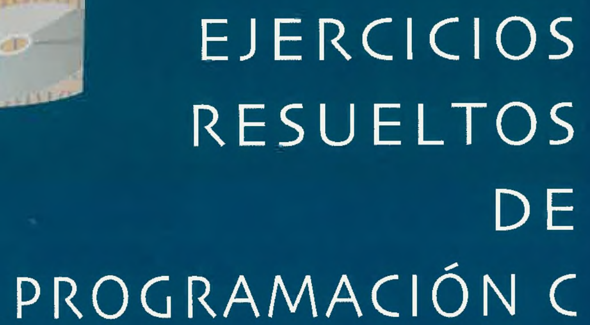


EJERCICIOS RESUELTOS DE PROGRAMACIÓN C

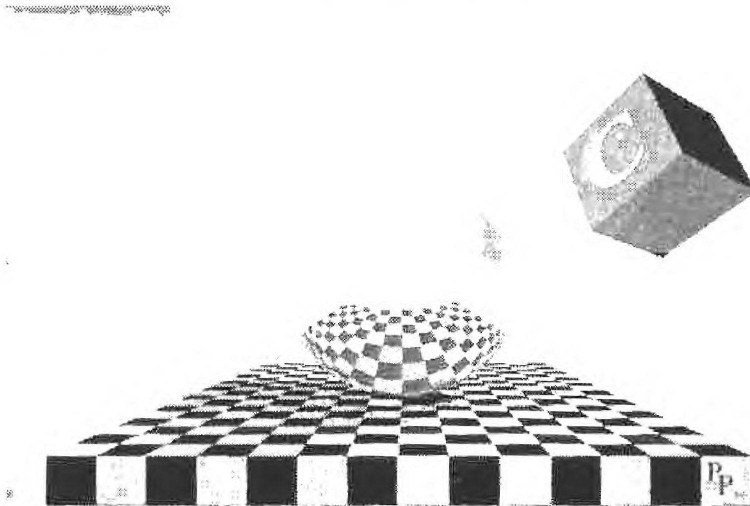
Pedro J. Sánchez Sánchez
José Galindo Gómez
Ignacio Turias Domínguez
Isidro Lloret Galiana

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE CÁDIZ

EJERCICIOS RESUELTOS DE PROGRAMACIÓN
C

Aprende a Programar, PROGRAMANDO

Pedro J. Sánchez Sánchez
José Galindo Gómez
Ignacio Turias Domínguez
Isidro Lloret Galiana



UNIVERSIDAD DE CÁDIZ
SERVICIO DE PUBLICACIONES
1997

Ejercicios resueltos de programación : C : aprende a programar, programando / Pedro J. Sánchez Sánchez ... [et al]. -- Cádiz : Universidad, Servicio de Publicaciones, 1997. -- 221 p.

1. Programación de ordenadores - Problemas, ejercicios, etc. I. Sánchez Sánchez, Pedro J. II. Universidad de Cádiz. Servicio de publicaciones. III. Título

681.3.066

I.S.B.N.: 84-7786-480-2

© SERVICIO DE PUBLICACIONES DE LA UNIVERSIDAD DE CÁDIZ

Pedro J. Sánchez Sánchez
José Galindo Gómez
Ignacio Turias Domínguez
Isidro Lloret Galiana

Diseño de Portada: Creasur, S.C.
I.S.B.N.: 84-7786-480-2
Imprime: Servicio de Autoedición e Impresión
Universidad de Cádiz

Índice General

Prólogo 5

Introducción al libro, Agradecimientos y Dedicatorias (3 en 1). 7

1 *Normas de Estilo* 9
- 1.1 Identificadores significativos 10
- 1.2 Constantes simbólicas 10
- 1.3 Comentarios, comentarios... 11
- 1.4 Estructura del programa 12
- 1.5 Indentación 14
- 1.6 Presentación 17

2 Ejercicios Básicos. 19
- 2.1 Operadores aritméticos. 19
- 2.2 Pitágoras. 20
- 2.3 La carretera. 21
- 2.4 El número π. 22
- 2.5 El número e. 23
- 2.6 Visualización de un valor con diferentes formatos. 23
- 2.7 Lectura y visualización de distintos tipos de datos. 24
- 2.8 Visualización de números reales. 25
- 2.9 Visualización de números reales con un tamaño determinado. 25
- 2.10 Visualización de caracteres como números. 26
- 2.11 Evaluación de expresiones con conversión automática de tipo. 27

3 Selección. 29
- 3.1 Par o impar. 29
- 3.2 Mayor de 3 enteros. 30
- 3.3 Mayor y menor de 3 enteros. 30
- 3.4 Sistema de ecuaciones con 2 incógnitas. 31
- 3.5 Conversión a mayúsculas. 32
- 3.6 Números divisibles. 33
- 3.7 Selección de destino de viaje por menú. 34
- 3.8 Operación entre números por menú. 35
- 3.9 Raíces reales de una ecuación de segundo grado. 36

4 Bucles. **39**
4.1 Tabla de caracteres ASCII. 39
4.2 Factorial. 40
4.3 Máximo Común Divisor. 41
4.4 Quitando espacios. 41
4.5 La pesca en la CEE. 42
4.6 Números Primos. 43
4.7 Números en un intervalo. 44
4.8 Mostrar números hasta una pulsación. 45
4.9 Factorización. 45

5 Arrays. **47**
5.1 MCD por argumentos en la línea de comandos. 47
5.2 Algoritmo para Ordenación de Arrays: Burbuja. 49
5.3 Media aritmética de 10 números. 50
5.4 Matriz traspuesta. 51
5.5 Búsqueda de números en un vector. 52
5.6 Manipulación de cadenas de caracteres. 53
5.7 Multiplicación de matrices 3x3. 54
5.8 Aproximación al número e. 56
5.9 Cálculo del centro de masas. 57
5.10 Lectura y comparación de cadenas y números. 58
5.11 Factorial con parámetros en la función `main()`. 59

6 Funciones para todo. **61**
6.1 Funciones en general. Conceptos básicos. 61
6.1.1 Función Perímetro. 61
6.1.2 Área de un círculo. 62
6.1.3 Operaciones aritméticas con funciones. 63
6.1.4 Conversión Km/h a m/s con función. 65
6.1.5 Mayor de dos números con función. 65
6.1.6 Números en un intervalo con función. 66
6.1.7 Cuadrante. 67
6.1.8 Años bisiestos. 69
6.1.9 Mínimo Común Múltiplo (MCM). 69
6.1.10 Números primos. 70
6.1.11 Raíces reales de una ecuación con funciones. 71
6.1.12 Fibonacci, iterativamente. 73
6.1.13 Función intercambio de valores: `swap()`. 74
6.1.14 Distancia euclídea. 74
6.1.15 Integral definida. 76
6.1.16 Operación XOR lógica. 77
6.1.17 El valor de las expresiones con punteros. 78
6.2 Funciones de cadenas de caracteres. 80
6.2.1 Comparación de cadenas. 80
6.2.2 Leer una cadena desde teclado. 81
6.2.3 Palíndromo, iterativamente. 82

6.2.4 Buscar subcadena. 84
6.2.5 Cadenas incluidas una en otra. 85
6.2.6 Obtener las Siglas de una cadena. 86
6.2.7 Cuenta caracteres. 88
6.2.8 Carácter más repetido en una cadena. 88
6.2.9 Manipulación de cadenas con menú de opciones. 89
6.2.10 Un `double` a cadena. 91
6.3 Funciones para manejo de arrays y matrices. 93
6.3.1 Máximo de un vector con funciones. 93
6.3.2 Posición del mayor valor. 94
6.3.3 Ordenación de un array por Selección. 95
6.3.4 Búsqueda Lineal en un array. 96
6.3.5 Búsqueda Binaria. 96
6.3.6 Cambio en monedas. 98
6.3.7 Relleno especial de matriz. 100
6.3.8 Producto especial de vectores. 102
6.3.9 Pares en un intervalo. 103
6.3.10 Media y varianza con funciones. 104
6.3.11 Media y desviaciones de un vector con funciones. 106
6.3.12 Lista de direcciones. 108
6.3.13 Búsqueda de personas en un vector de estructuras. 112
6.3.14 Copia de Matrices. 114
6.3.15 Matriz Traspuesta. 115
6.3.16 Comparar dos matrices. 116
6.3.17 Comprobar si una matriz es simétrica. 117
6.3.18 Multiplicación de matrices NxK por KxM. 119
6.3.19 Juego de las 3 en Raya (tic-tac-toe). 121
6.4 Funciones más complejas. 125
6.4.1 Visualizar la representación en binario de un entero. 125
6.4.2 Conversión de base general. 126
6.4.3 Intercambio entre valores de cualquier tipo. 127
6.4.4 Asignación de memoria dinámica. 129
6.4.5 Punteros a Funciones: Algo muy útil. 130

7 Manejo y Control de Ficheros. 135
7.1 Crear un fichero. 135
7.2 Mostrar un fichero: `type`. 136
7.3 Copiar de Ficheros: `copy`. 137
7.4 Borrar de un fichero: `del`. 138
7.5 Concatenación de ficheros. 139
7.6 Eliminación de tabuladores en un fichero. 140
7.7 Cambio de un carácter por otro en un fichero. 142
7.8 Lectura de datos desde fichero. 143
7.9 Cálculo del centro de masas con datos desde fichero. 144
7.10 Tamaño de un fichero 147
7.11 Inversión de un fichero. 148
7.12 Ordenar un fichero de palabras (en memoria). 149

7.13 Ordenacion de fichas en un archivo. 150

8 Conceptos básicos de Sonidos y Gráficos. 155

8.1 Noche de Paz. 155
8.2 Símbolo de la Paz. 159

9 Utilidades y programas más complejos. 163

9.1 Control del ratón para DOS. 163
9.2 Una calculadora sencilla. 170
9.3 Intérprete interactivo de una Máquina de Pila (calculadora). 186
9.4 Una simple Agenda. 199

A Tablas de consulta para programadores en C. 207

A.1 Tipos de datos del estándar ANSI C. 207
A.2 Caracteres de escape. 208
A.3 Operadores aritméticos. 208
A.4 Operadores relacionales. 209
A.5 Operadores lógicos. 209
A.6 Operadores a nivel de bits. 209
A.7 Precedencia de operadores. 210
A.8 Formatos para imprimir con `printf()` 211
A.9 Formatos para leer con `scanf()` . . 212
A.10 Frecuencias de las notas musicales. . 213

B *Reglas de Oro* de la Programación 215

C Bibliografía 217

C.1 Programación general 217
C.2 Programación en C básica 218
C.3 Programación C avanzada 220

Prólogo

Ayudar a la repoblación forestal, contribuir a la explosión demográfica, y dejar algún legajo, son algunas de las obligaciones que todo ser civilizado tiene para con este mundo.

Sobre el primer deber, estoy seguro que los autores algo habrán plantado en algún sitio, aunque sea un jaramago.

En relación con la segunda, lo del hijo, es sabido de uno de ellos, Pepe Galindo. De Pedro, por lo menos no consta.

Algunos comparan la segunda de las obligaciones con la tercera. Hablan de un libro como si fuera un hijo del autor o autores. Si comparamos el libro, por ejemplo, con Patricia, la hija de Pepe, (con la necesaria e inestimable colaboración de su esposa), pierde el libro seguro. Un ángel rubio, mucho más importante que cualquier conjunto de reglas y símbolos que conforman un libro de un lenguaje de programación. Si alguien piensa que el C es más importante, es que no la conoce.

De la última de las obligaciones, nuestros amigos son reincidentes. El presente libro sobre "ejercicios resueltos de C" no es su "ópera prima".

José Galindo Gómez es "Pp". Pedro Sánchez Sánchez es "Pedro J." (No tengo yo muy claro el motivo de la J). Ambos son Licenciados/Ingenieros en Informática por la Universidad de Granada, en tránsito por la Universidad de Cádiz.

> "Por caminos cifrados y confusos
>
> llegamos hasta aquí gente de fuera
>
> y fabuloso empeño.
>
> Nos atienden con mínimas palabras,
>
> experto en el trato de la imaginación ajena.
>
> (Si algo en común tenemos
>
> los duendes malheridos de esta casa
>
> es pasión por lo exacto y virtuoso.)..."
>
> La creación – Alfonso Sánchez Ferrajón –

Estos dos autores forman una pareja singular. Algo así como el gordo y el flaco. Enjuto uno, orondo otro. Dejamos al lector que descubra quien es el gordo y quien es el flaco.

Además, están Ignacio e Isidro o Isidro e Ignacio, "los algecireños".

Sobre la materia del libro, tenemos que reconocer que la informática ha conseguido algo inaudito: cambiar de sexo a una de las letras de nuestro abecedario. Todos conocíamos a

"la C". La tercera letra. Ahora tenemos "El C". La transformación no ha sido únicamente transexual. Los informáticos hemos conseguido condecorarla. Tenemos el C y el C++, con dos estrellas.

La obra no solo pretende, sino que consigue acercar el lenguaje C a las necesidades de un amplio colectivo. A lo largo de sus ocho capítulos y un centenar y medio de páginas, podemos encontrar un variado conjunto de ejercicios propuestos y resueltos. Por medio de los ejercicios, exponen los elementos del lenguaje C, así como las normas para construir un buen programa. Está dirigido a estudiantes universitarios que quieran adentrarse en el fantástico mundo de la programación de la mano del lenguaje C están dar, de un modo práctico. Destacamos del amplio número de ejercicios, la adecuada selección al nivel pretendido.

Con esta obra se continúa la contribución en la formación de nuestros estudiantes de informática. La formación es un derecho y un deber al mismo tiempo. La profesionalidad hacen de la formación, no sólo una actividad de necesaria utilidad, sino que la convierte en un elemento estratégico para conseguir los objetivos de eficacia, eficiencia y actualización permanente. No solo es la formación transmisión de conocimientos, también es un instrumento para desarrollar capacidades y promover actitudes.

El departamento de informática, de la mano de "Pp", "Pedro J." y "los algecireños" presenta otra obra más sobre el C, fruto del trabajo de cuatro de sus miembros. Nosotros hemos cumplido con nuestra parte.

Para sacar el fruto pretendido al libro es necesario trabajarlo. Primero hay que intentar realizar los ejercicios. No solo copiarlos. Ahora, señor/a lector/a, te toca trabajar a ti. Ánimo.

MANUEL FERNÁNDEZ BARCELL
Director del Departamento de Lenguajes y Sistemas Informáticos
de la Universidad de Cádiz.
Un defecto como otro cualquiera.

Introducción al libro, agradecimientos y dedicatorias.

Este libro pretende ser una ayuda y un estímulo a aquellos que han decidido aprender a programar en C. **Para aprender a programar hay que programar**. No hay mejor sistema (y quizás no hay otro). Naturalmente, es fundamental o, al menos, muy importante tener un buen profesor y/o un buen libro.

La bibliografía sobre el Lenguaje C es vasta, muy vasta. En un apéndice de esta obra ofrecemos un resumen de algunos de los libros más interesantes que hemos visto, agrupados por su nivel de dificultad. Algunos libros son realmente muy buenos y en todos se incluyen ejemplos para que se vea con claridad cómo funciona este lenguaje. Sin embargo, es muy frecuente que los neófitos en programación quieran o necesiten efectuar prácticas con otros programas y no saben o no se les ocurren qué tipo de programas pueden hacer y cómo hacerlos. Este libro pretende paliar, en parte, esa deficiencia, proponiendo ejercicios de muy diversa índole, agrupados en varios capítulos y ordenamos de menor a mayor dificultad. Además, de cada ejercicio se expone al menos una solución y se explica esa solución para solventar las dudas más frecuentes.

Esta obra está dividida en los siguientes capítulos:

1. Normas de estilo: Se explican detalladamente algunas de las normas más importantes para programar adecuadamente. Siguiendo estas normas se harán programas más legibles y más humanos.

2. Ejercicios básicos: Aquí se exponen una serie de ejercicios básicos en los que se hace incapié en la estructura general de un programa en C, las funciones de Entrada/Salida más típicas y los operadores más importantes.

3. Selección: Ejercicios un poco más complejos que usan los comandos de selección de los que dispone el C.

4. Bucles: Ejercicios en los que es necesario la iteración, es decir, repetir varias veces las mismas operaciones modificando algún valor (variable).

5. Arrays: Ejercicios centrados en el manejo de estructuras de datos array, de cualquier tipo, y en especial, las cadenas de caracteres.

