula el factorial de un entero. El factorial de un número n es el producto de todos los números enteros entre 1 y n. El código de esta función es:

```
int factorial (int n)
{
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

Cuando una función se llama a sí misma, se asigna un espacio en la pila para las nuevas variables locales y parámetros, y el código de la función se ejecuta con estas nuevas variables desde el principio. Una llamada recursiva no hace una nueva copia de la función. Sólo son nuevos los argumentos. Al volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la ejecución se reanuda en el punto de la llamada a la función dentro de la función. Podría decirse que las funciones recursivas tienen un efecto “telescópico” que las lleva fuera y dentro de sí mismas.

TEMA 5. TIPOS ESTRUCTURADOS DE DATOS.

- 5.1. Matrices.
 - 5.2. Cadena de caracteres (Strings).
 - 5.3. Estructuras.
 - 5.4. Enumerados.
 - 5.5. Tipos definidos por el usuario.
 - 5.6. Punteros.
-

5.1. MATRICES.

Un *array* o una *matriz* es una colección de variables del mismo tipo que se referencian por un nombre en común. A un elemento específico de un array se accede mediante un índice. En C todos los arrays constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la dirección más alta corresponde al último elemento. Los arrays pueden tener una o varias dimensiones.

Arrays unidimensionales.

La forma general de declaración de una array unidimensional es:

Tipo nombre_de_variable [tamaño].

Como en otros lenguajes, los arrays tienen que declararse explícitamente para que así el compilador pueda reservar espacio en memoria para ellos. Aquí, *tipo* declara el tipo base del array, que es el tipo de cada elemento del array. El valor de *tamaño* indica cuántos elementos mantendrá el array.

En C, todos los arrays tienen el 0 como índice de su primer elemento. Por tanto, cuando se escribe se está declarando un array de caracteres que tiene un número *n* de elementos de 0 a *n*-1.

La cantidad de memoria requerida para guardar un array está directamente relacionado con su tipo y su tamaño. Para un array unidimensional, el tamaño total en bytes se calcula con la siguiente expresión:

Total de bytes = sizeof (tipo) * tamaño del array

C no muestra los límites de los arrays. Se puede pasar cualquier extremo de un array y escribir en alguna otra variable de datos o incluso en el código del programa. Como programador, es tarea suya proporcionar una comprobación de límites cuando sea necesaria.

Los arrays unidimensionales son, en esencia, listas de información del mismo tipo que se guarda en posiciones contiguas de memoria según el orden del índice.

C nos permite inicializar matrices en el momento de su declaración y según el momento en el que se asigne el tamaño se clasifican en:

- Dinámicas: se definen en tiempo de compilación.
- Estáticas: se definen en tiempo de ejecución.

Un ejemplo de matriz unidimensional es:

```
Double balance [100];
```

Arrays bidimensionales.

C admite arrays multidimensionales. La forma más simple de un array multidimensional es el array bidimensional. Un array bidimensional es esencialmente un array de arrays unidimensionales. Para declarar un array de enteros bidimensional de tamaño x, e y se escribiría:

```
int enteros [x][y];
```

Los arrays bidimensionales se almacenan en filas y columnas, en las que el primer índice indica la fila y el segundo indica la columna.

En el caso de un array bidimensional, la siguiente fórmula da el número de bytes en memoria necesario para guardarlo:

Bytes tamaño del primer índice*tamaño del segundo índice*sizeof (tipo base)

Arrays multidimensionales.

C permite arrays de más de dos dimensiones. El límite exacto, si lo hay, viene determinado por el compilador. La forma general de declaración de un array multidimensional es:

```
Tipo nombre [tamaño1][tamaño2][tamaño3]...[tamañoN]
```

Los arrays de tres o más dimensiones no se utilizan a menudo por la cantidad de memoria que requieren para almacenarlos.

Con los arrays multidimensionales el cálculo de cada índice le lleva a la computadora una significativa cantidad de tiempo. Esto significa que el acceso a un elemento en un array multidimensional dura más que el acceso a un elemento en un array unidimensional.

Paso de arrays unidimensionales a funciones.

En C no se puede pasar un array completo como argumento a una función. Sin embargo, se puede pasar a una función un puntero a un array, especificando el nombre del array sin índice.

Si una función recibe un array unidimensional, se puede declarar el parámetro formal de tres formas: como un puntero, como un array delimitado o como un array no delimitado.

El resultado de los tres métodos de declaración es idéntico, porque cada uno le indica al compilador que se va a recibir un puntero a carácter. En la primera declaración se usa realmente un puntero. En la segunda se emplea la declaración estándar de array.

5.2. CADENA DE CARACTERES (STRINGS).

El uso más común de los arrays unidimensionales es, con mucho, como cadenas de caracteres. En C, una cadena se define como un array de caracteres que termina con un carácter nulo. Un carácter nulo se especifica como `'\0'` y generalmente es un cero. Por esta razón, para declarar arrays de caracteres es necesario que sean de un carácter más que la cadena más larga que pueda contener.

Aunque C no define un tipo de dato de cadenas, permite disponer de constantes de cadena. Una *constante de cadena* es una lista de caracteres encerrada entre dobles comillas.

No es necesario añadir explícitamente el carácter nulo al final de las constantes de cadena ya que el compilador de C lo hace automáticamente.

C incluye una gran cantidad de funciones de manejo de cadenas, las cuales se encuentran en la librería *string.h*. Las más importantes son:

Strcpy (c1,c2).

Copia la cadena c1 en la cadena c2. Su definición completa es la siguiente:

`Char *strcpy (char *s1, const char *s2).`

Un ejemplo en el que se muestra la utilización de esta cadena es el siguiente:

```
# include <stdio.h>
# include <string.h>
int main (void)
{
    char string [10];
    char *str1="abcdefghi";
    strcpy (string,str1);
    printf ("%s\n",string);
    return 0;
}
```

Strcat (c1,c2).

Concatena al final de la cadena c1 la cadena c2. Su definición completa es:

`char *strcat (char *s1, const char *s2)`

Un ejemplo de utilización de esta cadena es el siguiente:

```
# include <stdio.h>
# include <string.h>
int main (void)
{
    char destination [50];
    char *s1 = "Programación";
    char *s2 = "en";
    char *s3 = "C.";
    strcpy (destination, s1);
    strcat (destination, s2);
    strcat (destination, s3);
    printf ("%s\n",destination);
    return 0;
}
```

Strlen (c1).

Devuelve la longitud de c1 (solamente el número de caracteres: cadena[strlen(cadena)]='\0'). Su definición completa es:

```
int strlen (const char *s1);
```

En el siguiente programa se muestra un ejemplo de la utilización de esta función:

```
# include <stdio.h>
# include <string.h>
int main (void)
{
    char *string = " me gusta FI";
    printf ("%d\n",strlen(string));
    return 0;
}
```

Strcmp (c1,c2).

Compara la cadena c1 con la cadena c2 devolviendo:

- » 0 si son iguales.
- » Menor que 0 si c1<c2.
- » Mayor que 0 si c1>c2.

La definición completa de esta función es la siguiente:

```
int strcmp (const char *s1, const char *s2);
```

Un ejemplo de utilización de esta función es el que se muestra a continuación:

```
# include <stdio.h>
# include <string.h>
int main (void)
{
    char *buf1 = "aaa", *buf2 = "bbb";
    int comparador;
    comparador = strcmp (buf1,buf2);
    if (comparador < 0 ) printf("Buffer 1 precede a buffer 2");
    else
    {
        if (comparador > 0)
            printf ("buffer 2 precede a buffer 1");
        else printf ( "buffer 1 igual a buffer 2");
    }
    return 0;
}
```

Otras dos funciones importantes son:

- » Strchr (c1,car): devuelve un puntero a la primera ocurrencia de car en c1.
- » Strstr (c1,c2): devuelve un puntero a la primera ocurrencia de c2 en c1.

5.3. ESTRUCTURAS.

En lenguaje C, una estructura es una colección de variables que se referencia bajo un único nombre, proporcionando un medio eficaz de mantener junta una información relacionada. Una *declaración de estructura* forma una plantilla que puede utilizarse para crear estructuras. Las variables que componen la estructura se llaman miembros de la estructura (los miembros de la estructura también son llamados *elementos* o *campos*).

Para poder emplear una estructura es necesario hacer una declaración de la estructura mediante la palabra clave *struct*, El formato básico de la declaración es el siguiente:

```
Struct Nombre_estructura
{
    tipo elemento 1;
    tipo elemento 2;
    ...
    tipo elemento N;
};
```

Un ejemplo de declaración de un estructura es:

```
Struct empleado
{
    char nombre [50];
    char dirección [75];
    long telefono;
    float sueldo;
};
```

Puede observarse que la declaración termina con punto y coma, esto se debe a que la declaración de estructura es una sentencia.

A la hora de declarar variables del tipo estructura que hemos definido tenemos dos posibilidades:

- » En la propia declaración de la estructura. Por ejemplo:

```
Struct empleado
{
    char nombre [50];
    char dirección [75];
    long telefono;
    float sueldo;
} Pepe;          /* hemos declarado la variable Pepe como del
tipo empleado*/
```

» Aparte, tal y como se muestra en los siguientes ejemplos:

```
Struct empleado Juan, socios [30];
```

Referencia a los miembros de la estructura.

Los miembros de la estructura se referencian utilizando el operador `.` (denominado usualmente operador punto). Ejemplo: `Pepe.telefono = 969225094;`

Arrays de estructuras.

El uso más común de las estructuras es en los arrays de estructuras. Para declarar un array de estructuras, se debe definir la estructura y luego declarar la variable array de dicho tipo. Por ejemplo: `struct empleado socios [30];`

Para acceder a una determinada estructura, se indexa el nombre de la estructura. Por ejemplo para imprimir el teléfono del socio 3, sería necesaria la siguiente instrucción:

```
printf ("%d", socio [3].telefono);
```

Paso de estructuras completas a funciones.

Cuando se utiliza una estructura como argumento de una función, se pasa la estructura íntegra mediante el uso del método estándar de llamada por valor. Esto significa por supuesto, que todos los cambios realizados en los contenidos de la estructura dentro de la función a la que pasa no afectan a la estructura utilizada como argumento.

Un ejemplo de este caso se muestra en la siguiente función:

```
Struct lector                                Struct libro
{
    char nombre [80];
    char dirección [100];
    char telefono [20];
    int codigo;
    int edad;
    int prestamos;
};

void prestar (struct lector lec, struct libro lib)
/* función que realiza el préstamo de un libro a un lector*/
{
    if (lib.prestamo)
        printf ("el libro no está disponible");
    else
    {
        if (lec.prestamos>=3)
            printf ("ya tiene demasiado libros");
        else
        {
            lib.prestado=1;
            lec.prestamos++;
            lib.prestado_a = lec.codigo;
        }
    }
}
```

5.4. ENUMERADOS.

Una *enumeración* es un conjunto de constantes enteras con nombre que especifica todos los valores válidos que una variable de ese tipo puede tener. Las enumeraciones son normales en la vida cotidiana.

Las enumeraciones se definen de forma parecida a las estructuras: la palabra clave *enum* señala el comienzo de un tipo enumerado. La forma general de una enumeración es:

```
Enum etiqueta { lista de enumeraciones } lista de variables;
```

Aquí tanto la etiqueta de la enumeración como la lista de variables son opcionales. Como en las estructuras, se usa el nombre de la etiqueta para declarar variables de ese tipo. El siguiente fragmento de código define una enumeración llamada moneda y declara dinero de ese tipo:

```
Enum moneda {peñique, niquel, diez_centavos, cuarto,
medio_dólar, dólar} pelas;
Enum moneda dinero;
/* tanto pelas como dinero son enumerados de tipo moneda*/
```

La clave para entender la enumeración es que cada uno de los símbolos corresponde a un valor entero. De esta forma, pueden usarse en cualquier expresión entera. Cada símbolo recibe un valor que es mayor en uno que el símbolo que le precede. El valor del primer símbolo de la enumeración es el 0.

Se pueden especificar el valor de uno o más símbolos utilizando un inicializador. Esto se hace siguiendo el símbolo con un signo igual y un valor entero. A los símbolos que siguen a uno inicializado se les asigna valores consecutivos al valor previo de inicialización.

Una suposición errónea sobre los enumerados es que los símbolos pueden leerse y escribirse directamente.

Un ejemplo utilizando enumerados es la siguiente función:

```
Enum estilo {aventuras, amor, c_ficcion, historia};
```

```
Void listado_personalizado (enun estilo pref)
/*función que imprime un listado personalizado según los gustos
del lector*/
{
    int lib;
    for (lib=0;lib < total_lib;lib++)
    {
        if (lib [lib].tipo == pref)
            printf ("%s\n", lib [lib].titulo)
    }
}
```


5.5. TIPOS DEFINIDOS POR EL USUARIO.

C permite definir explícitamente nuevos nombres para los tipos de datos, usando la palabra clave *typedef*. Realmente no se crea un nuevo tipo de dato sino que se define un nuevo nombre para un tipo existente. Este proceso puede ayudar a hacer más transportables los programas con dependencias con las máquinas. Si se define un nombre de tipo propio para cada tipo de datos dependiente de la máquina que se utilice en el programa, entonces sólo habrá que cambiar las sentencias *typedef* cuando se compile en un nuevo entorno.

La forma general de la sentencia *typedef* es:

```
Typedef tipo nuevonombre;
```

Donde *tipo* es cualquiera de los tipos de datos válidos y *nuevonombre* es el nuevo nombre para ese tipo. Ejemplo:

```
Typedef float balance;
```

Otro ejemplo puede ser:

```
Typedef struct lector
{
    char nombre [80], telefono [20], dirección [100];
    int codigo, edad, prestamos,
    enum estilo pref.;
};
```

5.6. PUNTEROS.

Un **puntero** es una variable que contiene una dirección de memoria. Esta dirección es la posición de otro objeto (normalmente otra variable) en memoria. Por ejemplo, si una variable contiene la dirección de otra variable, entonces se dice que la primera variable *apunta* a la segunda.

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.

Si una variable puntero va a contener un puntero, entonces tiene que declararse como tal. Una declaración de puntero consiste en un tipo base, un * y el nombre de la variable. La forma general de declaración de una variable puntero es:

Tipo * nombre;

El tipo base del puntero define el tipo de variables a las que puede apuntar el puntero. Técnicamente, cualquier tipo de puntero puede apuntar a cualquier lugar de la memoria. Sin embargo, toda la aritmética de punteros está hecha en relación a su tipo base, por lo que es importante declarar correctamente el puntero.

Un ejemplo de definición de un puntero sería:

```
int *puntero;
```

Puntero es una variable que apuntará a la dirección de / un puntero a una variable de tipo entero.

Operadores de dirección e indirección.

El lenguaje C dispone del operador de dirección &, que permite obtener la dirección de la variable sobre la que se aplica.

También dispone del operador de indirección *, que permite acceder al contenido de la zona de memoria a la que apunte el puntero sobre el cual aplicamos dicho operador.

Algunos ejemplos con estos operadores son los que se muestran a continuación:

```
int i, j, *p;      /* p es un puntero*/
p=&i;              /* p apunta a la dirección de i*/
*p = 10;          /* i toma el valor 10 */
p=&j;              /* p apunta a la dirección de j */
*p = 11;          /* j toma el valor de 11 */
```

Operaciones con punteros.

Ni las constantes, ni las expresiones tienen dirección y por tanto, no se les puede aplicar &.

No se permite la asignación directa entre punteros que apuntan a distintos tipos de variables. Existe un tipo indefinido de puntero, *void*, que funciona como comodín para las asignaciones independientemente de los tipos.

Esto se muestra en el siguiente ejemplo:

```
int *p;
double *q;
void *r;
p=q;      /* ilegal*/
r=q;      /* legal*/
p=r;      /* legal*/
```

Aritmética de punteros.

Un puntero contiene no solo información de la dirección en memoria de la variable a la que apunta sino también de su tipo.

Las operaciones aritméticas que podemos hacer con punteros son solamente dos: la suma y la resta.

En estas operaciones las unidades que se suman y se restan son los bytes de memoria que ocupa el tipo de las variables a la que apunta. Este concepto quedará más claro con el siguiente ejemplo:

```
int *p;
double *q;
p++;      /* sumar 1 a p implica aumentar en 2 bytes la
dirección de memoria a la que apunta */
q--;      /* disminuir 1 a q implica disminuir 4 bytes en la
dirección apuntada por q */
```

Punteros y arrays.

Entre las matrices y los punteros existe una estrecha relación. Sea el array así definido:

```
Int array [8][5];
```

Y conociendo que los arrays se almacenan por filas, una fórmula para el direccionamiento de una matriz sería: " la posición en memoria del elemento array [7][2] será la posición array [0][0] + 7*5 + 2 ".

Este hecho quedará más claro en el siguiente ejemplo:

```
int vect [10], mat [3][5], *p;  
p = &vect [0];  
printf ("%d",*(p+2));          /* imprime mat [0][2]*/  
printf ("%d",*(p+5));          /* imprime mat [1][0]*/  
printf ("%d",*(p+14));         /* imprime mat [2][4]*/
```

Un aspecto a tener muy en cuenta es que el nombre de un vector es un puntero al primer elemento de dicho vector, es decir, estas dos expresiones son equivalentes:

```
P = vect;  
P = &vect[0];
```

Así pues, como `vect` apunta a `vect [0]` tenemos que `*(vect + 4) = vect [4]`.

Por otro lado, el nombre de una matriz bidimensional es un puntero al primer elemento de un vector de punteros. Hay un vector de punteros con el mismo nombre que la matriz y el mismo número de elementos que filas de la matriz cuyos elementos apuntan a los primeros elementos de cada fila de la matriz (fila de la matriz se puede considerar como una matriz unidimensional).

Este concepto quedará un poco más claro con los siguientes ejemplos.