6. Funciones para todo: Este es el tema más importante de todos. Utiliza todo lo visto en temas anteriores pero construyendo nuestras propias funciones. En C todo son funciones y es muy importante (como se explica en el capítulo 1) modularizar bien un programa. O sea, se deben construir tantas funciones como sean necesarias para que la legibilidad

del programa sea aceptable. Por eso, a este tema es al que le damos más importancia. Se pretende, entre otras cosas, que se vea claramente que cualquier cosa se puede hacer creando una función a la cual le damos unos valores y nos devuelve otros.

7. Ficheros: Un tipo de dato fundamental y muy útil. Aquí se exponen desde ejercicios muy básicos hasta funciones muy útiles que podremos necesitar en nuestros programas (cálculo del tamaño de un fichero...).

8. Conceptos básicos de sonidos y gráficos: A este tema no le hemos prestado demasiada importancia por considerar que todo lo relacionado con este tema no es estándar y las funciones que utilizan no son del estándar ANSI C. En este tema nos centramos en ejercicios para una arquitectura PC.

9. Utilidades y programas más complejos: En este capítulo se expondrán algunas aplicaciones más grandes en las que se vea cómo construir un programa completo que utilice todo lo necesario para conseguir el objetivo. Se explicarán, entre otras cosas cómo manejar el ratón desde un programa C y cómo construir una agenda electrónica muy sencilla.

En los Apéndices se añade información muy útil, cómo son un conjunto de tablas muy útiles para los programadores en C o una lista de Reglas de Oro para la programación, enumeradas sin perder el buen humor.

Por su estructura, este libro puede también verse como un elenco de programas que pueden ser usados por profesores (universitarios o no), para que éstos se los propongan a sus alumnos y estos los realizen en las clases prácticas. Por supuesto, también esto puede ser una fuente de preguntas de examen. Nosotros, los autores, como profesores de las Universidades de Cádiz, aseguramos que muchas de estas preguntas han sido preguntadas en algunos de nuestros exámenes, tanto teóricos como prácticos.

Muchos de los ejercicios aquí propuestos ya los teníamos preparados para nuestras clases y nuestros alumnos. El recopilarlos, explicarlos, clasificarlos, ordenarlos e idear nuevos ejercicios que fueran interesantes y didácticos, ha sido una tarea bastante más árdua de lo que nos pareció en principio, pero esperamos que haya valido la pena. Al menos, siempre queda aquello de ***Verba volant, scripta manent*** (Las palabras se las lleva el viento, lo escrito permanece). Y permanece aún más si tiene ISBN y Depósito Legal.

Queremos aprovechar este espacio para agradecer a todos los miembros del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz, que nos han apoyado y animado en todo momento. Si los nombráramos seguro que se nos olvidaría alguien. También, dar las gracias a toda la Universidad de Cádiz por que nos ha echo crecer y madurar, y aunque hay cosas que sólo pasan en la UCA, ahora no queremos recordarlas.

Para terminar, y cambiando de tema, nos gustaría dedicar este libro a todos aquellos que luchan firme y pacíficamente por la justicia siguiendo el ejemplo de M. K. Ghandi.

LOS AUTORES.

Capítulo 1

Normas de Estilo

Independientemente de los algoritmos usados, hay muchas formas y estilos de programar. La legibilidad de un programa es demasiado importante como para no dedicarle, al menos, un capítulo de esta obra.

Los programas, a lo largo de su vida, se van quedando obsoletos debido a cambios en su entorno. Un programa pensado para una determinada actividad, es muy normal tener que modificarlo porque cambie dicha actividad o porque decidamos incluirle nuevas posibilidades que antes no estaban previstas. De aquí las múltiples versiones que sacan al mercado las empresas de programación. Además, debido a la dificultad de algunos programas para probarlos exhaustivamente, a veces, se descubren errores cuando el programa lleva funcionando cierto tiempo.

Es entonces, cuando hay que modificar un programa escrito hace cierto tiempo, cuando salen los problemas de legibilidad de un programa. Para poder modificarlo, primero hay que comprender su funcionamiento, y para facilitar esta tarea el programa debe estar escrito siguiendo unas normas básicas. La tarea de mantenimiento del software (corregir y ampliar los programas) es una de las tareas más árduas del ciclo de vida del software, y al programar debemos intentar que nuestros programas sean lo más expresivos posibles, para ahorrarnos tiempo, dinero y quebraderos de cabeza a la hora de modificarlos.

Además, el C es un lenguaje que se presta a hacer programas complejos y difíciles de comprender. En C se pueden encapsular órdenes y operadores, de tal forma que, aunque consigamos mayor eficiencia su comprensión sea todo un reto. Como curiosidad, diremos que en EEUU existe un concurso de programación en C, en el que gana el que consiga hacer el programa más críptico.

Resumiendo y repitiendo, diremos que la claridad en un programa es de vital importancia, pues la mayor parte del tiempo de mantenimiento de un programa se emplea en estudiar y comprender el código fuente existente. Esto es especialmente importante cuando se trabaja en grupo donde posiblemente un programador tenga que corregir un programa que hizo hace ocho meses otro programador, que no está disponible para preguntarle dudas. Naturalmente, esto es más importante cuantas más líneas de código tenga el programa, pero incluso en programas pequeños puede perderse mucho tiempo intentando comprender su funcionamiento, si no están programados con mimo.

Unas normas de estilo en programación, son tan importantes que todas las empresas dedicadas a programación imponen a sus empleados una mínima uniformidad, para facilitar el intercambio de programas y la modificación por cualquier empleado, sea o no el programador

inicial. Por supuesto, cada programa debe ir acompañado de una documentación adicional, que aclare detalladamente cada módulo del programa, objetivos, algoritmos usados...

No existen un conjunto de reglas fijas para programar con legibilidad, ya que cada programador tiene su modo y sus manías y le gusta escribir de una forma determinada. Lo que sí existen son un conjunto de reglas generales, que aplicándolas, en mayor o menor medida, se consiguen programas bastante legibles. Aquí intentaremos resumir estas reglas.

1.1 Identificadores significativos

Un identificador es un nombre asociado a un objeto de programa, que puede ser una variable, función, constante, tipo de datos... El nombre de cada identificador debe *identificar* lo más claramente posible al objeto que identifica, valga la redundancia. Normalmente los identificadores deben empezar por una letra, no pueden contener espacios (ni símbolos raros) y suelen tener una longitud máxima que puede variar, pero que no debería superar los 10-20 caracteres para evitar lecturas muy pesadas.

Un identificador debe indicar lo más breve y claramente posible el objeto al que referencia. Por ejemplo, si una variable contiene la nota de un alumno de informática, la variable se puede llamar `nota_informatica`. Observe que no ponemos los acentos, los cuales pueden dar problemas de compatibilidad en algunos sistemas. El carácter `'_'` es muy usado para separar palabras en los identificadores.

Es muy normal usar variables como `i`, `j` o `k` para nombres de índices de bucles (`for`, `while`...), lo cual es aceptable siempre que la variable sirva sólo para el bucle y no tenga un significado especial. En determinados casos, dentro de una función o programa pequeño, se pueden usar este tipo de variables, si no crean problemas de comprensión, pero esto no es muy recomendable.

Para los identificadores de función se suelen usar las formas de los verbos en infinitivo, seguido de algún sustantivo, para indicar claramente lo que hace. Por ejemplo, una función podría llamarse `Escribir_Opciones`, y sería más comprensible que si le hubiéramos llamado `Escribir`, `Opciones` o `EscrOpc`. Si la función devuelve un valor, su nombre debe hacer referencia a este valor, para que sea más expresivo usar la función en algunas expresiones, como:

```
Precio_Total = (Precio_Unidad * Cantidad);
Precio_Total = Precio_Total + IVA(Precio_Total,16) +
               Gastos_Transporte(Destino);
```

1.2 Constantes simbólicas

En un programa es muy normal usar constantes (numéricas, cadenas...). Si estas constantes las usamos directamente en el programa, el programa funcionará, pero es más recomendable usar constantes simbólicas, de forma que las definimos al principio del programa y luego las usamos cuando haga falta. Así, conseguimos dos ventajas principalmente:

- Los programas se hacen más legibles: Es más legible usar la constante simbólica `PI` como el valor de π que usar 3.14 en su lugar:

```
Volumen_Esfera = 4/3. * PI * pow(radio,3);
```

- Los programas serán más fáciles de modificar: Si en un momento dado necesitamos usar `PI` con más decimales (3.141592) sólo tenemos que cambiar la definición, y no tenemos que cambiar todas las ocurrencias de 3.14 por 3.141592 que sería más costoso y podemos olvidarnos alguna.

En C, las constantes simbólicas se suelen poner usando una órden al Preprocesador de C, quedando definidas desde el lugar en que se definen hasta el final del fichero (o hasta que expresamente se indique). Su formato general es:

```
#define CONSTANTE valor
```

que se encarga de cambiar todas las ocurrencias de `CONSTANTE` por el valor indicado en la segunda palabra (`valor`). Este cambio lo realiza el preprocesador de C, antes de empezar la compilación. Por ejemplo:

```
#define PI 3.141592
```

Por convenio, las macros se suelen poner completamente en mayúsculas y las variables no, de forma que leyendo el programa podamos saber rápidamente qué es cada cosa. En general, se deben usar constantes simbólicas en constantes que aparezcan más de una vez en el programa referidas a un mismo ente que pueda variar ocasionalmente. Obsérvese, que aunque el valor de π es constante, podemos variar su precisión, por lo que es recomendable usar una constante simbólica en este caso, sobre todo si se va a usar en más de una ocasión en nuestro programa.

1.3 Comentarios, comentarios...

El uso de comentarios en un programa escrito en un lenguaje de alto nivel es una de las ventajas más importantes con respecto a los lenguajes máquina, además de otras más obvias. Los comentarios sirven para aumentar la claridad de un programa, ayudan para la documentación y bien utilizados nos pueden ahorrar mucho tiempo.

No se debe abusar de *comentarista*, ya que esto puede causar una larga y tediosa lectura del programa, pero en caso de duda es mejor poner comentarios de más. Por ejemplo, es absurdo poner:

```
Nota = 10; /* Asignamos 10 a la variable Nota */
```

Los comentarios deben ser lo más breve posible y evitando divagaciones. Se deben poner comentarios, cuando se crean necesarios, y sobre todo:

- En cada sentencia o bloque (bucle, `if`, `switch`...) que revista cierta complejidad, de forma que indique en el comentario qué se realiza o cómo funciona.

- Al principio de cada función cuyo nombre no explique suficientemente su cometido. Se debe poner no sólo lo que hace sino la utilidad de cada parámetro, el valor que devuelve (si lo hubiera) y si fuera oportuno, los requisitos necesarios para que dicha función opere correctamente.

- En la declaración de variables y constantes cuyo identificador no sea suficiente para comprender su utilidad.

- En los cierres de bloques (`}`), para indicar a qué sentencias de control de flujo pertenecen, principalmente cuando existe mucho anidamiento de sentencias y/o los bloques contienen muchas líneas de código.

- También se suele incluir un bloque de comentarios de presentación al principio del programa o de cada fichero del programa que permita seguir un poco la historia de cada programa, indicando: Nombre del programa, objetivo, parámetros (si los tiene), condiciones de ejecución, módulos que lo componen, autor o autores, fecha de finalización, últimas modificaciones realizadas y sus fechas... y cualquier otra eventualidad que el programador quiera dejar constancia.

No olvidemos que los comentarios son textos literarios, por lo que debemos cuidar el estilo, acentos y signos de puntuación.

1.4 Estructura del programa

Un programa debe ser claro, estar bien organizado y que sea fácil de leer y entender. Casi todos los lenguajes de programación son de formato libre, de manera que los espacios no importan, y podemos organizar el código del programa como más nos interese.

Para aumentar la claridad no se deben escribir líneas muy largas (que se salgan de la pantalla) y funciones con muchas líneas de código (especialmente la función principal). Una función demasiado grande demuestra, en general, una programación descuidada y un análisis del problema poco estudiado. Se deberá, en tal caso, dividir el bloque en varias llamadas a otras funciones más simples, para que su lectura sea más agradable. En general se debe modularizar siempre que se pueda, de forma que el programa principal llame a las funciones más generales, y estas vayan llamando a otras, hasta llegar a las funciones primitivas más simples. Esto sigue el principio de *divide y vencerás*, mediante el cual es más fácil solucionar un problema dividiéndolo en subproblemas (funciones) más simples.

A veces, es conveniente usar paréntesis en las expresiones, aunque no sean necesarios, para aumentar la claridad.

Cada bloque de especial importancia o significación, y cada función debe separarse de la siguiente con una línea en blanco. A veces, entre cada función se añaden una linea de asteriscos o guiones, como comentario, para destacar que empieza la implementación de otra función.

El uso de la sentencia `GOTO` debe ser restringido al máximo, pues lía los programas y los hace ilegibles. Como dicen Kerninghan y Ritchie en su libro de C, su utilización sólo está justificada en casos muy especiales, como salir de una estructura profundamente anidada, aunque para ello podemos, a veces, usar los comandos `break`, `continue`, `return` o la función `exit()`. Igualmente, esos recursos, para salir de bucles anidados deben usarse lo menos posible y sólo en casos que quede bien clara su finalidad. Si no queda clara su finalidad, será mejor

Figura 1.1: Antes de programar, se debe pensar qué algoritmo usar.

que nos planteemos de nuevo el problema y estudiemos otra posible solución que seguro que la hay.

Normalmente, un programa en C se suele estructurar de la siguiente forma:

- Primero los comentarios de presentación, como ya hemos indicado.
- Después, la inclusión de bibliotecas del sistema, los ficheros `.h` con el `#include` y entre ángulos (`<...>`) el nombre del fichero. Quizás la más típica sea:

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

- Bibliotecas de la aplicación. Normalmente, en grandes aplicaciones, se suelen realizar varias librerías con funciones, separadas por su semántica. Los nombres de fichero se ponen entre comillas (para que no las busque en el directorio de las librerías estándar) y se puede incluir un comentario aclarativo:

```
#include "rata.h"  /* Rutinas para control del ratón        */
#include "cola.h"  /* Primitivas para el manejo de una cola */
```

- Variables globales, usadas en el módulo y declaradas en otro módulo distinto, con la palabra reservada `extern`.

- Constantes simbólicas y definiciones de macros, con `#define`.
- Definiciones de tipos, con `typedef`.
- Declaración de funciones del módulo: Se escribirá sólo el prototipo de la función, no su implementación. De esta forma, el programa (y el programador) sabrá el número y el tipo de cada parámetro y cada función.
- Declaración de variables globales del módulo: Se trata de las variables globales declaradas aquí, y que si se usan en otro módulo deberán llevar, en el otro módulo, la palabra `extern`.
- Implementación de funciones: Aquí se programarán las acciones de cada función, incluida la función principal. Normalmente, si el módulo incluye la función principal, la función `main()`, ésta se pone la primera, aunque a veces se pone al final y nunca en medio. El resto de funciones, se suelen ordenar por orden de aparición en la función principal y poner juntas las funciones que son llamadas desde otras. Es una buena medida, que aparezcan en el mismo orden que sus prototipos, ya que así puede ser más fácil localizarlas.

Naturalmente, este orden no es estricto y pueden cambiarse algunos puntos por otros, pero debemos ser coherentes, y usar el mismo orden en todos los módulos. Otro punto muy importante es el referente a variables globales. En general es mejor no usar nunca este tipo de variables, salvo que sean variables que se usen en gran parte de las funciones (y módulos) y esté bien definida y controlada su utilidad. El uso de estas variables puede dar lugar a los llamados *efectos laterales*, que provienen de la modificación indebida de una de estas variables en algún módulo desconocido. Lo mejor es no usar nunca variables globales y pasar su valor por parámetros a las funciones que estrictamente lo necesiten, viendo así las funciones como *cajas negras*, a las que se le pasan unos determinados datos y nos devuelve otros, perfectamente conocidos y expresados en sus parámetros.

Por la misma razón que no debemos usar variables globales, no se deben usar pasos de parámetros por referencia (variable), cuando no sea necesario.

1.5 Indentación

La indentación o sangrado consiste en marginar hacia la derecha todas las sentencias de una misma función o bloque, de forma que se vea rápidamente cuales pertenecen al bloque y cuales no. Algunos estudios indican que el indentado debe hacerse con 2, 3 ó 4 espacios. Usar más espacios no aumenta la claridad y puede originar que las líneas se salgan de la pantalla, complicando su lectura.

La indentación es muy importante para que el lector/programador no pierda la estructura del programa debido a los posibles anidamientos.

Normalmente, la llave de comienzo de una estructura de control (`{`) se pone al final de la linea y la que lo cierra (`}`) justo debajo de donde comienza —como veremos más adelante— pero algunos programadores prefieren poner la llave `{` en la misma columna que la llave `}`, quedando una encima de otra. Eso suele hacerse así, en la implementación de funciones, donde la llave de apertura de la función se suele poner en la primera columna.

Veamos, a continuación, la indentación típica en las estructuras de control:

Sentencia `if-else`

```
if (condición) {
   sentencia1;
   sentencia2;
   ...
}
else {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `swith`

```
switch (expresión) {
  case expresión1: sentencia1;
                   sentencia2;
                   ...
                   break;
  case expresión2: sentencia1;
                   sentencia2;
                   ...
                   break;
  .
  .
  .
  case default   : sentencia1;
                   sentencia2;
                   ...
}
```

Sentencia `for`

```
for (exp1;exp2;exp3) {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `while`

```
while (condición) {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `do-while`

```
do {
   sentencia1;
   sentencia2;
   ...
} while (condición),
```

Aunque estos formatos no son en absoluto fijos, lo que es muy importante es que quede bien claro las sentencias que pertenecen a cada bloque, o lo que es lo mismo, donde empieza y termina cada bloque. En bloques con muchas lineas de código y/o con muchos anidamientos, se recomienda añadir un comentario al final de cada llave de cierre del bloque, indicando a qué sentencia cierra:

```
for (exp1;exp2;exp3) {
   sentencia1;
   sentencia2;
   ...

   while (condición_1) {
      sentencia1;
      sentencia2;
      ...
      if (condición) {
         sentencia1;
         sentencia2;
         ...

         while (condición_2) {
           sentencia1;
           sentencia2;
           ...
         } /*while (condición_2)*/

      } /*if*/
      else {
```

```
            sentencia1;
            sentencia2;
            ...

            if (condición) {
               sentencia1;
               sentencia2;
               ...
            }
            else {
               sentencia1;
               sentencia2;
               ...
            }

         } /*else*/
      } /*while (condición_1)*/
   } /*for*/
```

1.6 Presentación

Al hacer un programa debemos tener en cuenta quien o quienes van a usarlo o pueden llegar a usarlo, de forma que el intercambio de información entre dichos usuarios y el programa sea de la forma más cómoda, clara y eficaz posible.

En general, se debe suponer que el usuario no es un experto en la materia, por lo que se debe implementar un interfaz que sea fácil de usar y de aprender, intuitivo y que permita efectuar la ejecución de la forma más rápida posible.

Para ello, se suelen usar las siguientes técnicas:

- Usar argumentos en la línea de comandos para especificar las distintas opciones posibles.
- Avisar lo más detalladamente posible de todos los errores que se produzcan, los motivos de los mismos y cómo solucionarlos.
- Incluir una **ayuda** en línea (*on line*) en el programa, de forma que el usuario pueda consultar, en cualquier parte del programa, cómo ejecutar cada una de las acciones posibles en cada momento. Un estándar de hecho es usar la tecla de función `F1` para solicitar esta ayuda.
- Es muy usual incluir el argumento ? (o con una barra o guion delante) para solicitar que el programa nos muestre una ayuda sobre su funcionamiento, opciones, ficheros usados... Si el programa requiere algunos argumentos obligatorios, se puede mostrar esa ayuda si falla alguno de esos argumentos.
- Usar las teclas estándares: `ESC` para salir o deshacer las modificaciones hechas, `RePág` y `AvPág` para retroceder y avanzar una página, `Supr` (o `Del`) para borrar, las flechas para seleccionar o cambiar de posición, el tabulador...

- Usar las teclas de funciones (`F1`, `F2`...) para las operaciones más usuales y combinaciones de teclas para acceder a otras operaciones de forma rápida (como `Alt-P`, `Ctrl-F1`...).

- Cualquier programa debe ir siempre acompañado de un **manual de usuario**, en el que se explique detalladamente todo lo referente a la ejecución del programa: Requisitos mínimos de ejecución (de procesador, memoria RAM necesaria, espacio en disco que ocupa...), parámetros disponibles, ficheros que usa, mensajes de error, soluciones a los errores, variables de entorno (si las usa), ejemplos (si procede)...

- Ultimamente, la ejecución y manejo de programas se ha facilitado mucho con la llegada de entornos gráficos, en los que se usan comúnmente menús de opciones, iconos (pequeños dibujos), botones... Además, el manejo del ratón (*mouse*) hace más intuitivo y cómodo el manejo de determinados programas (aunque no sean en un entorno gráfico).

- Permitir al usuario volver hacia atrás siempre que lo desee: Deshacer algo o detener su ejecución. Es muy frecuente que ejecutando un programa se necesite modificar un dato que hemos indicado anteriormente o que simplemente deseemos ver algo anterior. Un buen programa debe permitir volver a cualquier punto anterior para mostrar o corregir cualquier dato, o para interrumpir la ejecución de algo que no se desea que continue. Para ello es muy frecuente el uso de la tecla de escape o `ESC` (carácter `'\x1B'` en un PC).

Capítulo 2

Ejercicios Básicos.

En éste capítulo y en los siguientes lo que pretendemos es exponer una serie de ejercicios. Cada uno de éstos viene resuelto (no se trata de la única solución posible). El nivel de dificultad de los ejercicios se ha intentado que fuera creciente dentro del capítulo, lo mismo que a lo largo del libro.

Recomendamos a aquellos que lean este libro que lo primero que deben de hacer es intentar resolver cada uno de los ejercicios por su cuenta. En el caso de no dar con la solución, es el momento de dar un vistazo a la solución propuesta para ver donde nos habíamos atascado.

Éste es el primer capítulo de ejercicios, en él se presenta cual es la estructura básica de un programa en *C*. Los únicos elementos del lenguaje que vamos a utilizar son los operadores básicos y las funciones básicas de entrada y salida de datos.

2.1 Operadores aritméticos.

Hacer un programa que pida por teclado dos números enteros e imprima en pantalla su suma, resta, multiplicación, división y el resto (módulo) de la división. Si la operación no es conmutativa, también se mostrará el resultado, invirtiendo los operadores.

Solución:

```
/****************************************************************/
/* Objetivo: Prueba y manejo de operadores aritméticos en C.  */
/* Entrada : Dos valores enteros.                              */
/* Salida  : Suma, Resta, Producto, División y Resto.          */
/****************************************************************/
#include<stdio.h>

void main()
{ int x,y;

  puts("\n ***** Operaciones *****");
  printf("\n- Dame un número entero: "),
  scanf("%i",&x);
  printf("\n- Dame otro entero: "),
```

```
  scanf("%i",&y);

  printf("\nSuma     : %i + %i = %i", x, y, x+y );
  printf("\nResta    : %i - %i = %i", x, y, x-y );
  printf("\n           %i - %i = %i", y, x, y-x );
  printf("\nProducto: %i * %i = %i", x, y, x*y );
  printf("\nDivisión: %i / %i = %f", x, y, x/(float)y );
  printf("\n           %i / %i = %f", y, x, y/(float)x );
  printf("\nMódulo   : %i %% %i = %i", x, y, x%y );
  printf("\n           %i %% %i = %i", y, x, y%x );

  puts("\n ***** FIN *****");
}
```

Obsérvese, como se usa un molde (*casting*) en la división. Esto se hace para convertir el denominador de tipo `int` a tipo `float`, de forma que la división no sea efectuada con decimales y el resultado sea mostrado más exactamente. Elimine el molde `(float)` y observe qué el resultado que se muestra es sólo la parte entera de la división. Para que el resultado de una división sea de tipo `float`, al menos un operando debe ser de ese tipo (o convertido a ese tipo).

2.2 Pitágoras.

Se desea averiguar la distancia euclídea de dos puntos en el plano (usando el teorema de Pitágoras), dando las coordenadas de sendos puntos en un eje cartesiano de coordenadas: `(x1,y1)` y `(x2,y2)`.

Solución:

```
/****************************************************/
/* Objetivo: Distancia euclídea.                    */
/* Entrada : Dos puntos cartesianos.                */
/* Salida  : Su distancia euclídea.                 */
/****************************************************/
#include <stdio.h>
#include <math.h>

void main()
{ double x1, y1, x2, y2;

  puts("\n ***** Distancia de dos puntos *****");
  printf("\n- Punto 1: x1 = "); scanf("%f",&x1);
  printf("\n-          y1 = "); scanf("%f",&y1);
  printf("\n- Punto 2: x2 = "); scanf("%f",&x2);
  printf("\n-          y2 = "); scanf("%f",&y2);

  x1=x1-x2;
```

```
  y1=y1-y2;
  printf("\n* La distancia es: %f\n", sqrt( (x1*x1)+(y1*y1) ));
  puts("\n ***** FIN *****");
}
```

La función `sqrt()` nos devuelve la raíz cuadrada de su argumento, y está incluida en la biblioteca `math.h`.

2.3 La carretera.

Hacer un programa que pida el total de kilómetros recorridos, el precio de la gasolina (por litro), el dinero de gasolina gastado en el viaje y el tiempo que se ha tardado (en horas y minutos), y que calcule:

- Consumo de gasolina (en litros y pesetas) por cada cien kilómetros.
- Consumo de gasolina (en litros y pesetas) por cada kilómetro.
- Velocidad media (en km/h y m/s).

Solución:

```
/*************************************************************/
/* Objetivo: Control del consumo de gasolina y velocidad media. */
/* Entrada : Km. recorridos, Precio gasolina, dinero gastado y  */
/*           Tiempo empleado en el viaje.                       */
/* Salida  : Consumo de gasolina cada km. y cada 100 km., y     */
/*           Velocidad media en el viaje.                       */
/*************************************************************/
#include <stdio.h>

void main()
{ float km,                /* Kilómetros recorridos        */
        preciolitro,       /* Precio por litro de gasofa   */
        litrosgastados,    /* Litros consumidos en el viaje */
        litros_km,         /* Litros consumidos por cada km */
        pelas_km;          /* Pelas gastadas por cada km    */
  int horas, minutos, dinero;

  puts("      -*****- LA  CARRETERA  -*****-\n");

  printf("- Total de Km. recorridos: "); scanf("%f",&km);
  printf("- Tiempo tardado: HORAS  : "); scanf("%i",&horas);
  printf("                  MINUTOS: "); scanf("%i",&minutos);

  minutos = 60*horas+minutos;

  printf("- Precio por litro de gasolina: "); scanf("%f",&preciolitro);
```

```
  printf("- Dinero gastado en gasolina  : "); scanf("%i",&dinero);
  litrosgastados = dinero/preciolitro;

  puts("\n  ***** RESULTADOS *****");

  printf("- Velocidad media       : %f km/h.\n",(km/minutos)*60);
  printf("- Total litros gastados: %f litros.\n",litrosgastados);
  printf("- Consumo de gasolina cada kilómetro: %f litros.\n",
          litros_km=litrosgastados/km);
  printf("                                      %f pesetas.\n",
          pelas_km=litros_km*preciolitro);
  printf("- Consumo de gasolina cada  100 Kms.: %f litros.\n",
          litros_km*100);
  printf("                                      %f pesetas.\n",
          pelas_km*100);

  getch();
  puts ("\n***** FIN *****");
}
```

Obsérvese como se puede usar una expresión de asignación dentro de un `printf()`. Recordemos que en C, toda expresión tiene siempre un valor. En este caso, una expresión de asignación toma el valor que asigna, por lo que serían equivalentes los siguientes fragmentos de programa:

```
printf("- Consumo de gasolina cada kilómetro: %f litros.\n",
        litros_km=litrosgastados/km);
```

y, primero asignar y luego visualizar:

```
litros_km=litrosgastados/km;
printf("- Consumo de gasolina cada kilómetro: %f litros.\n",litros_km);
```

2.4 El número π.

Hacer un programita que escriba en la salida estándar el valor de dicho número. Como todo el mundo sabe, el número π se puede obtener como resultado del arco coseno de –1.

Solución:

```
/***********************************************/
/* Objetivo: Escribir en la Salida Estándar    */
/*           el valor del número PI.           */
/***********************************************/
#include <stdio.h>
#include <math.h>
```

```
void main ()
{
  printf ("%f", acos(-1) );    /* Arco-coseno */
}
```

Si buscáramos mucha precisión tendríamos que emplear otra forma, pero con esta obtenemos suficiente precisión para la mayoría de los problemas habituales.

2.5 El número e.

Hacer un programa que escriba en la salida estándar el valor del número e, que es la base de los logaritmos naturales o neperianos.

Solución:

```
/***********************************************/
/* Objetivo: Escribir en la Salida Estándar    */
/*           el valor del número  e.           */
/***********************************************/
#include <stdio.h>
#include <math.h>

void main ()
{
  printf ("%f", exp(1) );    /* Exponencial: e elevado a 1 */
}
```

2.6 Visualización de un valor con diferentes formatos.

Mostrar en pantalla el dato de tipo `char` 'A', y a continuación el número 3 como dato de tipo `int`, `float` y `double`.

Solución:

```
/***********************************************/
/* Objetivo: Escribir en la Salida Estándar    */
/*           el carácter 'A' y el número 3     */
/*           tanto en entero, real y real en   */
/*           doble precisión                   */
/***********************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  clrscr();
```

```
  printf("\nchar: %c",'A');
  printf("\nchar: %c",65); /*char y un int (codigo ASCII 0-255)
                             se tratan igual*/
  printf("\nint: %d",3);
  printf("\nfloat: %f",3.); /*las constantes float y double deben
                              llevar punto decimal*/
  printf("\ndouble: %lf",3.);
}
```

2.7 Lectura y visualización de distintos tipos de datos.

Leer desde teclado un dato de tipo `char`, `int`, `float` ,`double` y mostrarlos en pantalla

Solución:

```
/**********************************************/
/* Objetivo: Leer un carácter, un entero,     */
/*           un real y un real en doble       */
/*           precisión desde la               */
/*           Entrada Estandar y mostrarlos por */
/*           la Salida Estandar               */
/* Entrada:  Un carácter, un entero, un real  */
/*           un real doble.                   */
/* Salida:   Mostrarlos por pantalla          */
/**********************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  char caracter;
  int entero;
  float flotante;
  double doble;

  clrscr();
  printf("\n\n\nIntroduzca un caracter: ");
  scanf("%c",&caracter); /* No olvide el símbolo & (ampersand)
                            en la función scanf() */
  printf("\nSe ha leido: %c",caracter);
  printf("\n\n\nIntroduzca un entero: ");
  scanf("%d",&entero);
  printf("\nSe ha leido: %d",entero);
  printf("\n\n\nIntroduzca un flotante: ");
  scanf("%f",&flotante);
  printf("\nSe ha leido: %f",flotante);
```

```
  printf("\n\n\nIntroduzca un doble precisión: ");
  scanf("%lf",&doble);
  printf("\nSe ha leido: %lf",doble); /*para printf() es lo mismo
                                         %lf que %f -no es lo mismo en scanf()- */
}
```

2.8 Visualización de números reales.

Leer un numero real y mostrarlo en pantalla con los siguientes códigos de formato de salida:

- con decimales sin exponente
- con decimales y exponente
- el más corto de los dos

Solución:

```
/***********************************************/
/* Objetivo: Leer un número real desde la      */
/*           Entrada Estandar y mostrarlo en   */
/*           sus diversos formatos.            */
/* Entrada:  Un número real.                   */
/***********************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  float numfloat;

  printf("\nEscribe un número:");
  scanf("%f",&numfloat);
  printf("\nSus formatos son %f, %e, %g",numfloat,numfloat,numfloat);
  printf("\nPulsa una tecla para terminar");
  getch();
  clrscr();
}
```

2.9 Visualización de números reales con un tamaño determinado.

Leer un numero de tipo `float` y mostrarlo con los formatos:

- al menos 5 digitos en parte entera, 0 en decimal (p.e. 23243.)
- 0 digitos en parte entera, al menos 5 en decimal (p.e. .45659)
- al menos 5 digitos en parte entera y al menos 5 en decimal (p.e. 45645.86755)

Solución:

```
/**********************************************/
/* Objetivo: Leer un número real y mostrarlo    */
/*           con una cantidad de dígitos para   */
/*           tanto para la parte real como la   */
/*           decimal y ambas a la vez.          */
/* Entrada:  Un número real.                    */
/**********************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  float numfloat;

  printf("\nEscribe un número:");
  scanf("%f", &numfloat);

  printf("\nSus formatos son %5.0f, %0.5f, %5.5f",
          numfloat, numfloat, numfloat);
  printf("\nPulsa una tecla para terminar");

  getch();
  clrscr();
}
```

2.10 Visualización de caracteres como números.

Leer por teclado un carácter y mostrar en pantalla el código ASCII del mismo y el carácter siguiente alfabéticamente.

Solución:

```
/**********************************************/
/* Objetivo: Leer un carácter y escribir su     */
/*           correspondiente en código ASCII    */
/*           y además escribir el siguiente     */
/*           carácter al leido                  */
/* Entrada:  Un carácter                        */
/* Salida:   Su código ASCII y el siguiente     */
/*           carácter                           */
/**********************************************/

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

```
void main(void)
{
  char varchar;

  clrscr();
  printf("\nIntroduce un caracter:");
  scanf("%c",&varchar);

  printf("\nint: %d",varchar);
  printf("\nchar: %c",varchar+1);
}
```

En el primer `printf()` debe mostrar un número, por eso se le pasa como argumento un caracter, que será evaluado como un entero (según las reglas de conversión automáticas), y mostrado en pantalla como tal. El segundo debe mostrar un caracter, el correspondiente a `varchar+1`, que se evaluará como entero, y se imprimirá en pantalla con formato de caracter `%c`.