```
int mat [5][3];  
**(mat++)= mat [1][0];  
*[(*mat)++] = mat [0][1];
```

Así pues, `mat` es un puntero a una variable de tipo puntero. Por tanto, `mat` es lo mismo que, `&mat [0]` y `mat [0]` lo mismo que `&mat [0][0]`. Análogamente `mat [1]` es `&mat [1][0]`,...

Si hacemos `p = mat;` tendremos:

```
*p = mat [0];  
*(p+1) = mat [1];  
**p = mat [0][0];  
**(p+1) = mat [1][0];  
*(*(p+1)+1) =mat [1][1];
```


TEMA 6. FICHEROS

- 6.1. Introducción.
 - 6.2. Puntero a fichero.
 - 6.3. Funciones para el tratamiento de ficheros.
-

6.1. INTRODUCCIÓN.

C distingue principalmente entre dos tipos de corrientes o flujos de datos:

- » **De texto:** Están constituidos por una secuencia indefinida de caracteres, con la peculiaridad de que es posible que se originen ciertas conversiones o transformaciones en los datos, pudiendo haber claras diferencias entre los caracteres leídos del dispositivo físico donde se hayan almacenado y los caracteres almacenados en la memoria principal. Por ejemplo, es posible la conversión del carácter LF (salto de línea) en dos caracteres CR y LF (retorno de carro y salto de línea) si el entorno del sistema así lo precisa.

- » **Binarios:** En este tipo de flujo o corriente existe una correspondencia exacta entre los caracteres leídos del dispositivo físico donde se hayan almacenados y los caracteres almacenados en memoria principal, es decir, que no se realiza conversión alguna.

6.2. PUNTERO A FICHERO.

Un puntero a fichero no es más que una variable de tipo puntero a través de la cual podemos realizar operaciones de E/S sobre el fichero referenciado. Este puntero se encuentra definido en la librería estándar "stdio.h".

Su formato de definición es el siguiente:

```
FILE * Nombre_Puntero_a_Fichero;
```

6.3. FUNCIONES PARA EL TRATAMIENTO DE FICHEROS.

Apertura y cierre de ficheros.

La apertura y cierre de ficheros se realiza con las siguientes funciones definidas en la librería "stdio.h".

» Función de apertura.

La función de apertura es la siguiente:

```
FILE *fopen (const char *Nombre_fichero, const char *Modo_apertura);
```

Podemos observar que esta función como argumentos lleva un primer argumento que es un puntero a una cadena de caracteres que contiene el nombre físico del fichero y un segundo argumento que también es un puntero a cadena correspondiente al modo de apertura del fichero.

La función nos devuelve un puntero a FILE, es decir, al fichero asociado a ese flujo de datos o el valor NULL si el fichero no puede abrirse.

En cuanto a la finalidad de la función es abrir un fichero, permitiéndonos el acceso a la información en él contenida. El acceso a dicha información vendrá limitado por el modo de apertura especificado.

Los principales modos de apertura son los que se expresan en la siguiente tabla:

MODO	SIGNIFICADO
r	Abre un fichero de texto para lectura. El fichero debe existir
w	Crea un fichero de texto para escritura. Si el fichero existe, la información en él contenida se destruye.
a	Abre un fichero de texto para añadir nuevos datos. Si el fichero no existe , se crea
rb	Abre un fichero binario para lectura. El fichero debe existir
wb	Crea un fichero binario par escritura. Si el fichero existe, la información en él contenida se destruye.
ab	Abre un fichero binario para añadir nuevos datos. Si el fichero no existe, se crea
r+	Abre un fichero de texto para lectura / escritura
w+	Crea un fichero de texto para lectura / escritura. Si el fichero existe se destruye
a+	Abre un fichero de texto para lectura / escritura. Si el fichero no existe, se crea
rb+	Abre un fichero binario para lectura / escritura
rw+	Crea un fichero binario para lectura / escritura
ra+	Abre un fichero binario para lectura / escritura

» Función de cierre.

La función completa es la siguiente:

```
int fclose (FILE *Nombre_fichero);
```

Como podemos observar el argumento de la función es un puntero de tipo FILE, correspondiente al nombre lógico del fichero. Esta función devuelve dos posibles valores, un 0 si la operación finaliza con éxito o EOF si se ha generado algún error.

La finalidad de esta función es, por tanto, cerrar el fichero (flujo de datos) abierto con *fopen* ().

Control de final de fichero.

Cuando en C hablamos de EOF (End Of File, final de fichero), hacemos referencia a una constante definida en el fichero cabecera "stdio.h" con valor -1.

En la práctica, EOF es un carácter especial definido en la tabla de códigos ASCII con valor decimal 26 utilizado por la mayor parte de los editores de texto a continuación del último registro o carácter almacenado en un fichero.

Ahora bien, para aquellos casos en los que no manejamos ficheros de texto, sino ficheros binarios, el uso de la constante EOF ya no es válido para localizar o determinar dónde se encuentra el final del fichero, ya que el valor de esta constante puede ser considerado como parte integrante de la información almacenada en el fichero y en este caso, puede que no se encuentre al final del último registro o carácter almacenado.

Para solventar este problema, C nos proporciona una función de nombre *feof* (), válida tanto para el tratamiento de ficheros binarios como para el tratamiento de ficheros de texto y cuyo objetivo es detectar el final de un fichero.

El formato de esta función es el siguiente:

```
int feof (FILE *Nombre_fichero);
```

El argumento de esta función es un puntero de tipo FILE, correspondiente al nombre lógico del fichero. La función devuelve un valor distinto de 0 si se ha alcanzado el final del fichero, en cualquier otro caso devuelve 1.

La finalidad de esta función es facilitar la detección del final de un fichero binario o un fichero de texto.

Acceso secuencial.

Las funciones de acceso secuencial las podemos dividir en cuatro tipos, que detallamos a continuación.

» Funciones de E/S.

Tenemos dos funciones principalmente de este tipo, que se describen a continuación.

La primera función tiene la forma siguiente:

```
int fputc (int c, FILE *Nombre_fichero);
```

Esta función tiene dos argumentos: un primer argumento correspondiente al carácter que queremos escribir y un segundo argumento correspondiente a un puntero de tipo FILE que indica el nombre lógico del fichero.

La función devuelve el carácter escrito si la operación se realizó con éxito o EOF en caso contrario.

La finalidad de esta función es escribir un carácter en el fichero (flujo de datos) abierto previamente.

La segunda función se define de la siguiente forma:

```
int fgetc (FILE *Nombre_fichero);
```

El argumento de ésta es un puntero de tipo FILE que indica el nombre lógico del fichero.

La función devuelve EOF una vez almacenado el final del fichero.

La finalidad de la misma es leer un carácter del fichero abierto en modo lectura.

Existen otras dos funciones cuyo funcionamiento y finalidad son equivalentes a *fputc ()* y *fgetc ()* y son *putc ()* y *getc ()*.

» Funciones de E/S con buffer.

En el sistema de E/S con buffer, C incluye dos funciones pertenecientes al ANSI que son *fputs ()* y *fgets ()* destinadas a la lectura y escritura de cadenas de caracteres en un flujo determinado.

La definición de la primera función es:

```
int fputs ( const char *Cadena, FILE *Nombre_fichero);
```

Esta función tiene dos argumentos como podemos observar: un primer parámetro que es una cadena de caracteres y un segundo parámetro que es un puntero de tipo FILE que indica el nombre lógico del fichero.

Si la función se ha ejecutado con éxito devuelve el último carácter escrito y en caso de producirse un error devuelve EOF.

La finalidad de dicha función es escribir la cadena de caracteres especificada como primer argumento, en un flujo o corriente de texto especificado como segundo argumento de la misma.

Esta función se caracteriza porque el indicador nulo ('\0') de 'Cadena', es convertido a salto de línea (LF) en el momento de ser escrita la cadena de caracteres en cuestión, sobre el flujo de datos indicado como segundo argumento de la función. Esta conversión se da exclusivamente si el modo de apertura es 'w' y no 'wb'.

La definición de la segunda función es:

```
char *fgets (char *Cadena, int Longitud, FILE *Nombre_ficheros);
```

Como puede observarse esta función tiene varios argumentos: un primer parámetro que es una cadena de caracteres o puntero a cadena de caracteres, un segundo parámetro que indica el número de caracteres que van a ser leídos del flujo de datos y un tercer parámetro que es un puntero de tipo FILE que indica el nombre lógico del fichero.

Si la función se ha ejecutado con éxito devuelve un puntero a 'Cadena' y en el caso de producirse un error o alcanzar el final de fichero devuelve NULL.

La finalidad de la función es leer una cadena de la longitud especificada de la corriente o flujo de texto indicado.

Esta función lee una cadena de caracteres hasta la lectura del carácter de salto de línea (LF) o hasta leer Num – 1 caracteres del flujo de datos especificado, almacenándolos en la cadena de caracteres apuntada por 'Cadena'. Una vez leído el número de caracteres especificados, se añade automáticamente al final de la cadena el terminador o carácter nulo ('\0').

» **Funciones para E/S formateada.**

Disponemos de dos funciones pertenecientes al ANSI C, que nos permiten formatear la E/S de la información. Estas funciones reciben el nombre de *fprintf ()* y *fscanf ()* y se comportan de igual manera que *printf ()* y *scanf ()*.

Mientras que *printf ()* y *scanf ()* permitían dar formato a la información mostrada o recogida de los dispositivos de E/S estándar (pantalla y teclado), estas últimas, *fprintf ()* y *fscanf ()*, están destinadas al tratamiento de flujos de datos (ficheros).

La definición de estas funciones es:

```
int fprintf(FILE *Nombre_fichero, const char *Cadena_formato,...)
```

```
int fscanf(FILE *Nombre_fichero, const char *Cadena_formato,...)
```

» **Funciones para la E/S de bloques de datos.**

El sistema de ficheros de ANSI C proporciona dos funciones para la escritura o la lectura de bloques de datos.

La definición de la primera función es:

```
size_t fread(void *Buffer, int Num_bytes, int Num, FILE *Nombre_ficheros);
```

La finalidad de esta función es leer del flujo de datos especificado, almacenando los caracteres leídos en la parcela de memoria apuntada por 'Buffer'.

La otra función es:

```
size_t fwrite(const void *Buffer, int um_bytes, int Num, FILE *Nombre_fichero);
```

La finalidad de esta función es escribir un número 'Num' de elementos de tamaño 'Num_bytes' de la parcela de memoria apuntada por 'Buffer'.

A continuación vamos a describir los parámetros de estas funciones:

- » Buffer: Es un puntero que señala a la parcela o región de memoria utilizada para la lectura / escritura de los datos.
- » Num_bytes: Indica el número de bytes que se van a leer o escribir.
- » Num: Determina el número de elementos, siendo cada uno de tamaño o longitud 'Num_bytes' que se van a leer o escribir.
- » Nombre_fichero: Puntero de tipo FILE, correspondiente al nombre lógico del fichero.

Una de las principales aplicaciones de las funciones *fread ()* y *fwrite ()* es la de leer y escribir datos estructurados, como por ejemplo, estructuras (registros), tablas, etc.

Acceso directo.

C proporciona dos funciones pertenecientes al ANSI que nos permiten un acceso directo a los datos almacenados en un fichero, proporcionándonos una ventaja frente a otros lenguajes de programación, que es el desplazamiento byte a byte a lo largo del fichero. Estas dos funciones reciben el nombre de *fseek ()* y *ftell ()*.

La definición de la primera función es:

```
int fseek (FILE *Nombre_fichero, lont Num_bytes, int Origen);
```

Esta función devuelve 0 si se ejecuta correctamente y un valor distinto de 0 si se produce algún error.

La finalidad de esta función es permitir el acceso directamente a una posición dentro del fichero partiendo de una posición origen.

Los parámetros de la función son:

- » Nombre_fichero: Es el nombre de la corriente o flujo de datos sobre el que queremos realizar un acceso directo.
- » Num_bytes: Es un valor numérico entero que indica el número de bytes que nos queremos desplazar a partir del origen, permitiéndonos acceder a una nueva posición directamente.
- » Origen: Es una macro definida en el fichero cabecera "stdio.h", que marca el lugar o punto dentro del fichero, desde el que se efectúa el desplazamiento. Esta macro toma tres posibles valores que son los mostrados a continuación.

NOMBRE DE LA MACRO	ORIGEN	VALOR EN "STDIO.H"
SEEK_SET	Principio del fichero	0
SEEK_CUR	Posición actual	1
SEEK_END	Final del fichero	2

No es recomendable el uso de esta función con ficheros de texto, ya que puede provocar un mal funcionamiento o resultados no esperados debido a las conversiones de

caracteres lo que puede originar errores de localización. El uso de esta función está recomendado sólo para ficheros binarios.

La otra función se define como:

```
int ftell (FILE *Nombre_fichero);
```

Esta función recibe un único parámetro que es un puntero de tipo FILE que indica el nombre lógico del fichero.

Esta función devuelve el valor actual del indicador de posición del flujo o fichero especificado. Este valor se corresponde con el número de bytes que hay desde el principio del fichero hasta el indicador de posición. En caso de producirse algún error, la función devuelve -1L.

Al devolvernos el valor actual del indicador de posición, nos permite en todo momento conocer nuestra situación exacta dentro del fichero.

EJERCICIOS

EJERCICIOS CON ESTRUCTURAS DE CONTROL

EJERCICIO 1.

Diseñar los siguientes algoritmos:

- » **Un cliente realiza un pedido a una fabrica. La fabrica examina la información que tiene sobre el cliente y solo cuando es solvente se acepta y cursa el pedido.**

SOLUCIÓN

El algoritmo es el siguiente:

ENTRADA: pedido a la fabrica.

Identificar cliente que pide: leer pedido, extraer nombre cliente.

Consultar base de datos de la fabrica: buscar registro sobre cliente, buscar campo solvente, si o no.

Si cliente solvente entontes ACEPTAR

Sino hacer NO ACEPTAR.

- » **Sumar todos los números del 1 al 1000.**

SOLUCIÓN

El algoritmo que resuelve este ejercicio es el siguiente:

ENTRADA: no existe

Inicializamos a y b a cero.

Mientras a sea menor o igual que 1000 hacer {incrementar a en 1 y sumarle a b el valor de a.

Imprimir por pantalla el valor de b.

- » **Identificar si un número es primo.**

SOLUCIÓN

El algoritmo correspondiente es:

ENTRADA: N

Inicializar D a 2.

Mientras el resto de D/N sea distinto de cero se incrementa D en 1.

Si N es igual que D imprimir SI

Sino imprimir NO.

» **Sumar todos los primos menores que 1000.**

SOLUCION

El algoritmo que resuelve este problema es:

ENTRADA: no existe.

Inicializar N a 1 y resultado a cero.

Mientras N este entre 1 y 1000 si N es primo se añade a resultado y se incrementa N en 1.

Imprimir resultado.

» **Identificar potencias de siete.**

SOLUCION

El algoritmo correspondiente a este ejercicio es:

ENTRADA: N

DIVIDENDO inicializar a N: DIVIDENDO/7 pasará a ser el COCIENTE y DIVIDENDO %7 pasará a ser el RESTO.

Mientras RESTO = 0 y ACABAR = 0 hacer:

Si COCIENTE = 7 entonces S=1 y ACABAR = 1.

Sino COCIENTE pasará a DIVIDENDO, DIVIDENDO/7 pasará a COCIENTE y DIVIDENDO %7 pasará a RESTO.

Si RESTO distinto de 0 entonces ACABAR = 0.

Si ACABAR = 1 imprimir SI

Sino imprimir No.

EJERCICIO 2.

Escribe un programa que lea un número entero positivo menor que 10 e indique si es par o impar.

SOLUCIÓN

Una alternativa para realizar este programa sería la siguiente: leemos un número, si este es mayor que 10 emite un mensaje de error, y si es menor que 10 determina si es par o impar. El algoritmo sería el siguiente:

```
#include<stdio.h>
main()
{
    int numero;
    printf("Introducir un número:\n");
    scanf("%d",&numero);
    if (numero<10)
    {
        if (numero%2==0)    /*el cero lo consideramos como un
número par*/
        {
            printf("%d es menor que 10 y par",numero);
        }
        else
            printf("%d es menor que 10 e impar",numero);
    }
    else
        printf("%d es mayor que 10);
}
```

Otra posibilidad de implementar este programa sería pidiendo un número mientras este fuera mayor que 10. En este caso el algoritmo sería el siguiente:

```
# include<stdio.h>
main()
{
    int numero;
    printf ("Introducir un numero: \n");
    scanf ("%d",&numero);
    while (numero>10)
    {
        printf("Error:el numero es mayor que 10 \n Introducir
un nuevo numero");
        scanf("%d",&numero);
    }
    if (numero%2==0)    /*el cero lo consideramos como un número
par*/
        printf("%d es menor que 10 y par",numero);
    else
        printf("%d es menor que 10 e impar",numero);
}
```

Como puede observarse con el while nos aseguramos que cuando salimos de él el número va a ser menor que 10, puesto que si no ocurre esto el programa estará pidiendo números de entrada repetidamente.

EJERCICIO 3.

Implementar un programa que dados tres números los ordene de mayor a menor mostrando la ordenación por pantalla.

SOLUCIÓN:

Este programa al igual que la mayoría se pueden resolver de varias formas aquí explicamos una posible cuyo algoritmo será:

```
# include<stdio.h>
main( )
{
    int primero,segundo,tercero, auxiliar;
    printf("Introducir el primer numero");
    scanf("%d",&primero);
    printf("Introducir el segundo numero");
    scanf("%d",&segundo);
    if (segundo>primero)
    {
        auxiliar=primero;
        primero=segundo;
        segundo=auxiliar;
    }
    printf("Introducir el tercer número: \n");
    scanf("%d",&tercero);
    if (tercero>primero);
    {
        auxiliar=tercero;
        tercero=segundo;
        segundo=primero;
        primero=auxiliar;
    }
    if (tercero>segundo)
    {
        auxiliar=tercero;
        tercero=segundo;
        segundo=auxiliar;
    }
    printf("La ordenación es:\n
    mayor: %d \n
    mediado: %d \n
    menor: %d \n",primero,segundo,tercero);
}
```

EJERCICIO 4.

Implementar un programa que dado un número, escriba por pantalla todos números menores que él y que sean impares.

SOLUCIÓN:

El algoritmo que realizaría este programa será el siguiente:

```
# include<stdio.h>
main( )
{
    int numero,contador;
    printf("Introducir un número:\n");
    scanf("%d",&numero);
    contador=1;
    while (contador<numero)
    {
        if (contador%2!=0)
            printf("%d es menor que %d e impar",contador,numero);

        contador=contador+1;
    }
}
```

Como puede observarse este algoritmo utilizamos dos variables, pero podríamos hacerlo solamente con una mediante un for de la forma siguiente:

```
# include<stdio.h>
main( )
{
    int numero;
    printf("Introducir un número:\n");
    scanf("%d",&numero);

    for( numero--;numero;numero--)
    {
        if (numero%2!=0)
            printf("%d es impar",numero);
    }
}
```

EJERCICIO 5.

Escribe un programa que permita visualizar en pantalla los divisores de un número dado.

SOLUCIÓN:

Para determinar los divisores de un número debemos dividir dicho número por todos los valores menores que él de manera que si esta división es entera el número es divisible por el número que lo estamos dividiendo. El algoritmo será por tanto el siguiente:

```
# include<studio.h>
main( )
{
    int numero,contador;
    printf("Introducir un numero");
    scanf("%d",&numero);
    contador=numero;
    while (contador>0)
    {
        if (numero%contador==0)
            printf("%d es un divisor de
%d",contador,numero);
        contador=contador-1;
    }
}
```

EJERCICIO 6.

Implementar un programa que convierta grados Fahrenheit a grados Centígrados y viceversa.

SOLUCIÓN:

Este programa se puede implementar con el siguiente algoritmo:

```
# include<stdio.h>
main ( )
{
    float  valor,resultado;
    char grados;
    printf("¿Grados que queremos obtener(C/F)?");
    scanf("%c",&grados);
    if (grados=='F')
    {
        printf("Introducir valor de los °C:\n");
        scanf("%f",&valor);
        resultado=(9/5)*valor+32;
        printf("%f °C son %f °F",valor,resultado);
    }
    else
    {
        printf("Introducir valor de los °F:\n");
        scanf("%f",&valor);
        resultado= (5/9)*(valor-32);
        printf("%f °F son %f °C",valor,resultado);
    }
}
```

Como puede observarse en este algoritmo controlamos si se trata de una conversión °C a °F ó °F a °C mediante un if.

EJERCICIO 7.

Implementar programa que escriba en pantalla la tabla de multiplicar, hasta el 10, de un número dado por pantalla.

SOLUCIÓN:

El algoritmo que implementaría este programa sería el siguiente:

```
# include<stdio.h>
main( )
{
    int numero,contador;
    printf ("Introducir un número:\n");
    scanf("%d",&numero);
    for (contador=0;contador<11;contador++)
        printf("%d * %d = %d", numero,contador,numero*contador);
}
```

EJERCICIO 8.

Implementar un programa que nos muestre la cifra mayor dentro de un número entero positivo dado por teclado.

SOLUCIÓN:

Para resolver este problema lo que debemos de hacer es extraer cada una de la cifras del número. Para ello dividiremos por 10 de manera que el resto de dicha división será una cifra del número. El algoritmo que implementaría este problema sería el siguiente:

```
# include<stdio.h>
main( )
{
    int numero, maximo,b;
    printf ("Introducir un número:\n");
    scanf ("%d",&numero);
    b=numero;    /*esta variable solo sirve para guardar el valor
inicial para luego mostrarlo en el printf final*/
    maximo=0;
    while (numero>0)
    {
        if ((numero%10>maximo)
        {
            máximo=numero%10;
            numero=numero/10;
        }
    }
    printf("La cifra mayor del número %d es %d",b,máximo);
}
```

Otro algoritmo que podríamos utilizar para resolver este problema sería el siguiente:

```
# include<stdio.h>
main( )
{
    int numero, maximo,b;
    printf ("Introducir un número:\n");
    scanf("%d",&numero);
    b=numero;    /*esta variable solo sirve para guardar el valor
inicial para luego mostrarlo en el printf final*/
    maximo=0;
    while (numero/10>0)
    {
        if ((numero%10>maximo)
        {
            máximo=numero%10;
            numero=numero/10;
        }
    }
    if (numero>maximo)
        maximo=numero;
    printf("La cifra mayor del número %d es %d",b,maximo);
}
```

EJERCICIO 9.

Realizar un programa que determine si un número introducido por pantalla es primo o no.

SOLUCIÓN:

Para determinar si un número es primo o no es primo lo que debemos de hacer es dividir dicho número por todos los menores que él de forma que si el resto de todas estas divisiones es únicamente cero cuando dividimos por si mismo y por la unidad. Es que el número será primo. Puesto que todos los números son divisible por la unidad esta posibilidad la excluimos de nuestro algoritmo.

```
# include<stdio.h>
main( )
{
    int numero,contador;
    contador=2;
    printf("Introducir el numero para determinar si es no
primo:\n");
    scanf("%d",&numero);
    while (numero%contador!=0)
        contador=contador+1;
    if (contador==numero)
        printf("El numero %d es primo",numero);
    else
        printf("El numero %d no es primo",numero);
}
```

EJERCICIO 10.

Implementar un programa que nos dé el desglose óptimo (mínimo número de monedas posibles) de una cantidad entera positiva en monedas de 100, 50, 25, 5 y 1 pesetas.

SOLUCIÓN:

Para resolver este problema lo que debemos hacer es restar primero a la cantidad introducida el mayor número de veces 100 para determinarnos la cantidad de monedas de 100 ptas que debe devolver, luego al dinero restante restarle 50 para obtener el número de monedas de 50 a devolver así sucesivamente con 25 y 5 ptas y el resto serán monedas de 1 ptas. Este procedimiento queda demostrado en el siguiente algoritmo:

```
# include<stdio.h>
main( )
{
    int                cantidad,monedas_100,monedas_50,monedas_25,
modenas_5,monedas_1;
    printf(" Introducir la cantidad a desglosar:\n");
    scanf("%d",&cantidad);
    monedas_100=monedas_50=monedas_25=monedas_5=monedas_1=0;
    while (cantidad>=100)
    {
        monedad_100+=1;
        cantidad - =100;
    }
    if (cantidad>=50)        /* cantidad < 100*/
    {
        monedas_50=1;
        cantidad - =50;
    }
    if (cantidad>=25)        /* cantidad <50*/
    {
        monedas_25=1;
        cantidad - =25;
    }
    while (cantidad>=5)        /* cantidad < 25*/
    {
        monedas_5+=1;
        cantidad - =5;
    }
    monedas_1=cantidad;        /* cantidad < 5*/

    printf("El desglose de la  cantidad introducida es: \n
        monedas 100 ptas: %d \n
        monedas 50 ptas: %d \n
        monedas 25 ptas:%d \n
        monedas 5 ptas:%d \n
        monedas 1 pta:%d \n",monedas_100, monedas_50, monedas_25,
monedas_5, monedas_1);
}
```

EJERCICIO 11.

Implementar un programa que nos dé como resultado el factorial de un número entero positivo dado por pantalla.

SOLUCIÓN:

El factorial de un número n se puede definir como sigue: $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$ de tal forma que este programa tiene dos alternativas, definirlo de forma estructurada o realizarlo con una función recursiva. A continuación presentamos las dos formas:

Forma estructurada:

```
# include <stdio.h>
main( )
{
    int numero,factorial,b;
    factorial=1;
    b=numero;
    printf("  Introducir el número cuyo factorial hay que
calcular:|n");
    scanf("%d",&numero);
    for( ;numero, numero--)
        fact*=numero;
    printf( "%d!=%d",b,factorial);
}
```

Forma recursiva:

Para realizar una definición de forma recursiva no emplearemos un programa norma de C sino que lo realizaremos mediante una función. La función que nos permite calcular el factorial de un número será:

```
int factorial (int numero)
{
    if (numero==2)
        return 2;
    else
        return numero*factorial(numero-1);
}
```

EJERCICIO 12.

Implementar un programa que determine cuantas cifras posee un número entero positivo introducido por pantalla.

SOLUCIÓN:

Para determinar las cifras que tiene un número lo que haremos es dividir sucesivamente dicho número por 10 hasta que no podamos hacerlo más veces de forma que contando el número de divisiones realizadas sabremos el número de cifras de dicho número. El algoritmo de este programa será por tanto:

```
# include <stdio.h>
main( )
{
    int numero,b,cifras=0;
    printf (" Introducir un numero");
    scanf("%d",&numero);
    b=numero;                               /*b simplemente nos sirve para
    guardar el valor inicial, que presentaremos posteriormente en el
    printf final*/
    while (numero>0)
    {
        cifras+=1;
        numero/=10;
    }
    printf ("El número %d tiene %d cifras",b,cifras);
}
```

EJERCICIO 13.

Un deposito contiene n litros de vino, introducido por pantalla. De él se extrae cada año un litro de vino y se añade 1 litro de agua y así se sucederá cada año. Implementar un programa que nos indique cuantos años tardará en igualarse la disolución y la proporción de la misma durante esos años.

SOLUCIÓN:

El algoritmo que realiza este programa será el que se presenta a continuación:

```
# include <stdio.h>
main ( )
{
    int ,n años,cont=0;    /*hemos restringido los litros que debe
    tener el deposito a un número entero*/

    printf ("Introducir el número de litros que tiene el depósito:
    \n");
    scanf("%d",&n);
    if (n%2!=0)
        printf(" No se llegará nunca a una solución con igual
    proporción de agua que de vino");    /*puesto que cada año se extraerán
    o añadirán litros enteros y no cantidades menores que este litro*/
    else
    {
        while (cont>n/2)
        {
            años+=1;
            contador+=1;
        }
        printf(" Tardará en alcanzar estas proporciones %d
    años",años
    )
    }
}
```

EJERCICIO 14.

Implementar un programa que liste los números primos menores e iguales que un número entero positivo dado por pantalla.

SOLUCIÓN:

El algoritmo que realizará esta función se presenta a continuación:

```
# include<stdio.h>
main ( )
{
    int numero, divisor;
    printf("Introducir un número:\n");
    scanf("%d",&numero);
    if (numero==1)
        printf("El 1 es un número primo");
    else
    {
        for( ;numero>1;numero--)
        {
            divisor=2;
            while (numero%divisor!=0)
                divisor+=1;
            if (numero==divisor)
                printf (" %d es un número primo",numero);
        }
        printf ("El 1 es primo");
    }
}
```

EJERCICIO 15.

Implementar un programa que introduciendo los 2 primeros caracteres de un día de la semana escriba dicho día completo.

SOLUCIÓN:

El algoritmo que implementa este programa será el siguiente:

```
# include<stdio.h>
main( )
{
    char dia[2];
    printf ("Introducir dos caracteres correspondientes a la
iniciales de un día de la semana: \n");
    scanf("%s",&dia);
    switch (dia)
    {
        case "Lu":
            printf ("Lunes"); break;
        case "Ma":
            printf ("Martes");
            break;
        case "Mi":
            printf ("Miercoles");
            break;
        case "Ju":
            printf ("Jueves");
            break;
        case "Vi":
            printf ("Viernes");
            break;
        case "Sa":
            printf ("Sabado");
            break;
        case "Do":
            printf ("Domingo");
            break;
        case default:
            printf (" No se corresponde con ningún día de la
semana");
    }
}
```

EJERCICIO 16.

Programa que escribe la suma de una serie de números recogidos del dispositivo estándar de entrada (teclado). La entrada de datos finaliza al evaluar la respuesta dada a un mensaje que diga "¿Continuar (S/N)?" mostrado una vez finalizadas las operaciones del bucle.

SOLUCION

El algoritmo que resuelve este problema es el siguiente:

```
# include <stdio.h>
main ()
{
    int num,suma=0;
    char resp;
    do
    {
        scanf("%d",&num);
        suma += num;
        printf("¿Continuar (S/N)?");
        while(getchar()!='\n');
        resp=getchar();
    }
    while(resp=='S');
    printf("La suma es: %d\n",suma);
}
```

EJERCICIO 17.

Hacer un programa que lea una serie de números enteros positivos de la entrada estándar y calcule el valor máximo de los mismos y cuántas veces aparece dicho valor repetido.

SOLUCION

El algoritmo que da solución a este ejercicio es el siguiente:

```
# include <stdio.h>
main()
{
    int numero, maximo, cantidad;
    printf("\nIntroduce un número:");
    scanf("%d",&maximo);
    cantidad=1;
    do
    {
        printf("Introduce otro número:");
        scanf("%d",&maximo);
        if(numero>maximo)
        {
            maximo=numero;
            cantidad=1;
        }
        else
        {
            if(numero==maximo)
                cantidad++;
        }
    }
    while(numero!=0);
    printf("El valor máximo es %d con %d repeticiones.",maximo,
cantidad);
}
```

EJERCICIOS CON ARRAYS

EJERCICIO 18.

Hacer un programa que:

- (a) Cree un array unidimensional de 20 elementos de tipo numérico entero y nombre 'numeros'.
- (b) Cargue el array con valores negativos, positivos y ceros.
- (c) Contabilice el número de valores positivos, negativos y ceros almacenados en el proceso de carga.
- (d) Muestre en pantalla los resultados obtenidos.

SOLUCION

EL algoritmo que implementa esta tarea es:

```
# include <stdio.h>
main()
{
    int numeros[20];
    int j;
    int pos,neg,cero;
    for(j=0;j<20;j++)
        scanf("%d",&numeros[j]);
    pos=neg=cero=0;
    for(j=0;j<20;j++)
    {
        if(numeros[j]<0)
            neg++;
        else if(numeros[j]>0)
            pos++;
        else
            cero++;
    }
    printf ("Hay %d números positivos, %d negativos y %d
ceros",pos,neg,cero);
}
```

EJERCICIO 19.

Hacen en C un programa que:

- (a) Cree un array unidimensional de 15 elementos de tipo numérico entero y nombre 'tabla'.
- (b) Cargue el array rellenándolo con valores enteros positivos y negativos.
- (c) Calcule en número de valores que sean pares y positivos presentando el resultado por pantalla.

SOLUCION

El algoritmo que nos resuelve este ejercicio es el que se presenta a continuación:

```
# include <stdio.h>
main ()
{
    int tabla[15];
    int j, contador=0;
    for (j=0;j<15;j++)
        scanf("%d",&tabla[j]);
    for (j=0;j<15;j++)
    {
        if (tabla[j]%2==0 && tabla[j]>0)
            contador++;
    }
    printf("El total de números pares positivos es
%d\n",contador);
}
```

EJERCICIO 20.

Hacer un programa que:

- (a) Cree un array unidimensional de 20 elementos de tipo numérico entero y nombre 'array'.**
- (b) Cargue el array con valores aleatorios.**
- (c) Muestre en pantalla todos aquellos elementos mayores de 30 junto con la posición que ocupan en la tabla. En caso de no existir ninguno, se mostrará el siguiente mensaje: *"No hay números mayores que 30"*.**

SOLUCION

El algoritmo que resuelve nuestro ejercicio es el siguiente:

```
# include <stdio.h>
main ()
{
    int array[20];
    int j;
    int sw=0;
    printf ("Introduce los valores que cargan la tabla.\n");
    for(j=0;j<20;j++)
        scanf("%d",&array[j]);
    printf("\nLos elementos mayores que 30 son:\n");
    for(j=0;j<20;j++)
    {
        if(array[j]>30)
        {
            sw=1;
            printf("Array[%d]=%d\n,j,array[j]);
        }
    }
    if(!sw)
        printf ("no hay números mayores que 30\n");
}
```

EJERCICIO 21.

Hacer un programa que:

- (a) Lea una secuencia de 15 números enteros, almacenándolos en un array de nombre 'numeros'.**
- (b) Los visualice en el dispositivo estándar de salida en orden inverso al de entrada.**

SOLUCION

El algoritmo que resuelve este programa es el siguiente:

```
# include <stdio.h>
main ()
{
    int numeros [15];
    int j;
    for (j=0;j<15;j++)
        scanf("%d",&numeros[j]);
    putchar('\n');
    for(j=14;j>=0;j--)
        printf("%d",numeros[j]);
}
```

EJERCICIO 22.

Hacer un programa que:

- (a) Lea una secuencia de 20 valores numéricos reales y los almacene en un array unidimensional de nombre 'numero'.**
- (b) Calcule cuál es el valor máximo, así como la posición que ocupa en la tabla.**
- (c) Muestre el valor máximo encontrado junto con la posición que ocupa en el array. En caso de aparecer repetido el valor máximo se mostrará el que ocupa la posición situada más a la izquierda.**

SOLUCION

El algoritmo que soluciona este ejercicio es el mostrado a continuación:

```
# include <stdio.h>
main ()
{
    float numero[20];
    int k, p;
    float maximo;
    for (k=0;k<20;k++)
    {
        printf("El valor de la componente [%d]:",k);
        scanf("%f",&numero[k]);
    }
    máximo=numero[0];
    p=0;
    for (k=1;k<20;k++)
    {
        if (maximo<numero[k])
        {
            máximo=numero[k];
            p=k;
        }
    }
    printf ("\nEl valor máximo es (numro[%d]=%f)\n",p,máximo);
}
```

EJERCICIO 23.

Hacer un programa que:

- (a) Cree un array bidimensional de 10x10 y nombre 'datos'.**
- (b) Inicialice el array de forma que cada componente o elemento guarde el número de fila al que pertenece.**
- (c) Muestre el contenido del mismo en pantalla.**

SOLUCION

EL algoritmo que da solución a este ejercicio es el siguiente:

```
# include <stdio.h>
main ()
{
    int datos[10][10];
    int f,c;
    for (f=0;f<10;f++)
    {
        for (c=0;c<10,c++)
            datos[f][c]=f;
    }
    for (f=0;f<10;f++)
    {
        for (c=0;c<10,c++)
            printf("%d", datos[f][c]);
        putchar('\n');
    }
}
```

EJERCICIO 24.

Hacer un programa que:

- (a) Cree un array bidimensional de longitud 5x5 y nombre 'diagonal'.**
- (b) Inicialice el array, de forma que los componentes pertenecientes a la diagonal de la matriz tomen valor uno y el resto valor cero.**
- (c) Muestre el contenido del array en pantalla.**

SOLUCION

EL algoritmo que resuelve este ejercicio se muestra seguidamente:

```
# include <stdio.h>
main()
{
    int diagonal[5][5];
    int f,c;
    for (f=0;f<5;f++)
    {
        for (c=0;c<5;c++)
            diagonal[f][c]=0;
    }
    for (f=0,c=4;f<5;f++)
        diagonal[f][c--]=1;
    for (f=0;f<5;f++)
    {
        for (c=0;c<5;c++)
            printf("%d",diagonal[f][c]);
        putchar('\n');
    }
}
```

EJERCICIO 25.

Hacer un programa que:

- (a) Cree un array bidimensional de longitud 10x10 y monbre 'matriz'.**
- (b) Cargue el array con valores aleatorios introducidos a través del dispositivo estándar de entrada (teclado).**
- (c) Muestre el contenido del array por pantalla.**
- (d) Sume todos los elementos de cada fila y todos los elementos de cada columna visualizando el resultado en pantalla.**

SOLUCION

El algoritmo correspondiente es:

```
# include <stdio.h>
main()
{
    int matriz[10][10];
    int f,c;
    int suma_c=0, suma_f=0;
    for (f=0;f<10;f++)
    {
        for(c=0;c<10;c++)
            scanf("%d",&matriz[f][c]);
    }
    for (f=0;f<10;f++)
    {
        for(c=0;c<10;c++)
            printf("%d",matriz[f][c]);
        putchar('\n');
    }
    for (f=0;f<10;f++)
    {
        for(c=0;c<10;c++)
            suma_c +=matriz[f][c];
        printf("Suma de las columnas de la fila
%d:%d\n",f,suma_c);
        suma_c=0;
    }
    putchar('\n');
```

```
    for (c=0;c<10;c++)
    {
        for(f=0;f<10;f++)
            suma_f +=matriz[f][c];
        printf("Suma de las filas de columna
%d:%d\n",c,suma_f);
        suma_f=0;
    }
}
```

EJERCICIO 26.

Hacer un programa que:

- (a) Cree un array bidimensional de longitud 5x5 y nombre 'suma'.**
- (b) Inicialice el array de tal forma que el valor de cada elemento sea la suma del número de su fila y del número de su columna.**
- (c) Muestre el contenido del array en pantalla.**

SOLUCION

El algoritmo que cumple las especificaciones del enunciado es:

```
# include <stdio.h>
main()
{
    int suma[5][5];
    int f,c;
    for (f=0;f<5;f++)
    {
        for (c=0;c<5;c++)
            suma[f][c]=f+c;
    }
    for (f=0;f<5;f++)
    {
        for (c=0;c<5;c++)
            printf("%d",suma[f][c]);
        putchar('\n');
    }
}
```

EJERCICIOS CON CADENAS DE CARACTERES

EJERCICIO 27.

Crear una función que permita obtener la parte izquierda de una cadena de caracteres definida por el número de caracteres que la forman.

SOLUCION

El código de esta función se presenta a continuación:

```
# include <stdio.h>
char *izquierda ();
main()
{
    char cadena[]="IMPORTE TOTAL";
    printf ("%s\n",izquierda(cadena,7));
}
char *izquierda(char *cadena, int num_car)
{
    char sub_cad[80];
    int n=0;
    while(n<num_car)
    {
        sub_cad[n]=cadena[n];
        n++;
    }
    sub_cad[n]='\0';
    return(sub_cad);
}
```

EJERCICIO 28.

Crear una función que permita obtener la parte derecha de una cadena de caracteres definida por el número de caracteres que la forman.

SOLUCION

El código de esta función es:

```
# include <stdio.h>
char *derecha();
main()
{
    char cadena[]="IMPORTE TOTAL";
    printf ("%s\n",derecha(cadena,5));
}
char *derecha (char *cadena, int num_car)
{
    char sub_cad[80];
    int n=0;
    while(n<num_car)
    {
        sub_cad[n]=cadena[strlen(cadena)-num_car+n];
        n++;
    }
    sub_cad[n]='\0';
    return(sub_cad);
}
```

Otra solución sería el siguiente:

```
# include <stdio.h>
char *derecha ();
main();
{
    char cadena[]="IMPORTE TOTAL";
    printf ("%s\n",derecha(cadena,5));
}
char *derecha(char *cadena, int num_car)
{
```

```
char sub_cad[80];
int n=num_car;
int c=strlen(cadena)-1;
sub_cad[n--]='\0';
while (n<=0)
{
    sub_cad[n]=cadena[c];
    n--;
    c--;
}
return (sub_cad);
}
```

EJERCICIO 29.

Crear una función que determine cuál es el centro de una cadena y devuelva la mitad derecha de la misma.

SOLUCION

EL algoritmo que presenta esta función es:

```
# include <stdio.h>
char *centro();
main()
{
    printf("/%s/\n",centro ("IMPORTE TOTAL"));
}
char *centro (char *cadena)
{
    char sub_cad[80];
    int cen, n=0;
    cen=strlen(cadena)/2;
    while (n<=cen)
    {
        sub_cad[n]=cadena[cen+1+n];
        n++;
    }
    sub_cad[n]='\0';
    return(sub_cad);
}
```

EJERCICIO 30.

Crear una función que limpie una cadena de caracteres de espacios en blanco o caracteres de cambio e línea.

SOLUCION

El código de esta función es el siguiente:

```
# include <stdio.h>
char *limpiar_espacios();
main ()
{
    char cadena[]="IMPORTE TOTAL";
    printf("/%s/\n",limpiar_espacios(cadena));
}
char *limpiar_espacios(char *cadena)
{
    char sub_cad[80];
    int i,j;
    i=j=0;
    while(i<=strlen(cadena))
    {
        if(cadena[i]!=' ' && cadena[i]!='\n')
            sub_cad[j++]=cadena[i++];
        else
            i++;
    }
    for(i=0;i<=strlen(sub_cad);i++)
        cadena[i]=sub_cad[i];
    return cadena;
}
```

EJERCICIO 31.

Crear una función que concatena dos cadenzas (cadena 1 y cadena 2) dejando el resultado de dicha operación sobre la primera (cadena 1).

SOLUCION

Esta función se puede determinar de dos formas la primera es la siguiente:

```
# include <stdio.h>
char *concatenar ();
main ()
{
    char cadena1[]="IMPORTE";
    char cadena2[]="TOTAL";
    concatenar(cadena1,cadena2);
    printf("[%s]\n",cadena1);
}
char *concatenar (char *cadena1, char *cadena2)
{
    int i,j;
    i=strlen(cadena1);
    j=0;
    while(j<=strlen(cadena2))
        cadena1[i++]=cadena2[j++];
    return cadena1;
}
```

El otro algoritmo es el siguiente:

```
# include <stdio.h>
char *concatenar ();
main ()
{
    char cadena1[]="IMPORTE";
    char cadena2[]="TOTAL";
    concatenar(cadena1,cadena2);
    printf("[%s]\n",cadena1);
}
```

```
char *concatenar (char *cadena1, char *cadena2)
{
    int i=0,j;
    while(cadena1[i]!='\0')
        i++;
    for(j=0;cadena2[j]!='\0';j++)
        cadena1[i++]=cadena2[j];
    cadena1[i]='\0';
    return cadena1;
}
```

EJERCICIO 32.

Dadas dos cadenas de caracteres (cadena1 y cadena2), crear una función que copia la segunda cadena de caracteres (cadena2) sobre la primera (cadena1).

SOLUCION

A la igual que el ejercicio anterior esta función se puede implementar de dos formas, la primera es la siguiente:

```
# include <stdio.h>
main ()
{
    char cadena1[]="IMPORTE";
    char cadena2[]="TOTAL";
    copiar(cadena1,cadena2);
    printf("[%s]\n",cadena1);
    printf("[%s]\n",cadena2);
}
char *copiar(char *cadena1, char *cadena2)
{
    int i;
    for(i=0;cadena2[i]!='\0';i++)
        cadena1[i]=cadena2[i];
    cadena[i]='\0';
    return cadena1;
}
```

El otro algoritmo es el siguiente:

```
# include <stdio.h>
char *copiar();
main()
{
    char cadena1[]="IMPORTANTE";
    cahr cadena2[]="TOTAL";
    copiar (cadena1,cadena2);
    printf("[%s]\n",cadena1);
    printf("[%s]\n",cadena2);
}
```

```
char *copiar (char cadena1[], char cadena2[])
{
    int i=0;
    while(i<=strlen(cadena2))
    {
        cadena1[i]=cadena2[i];
        i++;
    }
    return cadena1;
}
```

EJERCICIO 33.

Hacer un programa que compare dos cadenas y determine si son iguales o diferentes.

SOLUCION

Este programa tiene también dos soluciones, el primero es el siguiente:

```
# include <stdio.h>
int coincide();
main()
{
    char cadena1[]="IMPORTE";
    char cadena2[]="IMPORTE";
    char cadena3[]="IMPORT";
    if(!coincide(cadena1,cadena3))
        printf("cadena1 y cadena3 son diferentes.\n");
    else
        printf("cadena1 y cadena3 son iguales.\n");
    if(!coincide(cadena1,cadena2))
        printf("cadena1 y cadena2 son diferentes.\n");
    else
        printf("cadena1 y cadena2 son iguales.\n");
}
int coincide (char *s1, char *s2)
{
    int i=0;
    while(i<strlen(s1))
    {
        if(s1[i]!=s2[i])
            return 0;
        else
            i++;
    }
    return 1;
}
```

El otro algoritmo es el siguiente:

```
# include <stdio.h>

int coincide();

main()
{
    char cadena1[]="IMPORTE";
    char cadena2[]="IMPORTE";
    char cadena3[]="IMPORT";
    if(!coincide(cadena1,cadena3))
        printf("cadena1 y cadena3 son diferentes.\n");
    else
        printf("cadena1 y cadena3 son iguales.\n");
    if(!coincide(cadena1,cadena2))
        printf("cadena1 y cadena2 son diferentes.\n");
    else
        printf("cadena1 y cadena2 son iguales.\n");
}

int coincide (char *s1, char *s2)
{
    int i=0;
    while(s1[i]!='\0')
    {
        if(s1[i]!=s2[i])
            return 0;
        else
            i++;
    }
    return 1;
}
```

EJERCICIO 34.

Hacer un programa en el que dada dos matrices:

- (a) Un array bidimensional de longitud 6x8 de nombre 'matriculas', donde las filas representan grupos de alumnos y las columnas asignaturas.**
- (b) Un segundo array bidimensional de longitud 8x13 de nombre 'asignaturas', donde cada fila se corresponde con una asignatura.**

Muestre en pantalla una tabla con el total de alumnos matriculados por asignaturas.

SOLUCION

El algoritmo correspondiente es el siguiente:

```
# include <stdio.h>

main ()
{
    int matriculas[6][8]={45, 40, 42, 42, 44, 45, 44, 38, 50,
43, 50, 47, 46, 49, 50, 39, 35, 33, 36, 31, 34, 34, 35, 25, 43, 43,
43, 42, 44, 40, 42, 30, 55, 54, 53, 50, 54, 43, 34, 30, 32, 32, 34,
35, 34, 22}

    char asignaturas[][13]={
                                "Álgebra           ",
                                "Cálculo           ",
                                "Programación      ",
                                "Inglés            ",
                                "Estadística       ",
                                "Física            ",
                                "Historia          ",
                                "Tecnología        "};

    int f,c, suma=0;
    printf ("\nAlumnos matriculados en el curso 95/96 por
asignaturas:\n");
    printf("-----\n");
    printf("| ASIGNATURA                               N° DE MATRICULAS |\n");
    printf("-----\n");
    for(c=0;c<8;c++)
    {
        for(f=0;f<6;f++)
```



```
        suma += matriculas[f][c];  
        printf("%s %d\n", asignaturas[c], suma);  
        suma=0;  
    }  
}
```

EJERCICIOS CON PUNTEROS

EJERCICIO 35.

Escribir un programa que calcule y visualice la media aritmética de un vector de 10 elementos numéricos, utilizando una variable puntero que apunte a dicho vector.

SOLUCION

El algoritmo que implementa este programa es el siguiente:

```
# include <stdio.h>
void main (void)
{
    float med, suma=0;
    int tabla[10], j;
    int *pt=tab;
    for(j=0;j<10;j++)
    {
        printf("\nIntroducir el elemento %d: ",j+1);
        scanf("%d",pt+j);
    }
    for (j=0;j<10;j++)
        suma+=*(pt+j);
    med=suma/10;
    printf("La media vale:%f",med);
}
```

EJERCICIO 36.

Escribir un programa que ponga en mayúsculas el primer carácter de una cadena de caracteres y todo aquel carácter que siga a un punto, utilizando un puntero a dicha cadena.

SOLUCION

El algoritmo que resuelve este ejercicio es:

```
# include <stdio.h>
# include <ctype.h>
void main (void)
{
    char texto[80];
    char *ptext=texto;
    printf("\nIntroduccir un texto menor de 80 caracteres:\n");
    gets(texto);
    ptext=toupper(*ptext);
    while(*ptext!='\0')
    {
        ptext++;
        if(*ptext=='.')
            *(ptext+2)=toupper(*(ptext+2));
    }
    puts(texto);
}
```

EJERCICIO 37.

Escribir un programa que mediante el empleo de un menú realice operaciones aritméticas, llamando a las funciones correspondiente. Desarrollar este programa utilizando una tabla de punteros a funciones.

SOLUCION

El algoritmo que cumple las especificaciones del enunciado.