2.11 Evaluación de expresiones con conversión automática de tipo.

Realizar un programa que muestre las siguientes expresiones por pantalla:

- `3/2+'A'/2+9/2`
- `(int)(3/2+'A'/2+9.0/2)`
- `(float)3/2+'A'/2.0+9/2`

Solución:

```
/***********************************************/
/* Objetivo: Muestra por pantalla la evaluación */
/*           de unas expresiones               */
/***********************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  clrscr();
  printf("\n%d",3/2+'A'/2+9/2); /* El resultado de la expresión es
                                   un entero: 37 */
  printf("\n%d", (int) (3/2+'A'/2+9.0/2)); /* El resultado es un entero: 37 */
  printf("\n%f", (float) 3/2+'A'/2.0+9/2); /* El resultado es un real: 38.0 */
}
```

Capítulo 3

Selección.

En este capítulo nos vamos a centrar en un nuevo elemento del lenguaje. Este nuevo elemento son las sentencias de selección o bifurcación. Con ellas vamos a ser capaces de poder tomar una serie de decisiones para poder realizar programas más complejos.

Con las sentencias de selección se puede elegir o seleccionar un camino o flujo de control del programa, dependiendo de una condición. Así, un mismo programa, dependiendo de los datos de entrada puede variar el flujo de instrucciones a ejecutar, variando, por tanto, su comportamiento.

3.1 Par o impar.

Hacer un programa que lea de la entrada estándar (normalmente el teclado) un número entero positivo y escriba en la salida estándar (normalmente la pantalla o monitor) si es par o impar.

Solución:

```
/*******************************************************/
/* Objetivo: Indicar si un número es Par o Impar       */
/* Entrada : Un entero.                                */
/* Salida  : Si es Par o Impar.                        */
/*******************************************************/
#include <stdio.h>

void main ()
{ int x;

  printf ("\nDame un número entero: ");
  scanf ("%i", &x);

  if (x%2)
     puts ("\n- Es un número IMPAR.");
  else
     puts ("\n- Es un número PAR.");
}
```

Es importante tener en cuenta que en C cualquier valor distinto de 0 (cero) se tomará como Verdad y un valor 0 se tomará como Falso. Así, si el resto de dividir el entero `x` entre 2 es distinto de 0, será Impar y en caso contrario será Par. Por eso, en la comparación del `if` también se podría haber usado `(x%2 != 0)`, pero esta es menos eficiente.

En general, siempre que vayamos a comparar si una expresión es distinta de cero, se puede eliminar la comparación, dejando sólo la expresión.

3.2 Mayor de 3 enteros.

Hacer un programa que lea tres números enteros positivos distintos e imprima el mayor valor de los tres.

Solución:

```
/*****************************************************/
/* Objetivo: Hallar el mayor valor de 3 enteros.     */
/* Entrada : Tres enteros.                           */
/* Salida  : Un entero (el mayor de los 3).          */
/*****************************************************/
#include <stdio.h>

void main ()
{ int a, b, c;

  printf ("\nDame tres números enteros distintos: ");
  printf ("\n- Primero: "); scanf ("%i", &a);
  printf ("\n- Segundo: "); scanf ("%i", &b);
  printf ("\n- Tercero: "); scanf ("%i", &c);

  printf("\nEl mayor es: ");

  if (a > b && a > c)
     printf("%i\n",a);
  else if (b > a && b > c)
         printf("%i\n",b);
       else if (c > a && c > b)
              printf("%i\n",c);
            else printf("ERROR: Hay números iguales.\a\n");
}
```

3.3 Mayor y menor de 3 enteros.

Hacer un programa que lea 3 números enteros positivos distintos e imprima el mayor y el menor valor de los 3. Observe que hay muchas formas de hacerlo. Intente encontrar la que realice menos comparaciones entre los números (la más eficiente).

Puede basarse en el ejercicio anterior y modificarlo convenientemente. Normalmente es más rápido modificar un programa similar que escribir un programa nuevo.

Solución:

```
/********************************************************/
/* Objetivo: Hallar el mayor y menor valor de 3 enteros. */
/* Entrada : Tres enteros.                               */
/* Salida  : Dos enteros (el mayor y menor de los 3).    */
/********************************************************/
#include <stdio.h>

void main ()
{ int a, b, c, max, min;

  puts("\n*** Dame 3 números y te diré el Mayor y el Menor ***");
  printf("\n- Primer  número: "); scanf("%i",&a);
  printf("\n- Segundo número: "); scanf("%i",&b);
  printf("\n- Tercer  número: "); scanf("%i",&c);

  if (x>y) if (x>z) { max = x;
                      min = (y>z) ? z : y;
                    }
           else { max = z;
                  min = y;
                }
  else
           if (y>z) { max = y;
                      min = (x>z) ? z : x;
                    }
           else { max = z;
                  min = x;
                }

  printf("\n\n- Mayor: %i\n- Menor: %i\n",max,min);
  puts("\n***** FIN *****");
}
```

3.4 Sistema de ecuaciones con 2 incógnitas.

Hacer un programa que resuelva un sistema de ecuaciones con 2 incógnitas. El programa primero pedirá los 6 coeficientes a, b, c, d, e y f según el siguiente formato:

```
ax + by = c
dx + ey = f
```

y mostrará a continuación todas las posibles soluciones, si estas existen.

Solución:

```
/*********************************************************/
/* Objetivo: Resuelve un sistema de 2 ecuaciones con     */
/*           2 incógnitas.                               */
/* Entrada : 6 coeficientes reales                       */
/* Salida  : Dos valores para cada incógnita, si la      */
/*           solución es posible.                        */
/*********************************************************/
#include <stdio.h>

void main()
{ float a, b, c, d, e, f,   /* Coeficientes */
      denominador;

  puts("\n***** Sistema de Ecuaciones con 2 incógnitas *****");
  puts("       ax + by = c");
  puts("       dx + ey = f");
  printf("\n* Dame los coeficientes: a = ");scanf("%f",&a);
  printf("                         b = ");scanf("%f",&b);
  printf("                         c = ");scanf("%f",&c);
  printf("                         d = ");scanf("%f",&d);
  printf("                         e = ");scanf("%f",&e);
  printf("                         f = ");scanf("%f",&f);

  printf("\n* Ecuaciones:     %fx + %fy = %f",a,b,c);
  printf("\n                  %fx + %fy = %f",d,e,f);
  denominador = a*e - b*d;

  if (denominador) {
     printf("\n* Resultados: x = %f",(c*e - b*f)/denominador);
     printf("\n              y = %f",(a*f - c*d)/denominador);
  }
  else
    puts("\n\n***** Imposible solucionar esas ecuaciones *****");
}
```

3.5 Conversión a mayúsculas.

Escribir en una sentencia de asignación un sistema para asignar a la variable `c` de tipo `char`, el carácter que contiene dicha variable, pero convertido a mayúsculas, si el carácter está entre el carácter `'a'` y el `'z'`. En otro caso no se modificará el contenido de la variable `c`. No se debe usar la función `toupper()`, ya que si no es muy simple.

NOTA: Podemos considerar que usamos el código ASCII, el cual está ordenado de la siguiente forma:

```
caracteres de control, ..., símbolos de puntuación, ...,
'0', '1', '2', ..., '9', ..., 'A', 'B', 'C', ..., 'Z', ...,
'a', 'b', 'c', ..., 'z', ..., otros símbolos.
```

Para la solución debemos usar el operador ? : ya que nos pide que sea en una sentencia de asignación. Como vemos, las letras mayúsculas están todas juntas, empezando por el carácter `'A'`, igual que las minúsculas que empiezan por `'a'`. Para convertir a mayúsculas sólo hay que restar `'a'` y sumar `'A'`

Solución:

```
c = ( c >= 'a' && c <= 'z' ) ? ( c - 'a' + 'A') : c ;
```

3.6 Números divisibles.

Realizar un programa para que dados dos números enteros nos diga si el primero es divisible por el segundo o viceversa.

Solución:

```
/*******************************************************/
/* Objetivo: Dados dos números enteros decidir si son   */
/*           divisibles, es decir, si el segundo divide */
/*           al primero o viceversa.                    */
/* Entrada : Dos enteros                                */
/*******************************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  int num1,num2;

  clrscr(),
  printf("\nEscribe dos números:");
  scanf("%d%d",&num1,&num2);

  if (!(num1%num2)) /* condición simplificada de (num1%num2==0) */
    printf("\nEl segundo divide al primero");
  else if (!(num2%num1))
         printf("\nEl primero divide al segundo");
       else
   printf("\nNo son divisibles");
}
```

3.7 Selección de destino de viaje por menú.

Dada la tabla siguiente:

Viaje	Algeciras-Malaga	Algeciras-Granada	Algeciras-Madrid
Precio	1000	2000	8000

Realizar un programa que muestre los posibles destinos de viaje en pantalla, y que mediante una selección por menú, imprima en pantalla la cantidad que cuesta el viaje a un destino introducido desde teclado.

Después de esto el mismo programa debe pedir el dinero disponible por el viajero y en función de él, le indique el destino más lejano al que puede viajar.

Solución:

```
/*******************************************************/
/* Objetivo: Dado un menu con distintos destinos       */
/*           pedir una cantidad de dinero para realizar */
/*           el viaje y una vez dada decir cual es el   */
/*           destino más lejano al que podría ir        */
/* Entrada : Un número entero                           */
/*******************************************************/

#include <stdio.h>

void main(void)
{
  int destino;
  int dinero;

  printf("\n Los destinos son");
  printf("\n 1.Malaga 2.Granada 3.Madrid");
  printf("\n ¿A qué destino desea viajar?");
  scanf("%d",&destino);

  if (destino==1)
    printf("\n El precio es 1000 pesetas");
  else if (destino==2)
         printf("\n El precio es 2000 pesetas");
       else if (destino==3)
        printf("\n El precio es 8000 pesetas");

  printf("\n ¿Cual es el efectivo del que dispone?");
  scanf("%d", &dinero);

  if (dinero<1000)
    printf("\n No puede viajar a ningun destino");
  else if (dinero<2000)
```

```
        printf("\n El destino mas lejos es Malaga");
      else if (dinero<8000)
             printf("\n El destino mas lejano es Granada");
           else printf("\n El destino mas lejano es Madrid");
}
```

3.8 Operación entre números por menú.

Leer dos números por teclado y a contiuación un caracter (+,-,*,/) para indicar la operación a realizar con ellos. Se mostrará el resultado en pantalla.

Solución:

```
/*********************************************************/
/* Objetivo: Leer dos números enteros y luego realizar  */
/*           una de las operaciones aritméticas, las    */
/*           cuales se presentarán en un menú de        */
/*           selección.                                  */
/* Entrada : Dos números enteros                         */
/* Salida  : La correspondiente operación aritmética.   */
/*********************************************************/

#include <stdio.h>
#include <conio.h>

void main(void)
{
  int x,y,res,operand;

  clrscr();
  printf ("\nIntroduce dos números: "),
  scanf("%d%d",&x,&y);
  printf("\nPulsa un operador");
  operand=getch();

  switch (operand) { /* switch sólo admite enteros o caracteres */
    case '*': res=x*y;
              break;
    case '/': res=x/y;
              break;
    case '+': res=x+y;
              break;
    case '-': res=x-y;
              break;
    default:  exit(1);
  }
```

```
  printf("\nResultado= %d",res);
  getch();
}
```

Se observa como los datos de tipo caracter y enteros, pueden utilizarse como tipos iguales (en realidad un caracter se almacena en memoria como un número entero de tamaño un byte)

3.9 Raíces reales de una ecuación de segundo grado.

Determinar las raíces reales de una ecuación de segundo grado, leyendo los coeficientes (a,b y c) por teclado como enteros, y eliminando todos los posibles errores de ejecución.

Solución:

```
/*******************************************************/
/* Objetivo: Hallar las raices enteras de una ecuanción */
/*           de segundo grado con coeficientes enteros  */
/* Entrada : Tres números enteros                       */
/* Salida  : Soluciones reales si existen               */
/*******************************************************/

#include <stdio.h>
#include <math.h>

void main(void)
{
  int a,b,c;
  double x1, x2;
  double paso;

  clrscr();
  printf("Introduce el coeficiente de X2: ");
  scanf("%d", &a);
  printf("Introduce el coeficiente de X: ");
  scanf("%d", &b);
  printf("Introduce el termino independiente: ");
  scanf("%d", &c);

  paso = (double) b*b - (double) 4*a*c;

  if (paso < 0) {
    printf("\nNo existen soluciones reales a estos valores\n");
    printf("La ecuación %d X2 + %d X + %d no tiene soluciones.\n",
               a,b,c);
    return 0;
  }
```

```
  x1 = (-b + sqrt(paso)) / (2*a);
  x2 = (-b - sqrt(paso)) / (2*a);
  printf("\nLa ecuación %d X2 + %d X + %d tiene soluciones:\n", a,b,c);

  if (x1 != x2)
    printf("Soluciones: %.0lf y %.0lf", x1, x2);
  else
    printf("Solucion única: %.0lf", x1);
}
```

Capítulo 4

Bucles.

Con este tema se termina de ver las estructuras de control necesarias para la realización de un programa. En estas estructuras se basa el Teorema de **Jacopini-Bohm** que afirma que cualquier programa, por complejo que este sea, puede escribirse utilizando tan sólo estas tres estructuras de control: Secuencia, Selección e Iteración.

Con las estructuras de Iteración (repetitivas o bucles) lo que se pretende es repetir una serie de sentencias un numero de veces determinado.

4.1 Tabla de caracteres ASCII.

Hacer un programa que escriba en la salida estándar todos los caracteres ASCII que se estén usando. Supondremos un ASCII de 8 bits (lo más habitual), por lo que tendremos 256 valores posibles.

Además, cuando se llene la pantalla deberemos hacer una pausa hasta que se pulse una tecla.

Solución:

```
/***********************************************/
/* Objetivo: Escribir en la Salida Estándar    */
/*           la tabla de caracteres ASCII que  */
/*           se esté utilizando.               */
/***********************************************/
#include <stdio.h>
#include <conio.h>

void main ()
{ int n = -1;

  while (++n <= 255) {
    if ( n % 23 == 0 && n!=0 ) {
       printf ("\n- Pulse una tecla para continuar ...");
       getch();
       puts ("\n");
```

```
    }
    printf("\n\t %3d = '%c'", n, n);
  }
}
```

4.2 Factorial.

Escribir un programa que calcule el factorial de un número, `n!` a partir de un número positivo `n`. El factorial se calcula de la siguiente forma:

```
Si n = 0,     n! = 1
Si n > 0,     n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1
```

Solución:

```
/****************************************************************/
/* Objetivo: Hallar el factorial de un número.                 */
/* Entrada : Un unsigned (no existe el factorial de negativos). */
/* Salida  : Un long double, como su factorial.                 */
/****************************************************************/
#include <stdio.h>

void main()
{ unsigned int N = 0;          /* Número a hallar su factorial: N! */
  long double factorial=1;  /* Resultado */

  printf("\n Deme un número para hallar su factorial: ");
  scanf ("%u", &N);

  for ( ; N > 1; N--)
      factorial = factorial * N;

  printf ("\n- Factorial: %u! = %Lf\n", N, factorial);
}
```

4.3 Máximo Común Divisor.

Hacer un programa que calcule el M.C.D. (Máximo Común Divisor) de dos números que se piden por teclado.

Solución:

```
/*************************************************************/
/* Objetivo: Hallar el MCD de dos números.                   */
/* Entrada : Dos unsigned a los que hallarle su MCD.         */
/* Salida  : Un unsigned que sea el MCD buscado.             */
/*************************************************************/
#include <stdio.h>

void main()
{ unsigned int x, y;

  puts("\n***** MCD de dos números *****");
  printf ("\n- Introducir Primer número: ");
  scanf ("%u", &x);
  printf ("\n- Introducir Segundo número: ");
  scanf ("%u", &y);

  while (x != y) {
       if (x > y)
          x = x - y;
       else
          y = y - x;
  }

  printf ("El M.C.D. es: %u.", x);
}
```

4.4 Quitando espacios.

Escribir un programa en C que lea caracteres de su entrada y los escriba en su salida estándar, reemplazando cada cadena de uno o más espacios en blanco, por un solo espacio. Pruébelo desde un S.O. con línea de comandos (tipo DOS, UNIX...) utilizando las redirecciones.

Solución:

```
/*************************************************************/
/* Objetivo: Quita los espacios duplicados de un texto.      */
/* Entrada : Un texto por su entrada estándar.               */
/* Salida  : Un texto por su salida estándar.                */
/*************************************************************/
#include <stdio.h>
```

```
void main ()
{ int sp_ant = 0, /* Espacio anterior */
      sp_act = 0; /* Espacio actual   */
  char ch = getchar();

  sp_act = (ch==' ') ? 1 : 0;

  while (ch!=EOF) {
    if ( !(sp_ant && sp_act) )
       putchar(ch);
    ch = getchar();
    sp_ant = sp_act;
    sp_act = (ch==' ') ? 1 : 0;
  }
}
```

El algoritmo puede parecer un poco complejo, pero no lo es. Simplemente va leyendo caracteres y viendo si son o no son espacios. El carácter leído en cada momento sólo lo imprime si tanto dicho carácter como el anterior no son ambos espacios. Tanto si ninguno de los dos es espacio, como si uno de los dos es un espacio y el otro no, sí lo imprime.

4.5 La pesca en la CEE.

Suponga que la CEE impone a los pescadores un límite en los Kg. de pesca que pueden recoger en un día, para no agotar los recursos marinos. Desarrolle un programa al que primero se le dé ese límite y luego se le vayan dando los pesos de lo que se va pescando y vaya mostrando el total de Kg. pescados hasta ese momento. Cuando supere el máximo se debe dar la alarma y terminar el programa. También debe terminar el programa si introducimos un 0 como kilos pescados.

Solución:

```
/*****************************************************************/
/* Objetivo: Control del número de kilos de pescado recogidos,  */
/*           avisando cuando se supere el límite permitido.     */
/* Entrada : El límite y lo que se va pescando (en Kg.).        */
/* Salida  : Total pescado y avisa cuando se supere el límite.  */
/*****************************************************************/
#include <stdio.h>
#include <dos.h>

void main()
{ int total=0, limite, pesca;

  puts("\n***** La pesca en la CEE ***** ");
  printf("\n- Limite de Kg. de pesca permitidos: ");
```

```
  scanf("%i",&limite);
  printf("-El limite son %i Kg. (Pulsa 0 para FIN).\n\n",limite);

  while (1) {  /* ¡Ojo!, este NO es un bucle infinito */
       printf ("- Dame los kg. pescados ahora: ");
       scanf("%i",&pesca);
       if (pesca==0) break;

       total += pesca;
       if (total>=limite) {
          printf("\a\n LIMITE SUPERADO en %i Kg !!\n",total-limite);
          for ( total=1200; total>500; total--) {
               sound(total);
               delay(2);
          }
          nosound();
          break;
       }
       else printf("\a- TOTAL pescado por ahora: %i\n\n",total);
  }

  puts ("***** THE END *****");
}
```

Observe, que el bucle no es infinito, aunque la condición (1) sea siempre verdad, ya que siempre vale 1 que es distinto de cero. El bucle puede terminar por dos motivos, porque se introduzca un cero como kilos pescados y porque se supere el límite establecido. En ambos casos existe un `break` que rompe el bucle.

Si se supera el límite, se producirá un aviso sonoro formado por sonidos descendentes. Dichas funciones están incluidas en la biblioteca `dos.h`.

4.6 Números Primos.

Escribir un programa que muestre todos los números primos entre 3 y 32767.

NOTA: Hemos elegido ese número como tope porque suele ser el mayor número que se puede representar en una variable entera (usando una representación con signo, en 16 bits). Para más información sobre los tipos de datos ver Apéndice A.

Solución:

```
/**********************************************/
/* Objetivo: Escribir en la Salida Estándar     */
/*           todos los números PRIMOS entre     */
/*           3 y 32767.                         */
/**********************************************/
#include <stdio.h>
```

```
void main ()
{ int numero = 2,
      x;

  while (++numero <= 32767) {
    x = 1;

    while (++x < numero)
      if (numero % x == 0)  /* Si numero es divisible por x,          */
         break;             /* NO es PRIMO, y nos salimos del bucle  */

    if (x == numero)        /* No hemos encontrado ningún divisor    */
       printf("%i\n", numero);
  }
}
```

Lo que hacemos es, para cada número, buscamos un divisor entre los números que son menores que él. Si encontramos algún divisor, el número no es primo.

4.7 Números en un intervalo.

Leer dos números (desde el teclado) y mostrar en pantalla los números comprendidos entre ambos. Tenga en cuenta que no se sabe cual de los números es mayor (pueden leerse en cualquier orden).

Solución:

```
/*****************************************************************/
/* Objetivo: Leer dos números enteros y pretentar aquellos que   */
/*           se encuentran comprendidos entre ambos.             */
/* Entrada : Dos números enteros                                 */
/* Salida  : Todos los comprendidos entre ambos                  */
/*****************************************************************/

#include <stdio.h>
#include <conio.h>

main()
{
  int num1;
  int num2;
  int i;

  clrscr();
  printf("\nIntroduzca el primer número: ");
  scanf("%d", &num1);
  printf("\nIntroduzca el segundo número: ");
```

```
  scanf("%d", &num2);
  printf("\n");

  if (num1>num2)
    for(i=num1;i>=num2;--i)
      printf("%d ", i);
  else if (num2>num1)
         for(i=num1;i<=num2;++i)
           printf("%d ", i);
       else
         printf("Ambos números son iguales\n");
}
```

4.8 Mostrar números hasta una pulsación.

Mostrar los números desde el 32000 hasta el 0, mientras no se pulse una tecla.

Solución:

```
/*************************************************************/
/* Objetivo: Mostras una sucecion de números desde el 0 hasta  */
/*           el 32000 si no se pulsa una tecla                 */
/*************************************************************/

#include <stdio.h>
#include <conio.h>

void main (void)
{
  int i=32000;

  while (!kbhit()) { /* hbhit() no espera a que se pulse una tecla,
                        si no está pulsada vale 0 */
    printf("%d \t",i);
    i--;
  }
}
```

4.9 Factorización.

Realizar un programa que dado un número entero nos dé su descomposición en factores primos.

Todo número entero tiene una única descomposición dada por una multiplicación de potencias de números primos. Debemos de hacer un programa que encuentre esos números primos y además su potencia asociada.

Solución:

```
/******************************************************************/
/* Objetivo: Factorización de un número entero                    */
/* Entrada:  Un número entero                                     */
/* Salida:   Los factores junto a sus potencias correspondientes */
/******************************************************************/
#include <stdio.h>

void main()
{
  int n,
      factor=2,
      potencia = 0;

  printf("Introduce el número entero a factorizar: ");
  scanf("%d",&n);

  printf("La factorización del número %d es la siguiente\n",n);

  while(n>1) {  /* mientras no este factorizado el número */

    while( n % factor == 0 ) {/* buscar la potencia correspondiente al
                                 factor actualmente encontrado */
      potencia ++;
      n /= factor; /* buscamos el siguiente factor o
                      potencia del actual */
    }

    if(potencia > 0) {
      printf("Factor %d elevado a la potencia %d\n",factor,potencia);
      potencia = 0;
    }
    factor ++;
  }
}
```

Una primera solución sería probar entre los primeros números primos y ver si éstos dividen al número que deseamos factorizar. Nuestro problema es que no tenemos una lista de primos por los que probar para realizarlo de esta forma. La solución que se ha intentado es probar por todos los números empezando por el 2 que es el primer primo.

Como vamos eliminando aquellos factores que vamos encontrando si llegamos a dividir por un número que no sea primo, es decir, que también tenga una descomposición factorial, ya hemos tenido en cuenta sus factores y por tanto nunca dividirá a lo que nos que resulte de factorizar el número inicial.

Esta no es una solución eficiente y para números grandes o primos grandes puede que la respuesta tarde mucho en conseguirse. Como alternativa se podrían pensar nuevas mejoras para que el tiempo necesario en calcular la factorización de un número fuera más rápida.

Capítulo 5

Arrays.

En este capítulo se muestra una estructura compleja de datos muy útil a la hora de realizar nuestros programas. El array es un conjunto de datos del mismo tipo referenciados por un mismo nombre y ordenados por un índice.

El tipo de dato array es fundamental para la programación y de él se dispone en la mayoría de los lenguajes de programación, pero en C es especialmente importante, pues el tipo de dato **Cadena de caracteres** es un tipo particular de array (array de datos de tipo `char`).

A continuación vamos a ver distintos ejercicios para saber cómo se utiliza y qué valor tiene a la hora de aumentar la dificultad de nuestros programas, para poder resolver problemas más complejos.

En el tema siguiente hay dos apartados especiales para este tipo de datos: Uno específico para cadenas de caracteres y otro para arrays en general (vectores, matrices...).

5.1 MCD por argumentos en la línea de comandos.

Escribir un programa que calcule el Máximo Común Divisor de dos enteros que leerá en la línea de comandos del Sistema Operativo. Este programa ya se resolvió en un ejercicio anterior, pero aquel leía los datos de la entrada estándar y este los debe leer de la línea de comandos.

Nota 1: Deberá usar argumentos en la función principal (`argc` y `argv`). Recordamos que `argv` se puede ver como un **array** de cadenas de caracteres.

Nota 2: La línea de comandos es la línea de texto que escribe el usuario para ejecutar un programa. En Sistemas Operativos o entornos gráficos también existe línea de comandos, pero esta no tiene que escribirla el usuario siempre que quiera ejecutar el programa sino que se asocia a un icono (pequeño dibujo) y para ejecutar dicha línea de comandos bastará con el usuario la elija de alguna forma (normalmente con un doble clic con el ratón).

Solución:

```
/***************************************************************/
/* Objetivo: Hallar el MCD de dos argumentos enteros en la     */
/*           línea de comandos del S.O.                        */
/* Entrada : Dos enteros en la línea de comandos del S.O.      */
/* Salida  : El MCD de dichos enteros.                         */
/***************************************************************/
```

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

void main (int argc, char *argv)
{ long int x, y;

  if (argc < 3) {
     puts ("\nERROR: Sólo se permiten dos argumentos \n");
     puts ("Formato de la orden:  MCD <num1> <num2>");
          /* El programa se llamará MCD */
     return;
  }

  x = atol (argv [1]);
  y = atol (argv [2]);

  if (x <= 0 || y <= 0) {  /* O lógico (OR) */
     puts ("\nERROR: Ambos números deben ser mayores que cero");
     puts ("        y no deben contener letras.\n");
     return;
  }

  while (x != y)        /* Hallar el MCD */
        if (x > y) x = x - y;
        else       y = y - x;

  printf ("\nEl Máximo Común Divisor es: %i.\n", x);
}
```

La función `atol()` (*ascii to long*) está incluida en `stdlib.h` y su prototipo es el siguiente:

```
long atol (cons char *s);
```

Esta función convierte la cadena de caracteres `s` en un entero largo (`long`). Si hubiera algún error devuelve 0.

De forma similar existen algunas funciones para operaciones similares, como `atoi()` (*ascii to int*), `atof()` (*ascii to float*), `itoa()` (*int to ascii*) y `ltoa()` (*long to ascii*).

Recordamos que `argc` (*argument count*) es un argumento de la función principal que toma automáticamente el valor del número de argumentos que se ponen en la línea de comandos al ejecutar el programa desde el Sistema Operativo. En este número de argumentos se incluye el propio nombre de la función por lo que el valor mínimo de argc es 1 (el nombre del programa).

Con `argv[]` (*argument values*) podemos acceder a todos los argumentos que se pongan en la línea de comandos. Igualmente también se incluye el nombre del programa. Así `argv[0]` es una cadena de caracteres con el nombre del programa en ejecución, `argv[1]` es otra cadena de caracteres con el contenido del primer argumento, y así sucesivamente. Todos los argumentos de la línea de comandos se leen como si fueran cadenas de caracteres (si fueran números se deberá efectuar la conversión.

5.2 Algoritmo para Ordenación de Arrays: Burbuja.

Hacer un programa que ordene un array, de menor a mayor, por el método de ordenación **Burbuja**. Este método consiste en ir comparando dos a dos todos los elementos del array e intercambiándolos si están desordenados. Así, primero compara el elemento de posición 0 con el de posición 1, luego el 1 con el 2, y así sucesivamente hasta el final del array. Con esto conseguimos que en la primera pasada, el elemento mayor esté en su sitio, esto es, en la última posición. En la siguiente pasada, en la que no hay que comparar ya con el último elemento, conseguiremos que el siguiente elemento más grande esté en su sitio, esto es, en la penúltima posición.

De esta forma, si el array tiene n elementos, en $n - 1$ pasadas conseguimos ordenar completamente el array. Además, cada pasada requiere una comparación menos que la anterior.

El nombre de burbuja viene de que, los elementos mayores *suben* antes, para colocarse en su sitio primero, igual que las burbujas mayores suben más rápidamente a la superficie.

Podemos suponer que n valdrá siempre 10.

Solución:

```
/************************************************************/
/* Objetivo: Ordenar un array de menor a mayor por Burbuja.   */
/* Entrada : Elementos de un array de enteros.                */
/* Salida  : Elementos del mismo array, ordenados.            */
/************************************************************/
#include <stdio.h>

#define N_ELEMENTOS 10  /* Número de elementos del array a ordenar */

void main ()
{ int array [N_ELEMENTOS],
      indice,                /* Para movernos por el array */
      aux,                   /* Variable auxiliar para intercambios */
      v = 1;                 /* Número de vuelta a realizar */
  char no_modificado;        /* Indica si se ha modificado o no el array */

  printf ("\nEntrada de los %i números del array:\n",N_ELEMENTOS);
  for (indice = 0; indice < N_ELEMENTOS; indice++) {
      printf ("- Elemento %i: ",indice);
      scanf  ("%i", &array [indice]);
  }

  /* Bucle del Algoritmo BURBUJA */
  do { no_modificado = indice = 0;
       do { if (array[indice] > array[indice+1]) { /* Intercambio */
              aux = array[indice];
              array[indice] = array[indice+1];
              array[indice+1] = aux;
              no_modificado = 1; /* Indica que el array se ha modificado */
```