```
# include <stdio.h>
void main (void)
{
    void suma(float x, float y);
    void resta(float x, float y);
    void prod(float x, float y);
    void divi(float x, float y);
    void(*p[4])(float x, float y)={suma, resta, prod, divi};
    float num1, num2;
    int opcion;
    printf ("\nIntroducir dos números \n");
    scanf("%f%f",&num1,&num2);
    printf("1.-Suma\n");
    printf("2.-Resta\n");
    printf("3.-Producto\n");
    printf("4.-División\n");
    do
    {
        printf("Seleccionar una opción\n");
        scanf("%d",&opcion);
    }
    while(opcion<1 || opcion >4);
    (*p[opcion-1])(num1,num2);
}

void suma(float x, float y)
{
    printf("La suma vale:%f",x+y);
}
```

```
void resta(float x, float y)
{
    printf("La resta vale:%f",x-y);
}
void prod(float x, float y)
{
    printf("El producto vale:%f",x*y);
}
void divi(float x, float y)
{
    if(y==0)
        printf("La división no es posible. Divisor=0");
    else
        printf("La división vale:%f",x/y);
}
```

EJERCICIO 38.

Escribir un programa que llame a tres funciones de nombre 'Pepe', 'Ana' y 'Maria' mediante el uso de una tabla de punteros a funciones.

SOLUCION

El algoritmo que cumple las especificaciones del enunciado es:

```
# include <stdio.h>
int Pepe(), Ana(), Maria();
int (*mf[])()={Pepe, Ana, Maria};
void main (void)
{
    int j;
    for (j=0;j<3;j++)
        (*mf[j])();
    /*Las dos lineas anteriores pueden simplificarse en:
        for (j=0;j<3;(*mf[j++])());*/
}
Pepe()
{
    printf("soy Pepe\n");
}
Ana ()
{
    printf("soy Ana\n");
}
Maria()
{
    printf("soy Maria\n");
}
```

EJERCICIOS CON ESTRUCTURAS

EJERCICIO 39.

Programa que define una tabla de proveedores teniendo asignados cada proveedor un nombre, cantidad vendida del artículo, precio unitario (introducidos por teclado) e importe (calculado a partir de los datos anteriores). Se pretende visualizar los datos de cada proveedor, el importe total de compra, así como el nombre del proveedor más barato y el del más caro.

SOLUCION

El algoritmo que determina este programa es el siguiente:

```
# include <stdio.h>
# define NUM 3
# define DIM 16
void main (void)
{
    struct proveedor
    {
        char prov[DIM];
        int cant_p;
        float precio;
        float importe;
    };
    struct proveedor provee[NUM];
    int cont, k=0, j=0;
    float preciomín, preciomax;
    float total=0.0;
    /* Carga de la tabla de datos de proveedores*/
    for(cont=0;cont<NUM;cont++)
    {
        printf("\nIntroducir nombre del proveedor %d:",
cont+1);

        gets(provee[cont].prov);
        printf("Introducir cantidad de piezas: ");
        scanf("%d", &provee[cont].cant_p);
        printf("Introducir precio unitario:");
        scanf("%d",&provee[cont].precio);
```

```
        provee[cont].importe=provee[cont].cant_p*provee[cont]
        .precio;
        while(getchar()!='\n');
    }
    /*Visualizar datos de proveedores*/
    for(cont=0;cont<NUM;cont++)
        printf("\n%s    %s    %s    %s",    provee[cont].prov,
        provee[cont].cant_p,                provee[cont].precio,
        provee[cont].importe);
    total += provee[cont].importe;
    printf("\nEl importe total es:%f",total);
    /*Calcular el proveedor más barato y el más caro*/
    preciomin=provee[0].precio;
    preciomax=provee[0].precio;
    for(cont=1;cont<NUM;cont++)
    {
        if(preciomin>provee[cont].precio)
        {
            preciomin=provee[cont].precio;
            k=cont;
        }
        if(preciomax<provee[cont].precio)
        {
            preciomax<provee[cont].precio;
            j=cont;
        }
    }
    printf("\n    El    proveedor    mas    barato    es:    %s",
    provee[k].prov);
    printf("\n El proveedor más caro es: %s", provee[j].prov);
}
```

EJERCICIO 40.

Programa que define una tabla de proveedores empleando una estructura que anida los datos del proveedor (nombre, dirección , y teléfono), cantidad vendida, precio unitario e importe (calculado). Los datos no calculados se introducen por teclado.

Se pretende visualizar en pantalla los datos de cada proveedor, el importe total de las compras y el nombre y teléfono del proveedor más barato.

SOLUCION

El algoritmo que resuelve este problema es el siguiente:

```
# include <stdio.h>
# define NUM 3
struct dat_p
{
    char nom_p[16];
    char dir_p[30];
    char telef_p[10];
};
struct entrada
{
    struct dat_p datos;
    int cant;
    float precio;
    float importe;
};
void main (void)
{
    struct entrada sumin[NUM];
    int cont, j=0;
    float total=0.0;
    float preciomin;
    printf("\nIntroducir datos de proveedores\n");
    for(cont=0;cont<NUM;cont++)
    {
        printf("Introducir proveedor num:%d",cont+1);
        printf("\nNombre:");
        gets(sumin[cont].datos.nom_p);
```

```

        printf("\nDirección:");
        gets(sumin[cont].datos.dir_p);
        printf("\nTelefono:");
        gets(sumin[cont].datos.telef_p);
    }
    printf("\n*****");
    printf("\nIntroducir cantidades y precios de cada
proveedor");
    for(cont=0;cont<NUM;cont++);
    {
        printf("\nProveedor: %s",sumin[cont].datos.nom_p);
        printf("\nCantidad suministrada:");
        scanf("%d",&sumin[cont].cant);
        printf("Precio unitario:");
        scanf("%f",&sumin[cont].precio);
        while(getchar()!='\n');
        sumin[cont].importe=sumin[cont].cant*
*sumin[cont].precio;
    }
    printf("%s    %s    %s    %s\n","Proveedor","Cant","Precio",
"Importe");
    for(cont=0;cont<NUM;cont++)
    {
        printf ("%s  %d  %f  %f\n",sumin[cont].datos.nom_p,
sumin[cont].cant, sumin[cont].precio, sumin[cont].importe);
        total += sumin[cont].importe;
    }
    printf("\nEl importe total es:%f", total);
    preciomin=sumin[0].precio;
    for(cont=0;cont<NUM;cont++)
    {
        if (preciomin>sumin[cont].precio)
        {
            preciomin=sumin[cont].precio;
            j=cont;
        }
    }
    printf("\nEL proveedor mas barato es: %s,
sumin[j].datos.nom_p);
    printf("\nY su teléfono es: %s", sumin[j].datos.telef_p);
}

```

EJERCICIO 41.

Programa que visualiza en binario, según el código ASCII los caracteres introducidos por teclado (hasta que un carácter sea cero). Se utilizará una unión que contenga un carácter y una estructura de campos de bits para contener un octeto o byte.

SOLUCION.

EL algoritmo que resuelve este problema es el siguiente:

```
# include <stdio.h>
struct octeto
{
    int a:1;
    int b:1;
    int c:1;
    int d:1;
    int e:1;
    int f:1;
    int g:1;
    int h:1;
};
union bits
{
    char car;
    struct octeto bits x;
}codigo;
void visuabit(union bits x);
void main (void)
{
    printf("Introducir un caracter\n");
    scanf("%c",&(codigo.car));
    while(codigo.car!='0');
    {
        printf("El codigo ASCII es:");
        visuabit(codigo);
    }
}
```

```
        while (getchar() != '\n');
        {
            printf("Introducir otro carácter. Cero para
terminar\n");
            scanf("%c",&(codigo.car));
        }
    }
}

void visuabit(union bits x)
{
    if(x.bit.h) printf("1");
    else printf("0");
    if(x.bit.g) printf("1");
    else printf("0");
    if(x.bit.f) printf("1");
    else printf("0");
    if(x.bit.e) printf("1");
    else printf("0");
    if(x.bit.d) printf("1");
    else printf("0");
    if(x.bit.c) printf("1");
    else printf("0");
    if(x.bit.b) printf("1");
    else printf("0");
    if(x.bit.a) printf("1");
    else printf("0");
    printf("\n");
}
```

EJERCICIOS CON FICHEROS

EJERCICIO 42.

Escribir un programa que crea un fichero de texto de nombre 'datos.txt' en el que se almacena una secuencia de caracteres hasta que el valor final de fichero (EOF) es introducido por teclado. Dicho valor se introduce pulsando la tecla F6 o CTRL+Z.

SOLUCION

El algoritmo que resuelve este problema es el siguiente:

```
# include <stdio.h>
void main (void)
{
    FILE *ptr;
    char c;
    if ((ptr=fopen("datos.txt","w"))!=NULL)
    {
        while ((c=getchar())!=EOF)
            fputc(c,ptr);
        fclose(ptr);
    }
    else
        printf("El fichero 'datos.txt' no puede abrirse.\n");
}
```

EJERCICIO 43.

Programa que visualiza en pantalla el contenido de un fichero cuyo nombre es introducido por teclado. Este programa es válido tanto para ficheros de texto como para ficheros binarios.

SOLUCION

El algoritmo que resuelve este ejercicio es el siguiente:

```
# include <stdio.h>
main()
{
    FILE *ptrf;
    char ch;
    char nomb_fich[25];
    printf ("Introduce nombre del fichero");
    gets(nom_fich);
    if ((ptrf=fopen(nom_fich,"r"))==NULL)
    {
        printf("No es possible abrir '%s'.\n",nov_fich);
        exit(0);
    }
    else
    {
        while(!feof(ptrf))
            fputc(fgetc(prf),stdout);
        printf("Final del fichero '%s'.\n",nov_fich);
    }
    fclose(ptrf);
}
```

EJERCICIO 44.

Diseñar una función que reciba dos cadenas de caracteres como parámetros, correspondientes a los nombres de dos ficheros de texto (*origen* y *destino*) cuya función es copiar el contenido del primer fichero sobre el segundo dejando el fichero original intacto.

SOLUCION

El algoritmo que soluciona este ejercicio es el siguiente:

```
int copiar(char *origen, char *destino)
{
    FILE *f1, *f2;
    if((f1=fopen(origen,"r"))==NULL)
    {
        printf("No es possible abrir '%s'\n",origen);
        return -1;
    }
    if((f2=fopen(destino,"w"))==NULL)
    {
        printf("No es possible abir '%s'\n",destino);
        fclose(f1);
        return -2;
    }
    while(!feof(f1))
        fputc(fgetc(f1),f2);
    fclose(f1);
    fclose(f2);
    return 0;
}
```

EJERCICIO 45.

Programa que visualiza el contenido de un fichero cuyo nombre es introducido por teclado (este programa sólo es válido para ficheros de texto).

SOLUCION

El algoritmo que cumple con las especificaciones del enunciado es:

```
# included <stdio.h>
main()
{
    FILE *ptrf;
    char character;
    char nomb_fich[25];
    printf("Introduce nombre del fichero");
    gets(nomb_fich);
    if((ptrf=fopen(nomb_fich, "r"))==NULL)
    {
        printf("No es posible abrir '%s'.\n",nomb_fich);
        exit(0);
    }
    else
    {
        character=getc(ptrf);
        while(character!=EOF)
        {
            putchar(character, stdout);
            character=getc(ptrf);
        }
        printf("Final del fichero '%s'.\n",nomb_fich);
    }
    fclose(ptrf);
}
```

EJERCICIO 46.

Codificar una función que reciba como parámetros dos cadenas de caracteres correspondientes a los nombres de dos ficheros de texto y los concatena dejando el resultado sobre el primero.

SOLUCION

El código correspondiente a esta función es el siguiente:

```
int concatenar(char *fich_uno, char *fich_dos)+
{
    FILE *primero, *segundo;
    if((primero=fopen(fich_uno,"a"))==NULL)
    {
        printf("No es posible abrir '%s'\n",fich_uno);
        return -1;
    }
    if((segundo=fopen(fich_dos,"r"))==NULL)
    {
        printf("No es posible abrir '%s'\n");
        fclose(primero);
        return -2;
    }
    while(!feof(segundo))
        fputc(fgetc(segundo),primero);
    fclose(primero);
    fclose(segundo);
    return 0;
}
```

EJERCICIO 47.

Escribir un programa que conste de dos funciones secundarias de nombre 'crear_fichero()' y 'listar_fichero()' donde:

- (a) La primera función almacena en un fichero de texto, cuyo nombre se introduce por teclado desde la función principal y es pasado como parámetro a esta función los nombres (nombre y apellidos) de los alumnos de un centro de estudios.**
- (b) La segunda función muestra en pantalla el contenido del fichero de texto creado en el apartado anterior.**

SOLUCION

El código del programa principal es:

```
# include <stdio.h>
main ()
{
    char nomb[13];
    printf("Introduce el nombre del archivo que deseas
crear:");
    gets(nomb);
    if(crear_fichero(nomb)==-1)
        exit(0);
    else
    {
        if(!listar_fichero(nomb))
            printf("La operación ha concluido
correctamente.\n");
    }
}

int crear_fichero(char *nomb_fich)
{
    FILE *fich;
    char nombre[65];
    int i;
    if((fich=fopen(nomb_fich,"w"))==NULL)
    {
        printf("No es posible abrir fichero '%s'\n",
nomb_fich);
```

```
        return -1;
    }
    i=1;
    printf("\n<<Introduce nombre y apellidos>>\n");
    while(1)
    {
        printf("%d.-", i++);
        fgets(nombre, 65, stdin);
        if (feof(stdin))
            break;
        else
            fputs(nombre, fich);
    }
    fclose(fich);
    return 0;
}

int listar_fichero(char nomb_fich)
{
    FILE *fich;
    char nombre[65];
    int i;
    if ((fich=fopen(nomb_fich, "r")) == NULL)
    {
        printf("No es posible abrir fichero\n", nomb_fich);
        return -1;
    }
    while(1)
    {
        fgets(nombre, 65, fich);
        if (feof(fich))
        {
            printf("Final de Fichero...\n");
            break;
        }
        else
            fputs(nombre, stdout);
    }
    fclose(fich);
    return 0;
}
```

EJERCICIO 48.

Escribir el código corresponde a una función que recibe el nombre de un fichero como parámetros a partir del cual crea un segundo fichero de nombre 'copia.txt', cuyo contenido es la información almacenada en el fichero origen, pero sin espacios en blanco, tabuladores y cambios de línea.

SOLUCION

El algoritmo de este ejercicio es el siguiente:

```
int eliminar_blanco(char *fnomb)
{
    FILE *ptf1, *ptf2;
    char ch;
    ptf1=fopen(fnomb,"r");
    if (ptf1!=NULL)
    {
        ptf2=fopen("copia.txt","w");
        if(ptf2==NULL)
        {
            printf("No puede abrirse 'copia.txt'.\n");
            fclose(ptf1);
            return -1;
        }
        else
        {
            while(!feof(ptf1))
            {
                ch=fgetc(ptf1);
                if(ch!=' ' && ch!='\n' && ch!='\t')
                    fputc(ch,ptf2);
            }
            fclose(ptf1);
            fclose(ptf2);
        }
    }
    else
    {
        printf("No puede abrirse '%s'\n",fnomb);
    }
}
```

```
        return -1;
    }
}
```

EJERCICIO 49.

Dado un fichero de texto, escribir un programa que convierta los caracteres alfabéticos que aparecen en mayúsculas por caracteres alfabéticos en minúsculas y viceversa, dejando el resultado obtenido en un segundo fichero también de texto. El nombre de ambos ficheros deberá leerse del dispositivo estándar de entrada (teclado). En aquellos casos en los que el fichero destino especificado ya existia, se destruirá su contenido, dando origen a un nuevo fichero de idéntico nombre y nuevo contenido.

SOLUCION

El algoritmo que resuelve este programa es el siguiente:

```
# include <stdio.h>
void main (void)
{
    char fich1[13];
    char fich2[13];
    FILE *ptr1, *ptr2;
    char c;
    printf("Nombre del fichero origen:");
    gets(fich1);
    printf("Nombre del fichero destino:");
    gets(fich2);
    if((ptr1=fopen(fich1,"r"))==NULL)
    {
        printf("Imposible abrir fichero '%s'.\n",fich1);
        exit 0;
    }
    else
    {
        if((ptr2=fopen(fich2,"w"))==NULL)
        {
            printf("Imposible abrir fichero '%s'.\n",fich2);
            fclose(ptr1);
            exit 0;
        }
        else
        {
```

```
while ((c=fgetc(ptr1)) != EOF)
{
    if (c>='A' && c<='Z')
        fputc(c+32, ptr2);
    else
    {
        if (c>='a' && c<='z')
            fputc(c-32, ptr2);
        else
            fputc(c, ptr2);
    }
}
fclose(ptr1);
fclose(ptr2);
}
```

EJERCICIO 50.

Hacer un programa que crea un fichero de texto cuyo nombre es introducido por teclado y cuyo objetivo es almacenar los datos personales de los empleados de una empresa.

La estructura de los registros del fichero es la mostrada a continuación, donde cada uno de los registros guarda los datos personales de un empleado de la empresa.

CÓDIGO EMPLEADO	NOMBRE	EDAD	SUELDO
-----------------	--------	------	--------

SOLUCION

El algoritmo que resuelve este problema es el siguiente:

```
# include <stdio.h>
void vaciar_buffer(void)
void main (void)
{
    FILE *ptrf;
    char fichero[13];
    unsigned int codigo;
    char nombre[45];
    int edad;
    long int sueldo;
    printf("Nombre del archivo que deseas crear:");
    gets(fichero);
    if((ptrf=fopen(fichero,"w"))==NULL)
    {
        printf("No es posible abrir fichero '%s'\n",fichero);
        exit(0);
    }
    printf("Código de empleado:");
    scanf("%u",&codigo);
    while(codigo!=0)
    {
        vaciar_buffer();
        printf("Nombre:");
        gets(nombre);
```



```
        printf("Edad:");
        scanf("%d",&edad);
        printf("Sueldo:");
        scanf("%d",&sueldo);
        fprintf(ptrf,"%u %s %d %d\n",codigo, nombre, edad,
sueldo);

        printf("-----\n");
        vaciar_buffer();
        printf("Codigo de empleado:");
        scanf("%u",&codigo);
    }
    fclose(ptrf);
}

void vaciar_buffer(void)
{
    while(getchar()!='\n');
    return;
}
```

EJERCICIO 51.

Hacer un programa que muestre en pantalla el contenido de un fichero con las misma estructura y características del ejercicio anterior.

SOLUCION

El algoritmo que satisface las especificaciones del enunciado, es el que se presenta a continuación:

```
# include <stdio.h>
void vaciar_buffer(void)
void main (void)
{
    FILE *ptrf;
    char fichero[13];
    unsigned int codigo;
    char nombre[45];
    int edad;
    long int sueldo;
    printf("Nombre del archivo que deseas crear:");
    gets(fichero);
    if ((ptrf=fopen(fichero,"r"))==NULL)
    {
        printf("No se puede abrir el fichero '%s'\n",fichero);
        exit (0);
    }
    while(1)
    {
        fscanf(ptrf,                                "%u%s%d%d",
&codigo,nombre,&edad,&suelo);
        if (feof(ptrf))
            break;
        else
        {
            printf("Código:%d\n",codigo);
            printf("Nombre:%s\n",nombre);
            printf("Edad:%d\n",edad);
            printf("Sueldo:%d\n");
        }
    }
}
```

```
        printf("-----\n");
    }
}
fclose(ptrf);
}
void vaciar_buffer(void)
{
    while(getchar() != '\n')
        return
}
}
```

EJERCICIO 52.

Hacer un programa que cree un fichero de nombre 'result.tmp' qu contenga valores numéricos reales en operaciones matemáticas. Una vez creado el fichero, el programa deberá mostrar su contenido en pantalla.

SOLUCION

El algoritmo que soluciona este ejercicio es el siguiente:

```
# include <stdio.h>
void main (void)
{
    FILE *fich;
    float num;
    if ((fich=fopen("result.tmp","w"))==NULL)
    {
        printf("No es posible abrir fichero '%s' \n",fich);
        exit(1);
    }
    else
    {
        printf("<<Introduce un valor numérico por
linea>>\n");

        scanf("%f",&num);
        while(num!=0)
        {
            fprintf(fich,"%f\n",num);
            scanf("%f",&num);
        }
    }
    fclose(fich);
    if ((fich=fopen("result.tmp","r"))==NULL)
    {
        printf("No es posible abrir el fichero '%s'\n",fich);
        exit(1);
    }
    else
    {
        printf("<<Contenido del fichero 'result.tmp'>>\n");
```

```
        do
        {
            fscanf(fich,"5f",&num);
            if (feof(fich))
                break;
            else
                printf("%f\n",num);
        }
        while(1);
    }
    fclose(fich);
}
```

EJERCICIO 53.

Programa que almacena en un fichero de texto de nombre 'alum.dat' los datos personales de los alumnos de una clase.

En el fichero se almacenará por alumno la siguiente información:

Código del alumno.

Nombre del alumno.

Primer apellido.

Segundo apellido.

Edad del alumno.

Una vez finalizada la operación que almacena los datos en el fichero, visualizar el contenido del mismo en pantalla.

SOLUCION

El algoritmo que solucina este ejercicio es el mostrado a continuación.

```
# include <stdio.h>
main ()
{
    int codigo;
    char nom_alum[20];
    char apell_1[15];
    char apell_2[15];
    int edad;
    char seguir;
    FILE *fich;
    if((fich=fopen("alum.dat","w"))==NULL)
    {
        printf("No es posible abrir 'alum.dat' \n");
        exit(0);
    }
    do
    {
        printf("Código del alumno:");
        scanf("%d",&codigo);
        while(getchar()!='\n');
        printf("Nombre del alumno:");
        gets(nomb_alum);
        printf("Primer apellido:");
```

```

        gets(apell_1);
        printf("Segundo apellido");
        gets(apell_2);
        printf("Edad:");
        scanf("%d",&edad);
        fprintf(fich, "%d %s %s %s %d\n",codigo,nomb_alum,
apll_1, apell_2, edad);
        while(getchar()!='\n');
        printf("Deseas continuar (S/N)?\n");
        seguir=getchar();
    }
    while(seguir!='N' && seguir!='n');
    fclose(fich);
    if((fich=fopen("alum.dat","r"))==NULL)
    {
        printf("No es posible abrir 'alum.dat'\n");
        exit(0);
    }
    while(!feof(fich));
    {
        fscanf(fich, "%d%s%s%s%d", &codigo, nomb_alum,
apell_1, apell_2, &edad);
        printf ("Codigo de alumno: %d\n", codigo);
        printf ("Nombre del alumno: %s\n", nomb_alum);
        printf ("Primer apellido: %s\n", apell_1);
        printf ("Segundo apellido: %s\n", apell_2);
        printf ("Edad: %d\n", edad);
        printf ("*****");
        if(feof(fich))
        {
            printf("\nDetectado final de fichero\n");
            break;
        }
    }
    fclose(fich);
}

```

EJERCICIO 54.

Se desea crear un programa que almacene en un fichero de nombre 'agenda.dat' los datos de una agenda telefónica. El fichero estará formado por registros con la siguiente estructura.

NOMBRE	TIPO	LONGITUD
Nombre	Cadena	15
Apellidos	Cadena	25
Telefono	Cadena	14

SOLUCION

El algoritmo que nos permite conseguir la agenda telefónica es el siguiente:

```
# include <stdio.h>
main()
{
    struct Tagenda
    {
        char nomb[15];
        char apellidos[25];
        char Tf[14];
    } agenda;
    char seguir;
    FILE *pf;
    if ((pf=fopen("agenda.dat","w"))==NULL)
    {
        printf("No es posible abrir 'agenda.dat'\n");
        exit(0);
    }
    do
    {
        printf("Nombre:");
        gets(agenda.nomb);
        printf("Apellidos:");
        gets(agenda.apellidos);
        printf("Telefono:");
```



```
        gets (agenda.Tf);
        fwrite(&agenda,sizeof(struct Tagenda),1,pf);
        printf("¿Deseas continuar (S/N)?\n");
        seguir=getchar();
        while(getchar()!='\n');
    }
    while(seguir!='N' && seguir!='n');
    fclose(pf);
    if((pf=fopen("agenda.dat","r"))==NULL)
    {
        printf("No es possible abrir 'agenda.dat' \n");
        exit(0);
    }
    while(1)
    {
        fread(&agenda,sizeof(struct Tagenda),1,pf);
        if(feof(pf))
            break;
        printf("Nombre: %s\n",agenda.nomb);
        printf("Apellidos: %s,\n",agenda.apellidos);
        printf("Telefono:%s\n",agenda.Tf);
        putchar('\n');
    }
    fclose(pf);
}
```

EJERCICIO 55.

Realizar un programa que calcula el área de un triángulo, donde 'b' e la base y 'h' la altura. El resultado debe ser almacenado en un fichero de nombre 'calculos.dat' y visualizado posteriormente.

SOLUCION

Este ejercicio se resuelve con el siguiente algoritmo:

```
# include <stdio.h>
main ()
{
    FILE *ptrf;
    float area,b,h;
    printf("Introduce valor de la base:");
    scanf("%f",&b);
    printf("Introduce valor de la altura:");
    scanf("%f",&h);
    area=b*h/2;
    ptrf=fopen("calculos.dat","w");
    if(ptrf==NULL)
    {
        printf("Error de apertura en 'calculos.dat'\n");
        exit(0);
    }
    fwrite(&area, sizeof(area),1,ptrf);
    fclose(ptrf);
    ptrf=fopen("calculos.dat","r");
    if(ptrf==NULL)
    {
        printf("Error de apertura en 'calculos.dat'\n");
        exit(0);
    }
    fread(&area,sizeof(area),1,ptrf);
    printf("El area del triangulo es %f,\n",area);
    fclose(ptrf);
}
```

EJERCICIOS AVANZADOS Y VARIADOS

EJERCICIO 56.

Diseñar un programa que realiza la media aritmética.

SOLUCION

EL algoritmo que resuelve este ejercicio será:

```
# include <stdio.h>
main ( )
{
    float suma=0, total=0, num, media;
    scanf ("%f",&num);
    while (num != 0)
    {
        suma += num;
        total++;
        scanf ("%f",&num);
    }
    media = suma / total;
    printf ("La media es %f", media);
}
```

EJERCICIO 57.

Realizar una función que calcule la raíz cuadrada de un número dando su aproximación entera más cercana.

SOLUCION

El algoritmo que resuelve este ejercicio es el que se presenta a continuación:

```
long raizcuad (long valor)
{
    long r;
    for (r=1;r*r < valor; r++)
        if ((valor - (r-1)*(r-1))< (r*r - valor))
            return r-1;
    else
        return r;
}
```

EJERCICIO 58.

Realizar un programa que calcule la potencia de un número, dada la base y el exponente, siendo este entero y positivo.

SOLUCION

El programa que resuelve este ejercicio presenta el siguiente código:

```
# include <stdio.h>
main ( )
{
    int x, n, producto;
    printf (" Introducir el valor de la base");
    scanf ("%d",&x);
    printf (" Introducir el valor del exponente");
    scanf ("%d",&n);
    producto = 1;
    for ( ;n; n--)
        producto *= x;
    printf ("La potencia es %d",producto);
}
```

Este programa lo podemos implementar también como una función recursiva tal y como se muestra a continuación:

```
int potencia (int x, int n)
{
    if (n == 0) return 1;
    else return x*potencia (x, n-1);
}
```

EJERCICIO 59.

Implementar la función que devuelva el valor absoluto de un número.

SOLUCION

Esta función tiene el siguiente código:

```
int valorabsoluto (int x)
{
    if (x>=0)
        return x;
    else
        return -x;
}
```

Otra forma de expresar esta función sería mediante el operador `?:`, tal y como se muestra a continuación:

```
int valorabsoluto (int x)
{
    (x>0) ? return x: return -x;
}
```

EJERCICIO 60.

Implementar la función que realiza la potencia de un número pudiendo ser el exponente positivo o negativo, pero siempre entero.

SOLUCION

El código de esta función es el que se muestra a continuación:

```
float pot (int x, int n)
{
    int potencia (int x, int n);
    int valorabsoluto (int x);
    if (n > 0)
        return potencia (x, n);
    else
        return (1 / potencia (x, valorabsoluto (n)));
}
```

Las funciones *potencia* y *valorabsoluto* se han obtenido en los dos programas anteriores por lo que nos las vamos a especificar en este ejercicio.

EJERCICIO 61.

Realizar dos funciones, una que calcule el m.c.d y otra el m.c.m de dos números dados.

SOLUCION

Para calcular el m.c.d de dos números dados vamos a emplear el algoritmo de Euclides, que vamos a mostrar con el siguiente ejemplo:

m.c.d (72, 540)

73 468

73 396

73 324

73 232

73 180

73 108

72 36

36 36

Según este proceso, puede deducirse fácilmente que la función que calculará el m.c.d de dos números será recursiva, y su código se presenta a continuación:

```
int mcd (int x, int y)
{
    if (x == y) return x;
    else
    {
        if (x > y) return mcd (x-y,y);
        else return mcd (x, x-y);
    }
}
```

Para obtener la función que calcula el m.c.m. de dos números dados debemos tener en cuenta que:

$mcd * mcm = x * y$ siendo x e y los números.

Así pues la función que calcula el mcm de dos números será:


```
int mcm (int x, int y)
{
    return (x*y / mcd (x,y));
}
```

EJERCICIO 62.

Realizar una función que identifique palíndromos.

Nota: un palíndromo es una palabra que se lee igual de derecha a izquierda que de izquierda a derecha. La función debe devolver 1 si la palabra es un palíndromo y 0 si no lo es.

SOLUCION

Podemos resolver este ejercicio de dos formas distintas dependiendo de los parámetros de entrada que le demos a la función.

En la primera forma i marca el comienzo de la palabra y j el final de la misma. La forma de realizar esta función es recursiva puesto que se debe realizar comprobaciones sucesivas, el código de la misma es:

```
int palindromo (int i, int j, char matriz [50])
{
    if (j<=1) return 1;
    else
    {
        if (matriz [i]==matriz[j])
            return palindromo (i+1, j-1, matriz [50]);
        else return 0;
    }
}
```

La otra forma de hacerlo sería dándole a la función la longitud de la palabra que queda por comprobar para determinar si es palíndromo o no además de la palabra. Su código es:

```
int palindromo (int long, char matriz [50])
{
    char matrizB [50];
    int i;
    if (long <=1) return 1;
    else
    {
        if (matriz [0]==matriz [long-1])
        {
            for (i=0; i<long; i++)
                matrizB [i]=matriz [i+1];
            return palindromo (long-2, matrizB [50]);
        }
        else return 0;
    }
}
```

EJERCICIO 63.

Escribir en C el código de una función con parámetros formales *size* (entero) y *matriz* (matriz 10 x 10 de enteros) y que devuelva el valor del determinante de la submatriz principal (*size x size*) de *matriz*. Puede utilizarse la función *int potencia (int base, int exponente)*.

SOLUCION

El código de esta función es el que se presenta a continuación:

```
int determinante (int size, int matriz [10][10])
{
    int newmatrix [10][10];
    int i, j, k, resultado=0;
    if (size==1) resultado = matrix [0][0];
    else
    {
        for (k=0; k<size; k++)
        {
            for (i=0; i<size-1; i++)
            {
                for (j=0; j<size-1; j++)
                {
                    if (j<k)
                        newmatrix[i][j]=matrix[i+1][j];
                    else
                        newmatrix[i][j]=matrix[i+1][j+1];
                }
            }
            resultado += matrix[0][k]*potencia
                (-1,k)*determinante(size-1, newmatrix [10][10]);
        }
    }
    return resultado;
}
```

Otra forma de realizar esta función sería en vez de con índices con punteros en este caso el código sería:

```
int determinante (int size, int matriz [10][10])
{
    int newmatrix [10][10];
    int i, j, k, resultado=0;
    if (size==1) resultado = matriz [0][0];
    else
    {
        for (k=0; k<size; k++)
        {
            for (i=0; i<size-1; i++)
            {
                for (j=0; j<size-1; j++)
                {
                    if (j<k)
                        (*(newmatrix+i)+j)=*(*(matriz+i+1)+j);
                    else
                        (*(newmatrix+i)+j)=*(*(matriz+i+1)+j+1);
                }
            }
            resultado += matriz[0][k]*potencia
                (-1,k)*determinante(size-1, newmatrix [10][10]);
        }
    }
    return resultado;
}
```

EJERCICIO 64.

Sea char *mat[14]*, localícese la posición del primer dígito que forma el entero mayor incluido en *mat*. Nota: nos movemos con punteros.

SOLUCION

Lo único que debemos saber para resolver este ejercicio es que si la resta de dos *char* se almacena en un tipo *int*, en éste queda almacenado la diferencia entre el código ASCII de cada uno de los *char*. La función que realiza esta tarea es la siguiente:

```
char mat[14]
int funcion (char *mat)
{
    int parcial, sumaparcial=0, sol, j, sumamayor=0;
    for (i=0; i<15; i++)
    {
        parcial=*mat-'0';
        if ((parcial>=0)&&(parcial<=9))
        {
            sumaparcial=10*sumaparcial+parcial;
            j++;
        }
        else
        {
            if(sumaparcial>sumamayor)
            {
                sol=i-j;
                sumamayor=sumaparcial;
            }
            sumaparcial=0;
            j=0;
        }
        mat++;
    }
    return sol;
}
```

EJERCICIO 65.

Escribir en lenguaje C el código de una función que devuelva el número de días entre dos fechas escritas en el tipo fecha que se debe definir adecuadamente.

SOLUCION

El tipo fecha será una estructura que vamos a definir de la siguiente forma:

```
Struct fecha
{
    int dd;
    enum mes {Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
    Agosto, Septiembre, Octubre, Noviembre, Diciembre}mm;
    int aa;
};
```

Para resolver este problema necesitamos dos funciones que son las siguientes:

```
int auxiliar (struct fecha f) //Cuenta los días desde fecha 0
{
    int resultado;
    resultado = 365 * f.aa;
    resultado += f.aa /4;
    resultado -= f.aa /100;
    resultado += f.aa /1000;
    if (f.mm == Febrero) resultado +=31;
    else
    {
        switch (f.mm)
        {
            case Diciembre: resultado += 30;
            case Noviembre: resultado += 31;
            case Octubre: resultado += 30;
            case Septiembre: resultado += 31;
            case Agosto: resultado += 31;
            case Julio: resultado += 30;
            case Junio: resultado += 31;
            case Mayo: resultado += 30;
            case Abril: resultado += 31;
            case Marzo: resultado +=28;
            default: resultado += 59;
        }
        if (f.aa%1000==0 || (f.aa%4==0 && f.aa%100!=0))
            resultado++;
    }
    resultado += f.dd;
    return resultado;
}

int funcion (struct fecha f1, struct fecha f2)
{
    int dias1, dias2, diferencia;
    dias1 = auxiliar (f1);
    dias2 = auxiliar (f2);
    diferencia = (dias1 > dias2)? dias1-dias2 : dias2-dias1;
    return diferencia;
}
```


EJERCICIO 66.

Escribir una función que dado un número nos determine si es o no es capicúa.
Nota: devuelve un 1 si lo es y un 0 si no lo es.

SOLUCION

La función correspondiente tendrá el siguiente código:

```
int capicua (int x)
{
    int suma=0;
    while (x>=10)
    {
        suma = 10*suma+x%10;
        x=x/10;
    }
    suma = 10*suma + x;
    if (suma==x)
        return 1;
    else
        return 0;
}
```

EJERCICIO 67.

Diseñar una función que dado un número y el número de decimales correspondiente nos devuelva el logaritmo en base 2 del mismo.

SOLUCION

En la función que se nos pide vamos a utilizar otra función que nos calcula la aproximación entera del logaritmo en base 2 del número dado, y cuyo código es el siguiente:

```
int long (float numero)
{
    int j=-1, pot=1;
    if (numero >= 1)
    {
        while (pot < numero)
        {
            pot *=2;
            j++;
        }
        return j;
    }
    else
    {
        while ((1/pot)>numero)
        {
            pot *=2;
            j--;
        }
        j +=2;
        return j;
    }
}
```

La función especificada en el enunciado será:

```
float logaritmobase2 (float numero, int x)
{
    int log (float numero);
    int potencia (float base, int exponente);
    int z;
    z = potencia (10, x)
    return (log(potencia (numero, z))/z);
}
```

EJERCICIO 68.

Escribir en lenguaje C el código de un programa que lea texto (carácter a carácter) desde el teclado y muestre por pantalla la suma de los números encontrados hasta que lea el carácter '\$'. Ejemplo: ante la entrada *c91d908ah2m023q\$* produce (91+908+2+233) 1024.

SOLUCION

El código del programa que realiza esta tarea es el siguiente:

```
# include <stdio.h>
main ( )
{
    char c;
    int digito, parcial=0;
    int total = 0;
    do
    {
        c = getchar ();
        digito = c - '0';
        if ((digito <= 9) && (digito >=0))
            parcial = 10*parcial + digito;
        else
        {
            total += parcial;
            parcial =0;
        }
    }
    while (c!='$')
    printf ("La cadena produce: %d \n", total);
}
```

EJERCICIO 69.

Escribir en lenguaje C un programa que lea un entero. Mediante la función *f*, lo escribe en base 2 en un vector de 0 ó 1 de tamaño 8, *matriz*, definido en main, de forma que el primer dígito de matriz sea el menos significativo. Mediante la función *g*, modifica matriz para que contenga la representación en binario de entero leído + 1. Finalmente muestra por pantalla la representación en base 2 de entero leído + 1. Los parámetros formales de *f* son el entero *num* y el puntero al primer elemento de matriz *pun1*. El parámetro formal de *g* es el puntero al primer elemento de matriz, *pun2*. Las funciones *f* y *g* no devuelven ningún valor de forma explícita.

SOLUCION

El programa que resuelve este ejercicio es el que se presenta a continuación:

```
# include <stdio.h>

void f (int num, int *pun1)
{
    int j;
    for (j=0; j<=7; j++)
    {
        *pun1=num%2;
        num/=2;
        pun1++;
    }
}

void g (int *pun2)
{
    while (*pun2 == 1)
    {
        *pun2=0;
        pun2++;
    }
    *pun2=1;
}

void main ( )
{
    int enteroleido, j, n=0;
    int matriz [8];
    scanf ("%d", &enteroleido);
    f (enteroleido, matriz)
    g (matriz)
    while (matriz [7-n] == 0)
        n++;
    for (j=n; j<=7; j++)
        printf ("%d", matriz [7-j]);
}
```

EJERCICIO 70.

Escribir en C un programa que muestre por pantalla todos los números de 4 dígitos que interpretados de izquierda a derecha en base 7 representan la misma cantidad que interpretados de derecha a izquierda en base 5.

SOLUCION

El programa que resuelve este ejercicio es el que se muestra a continuación, en el que únicamente se debe tener en cuenta que en base 5 los números solo pueden contener los dígitos del 0 al 4 y en base 7 del 0 al 6.

```
# include <stdio.h>
main ()
{
    long int i,j,k,m imprime;
    for (i=0; i<=4; i++)
    {for (j=0; j<=4; j++)
        {for (k=0; k<=4; k++)
            {for (m=0; m<=4; m++)
                {
                    if (343*i+49*j+7*k+m==i+5*j+25*k+125*m)
                    {
                        imprime=1000*i+100*j+10*k+m;
                        printf("%d", imprime);
                    }
                }
            }
        }
    }
}
```

EJERCICIO 71.

Programa en C que lea un entero. Mediante la función *f* escriba ordenadamente los dígitos que forman su representación en base 2 en un vector de tamaño 8, *matriz*, definido en main de forma que el primer dígito de la matriz sea el más significativo. Mediante la función *g* modificamos matriz para que contenga la representación en base 2 correspondiente al entero leído más 2^3 . Finalmente muestra por pantalla si entero leído+ 2^3 es o no capicúa y si es o no potencia de 2. Los parámetros formales de *f* son el entero *num* y el puntero al primer elemento de matriz *punt1*. El parámetro formal de *g* es el puntero al primer elemento de matriz *punt2*. Las funciones *f* y *g* no devuelven valores de forma explícita.

SOLUCION

El programa que cumple las especificaciones del enunciado es:

```
# include <stdio.h>
void f(int num, int *punt1)
{
    int j;
    punt1 +=7;
    for (j=0; j<=7; j++)
    {
        *punt1=num%2;
        num /=2;
        punt1--;
    }
}
void g (int *punt2)
{
    punt2 +=4;
    while (*punt2 == 1)
    {
        *punt2=0;
        punt2--;
    }
    *punt2=1;
}
void main ()
{
```



```
int enteroleido, j, reves=0, n=0;
int matriz [8];
scanf ("%d", &enteroleido);
f (enteroleido, matriz);
g (matriz);
for (j=0; j<=7; j++)
{
    reves+=(*matriz)*(potencia(2,j));
    matriz++;
}
if (reves == (enteroleido+8))
    printf ("Es capicua");
else
    printf ("No es capicua");
while (matriz [n] == 0)
    n++;
while (matiz [n+1] == 0)
    n++;
if (n>8)
    printf ("Es potencia de 2");
else
    printf ("No es potencia de 2");
}
```

EJERCICIOS DE APLICACIÓN

EJERCICIO 72.

Diseñar un programa que realice la conversión de un número decimal a hexadecimal.

SOLUCION

El algoritmo que nos soluciona este programa es el siguiente:

```
# include <stdio.h>

void ahex (int n)

main ()
{
    int n; /*numero leído*/
    int a, b; /*almacenamiento temporal*/
    printf ("Introduzca un entero del 0 al 255: ");
    scanf ("%d",&n);
    a = n/16; /*obtiene el valor de los 4 bits superiores*/
    b = n%16; /*obtiene el valor de los 4 bits inferiores*/
    ahex(a);
    ahex(b);
}

void ahex (int n)
{
    if ((n>=0)&&(n<=9))
        printf ("%d",&n);
    else
    {
        switch (n)
        {
            case 10: printf ("A");break;
            case 11: printf ("B");break;
            case 12: printf ("C");break;
            case 13: printf ("D");break;
            case 14: printf ("E");break;
            case 15: printf ("F");break;
            default: printf ("Error!\n");
        }
    }
}
```

EJERCICIO 73.

Desarrollar un programa en C que realice la función de una máquina expendedora. Esta máquina acepta monedas de 5, 10 y 25 pesetas. Todos los productos de la misma cuestan 50 pts. Cuando se compra un producto, se utiliza como cambio el mínimo número de monedas de 5, 10 y 25 pesetas.

SOLUCION

El algoritmo que implementa la función de esta máquina expendedora es:

```
# include <stdio.h>
main ()
{
    int cantidad=0, D5=0, D2=0, D1=0;
    char moneda;
    printf ("Deposite moneda (1-2-5 duros) o pida articulo
(P) ");
    do
    {
        moneda = getchar ();    /*obtiene respuesta usuario*/
        printf ("\n");
        switch (moneda)
        {
            case '5': cantidad += 25; break;
            case '2': cantidad += 10; break;
            case '1': cantidad += 5; break;
            case 'P': if (cantidad<50)
                printf ("Por favor introducir más dinero");
        }
        if (moneda!='P')
            printf ("Cantidad =%f\n",cantidad/100.0);
    }
    while ((moneda!='P') || (cantidad <50));
    cantidad -=50;    /*compra el articulo*/
    while (cantidad >= 25)
    {
        cantidad -= 25;
        D5++;
    }
```

```
    if (D5!=0)
        printf ("Monedas de 25 pts: %d",D5);
    while (cantidad >=10)
    {
        cantidad -= 10;
        D2++;
    }
    if (D2!=0)
        printf ("Monedas de 10 pts: %d",D2);
    while (cantidad >= 5)
    {
        cantidad -= 5;
        D1++;
    }
    if (D1!=0)
        printf ("Monedas de 5 pts: %d",D1);
}
```

EJERCICIO 74.

Escribir un programa en C que genere la serie numérica de Fibonacci.

SOLUCION

El algoritmo que nos genera la serie de Fibonacci es el siguiente:

```
# include <stdio.h>
main ()
{
    int n ;                /*numero leído*/
    int anterior, nuevo;   /*variables de la serie de
Fibonacci*/
    int temp.;            /*almacenamiento temporal*/
    int j;                /*Contador de bucle*/
    printf ("Introduzca el numero de términos que se
generan:");
    scanf ("%d",&n);
    anterior=0;
    nuevo=1;
    printf ("%d %d ",anterior, Nuevo);
    for (j=1;j<=n-2;j++)
    {
        temp=anterior+nuevo;
        anterior=nuevo;
        nuevo=temp;
        printf ("%d",nuevo);
    }
}
```

EJERCICIO 75.

Implementar una máquina calculadora en C.*SOLUCION*

El algoritmo que implementa esta máquina es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#define FALSO 0
#define VERDADERO 1
main ()
{
    int estado;          /*Estado actual de la máquina*/
    char car;            /*Carácter leído*/
    int err;              /*Indicador de error*/
    int op;               /*operación matemática seleccionada*/
    int X,Y,Z;            /*se usan en los cálculos*/
    printf ("Introduzca una ecuación de la forma:\n2");
    printf ("X + Y = o X - Y =\n");
    printf ("donde X e Y son los enteros de un solo
digito.\n");
    estado = 1;
    err = FALSO;
    do
    {
        do                /*salta los espacios en blanco*/
        {
            car=getchar();
            printf ("%c",car);
        }
        while (car == ' ')
        switch (estado)
        {
            case 1: if ((car>='0')&&(car<=9))
                    {
                        X=atoi(&car);
                        estado=2;
                    }

```

```
        else err = VERDADERO;
        break;
    case 2: op=0;
        if (car=='+') op=1;
        if (car=='-') op=2;
        if (op==0) err=VERDADERO;
        estado=3;
        break;
    case 3: if ((car >='0') && (car <='9'))
        {
            Y=atoi(&car);
            estado=4;
        }
        else err=VERDADERO;
        break;
    case 4: if (car=='=') estado = 5;
        else err=VERDADERO;
    }
}
while ((estado!=5) && !err);
if (!err)
{
    Z=(op==1)?X+Y:X-Y;
    Printf ("%d",Z);
}
else printf ("\n Carácter ilegal en la entrada de datos.");
}
```

EJERCICIO 76.

Implementar un programa en C para determinar un número entre 1 y 1024 elegido por el usuario.

SOLUCION

Este programa tendrá la siguiente estructura:

```
# include <stdio.h>
# define MAXMUN 1024
int adivinar (int inferior, int superior);
main ()
{
    int intentos;
    printf ("Por favor piense un numero del 1 al%d \n",MAXMUN);
    intentos=adivinar (1,MAXMUN);
    printf ("\n He necesitado %d intentos.\n",intentos);
}
int adivinar (int inferior, int superior)
{
    int prediccion;
    char respuesta;
    prediccion=inferior+(superior-inferior)/2;
    printf ("\nEs su numero %d?",prediccion);
    respuesta =getchar();
    switch (respuesta)
    {
        case 'S': return 1; break;
        case 'm': return 1+adivinar(inferior,prediccion-
1);break;
        case 'M': return 1+adivinar
(prediccion+1,superior);break;
    }
}
```


EJERCICIO 77.

Escribir en C un programa que genere una secuencia de números pseudo-aleatoria.

SOLUCION

El algoritmo que implementa esta función es el siguiente:

```
# include <stdio.h>
void generador_aleatorios (unsigned char *aleatorio);
main ()
{
    unsigned char aleatorio; /*almacenamiento para el n°
aleatorio*/
    int contador;
    printf ("Introduzca la semilla de aleatorios (del 0 al
255)");
    scanf ("%d",&aleatorio);
    for (contador=1;contador<=10;contador++)
    {
        generador_aleatorios(&aleatorio);
        printf ("El proximo numero aleatorio es
%d\n",aleatorio);
    }
}
void generador_aleatorios (unsigned char *aleatorio)
{
    unsigned char temp1,temp2;
    temp1=*aleatorio &0x80;
    temp2=*aleatorio &0x04;
    *aleatorio <<=1;
    if ((0==temp1 | temp2) || (0x84==temp1 | temp2))
        *aleatorio = *aleatorio | 0x01;
}
```

EJERCICIO 78.

Realizar un programa en C que se comporte como un comprobador de códigos ISBN.

SOLUCION

El formato de un código ISBN es el siguiente: código de grupo (1 dígito), código del editor (4 dígitos), código del libro (4 dígitos), carácter de control (1 dígito).

El último carácter se obtiene en dos partes, una primera donde se multiplica cada valor del código ISBN por el valor correspondiente a la serie 1,2,3,4,5,6,7,8,9 y todas estas multiplicaciones se suman. En el paso 2, esta suma se divide por 11 y se conserva el entero del resto, siendo éste el dígito de control calculado.

El algoritmo que se encarga de comprobar estos códigos es:

```
# include <stdio.h>
main ()
{
    char entrada[10];
    int total=0;
    int j, rem;
    printf ("Introduzca los 9 primeros digitos del codigo
ISBN");

    scanf ("%s",&entrada);
    for (j=0;j<9;j++)
        total +=(entrada[j]-0x30)*(j+1);
    rem=total%11;
    printf("El ultimo carácter debía ser");
    if (rem!=10)
        printf("%c", (rem+0x30));
    else
        printf ("X");
}
```

EJERCICIO 79.

Realizar un programa en C que nos permita contar vocales.

SOLUCION

Deben contarse las vocales mayúsculas y minúsculas, puesto que son idénticas. Se usa un vector entero de cinco elementos para mantener los contadores de las vocales. Para reducir el número de comparaciones necesarias, cada carácter de la cadena de caracteres se convierte a mayúsculas antes de hacer dichas comparaciones. Los caracteres ASCII se convierten fácilmente a mayúsculas haciendo un AND a nivel de bit de sus códigos con 0xdf.

```
# include <stdio.h>
# include <string.h>
# include <ctype.h>
main ()
{
    char texto []="NUESTRO profesor EXPLICA cada CLASE con
CLARIDAD.";
    char vstr []="AEIOU";
    static int vocales;
    static int vcount [5];
    int j;
    printf ("EL texto de entrada es \"%s\"",texto);
    for(j=0; j < strlen(texto); j++)
    {
        if (isalpha (texto[j])!=0)
            texto[j]=texto[j]&0xdf;
        if (strchr(vstr,texto[j])!='\0')
            vocals++;
        switch (texto[j])
        {
            case 'A': vcount[0]++;break;
            case 'E': vcount[1]++;break;
            case 'I': vcount[2]++;break;
            case 'O': vcount[3]++;break;
            case 'U': vcount[4]++;break;
        }
    }
}
```

```
        printf ("\n EL texto de entrada contiene %d\n",vocale);  
        printf ("Hay %d As, %d Es, %d Is, %d Os, %d Us",vcount[0],  
vcount[1], vcount[2], vcount[3], vcount[4]);  
    }
```

EJERCICIO 80.**Implementar un analizador de léxico.****SOLUCION**

La finalidad del analizador léxico es romper la sentencia de entrada para extraer los componentes más pequeños admitidos en el lenguaje C.

El analizador de léxico es muy limitado, no analizará correctamente cualquier sentencia de C que usted introduzca, solamente aquellas que contengan los símbolos definidos en los vectores *sencillos* y *dobles*. El funcionamiento del programa es: si un carácter de entrada es alfabético, se supone que es el principio del nombre de una variable o función. En otro caso, se busca en los vectores *sencillos* y *doble* alguna equivalencia. Si no se encuentra, el texto de entrada se imprime en la salida indicado que es un componente léxico indefinido.

```
# include <stdio.h>
# include <string.h>
# include <ctype.h>
main ( )
{
    char sencillos[]=";{} ,=() []*?'\"\\_";
    char dobles[]="*/= % = += -= || && == != <= >= << >> ++ --";

    char ch, entrada[80], ds[3];
    int pos=0;
    printf ("Introduzca una sentencia en C ");
    gets (entrada);
    while (pos < strlen(entrada))
    {
        printf("Componentes léxico");
        do
        {
            ch=entrada[pos];
            pos++;
        } while(ch == ' ');    /*Saltar blancos*/
        /*¿Es el nombre del símbolo?*/
        if (isalpha(ch)!=0)
        {
            printf("%c",ch); /*Imprimir la primera letra*/
            /*Imprimir la letras/dígitos restantes*/
```

```

        while(isalnum(entrada[pos]))
        {
            printf("%c", entrada[pos]);
            pos++;
        }
    }
    else
    /*¿Es un número?*/
    if(isdigit(ch))
    {
        printf("%c", ch); /*imprimir primer digito*/
        /*imprimir los dígitos restantes*/
        while(isdigit(entrada[pos]))
        {
            printf ("%c", entrada[pos]);
            pos++;
        }
    }
    else
    /*Comprobar que es un componente léxico sencillo*/
    if(strchr(sencillos, ch) != 0);
    {
        printf("%c", ch); /*imprimir componentes
léxicos*/

        /*¿Componente léxico de tipo cadena o
carácter?*/

        if((ch == '\\''))
        {
            do /*imprimir el texto del componente*/
            {
                ch=entrada[pos];
                printf("%c", ch);
                pos++;
            }
            while((ch=='') && (ch!='\\'));
        }
        else
        /*Comprobar que es un componente léxico doble*/
        {
            ds[0]=ch;
            ds[1]=entrada[pos];

```

```
        ds[2]='\0';
        if(strstr(dobles,ds)!='\0')
        {
            printf("%s",ds);
            pos++;
        }
        else
            printf("%c",ch);
    }
    printf("%c",ch);
}
```

EJERCICIO 81.