```
                    /* por tanto: NO está ordenado todavía. */
          } /*if*/
          indice++;
      } while ( indice < N_ELEMENTOS - v );
      v++;        /* Se incrementa el número de vuelta */
  } while (no_modificado);

  printf ("\nSalida de los %i números del array ordenados:",N_ELEMENTOS);
  for (indice = 0; indice < N_ELEMENTOS; indice++)
      printf ("\n- Elemento %d: %d ", indice, array[indice]);

  puts ("\n\n***** Fin del Programa. *****");
}
```

Veamos cómo se comporta el anterior algoritmo con un array de sólo 5 elementos:

v	array [0]	array [1]	array [2]	array [3]	array [4]	no_modificado	indice	Notas
1	5	2	3	1	2	0	0	valores iniciales
	2	5	3	1	2	1	1	Intercambio (0,1)
	2	3	5	1	2	1	2	Intercambio (1,2)
	2	3	1	5	2	1	3	Intercambio (2,3)
	2	3	1	2	5	1	4	Intercambio (3,4)
	2	3	1	2	5	1	4	No $4 < 5-1$
2	2	3	1	2	5	0	0	Vuelta 2
	2	3	1	2	5	0	1	No cambia (0,1)
	2	1	3	2	5	1	2	Intercambio (1,2)
	2	1	2	3	5	1	3	Intercambio (2,3)
	2	1	2	3	5	1	3	No $3 < 5-2$
3	2	1	2	3	5	0	0	Vuelta 3
	1	2	2	3	5	1	1	Intercambio (0,1)
	1	2	2	3	5	1	2	No cambia (1,2)
	1	2	2	3	5	1	3	No $2 < 5-3$
4	1	2	2	3	5	0	0	Vuelta 4
	1	2	2	3	5	0	1	No cambia (0,1)
	1	2	2	3	5	0	1	No $1 < 5-4$
	1	2	2	3	5	0	1	no_modificado == 0

5.3 Media aritmética de 10 números.

Leer 10 números enteros y mostrar la media de ellos.

Cuando se trata de varios datos de igual tipo y que requieren igual tratamiento, en general no se deben declarar todos como variables independientes sino como un array de tantos elementos como sean necesarios. Esto hace que los programas sean más fáciles de modificar y más fácil de entender.

Naturalmente, este ejercicio es muy simple y no requeriría el uso de arrays ya que ni siquiera hay que almacenar todos esos números, pero hagámoslo con arrays a modo de ejemplo.

Solución:

```
/*************************************************************/
/* Objetivo: Leer 10 números enteros y realizar la media      */
/*           aritmética de los mismos.                        */
/* Entrada : 10 números enteros                               */
/* Salida  : Un números real que es la media                  */
/*************************************************************/

#include <stdio.h>

void main(void)
{
  int tabla[10];
  int i, suma = 0;

  clrscr();

  for(i=0;i<10;i++){
    printf("Introduce el número %d: ", i+1);
    scanf("%d", &tabla[i]);
    suma = suma + tabla[i];
  }

  /* Después de esto en el array tabla tenemos los 10 enteros */

  printf("\nLa media de todos ellos es %d", suma/10);
}
```

5.4 Matriz traspuesta.

Realizar un programa que lea una matriz de 3x3 números de tipo float y calcule una segunda matriz transpuesta de la primera. A continuación se debe mostrar en pantalla primero la matriz transpuesta y segundo la matriz original.

Solución:

```
/*************************************************************/
/* Objetivo: Leer una matriz de 3x3 de números reales .       */
/*           Calcular la matriz traspuesta y presentarla por  */
/*           pantalla. Presentar también la original          */
/* Entrada : Una matrix 3x3 de reales                         */
/* Salida  : Matriz traspuesta y la original                  */
/*************************************************************/
```

```
#include <stdio.h>

main()
{
  int i, j;
  float f;
  float matriz[3][3];

  clrscr();
  for(i=0;i<3;i++)
    for(j=0;j<3;j++){
      printf("Introduzca valor [%d][%d]: ", i+1, j+1);
      scanf("%f", &matriz[i][j]);
    }

  clrscr();
  printf("MATRIZ TRASPUESTA\n"),
  printf("*****************\n"),
  for(i=0;i<3;i++){
    printf("\n");
    for(j=0;j<3;j++)
      printf("%5.2f ", matriz[j][i]);
  }

  printf("\n\n\n");
  printf("MATRIZ ORIGINAL\n");
  printf("***************\n");
  for(i=0;i<3;i++){
    printf("\n");
    for(j=0;j<3;j++)
      printf("%5.2f ", matriz[i][j]);
  }
}
```

5.5 Búsqueda de números en un vector.

Leer un vector de 15 números enteros y a continuación se entre en un bucle de forma que mientras se introduzcan por teclado números distintos de cero, se deberán buscar estos números en el vector e indicar en pantalla si están dentro y en qué posición.

Solución:

```
/**************************************************************/
/* Objetivo: Leer un vector de 15 números enteros. Muentras     */
/*           se introduzcan por teclado números distintos de 0  */
/*           buscar dicho número en el vector, indicando si se  */
/*           encuentra y en que posición                        */
```

```
/* Entrada : Un vector de 15 números enteros y un entero.        */
/* Salida  : Si se encuentra el número decir la posición         */
/*****************************************************************/

#include <stdio.h>

main()
{
  int i, j, vector[15], chivato = 0;

  clrscr();
  for(i=0;i<15;i++){
    printf("Introduzca valor [%d]: ", i+1),
    scanf("%d", &vector[i]);
  }

  clrscr();
  printf("Busqueda de valores en VECTOR\n");
  printf("****************************\n");
  printf("\nIntroduzca número: ");
  scanf("%d", &j);

  while(j) {
    for(i=0;i<15;i++) {
      if(vector[i] == j) {
        printf("Número %d encontrado en Vector en
                   posición %d\n", j, i+1);
        chivato = 1;
      }
    }
    if (!chivato)
      printf("Número %d no encontrado en Vector\n", j);
    chivato = 0;
    printf("\nIntroduzca número: ");
    scanf("%d", &j);
  }
}
```

5.6 Manipulación de cadenas de caracteres.

Leer una serie de cadenas de caracteres y mostrar en pantalla para cada una de ellas su longitud, y la concatenación de esa cadena con la cadena ".EXE". Cuando se lea la cadena *nula* el programa terminará.

Solución:

```
/*****************************************************************/
```

```
/* Objetivo: Leer una cadena de caracteres y decir cual es su   */
/*           longitud. Concatenarla gOnOla cadena ".EXE". El    */
/*           programa terminará cuando se lea la cadena nula    */
/* Entrada : Una cadena de caracteres                           */
/* Salida  : Longitud de la cadena y concatenación con la       */
/*           ".EXE"                                             */
/****************************************************************/
#include <stdio.h>

main()
{
  char cadena[80];

  clrscr();
  printf("\nIntroduzca cadena: ");
  gets(cadena); /* Con scanf() con %s sólo se lee una palabra */

  while(strlen(cadena)){
    printf("La cadena contiene %d caracteres.\n", strlen(cadena));
    printf("Concatenación %s\n", strcat(cadena, ".EXE"));
    printf("\nIntroduzca cadena: ");
    gets(cadena);
  }
}
```

5.7 Multiplicación de matrices 3x3.

Realizar un programa que lea desde teclado dos matrices 3x3 enteras y como resultados nos de la matriz multiplicación de las anteriores.

Es una muy buena técnica de programación (como ya se ha indicado antes) no usar constantes en un programa. Para ello es mejor usar constantes simbólicas definidas con macros con `#define`. Así use una constante simbólica `RANGO` para indicar el tamaño de la matriz. En este caso `RANGO` deberá tomar el valor de la constante 3.

Solución:

```
/****************************************************************/
/* Objetivo: Leer dos matrices 3x3 de enteros. Realizar la      */
/*           multiplicación.                                    */
/* Entrada : Dos matrices enteras 3x3                           */
/* Salida  : Una matriz resultado de multiplicarlas             */
/****************************************************************/
#include <stdio.h>
#include <stdlib.h>

#define RANGO 3
```

```
void main(void)
{
  int mat_a[RANGO][RANGO],
      mat_b[RANGO][RANGO],
      mat_c[RANGO][RANGO];
  char cadena[20];
  int i,j,k;

/*lectura de mat_a por teclado*/
  for(i=0;i<RANGO;i++)
    for(j=0;j<RANGO;j++) {
      printf("\nIntroduce mat_a[%d][%d]= ",i,j);
      gets(cadena);
      mat_a[i][j]=atoi(cadena);
    }

/*lectura de mat_b por teclado*/
  for(i=0;i<RANGO;i++)
    for(j=0;j<RANGO;j++) {
      printf("\nIntroduce mat_b[%d][%d]= ",i,j);
      gets(cadena);
      mat_b[i][j]=atoi(cadena);
    }

/*inicializacion de mat_c a ceros*/
  for(i=0;i<RANGO;i++)
    for(j=0;j<RANGO;j++)
      mat_c[i][j]=0;

/*calculo de mat_c = mat_a(mat_b */
  for(i=0;i<RANGO;i++)
    for(j=0;j<RANGO;j++)
      for (k=0;k<RANGO;k++)
        mat_c[i][j]+=mat_a[i][k]*mat_b[k][j];

/*mostrar datos en pantalla*/
  printf("\nResultado de mat_c = mat_a(mat_b");
  for(i=0;i<RANGO;i++) {
    printf("\n");
    for(j=0;j<RANGO;j++)
      printf("\t\t%d",mat_c[i][j]);
  }
}
```

5.8 Aproximación al número e.

Generar una secuencia (de 0 a 100) de aproximación al número e. En cada paso de la secuencia, calcular el error entre la aproximación y el número e exacto (con 32 bits), almacenando la aproximación y el error cometido en un vector de estructuras. Al final se deberá imprimir en pantalla la aproximación calculada y el error en todos los pasos.

Solución:

```
/************************************************************/
/* Objetivo: Generar una aproximación al número e. Generar 100  */
/*           secuencias calculando el error. Almacena el número */
/*           generado y su error                                */
/************************************************************/
#include <stdio.h>
#include <math.h>
#include <conio.h>

void main ()
{
  int i;
  double auxi , error , e;
  struct dato
  {
    double aproximacion;
    double error;
  };
  struct dato tabla [ 100 ];

  clrscr();
  e = exp ( ( double ) 1 );

  for ( i = 1; i <= 100; i++ ) {
    auxi = ( 1 + ( double ) ( 1. / i ) );
    auxi = pow ( auxi , ( double ) i );
    error = e - auxi;
    tabla [ i ].aproximacion = auxi;
    tabla [ i ].error = error;
    printf( "Aproximacion %d : %f\t\t", i , tabla [ i ].aproximacion );
    printf ("Error %d: %1.10f\n", i , tabla [ i ].error );
  }
}
```

NOTA: Un número de 32 bits debe ser de tipo `double`.

5.9 Cálculo del centro de masas.

Calcule el centro de masas de un sistema de N partículas (N será una constante igual a 10), de tal forma que cada una de ellas vendrá dada por su posición en el espacio cartesiano tridimensional y por su masa. Tanto las coordenadas cartesianas como la masa se deberán leer por teclado. El resultado se deberá imprimir en pantalla.

Solución:

```
/*************************************************************/
/* Objetivo: Calcular el centro de masas de N=10 particulas.   */
/*           Cada particula viene dada en sus coordenadas       */
/*           cartesianas y por su masa.                         */
/* Entrada : Cada particula y su masa                          */
/* Salida  : Calculo del centro de masas                       */
/*************************************************************/
#include <stdio.h>
#include <math.h>

#define N 10

void main ()
{
  int i;
  float xcm = 0 , ycm = 0 , zcm = 0;
  struct particula {
    int x;
    int y;
    int z;
    int masa;
  };
  struct particula tabla [ N ];

  clrscr();
  for ( i = 1; i <= N; i++ ) {
    printf ( "Introduce coordenada x de la particula %d: ", i );
    scanf ( "%d", &tabla [ i ].x );
    printf ( "Introduce coordenada y de la particula %d: ", i );
    scanf ( "%d", &tabla [ i ].y );
    printf ( "Introduce coordenada z de la particula %d: ", i );
    scanf ( "%d", &tabla [ i ].z );
    printf ( "Introduce la masa  de la particula %d: ", i );
    scanf ( "%d", &tabla [ i ].masa );
   }

  for ( i = 1; i <= N; i++ ) {
    xcm += ( ( float ) tabla [ i ].x * ( float ) tabla [ i ].masa ) / N;
```

```
    ycm += ( ( float ) tabla [ i ].y * ( float ) tabla [ i ].masa ) / N;
    zcm += ( ( float ) tabla [ i ].z * ( float ) tabla [ i ].masa ) / N;
  }
  printf ( "\n\nLas coordenadas del centro de masa son:
                   %g, %g, %g", xcm , ycm , zcm );
}
```

5.10 Lectura y comparación de cadenas y números.

Leer dos cadenas de caracteres. Si el primer carácter es numérico supondremos que las cadenas contienen un número cada una y se convierten a su valor numérico y comparan los como números. En otro caso se comparan como cadenas.

Solución:

```
/*****************************************************************/
/* Objetivo: Leer dos cadenas de caracteres. Si el primer       */
/*           es numérico suponemos que son dos números y los    */
/*           comparamos. En otro caso hacemos una comparación   */
/*           de cadenas.                                        */
/* Entrada : Dos cadenas de caracteres                          */
/*****************************************************************/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

#define TAM 40

void main(void)
{
  char s1[TAM], s2[TAM];
  double n1,n2;

  printf("\nIntroduzca una cadena: ");
  gets(s1);
  printf("\nIntroduzca otra cadena: ");
  gets(s2);

  if (isalpha(s1[0])) {
    if (!strcmp(s1,s2))
      printf("\nSon iguales");
    else if (strcmp(s1,s2)>0)
           printf("\nLa primera es mayor");
         else
           printf("\nLa segunda es mayor");
  }
```

```
  else {
    n1=atof(s1);
    n2=atof(s2);
    if (n1==n2)
       printf("\nSon iguales");
    else if (n1>n2)
            printf("\nEl primero es mayor");
         else
            printf("\nEl segundo es mayor");
  }
}
```

Es más práctica la entrada de datos, del tipo que sean, como cadenas, convirtiéndose a su valor numérico en el caso de que las cadenas contengan números.

5.11 Factorial con parámetros en la función main().

Realizar un programa que calcule el factorial de un número, recibiéndolo como parámetro de la función main().

Ya se ha expuesto anteriormente un programa similar que calculaba el Máximo Común Divisor.

Solución:

```
/*****************************************************/
/* Objetivo: hallar el factorial de un número que    */
/*            se lee en la línea de comandos del S.O. */
/*****************************************************/
/* PARAMETROS de main(): int argc  y  char **argv    */
#include <math.h>
#include <stdio.h>

double factorial(double x);

void main(int argc,char **argv)
{
  if (argc<2)
     puts("\t\t\t Requiere un argumento.");
  else
    printf("\t\t\t = %lf",factorial(atof(argv[1])));
}

/*****************************************************/
/* Objetivo: hallar el factorial de un número        */
/* Requisitos: x>=1                                  */
/* Entrada: x (factorial a hallar)                   */
/* Salida: con return el valor del factorial         */
```

```
/**************************************************/
double factorial(double x)
{
  if (x==1)
    return 1.;
  else
    return x*factorial(x-1); /* OJO: Llamada recursiva */
}
```

A veces es útil que el programa ejecutable pueda ser invocado desde el Sistema Operativo pasándole unos argumentos. Esto se consigue incluyendo en la función `main()` dos parámetros formales cuyas definiciones ya se han explicado en un ejercicio anterior pero que aquí repetimos:

- `int argc`, su valor indica cuántos argumentos contiene la orden en la línea de comandos del Sistema Operativo, la que ha tecleado el usuario para la ejecución del programa (incluido el nombre del programa), y
- `char **argv` (puntero a puntero a carácter) que se puede entender como un vector de cadenas de caracteres, que contiene cada una de las cadenas de caracteres de los argumentos indicados por argv.

Por ejemplo, si un programa se llamase `program.exe` en DOS, y pudiese ejecutarse con la orden `program 100 -c`, entonces, `argc` valdría 3 y `argv` contendría tres cadenas de caracteres de la siguiente forma:

- `argv[0] --> "program"`
- `argv[1] --> "100"`
- `argv[2] --> "-c"`

En este ejemplo se utiliza para pasarle el número al que se le quiera calcular el factorial a la función `factorial()`. Observe que dicha función se llama a sí misma con otro argumento. Este tipo de funciones son llamadas funciones recursivas.

Capítulo 6

Funciones para todo.

Como sabemos, en C todo son funciones. Hasta el módulo principal (`main`) es una función. El programa más simple en C incorporará como mínimo una función (la principal), pero en cuanto el programa se complique ligeramente, surgiran otras funciones que serán llamadas desde la principal o desde otras que son llamadas desde esta y así sucesivamente. No hay límite teórico: La función `main()` puede llamar a otra y esta a otra y así indefinidamente.

El C reconoce a las funciones por los paréntesis que les suceden. Siempre que encontremos un identificador seguido de paréntesis (aunque en medio de estos no haya ningún valor), podremos asegurar que eso es una función. No confundamos los identificadores con las palabras reservadas (recordemos que después de la palabra reservada `if` siempre deben ir paréntesis, pero eso no es una función).

Así, tenemos funciones estándares para las operaciones más típicas, que podremos usar libremente, como son: `printf()`, `scanf()`, `puts()`, `gets()`, `pow()`, `exp()`... y muchas más. Nosotros podremos usarlas o incluso construirnos nuestras propias funciones para esas u otras finalidades.

Además, como ya se ha dicho en el capítulo 1, es muy importante modularizar correctamente un programa. Es decir, aunque se pueda hacer un programa con pocas funciones lo ideal es hacer tantas como sean necesario para que el programa sea fácil de entender (y por tanto fácil de modificar y corregir errores).

En este capítulo proponemos algunas funciones que pueden ser muy útiles en algunos otros programas y que nos parecen muy didácticas para comprender la filosofía fundamental de la programación en C.

6.1 Funciones en general. Conceptos básicos.

En este apartado se tratarán los conceptos elementales del manejo de funciones, pero también se incluirán funciones en las que lo más importante es el algoritmo que se ha seguido para solucionar el problema.

6.1.1 Función Perímetro.

Haga una función que tenga un argumento de tipo `unsigned` que será el radio de una circunferencia, y devuelva un número real (`float`) que será el perímetro de la circunferencia. Además, programe una función principal que use la función anterior.

Solución:

```
#include <stdio.h>

float perimetro (unsigned); /* Cabecera o Prototipo de la función */
                            /* Aquí se pone esta cabecera y la definición
                               de la función se incluye despues de main(). */

void main()
{ int radio;

  printf ("\n- Deme el radio de una circunferencia: ");
  scanf("%i",&radio);

  printf ("\n El perímetro es: %f.", perimetro(radio) );
  puts("\n ***** FIN *****");
}

/*****************************************************/
/* Objetivo: Calcular el perímetro de una circunferencia */
/* Entrada : El radio de la circunferencia.            */
/* Salida  : Devuelve el perímetro de la circunferencia  */
/*****************************************************/
float perimetro (unsigned radio) /* Definición o Implementación de la función */
{
  return 2*3.141592*radio;
}
```

Observe que la operación `2*3.141592`, al ser entre constantes produce siempre el mismo resultado, por lo que sería más eficiente poner directamente el resultado:

```
  return 6.283184*radio;
```

6.1.2 Área de un círculo.

Calcular el área de un círculo, conocido su radio, mediante una función con un parámetro entero que indicará el radio de un círculo. La función devolverá un valor real que debe ser el radio del círculo. Implemente además una pequeña función principal que utilize (llame) a la anterior función.

Solución:

```
/************************************************************/
/* Objetivo: calcular el area de un circulo a partir del radio */
/* Entrada: r: el radio.                                      */
/* Salida: valor del area devuelto con return.                */
/************************************************************/
#include <stdio.h>
```

```
#define PI 3.141592

float calculo_area(int r)
{
  float a;

  a = PI * r * r;
  return a;
}

void main (void)  /* función principal */
{ float area = 0.0;
  int r;

  printf ("Introduzca el radio: "),
  scanf ("%d", &r);
  area = calculo_area(r );     /* llamada a la función que calcula el área */
  printf ("\nEl área del círculo es: %f";area);
}
```

Observe que en este ejemplo hemos puesto la implementación de la función antes de la función principal. De esta forma no es necesario poner antes el prototipo o declaración de la función. Aunque esta es otra forma válida, recomendamos poner los prototipos de las funciones que implementemos, ya que esto nos ayudará a tener una visión más clara del programa, facilitándonos en todo momento el número de funciones que tenemos y como acceder a cada una de ellas.

6.1.3 Operaciones aritméticas con funciones.

Dados dos números enteros, introducidos desde el teclado calcular e imprimir el resultado de suma (+), diferencia (-), producto (*) y división (/). Utilizar una función con parámetros para cada operación.

Observe que la división entre dos números enteros puede ser un número real.

Solución:

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

/****************************************************/
/* Objetivo: sumar dos números.                     */
/* Entrada: v1,v2 (numeros a sumar).                */
/* Salida: suma devuelta con return.                */
/****************************************************/
int suma(int v1, int v2)
{
  return v1+v2;
}
```

```
/****************************************************/
/* Objetivo: restar dos números.                    */
/* Entrada: Dos números enteros.                    */
/* Salida: Devuelve la resta.                       */
/****************************************************/
int diferencia(int v1, int v2)
{
  return v1-v2;
}

/****************************************************/
/* Objetivo: Dividir dos números.                   */
/* Entrada: Dos números enteros.                    */
/* Salida: Devuelve la división (en un float).      */
/****************************************************/
float division(int v1, int v2)
{
  return v1/ (float)v2;
}

/****************************************************/
/* Objetivo: Multiplicar dos números.               */
/* Entrada: Dos números enteros.                    */
/* Salida: Devuelve la multiplicación.              */
/****************************************************/
int multiplicacion(int v1, int v2)
{
  return v1*v2;
}

void main()
{ int a,b;

  clrscr();
  printf("\nIntroduzca dos valores: ");
  scanf("%d",&a);
  scanf("%d",&b);

  printf("\n\nLa suma es: %d", suma(a,b));
  printf("\nLa diferencia es: %d",diferencia(a,b));
  printf("\nEl producto es: %d", multiplicacion(a,b));
  printf("\nLa división es: %f", division(a,b));
}
```

Téngase en cuenta que el resultado de la operación entre dos enteros es un entero, por lo que la división entre enteros resulta sin decimales. Para que contenga decimales lo que hacemos

es convertir a float el denominador de la división con un operador de moldeado (*casting*).

Este es un ejercicio sin demasiada importancia, ya que en ningún caso merece la pena hacer una función para efectuar estas operaciones básicas. Lo importante de este ejercicio es ver cómo se pueden relacionar las funciones entre sí y como se comunican datos entre ellas en ambos sentidos de la comunicación.

6.1.4 Conversión Km/h a m/s con función.

Convertir Kms/h. a mts/seg, llamando a una función que realice la conversión y muestre el resultado en pantalla. Implemente una función principal que lea un valor para los Km/h y efectúe una llamada a la función anterior.

Solución:

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

void funcion_float(float);

void main(void)
{ float kmh;

  clrscr();
  printf("Introduzca los Kms/h para efectuar conversión: ");
  scanf("%f", &kmh);
  funcion_float(kmh);
}

/********************************************************/
/* Objetivo: convertir km/h a m/s y mostrar en pantalla. */
/* Entrada: Un real: kmh (km/h).                         */
/********************************************************/
void funcion_float(float kmh)
{
  float mts;

  mts = (float) (kmh * 1000 / 3600);
  printf("\nLa conversión resultante es: %3.2f Kms/h corresponden a %5.2f Mts/seg."
```

6.1.5 Mayor de dos números con función.

Leer dos números (desde la entrada estándar) y determinar cuál es el mayor, y la diferencia entre ambos, llamando a una función que determine el mayor y muestre la diferencia. Implemente una función principal para probar la función anterior.

Solución:

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

```
#include <conio.h>

void chequeo(int, int);

void main(void)
{ int num1;
  int num2;

  clrscr();
  printf("Introduzca el primer número: ");
  scanf("%d", &num1);
  printf("Introduzca el segundo número: ");
  scanf("%d", &num2);
  chequeo(num1, num2);
}

/****************************************************************/
/* Objetivo: comprueba cuál es el mayor de 2 números y muestra la */
/*           diferencia en pantalla.                             */
/* Entrada: num1, num2 (los dos números).                        */
/****************************************************************/
void chequeo(int num1, int num2)
{
  int diferencia;

  if (num1>num2){
     diferencia = num1 - num2;
     printf("El %d es mayor que %d en %d unidades\n", num1, num2, diferencia);
  }
  else
    if (num2>num1) {
       diferencia = num2 - num1;
       printf("El %d es mayor que %d en %d unidades\n", num2, num1, diferencia);
    }
    else
       printf("Ambos números son iguales\n");
}
```

6.1.6 Números en un intervalo con función.

Programa que lea dos números (desde la entrada estándar) y muestre en pantalla los números comprendidos entre ambos. La función principal deberá llamar a una función que muestre los números comprendidos entre los dos. Tenga en cuenta que no se sabe cual de los números es mayor (pueden leerse en cualquier orden).

Solución:

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

void desde_hasta(int, int);

void main(void)
{ int num1;
  int num2;

  clrscr();
  printf("Introduzca el primer número: ");
  scanf("%d", &num1);
  printf("Introduzca el segundo número: ");
  scanf("%d", &num2);
  printf("\n");
  desde_hasta(num1, num2);
}

/*****************************************************/
/* Objetivo: mostrar los números comprendidos entre dos. */
/* Entrada: num1,num2 (los dos números).               */
/*****************************************************/
void desde_hasta(int num1, int num2)
{
  int i;

  if (num1>num2)
     for (i=num1;i>=num2;--i)
         printf("%d ", i);
  else
     if (num2>num1)
        for (i=num1;i<=num2;++i)
            printf("%d ", y);
     else
        printf("Ambos números son iguales\n");
}
```

6.1.7 Cuadrante.

Escribir una función que reciba como entradas las coordenadas cartesianas de un punto del plano `(x,y)`, y devuelva como resultado el cuadrante al cual pertenece el punto (1, 2, 3 ó 4). La función devolverá 0 si el punto está en un eje. Realice una función principal para probar que esta función devuelve correctamente el resultado.

Solución:

```
#include <stdio.h>

int cuadrante (int, int);

void main()
{ int x, y, /* Coordenadas de un punto */
      cuad; /* Cuadrante del punto     */

  printf("\n - Dame las coordenadas del punto: X = "); scanf("%i",&x);
  printf("\n                                   Y = "); scanf("%i",&y);

  if (cuad=cuadrante(x,y))
       printf("\n - El punto (%i,%i) está en el cuadrante %i.\n",x,y,cuad);
  else printf("\n - El punto está en un eje de coordenadas.\n");
}

/*************************************************************/
/* Objetivo: Averiguar el cuadrante de un punto.            */
/* Entrada : Coordenadas del punto (x,y).                   */
/* Salida  : Cuadrante del punto. 0 si está en un eje.      */
/*************************************************************/
int cuadrante (int x, int y)
{
  if (x == 0 || y == 0) return 0; /* Está en un eje */

  if (x>0) {
     if (y>0) return 1;
     return 4;
  }

  if (y>0) return 2;

  return 3;
}
```

Hay que tener en cuenta, que cuando en una función se ejecuta una sentencia `return`, lo que hace es devolver el valor a la función que efectuó la llamada (si hay tal valor), y por tanto, termina la ejecución de dicha función. Por este motivo, no es necesario usar la palabra reservada `else` en los `if` del programa.

Observe como se usa el operador asignación en la función principal cuando se ejecuta la llamada a la función `cuadrante()`. En este caso, el programa debe evaluar la expresión de asignación: primero se evalúa la función y luego se asigna el valor que esta devuelva a la variable `cuad`. El operador de asignación (=) devuelve el valor que asigna. Así, si la función devuelve un 0, la expresión de asignación se evaluará como 0, es decir, como falsa e irá a ejecutar la parte `else`. En cualquier otro caso se evaluará como verdadera (distinta de cero).

6.1.8 Años bisiestos.

Realizar una función en C que nos diga si un año es bisiesto o no. La función aceptará un único argumento que será el año que queremos saber si es o no bisiesto y devolverá un valor de verdad si es bisiesto y falso si no lo es. Recuérdese que en C cualquier valor distinto de cero es verdad y el cero se toma como falso.

Como todo el mundo sabe, un año es bisiesto:

1. Si es divisible por 4 y no por 100.
2. Caso de ser divisible por 100 que lo sea también por 400.

La explicación es que una vuelta de la Tierra al Sol dura *algo menos* de 365 días y 6 horas. Esas 6 horas se acumulan cada 4 años (6*4=24) para dar lugar a un nuevo día (el 29 de Febrero) en los llamados años bisiestos. Pero como no son 6 horas justas, sino que es *algo menos*, cada cierto tiempo hay que evitar que un año sea bisiesto. Al final de cada siglo (los divisibles por 100) los años no son bisiestos, salvo que sean divisibles por 400. Por ejemplo, el año 2000 será bisiesto y el año 1900 no lo fué.

Solución:

```
/****************************************************/
/* Objetivo: Averiguar si un año es o no Bisiesto.  */
/* Entrada : El año (entero positivo)               */
/* Salida  : Devuelve: 1 (verdad) si es bisiesto.   */
/*                     0 (falso)  si no lo es.      */
/****************************************************/
char bisiesto (unsigned a)
{
  if ( !(a%4)   &&  (a%100) ||
       !(a%100) && !(a%400)
     )
     return 1;

  return 0;
}
```

La función devuelve un 0 ó un 1. Como sólo vamos a devolver esos valores, un tipo de dato `char` es más que suficiente. Podríamos haber puesto que se devolviera un `int` o un `unsigned int` y también estaría correcto.

6.1.9 Mínimo Común Múltiplo (MCM).

En un ejercicio anterior se ha visto cómo calcular el Máximo Común Divisor (MCD) de dos números. Ahora proponemos realizar una función cuyo prototipo sea:

```
unsigned long int MCM (unsigned, unsigned);
```

que devuelva el Mínimo Común Múltiplo (MCM) de dos enteros sin signo que se pasan como argumentos.

Solución:

```
/**************************************************/
/* Objetivo: Hallar el Mínimo Común Múltiplo.      */
/* Entrada : Dos enteros positivos (sin signo).    */
/* Salida  : Devuelve el MCM de ambos números.     */
/* Requisitos: Que el producto de ambos números se */
/*           pueda almacenar en el tipo de dato de */
/*           salida (unsigned long).               */
/**************************************************/
unsigned long int MCM (unsigned x, unsigned y)
{ unsigned long MCM = x * y,        /* MCM por el momento */
num = MCM - 2,       /* Número posible MCM */
max = x>y ? x : y;  /* MCM más pequeño posible */

  while (num >= max) {
    if ( !(num%x) && !(num%y) ) /* Si num es múltiplo de x e y */
       MCM = num;
    num = num - 2;
  }

  return MCM;
}
```

Está claro que el MCM de dos números estará siempre comprendido entre el mayor de los dos (`max`) y el producto de ambos, siendo ese producto un común múltiplo seguro. Se trata de examinar uno a uno los números de ese rango y escoger el que sea más pequeño de todos que sea múltiplo de los dos números en cuestión.

Como caso extremo está la situación en la que no encontramos ningún número en ese rango que sea múltiplo de ambos números. En ese caso, el MCM será el producto de ambos. Esa es la razón del requisito que se establece en la función, ya que el producto de ambos puede ser un número bastante grande y puede que sea el MCD. Si se produce un desbordamiento (*overflow*) los resultados son impredecibles.

Si uno de los números de entrada es par (o los dos), el MCM será también par, por lo que no es necesario examinar los números impares. Igualmente, si ninguno de los números es par, el MCM será impar, por lo que no tenemos que examinar los números pares. Esa es la razón de que se vayan examinando números de 2 en 2, restando 2 a la variable `num`.

Observe que a una variable de una función se le puede poner el mismo identificador que a esa función. En general, es mejor distinguirlas con distinto identificador. Como norma, en una función no puede existir una llamada a una función con el mismo identificador que una variable.

6.1.10 Números primos.

Determinar si un número es primo o no, utilizando una función que devuelva un valor 1 ó 0 (verdad o falso), según el caso. Implementar una función principal para probar su funcionamiento.