Implementar en C un codificador de texto.**SOLUCION**

En aras de la seguridad, muchas organizaciones codifican actualmente sus datos de los computadores para evitar que los espías puedan obtener información. Existen muchas técnicas para codificar (o cifrar) un texto, algunas de las cuáles tienen cientos de años.

Unas de las técnicas más simples es la codificación por *transposición*. En esta técnica, el texto de entrada se escribe como una matriz de caracteres bidimensional. Luego se transpone la matriz (filas y columnas invertidas) y se extraen los caracteres.

El siguiente programa lleva a cabo una codificación por transposición creando una matriz cuadrada cuyas dimensiones se basan en la longitud del texto de entrada. Los elementos no usados de la matriz se rellenan con blancos para eliminar la posibilidad de que aparezcan caracteres extraños en la salida.

```
# include <stdio.h>
# include <string.h>
# include <math.h>
main()
{
    char entrada[80];
    char codificador[9][9];
    int r,c,p,n,i;
    printf("Introduzca el texto a codificador ");
    gets(entrada);
    i=0;
    for(n=0;n<strlen(entrada);n++)
    {
        if(entrada[n]==' ')
            i++;
    }
    n=1+sqrt(strlen(entrada)-i);
    for(r=0;r<9;r++)
    {
        for(c=0;c<9;c++)
            codificador[r][c]=' ';
    }
    p=0;
```



```
    r=0;
    c=0;
    while(entrada[p]!=''\0')
    {
        if(entrada[p]!=' ')
        {
            codificador[r][c]=entrada[p];
            r++;
            if(r==0)
            {
                r=0;
                c++;
            }
        }
        p++;
    }
    printf("Codificación por transposición: ");
    for(r=0;r<9;r++)
    {
        for(c=0;c<9;c++)
        {
            if (codificador[r][c]!=' ')
                printf("%c",codificador[r][c]);
        }
    }
}
```

EJERCICIO 82.

Implementar en C un programa que nos permita acceder a la FAT y sea capaz de almacenar información sobre 64 sectores. Se utilizará un función denominada *enlaces()*, para buscar a través de la FAT hasta que se encuentra el valor 99 y otra función llamada *sectores_libres()* que se utilizará para determinar cuántos sectores están disponibles.

SOLUCION

Para este programa se utiliza la técnica de organización encadenada, utilizada en el sistema operativo DOS para el almacenamiento y recuperación eficiente de archivos. Se basa en el uso de una estructura de datos denominada tabla de asignación de archivos (file allocation table, FAT) que está almacenada en una especial del disco. La FAT es una lista de números que describe cómo están asignados los sectores (o unidades de asignación) del disco.

Cada posición en de la FAT con un valor distinto de cero indica que el sector está asignado. El valor 99 especifica que se trata del último sector del archivo. Los elementos con un valor igual a cero se corresponden con sectores libres. Teniendo esto es cuenta el algoritmo será:

```
# include <stdio.h>
void enlaces(int cp)
void sectores_libres(void);
inf fac[64]={0, 0, 99, 54, 5, 6, 38, 24, 0, 10, 49, 15, 59, 0,
12, 0, 0, 99, 0, 0, 2, 0, 44, 0, 29, 0, 11, 0, 0, 3, 0, 0, 4, 17, 0,
99, 0, 0, 39, 9, 0, 0, 6, 0, 45, 46, 26, 0, 0, 33, 0, 0, 0, 0, 55, 56,
57, 99, 0, 0, 61, 20, 0, 0}
main()
{
    int cadenzas[]={7, 32, 60, 14};
    int j;
    for(j=0;j<4;j++)
    {
        printf("Cadena enlazada del archivo #%d: ",j+1);
        enlaces(cadenas[j]);
    }
    printf("\n");
    sectores_libres();
}
```

```
void enlaces(int cp)
{
    do
    {
        printf("%d\t",cp);
        if((cp!=0)&&(cp!=99))
            cp=fat[cp];
    }
    while ((cp!=0)&&(cp!=0));
    if (cp==0)
        printf ("?\nError! Cadena perdida");
    printf("\n");
}

void sectores_libres(void)
{
    int j,libres=0;
    for(j=1;j<64;j++)
    {
        if (fat[j]==0)
            libres++;
    }
    printf ("Hay %d unidades libres de asignación",libres);
}
```

EJERCICIO 83.

Realizar un programa que implementa la función de un generador de histogramas de una lista de 100 valores, donde cada valor está comprendido en el intervalo del 1 al 10.

SOLUCION

Cuando se examinan grandes volúmenes de datos, una técnica para evaluar el conjunto de los datos es el histograma, que contiene la frecuencia de aparición de cada valor en el conjunto de datos. El algoritmo correspondiente será:

```
# include <stdio.h>
# include <stdlib.h>
main()
{
    int valores[100];
    static int hist[10];
    int j;
    for(j=0;j<100;j++)
        valores[j]=1+rand()%10;
    printf ("El conjunto inicial de numeros es:\n");
    for(j=0;j<100;j++)
    {
        if(0==j%10)
            printf("\n");
        printf("%d",valores[j]);
    }
    for(j=0;j<100;j++)
        hist[valores[j]-1]++;
    printf("\n\n El histograma para los numeros dados
es:\n\n");
    for(j=0;j<10;j++)
        printf("%d:\t%d\n",j+1,hist[j]);
}
```

EJERCICIO 84.

Escribir en C un program que implemente una tabla de dispersión de 32 posiciones.

SOLUCION

El término “tabla de dispersión” es un nombre algo peculiar utilizado para denominar a una estructura de datos que almacena la información de forma que la posición en la tabla donde se guarda un determinado dato, se calcula mediante el uso de una *función de dispersión*. Una tabla de dispersión puede construirse sobre un vector de una dimensión o sobre una estructura de datos más compleja. En este caso se utilizará una función de dispersión sencilla que genere un índice para una tabla de dispersión que contenga 32 posiciones. El algoritmo será:

```
# include <stdio.h>
# include <string.h>
int dispersion (char nombre[]);
main ()
{
    static int tabla_dispersión[32], fin;
    int dirección;
    char var[10];
    do
    {
        printf ("Introduzca el nombre de una variable ");
        gets(var);
        if(0!=strcmp(var,"fin"))
        {
            direccion=dispersion(var);
            if(0==tabla_dispersión[direccion])
            {
                printf("Variable nueva, dirección en la
tabla");
                printf("%d\n",dirección);
            }
            else
            {
                printf("Conflicto con %d otras
variables!\n",tabla_dispersión[dirección]);
```

```
                                printf("Dirección      de      la
tabla=%d\n",dirección);
                                }
                                tabla_dispersión[dirección]++;
                                }
                                else
                                fin++;
                                }
                                while(!fin);
}
int dispersión (char nombre[])
{
    int val, mitad;
    mitad=strlen(nombre)/2;
    val=nombre[0]+nombre[mitad]+nombre[strlen(nombre)-1];
    val>>=2;
    val&=31;
    return val;
}
```

EJERCICIO 85.

Escribir en C un programa que nos permita representar "cuadrados mágicos".

SOLUCION

Un cuadrado mágico es una matriz cuadrada con un número impar de filas y columnas, cuyas filas y columnas (e incluso sus diagonales) suman el mismo valor.

La técnica que se utiliza para generar estos los cuadrados mágicos es muy simple. Se comienza fijando a 1 el elemento central de la primera fila. A continuación se van escribiendo los sucesivos valores (2,3,...) desplazándose desde la posición anterior una fila hacia arriba y una columna hacia la izquierda. Estos desplazamientos se realizan tratando la matriz como si estuviera envuelta sobre sí misma, de forma que moverse una posición hacia arriba desde la fila superior lleva a la inferior y moverse una posición a la izquierda desde la primera columna conduce a la última. Si la nueva posición está ya ocupada, en lugar de desplazarse una fila hacia arriba, se moverá hacia abajo y continuará el proceso.

Puesto que no se permiten matrices con un tamaño par, el programa comprueba el valor introducido por el usuario para asegurar que se trata de un número impar. Según esto el algoritmo es:

```
# include <stdio.h>
main ()
{
    static int cuadrado_magico[9][9];
    int fila, columna, k, n, x, y;
    printf ("Introduzca el tamaño del cuadrado mágico ");
    scanf ("%d",&n);
    if (0==n%2)
    {
        printf("Lo siento, debe introducir un numero impar");
        exit(0);
    }
    k=2;
    fila=0;
    columna=(n-1)/2;
    cuadrado_magico[fila][columna]=1;
    while(k<=n*n)
    {
        x=(fila-1<0)?n-1:fila-1;
        y=(columna-1<0)?n-1:columna-1;
```

```
        if (cuadrado_mágico[x][y] != 0)
        {
            x = (fila+1 < n) ? fila+1 : 0;
            y = columna;
        }
        cuadrado_mágico[x][y] = k;
        fila = x;
        columna = y;
        k++;
    }
    for (fila = 0; fila < n; fila++)
    {
        for (columna = 0; columna < n; columna++)
            printf("\t%d", cuadrado_mágico[fila][columna]);
        printf("\n");
    }
}
```

EJERCICIO 86.

Diseñar en C un programa que nos permita mostrar un archivo de texto pantalla a pantalla. El programa se para hasta que el usuario pulse una tecla y entonces se muestra la siguiente página.

SOLUCION

Este programa es útil para ver archivos de texto largos. El algoritmo es el siguiente:

```
# include <stdio.h>
main (int argc, char *argv[])
{
    FILE *fptr;
    char fchar, temp;
    int lineas=0;
    if (argc>1)
    {
        fptr=fopen(argv[1]);
        if (fptr==NULL)
        {
            printf("No puedo abrir %s\n", argv[1]);
            exit();
        }
    }
    else
    {
        printf ("no se ha especificado ningun archivo.\n");
        exit();
    }
    fchar=getc(fptr);
    while (fchar!=EOF)
    {
        printf("%c", fcahr);
        if (fchar=='\n')
        {
            lineas++;
            if (lineas==23)
            {
                lineas=0;
            }
        }
    }
}
```

```
        printf("\nMas...");
        temp=getch();
        /*Borrar "Mas..." de la pantalla*/
        printf("\b\b\b\b\b\b\b\b");
        printf("      ");
        printf("\b\b\b\b\b\b\b\b");
    }
}
fchar=getc(fptr);
}
fclose(fptr);
}
```

EJERCICIO 87.

Diseñar en C un programa que sustituya cadenas de blancos por tabuladores para dejar un texto bien ajustado y viceversa.

SOLUCION

Para implementar este programa se utilizan tres funciones: *entabula*, para poner tabuladores, y *desentabula*, para quitar tabuladores, además de *treat* que realiza el procesamiento básico para las otras dos. El algoritmo es el siguiente:

```
# include <stdio.h>
int main (int argc, char *argv[])
{
    char bufin [SIZEBUF];
    char bufout[SIZEBUF];
    FILE *filein=NULL;
    int initial=1;
    boolean exiting=FALSE;
    boolean desentab=FALSE;
    boolean stdinend= TRUE;
    while(!exiting&&initial<argc)
    {
        if (strcmp(argv[initial],"-n")==0)
            desentab=TRUE;
        else
        {
            stdinend =FALSE;
            if(strcmp(argv[initial],"-")==0)
                filein=stdin;
            else
            {
                filein=fopen(argv[initial],"r");
                if(!filein)
                    perror(argv[0]);
            }
            if(filein)
            {
                treat(desentab?desentabula:entabula,
sizeof(bufin),bufin,bufout,filein,stdout);
```

```
        if(filien!=stdin)
            fclose(filein);
    }
}
initial++;
}
if(stdinend)
    treat(desentab?desentabula:entabula,sizeof(bufin),
bufin,bufout,stdin,stdout);
    exit(0);
}
```

EJERCICIO 88.**Implementar en C un formateador de texto.***SOLUCION*

El propósito de un formateador de texto es ajustar un bloque de texto, para determinar la forma la forma en que será mostrado por pantalla, o impreso, mediante la inserción del número adecuado de blancos entre palabras en cualquiera de las líneas de forma que todas ellas se ajusten perfectamente a los márgenes derecho e izquierdo predefinidos.

El algoritmo que implementa esta aplicación es:

```
# include <stdio.h>
# include <string.h>
# define EXP 1
# define NOEXP 0
# define ANCHO 45
void inicalmacen(char almacen[]);
int obtener(char datos[],int ptr, caher palabra[]);
void cargapalabra (char almacen[],caher palabra[]);
void expandir_linea(char almacen[],in como);
char text[ ] =
    "El microcomputador es una parte importante de cualquier
    curriculum de ingeniería. La maquina en si misma puede
    usarse para enseñar muchos temas del curso relacionados
    con el hardware, tales como interfaz de usuario, diseño de
    memoria y fundamentos de microprocesadores. Los paquetes
    de software disponibles actualmente ofrecen a los
    estudiantes una amplia variedad de aplicaciones,
    incluyendo gráficos, programación, y diseño analógico
    digital. Con todo ello, el microcomputador representa una
    ayuda apreciable para la enseñanza y la productividad.";
char alm_salida[80];
char siguiente palabra[80];
int espacio_restante, donde, siguiente;
main ()
{
    espacio_restante = ANCHO;
    donde = 0;
```

```

    inicalmacen(alm_salida);
do
{
    siguiente = obtener(text, donde, siguiente palabra);
    if(espacio_restante >= strlen(siguiente palabra));
    {
        cargapalabra(alm_salida,
siguiente palabra);
        donde = siguiente;
    }
    else
    {
        expandir_línea (alm_salida, EXP);
        inicalmacen (alm_salida);
        espacio_restante = ANCHO;
    }
}
while(0!=strlen(siguiente palabra));
if(0!=strlen(alm_salida))
    expandir_línea(alm_salida, NOEXP);
}

void inicalmacen(char almacen[])
{
    int z;
    for(z = 0; z <= ANCHO; z++)
        almacen[z] = '\0';
}

int obtener(char datos[], int ptr, char palabra[])
{
    int i=0;

    /* Saltar blancos entre palabras */
    while((datos[ptr] != ' ') && (datos[ptr]!='\0'))
        ptr++;
    /* Copiar caracteres hasta el blanco o el NULO */
    while((datos[ptr]!=' ') && (datos[ptr]!='\0'))
    {
        palabra[i] = datos[ptr];
        i++;
        ptr++;
    }
}

```

```

        palabra[i]='\0'; ;      /* Marcar el fin de la cadena*/
        return ptr;
    }
void cargapalabra(char almacen[], char palabra[])
{
    strcat(almacen,palabra) /* Copiar la palabra a la salida*/
        espacio_restante -= strlen(palabra); /*
        Ajustar el espacio restante del almacen de salida*/

    if(ANCHO > strlen(almacen)) /* Está el almacen salida
llenado?*/
    {
        strcat(almacen," "); /* Añadir blanco después de
palabra*/

        espacio_restante--;
    }
}
void expandir_linea(char almacen[], int como)
{
    int n,k,exp;
    if(como == 0) /* ¿No expandir alm_salida?*/
        printf("%s\n",almecen);
    {
        /* Llenar fin de alm_salida con NULOS */
        n = ANCHO;
        while((almacen[n] == '\0') || (almacen[n] == ' '))
        {
            almacen[n] = '\0';
            n -= 1;
        }
        /* Determinar el número de blancos a expandir*/
        exp = ANCHO - strlen(almacen);
        n=0;
        while(exp > 0) /* Mientras haya blancos que
insertar...*/
        {
            do /* Saltar palabra actual*/
            {
                n++;
            }while((almacen[n]!=' ' ) &&
(almacen[n]!='\0'));

```

```
        /*a cero el indice de alm_salida si está al
final*/
        if (almacen[n]=='\0')
            n=0;
        else
        {
            //si no insertar un blanco en alm_salida
            do
            {
                n++;
            } while (almacen[n]!=' ');

            for (k=ANCHO-1;k>n;k--)
                almacen[k]=almacen[k-1];
            almacen[b]=' ';
            n++;
            exp--; /*Un blanco menos a insertar*/
        }
    }
    printf("%s\n",almacen); /*imprimir la salida
expandida*/
}
```


EJERCICIO 89.

Programa que muestra en pantalla la tabla de códigos ASCII (ASCII extendido o ASCII de 8 bits).

SOLUCION

El algoritmo correspondiente a las especificaciones del enunciado se muestra a continuación:


```
# include <stdio.h>
main()
{
    int cont;
    for(cont=32;cont<=255;cont++)
    {
        if(cont%2==0)
            putchar('\n');
        printf("0x%x ... %3o ... %3d ... %c", cont, cont, cont
,cont);
    }
}
```


BIBLIOGRAFÍA

BIBLIOGRAFÍA

 Schildt, H. "C MANUAL DE REFERENCIA". Ed. McGraw-Hill.

 James L. Antonakos. "PROGRAMACIÓN ESTRUCTURADA EN C". Ed. Prentice Hall.

 Enrique Quero Catalinas. "PROGRAMACIÓN EN LENGUAJE C: EJERCICIOS Y PROBLEMAS". Ed. Paraninfo.

 Gottfried, B. S. "PROGRAMACIÓN EN C". Ed. McGraw-Hill.