Solución:

```
#include <stdio.h>

int primo(int);

void main(void)
{ int i;

  printf("Introduzca el número: ");
  scanf("%d", &i);
  if (primo(i))
     printf("\n\nEl %d es un número primo", i);
  else
     printf("\n\nEl %d no es un número primo", i);
}

/**************************************************/
/* Objetivo: determinar si un número es primo o no.  */
/* Requisitos: y >= 2                               */
/* Entrada: y (numero a determinar si es primo).    */
/* Salida: devuelta con return (0 no primo, 1 primo).*/
/**************************************************/
int primo (int y)
{ int contador;

  for (contador = 2; contador < i; contador++)
      if (!(i%contador)) return 0;

  return 1;
}
```

6.1.11 Raíces reales de una ecuación con funciones.

Calcular las raíces de una ecuación de segundo grado (sólo en el caso de solución real):

$$ax^2 + bx + c = 0$$

El programa leerá los coeficientes a, b y c desde el teclado, y llamará a una función que debemos definir cuyo prototipo es:

```
void raiz_seg_grado (int a, int b, int c, float *x1, float *x2);
```

Solución:

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

```
void raiz_seg_grado(int a, int b, int c, float *x1, float *x2);

void main(void)
{ int a,b,c;
  float x1;
  float x2;
  double paso;

  clrscr();
  printf("Introduce el coeficiente de Xý: ");
  scanf("%d", &a);
  printf("Introduce el coeficiente de X: ");
  scanf("%d", &b);
  printf("Introduce el termino independiente: ");
  scanf("%d", &c);
  paso = (double) b*b - (double) 4*a*c;

  if (paso < 0) {
     printf("\nNo existen soluciones reales a estos valores\n");
     printf("La ecuación %d X^2 + %d X + %d no tiene soluciones.\n", a,b,c);
     return 0;
  }

  raiz_seg_grado(a, b, c, &x1, &x2);
  printf("\nLa ecuación %d X^2 + %d X + %d tiene soluciones:\n", a,b,c);
  if (x1 != x2)
     printf("Soluciones: %5.1f y %5.1f", x1, x2);
  else
     printf("Solucion única: %5.1f", x1);
}

/*************************************************************/
/* Objetivo: determinar las raíces de una ecuanción de grado 2. */
/* Requisitos: los coeficientes deben corresponder a una       */
/*             ecuación con solución.                           */
/* Entrada: a,b,c (coeficientes de la ecuación).               */
/* Salida: x1,x2 (soluciones o raíces).                         */
/*************************************************************/
void raiz_seg_grado(int a, int b, int c, float *x1, float *x2)
{ double paso = 0;

  paso = (double) b*b - (double) 4*a*c;
  *x1 = (-b + sqrt(paso)) / (2*a);
  *x2 = (-b - sqrt(paso)) / (2*a);
}
```

6.1.12 Fibonacci, iterativamente.

Realizar una función, `fibonacci(n)` que calcule el término n-ésimo de la sucesión de Fibonacci, de forma iterativa (no recursiva). La sucesión de Fibonacci está definida de la siguiente forma:

```
fibonacci (0) = 0
fibonacci (1) = 1
fibonacci (i) = fibonacci (i-1) + fibonacci (i-2)
```

Además, codificar un programa de prueba que pida sucesivamente un número `n` y de el valor del término `n`-ésimo de esta sucesión, hasta que se le introduzca un número negativo.

Solución:

```
#include <stdio.h>

unsigned long fibonacci (int);

void main ()
{ int x;

  printf ("Introducir número (negativo para terminar): ");
  scanf ("%i", &x);

  while (x >= 0) { /* Mientras sea positivo */
    printf ("Fibonacci (%i) = %lu.\n\n", x, fibonacci (x));
    printf ("Introducir número (negativo para FIN): ");
    scanf ("%i", &x);
  }

  puts ("\n***** Fin del programa. *****");
}

/************************************************************/
/* Objetivo: Hallar el término n-ésimo de la sucesión      */
/*           de Fibonacci.                                  */
/* Entrada : Número n.                                      */
/* Salida  : n-ésimo término de la sucesión de Fibonacci.  */
/* Observaciones: Si el número resultante es muy grande    */
/*           el valor devuelto será impredecible.           */
/************************************************************/
unsigned long fibonacci (unsigned n)
{ unsigned long s1=0, s2=1, s3;
  unsigned i = 2;

  if (n == 0) return 0; /* fibonacci (0) = 0 */
  if (n == 1) return 1; /* fibonacci (1) = 1 */
```

```
  for ( ; i <= n ; i++) {
     s3 = s1 + s2;
     s1 = s2;
     s2 = s3;
  }

  return s3;
}
```

Inicialmente, en `s1` y `s2` se almacenan el primer y segundo valor de la sucesión de Fibonacci, respectivamente. Para un n>1, lo que hace es hallar sucesivamente el siguiente término y almacenarlo en `s3`, con la instrucción `s3 = s1 + s2;`, y a continuación cambia los valores de `s1` y `s2` para igualarlos a los 2 últimos términos hallados.

6.1.13 Función intercambio de valores: `swap()`.

Codificar una función que sirva para intercambiar los valores de dos variables enteras. Su prototipo será:

```
void swap (int *, int *);
```

Para la llamada a la función se utilizará el operador de dirección (&): `swap (&a, &b);` siendo `a` y `b` dos variables enteras.

Solución:

```
/*********************************************************/
/* Objetivo: Intercambiar valores de dos variables enteras. */
/* Entrada : Las direcciones de dos variables enteras.     */
/*********************************************************/
void swap (int *X, int *Y)
{ int intermedia = *X;

  *X = *Y;
  *Y = intermedia;
}
```

NOTA: Esta función se usará en la función de ordenación de un array por selección, en un ejercicio posterior.

6.1.14 Distancia euclídea.

Determinar la distancia entre dos puntos P y Q (leidos por teclado), con una función cuyo prototipo sea:

```
float distancia_euclidea (punto P, punto Q);
```

y que imprima el resultado con una función cuyo prototipo sea.

```
void imprime_distancia (float distancia);
```

El tipo `punto` se tendrá que definir como una estructura con dos enteros con `typedef`

Solución:

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

typedef struct{
          int x;
          int y;
}punto; /* definición del tipo de datos punto, como una estructura*/

/**********************************************************/
/* Objetivo: hallar la distancia euclidea entre dos puntos. */
/* Entrada: P,Q (variables de tipo punto).                  */
/* Salida: con return la distancia.                         */
/**********************************************************/
float distancia_euclidea (punto P, punto Q)
{ float d;

  d =(float) sqrt(pow((double)(Q.x-P.x),2.0)+pow((double)(Q.y-P.y),2.0));
  return d;
}

/***************************************************/
/* Objetivo: mostrar un valor flotante en pantalla.  */
/* Entrada: distancia (valor a mostrar).             */
/***************************************************/
void imprime_distancia(float distancia)
{
  printf("\nLa distancia euclidea entre los dos puntos es: %g",distancia);
}

void main (void)
{ punto P,Q;
  float distancia;

  printf("\nIntroduce las coordenadas de los dos puntos.");
  printf("\nCoordenadas P (x,y) (enteras): ");
  scanf("%d %d",&P.x,&P.y),
  printf("\nCoordenadas Q (x,y) (enteras): ");
  scanf("%d %d",&Q.x,&Q.y),
  distancia=distancia_euclidea(P,Q);
  imprime_distancia(distancia);
}
```

6.1.15 Integral definida.

Calcular la integral definida de una función f(x). El programa deberá leer por teclado el intervalo en el que se pretende calcular la integral (el límite inferior a y el límite superior b), y también deberá leer el número de pasos en que se pretende dividir el intervalo (npasos). Se deberá imprimir en pantalla el resultado de la integral. En cada paso se computará una aproximación al área bajo la curva, de tal forma que será igual al área de un rectángulo que como base tendrá la medida (b-a)/npasos, y como altura el valor de la función en el primer punto de ese paso, es decir, por ejemplo, si estamos en el primer paso estaremos considerando el trozo de intervalo (a,a+(b-a)/npasos), con lo que la altura del rectángulo será f(a).

La función f(x) está dada, por ejemplo, por el siguiente código en C:

```
float f(float x)
{
  return x * x / 2 + 3;
}
```

Solución:

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

float f ( float x );

void main ()
{ int a, b,
      npasos;
  float i,
        dif , x , integral,
        solucion = 0;

  clrscr ();
  printf ( "L¡mite inferior: ");
  scanf ( "%d" , &a );
  printf ( "L¡mite superior: ");
  scanf ( "%d" , &b );
  printf ( "Número de pasos: ");
  scanf ("%d", &npasos);

  dif = ( float ) ( b - a ) / ( float ) npasos;

  for ( i = a + dif; i < b + dif; i = i + dif ) {
      integral = dif * f ( i );
      solucion = solucion + integral;
  }

  printf ( "%f", solucion );
}
```

```
/**************************************************/
/* Objetivo: hallar el valor de una funcion f(x)    */
/* Entrada: x (valor de abscisa).                   */
/* Salida: con return valor de f(x).                */
/**************************************************/
float f ( float x )
{
  return x * x / 2 + 3;
}
```

6.1.16 Operación XOR lógica.

En C existen los 3 operadores lógicos básicos, que son los siguientes (enumerados de mayor a menor precedencia):

- Negación lógica (NOT): `!`
- Y lógico (AND): `&&`
- O lógico (OR): `||`

Existe otro operador lógico llamada O exclusivo, **XOR** (*eXclusive OR*), que toma el valor verdad si es verdad uno sólo de sus operandos, es decir, si V es Verdad (1) y F es Falso (0):

A	B	A XOR B
F	F	F
F	V	V
V	F	V
V	V	F

Escribir una función que acepte 2 valores enteros `A` y `B` y que realize la operación anterior, devolviendo el resultado de `A` XOR `B`.

Solución:

```
int XOR (int A, int B)
{
  return (A && !B) || (!A && B);
}
```

La explicación es sencilla, ya que el XOR devolverá verdad (1) si es verdad `A` **Y** NO `B`, **O** si es verdad NO `A` **Y** `B`.

Los paréntesis no son necesarios, ya que la operación `&&` tiene mayor prioridad que la `||`, pero se han puesto para aumentar la claridad.

Al usar la función anterior, podemos usar en sus argumentos no sólo valores enteros típicos, sino también expresiones lógicas o relacionales, pudiendo por ejemplo hacer algo como lo siguiente:

```
if (XOR (x>0, y>0) ) {

    . /* Esto se ejecuta si x O y son mayores que cero, */
    . /* pero no si lo son ambas.                        */

}
```

En el `if` anterior, la condición será cierta si `x` es mayor a cero, pero no lo es `y`, o viceversa.

6.1.17 El valor de las expresiones con punteros.

Suponga que tenemos la siguiente declaración de variables en un programa en C:

```
int a = 6,
    b = 7,
    *pa, *pb, *pc;

pa = &a,
pb = &b,
pc = pa;
```

Indique el valor de las siguientes expresiones, rellenando la tabla siguiente.
NOTA: Recuerde que las expresiones relacionales valen 0 si son falses y 1 si son verdaderas.

Número	Expresión	Valor
1.	`a`	
2.	`*(&b)`	
3.	`*pa * a`	
4.	`*pb + *pa`	
5.	`*pc + b`	
6.	`*pc + *(&(*pa))`	
7.	`a > *pb`	
8.	`*pb >= (*pc - 1)`	
9.	`pc == &a`	
10.	`*pb == &a`	
11.	`(*pc - *pb) == (*pa - b)`	
12.	`a >= *pc`	
13.	`&pc == &a`	
14.	`&pc == &pa`	
15.	`&pc == pa`	
16.	`&pc == pb`	
17	`&pc == pc`	

Solución:

Número	Expresión	Valor
1.	`a`	6
2.	`*(&b)`	7
3.	`*pa * a`	36
4.	`*pb + *pa`	13
5.	`*pc + b`	13
6.	`*pc + *(&(*pa))`	12
7.	`a > *pb`	0
8.	`*pb >= (*pc - 1)`	1
9.	`pc == &a`	1
10.	`*pb == &a`	0
11.	`(*pc - *pb) == (*pa - b)`	1
12.	`a >= *pc`	1
13.	`&pc == &a`	0
14.	`&pc == &pa`	0
15.	`&pc == pa`	0
16.	`&pc == pb`	0
17.	`&pc == pc`	0

La explicación a cada apartado es la siguiente:

1. La expresión `a` toma el valor de la variable `a` que es 6, el valor que se asignó a dicha variable en la declaración
2. La expresión indica el valor que hay almacenado en la dirección de memoria de `b` el cual es el valor de `b`, o sea 7
3. Producto de dos valores: El almacenado en la dirección `pa`, que es 6 por `a`. `6 * 6 = 36`
4. Suma el valor almacenado en la dirección `pb` con el almacenado en la dirección `pa`. `7 + 6 = 13`
5. Suma el valor almacenado en la dirección `pc` con el almacenado la variable `b`. `6 + 7 = 13`
6. Suma el valor almacenado en la dirección `pc` con el almacenado en la dirección `pa`. `6 + 6 = 12`
7. El valor de la variable `a` (6) NO es mayor que el contenido de la dirección de memoria `pb` (7)
8. El contenido de la dirección `pb` (7) es mayor o igual que el contenido de la dirección `pc` (7) menos 1
9. La variable `pc` almacena una dirección que es la misma que la dirección de `a`
10. No es igual el contenido de la dirección `pb` (que es de tipo entero) con la dirección de la variable `a` (que es de tipo puntero)
11. Resumiendo, es cierto porque: `(6-7) == (6-7)`

12. Es cierto porque: `6 >= 6`

13. La dirección donde se almacena el puntero `pc` no es igual a la dirección de `a`. Si fuera así, ambas variables se almacenarían en el mismo sitio, cosa que no tiene ningún sentido y es imposible

14. La dirección donde se almacena el puntero `pc` no es igual a la dirección donde se almacena la variable puntero `pa`

15. La dirección donde se almacena el puntero `pc` no es igual a `pa`, que es la dirección de `a`

16. La dirección donde se almacena el puntero `pc` no es igual a `pb`

17. La dirección donde se almacena el puntero `pc` no es igual a `pc`

6.2 Funciones de cadenas de caracteres.

Practicamente en cualquier programa de cierta complejidad será necesario manejar cadenas de caracteres: Para almacenar un nombre, dirección, la matrícula de un coche, un número de teléfono (con guiones...), una ciudad... En general, cualquier cosa que no sea numérica o que no requiera un tratamiento numérico.

Las constantes de tipo carácter (`char`) se encierran entre comillas simples, como por ejemplo los caracteres `'A'`, `'a'`, `'\n'`, `'+'`, etc... Las constantes de tipo cadena de caracteres se encierran siempre entre comillas dobles, como por ejemplo la constante:

```
"Como esto está entre comillas dobles es una constante de tipo cadena."
```

En C las cadenas de caracteres son simples arrays de caracteres en los que situamos el carácter nulo `'\0'` (valor entero cero) en la última posición de la cadena (no del array). Así, podemos tener un array de caracteres de 50 caracteres y almacenar ahí el valor `"Paz"`. Para ello tendremos que asignar la `'P'` a la posición 0 del array, la `'a'` a la posición 1, la `'z'` a la posición 2 y el carácter nulo `'\0'` a la posición 3 del array.

No debemos olvidar nunca el carácter nulo pues muchas funciones lo utilizan para saber hasta donde llega la cadena. Si este, por error, no existe la función mirará las posiciones de memoria consecutivas en la memoria hasta encontrarlo por casualidad. Este comportamiento puede dar resultados inesperados.

A continuación proponemos una serie de ejercicios que clarifican el uso de cadenas de caracteres en C.

6.2.1 Comparación de cadenas.

Hay una función de biblioteca (de `string.h`) que sirve para comparar cadenas. Se llama `strcmp()` (*string comparison*) y tiene dos argumentos de entrada, las dos cadenas a comparar. Devuelve un entero que será:

strcmp(s1,s2)	Condición
0	Ambas cadenas son iguales
< 0	Cadena s1 menor que s2
> 0	Cadena s1 mayor que s2

El valor devuelto es la diferencia entre los dos primeros caracteres que sean distintos. Ejemplos:

s1	s2	strcmp(s1,s2)
ABC	ABC	0
ABC	ABD	-1
ABD	ABC	1
AB	ABA	-65
ABA	AB	65
ABB	AB	66

El objetivo es realizar una función `strcomp()` que obtenga los mismos resultados que `strcmp()`.

Solución:

```
/*********************************************************/
/* Objetivo: Comparar cadenas de caracteres.             */
/* Entrada : Dos cadenas de caracteres.                  */
/* Salida  : Devuelve la diferencia entre los dos primeros */
/*           caracteres que sean diferentes de las cadenas */
/*********************************************************/
int strcomp (char s1[], char s2[])
{ int i=0;

  for ( ; s1[i] ; i++)
      if (s1[i]!=s2[i])
         return (s1[i] - s2[i]);

  return (s1[i] - s2[i]);
}
```

6.2.2 Leer una cadena desde teclado.

Hacer un programa que simule de la mejor forma posible las capacidades que tiene la función de biblioteca `gets()`. Esta función es capaza de leer una cadena desde la entrada estandar y almacenerla en una variable cadena que se le pasa a la función.

Solución:

```
/****************************************************************/
/* Objetivo: Hacer una función que lear una cadena desde la     */
/*           entrada estandar                                   */
/* Salida :  Una cadena                                         */
```

```
/*************************************************************/
char *xgets(char *c)
{ char car, *p;
  int t;

  p = c; /* se devuelve un puntero a la cadena, es decir
            se tiene una cadena de caracteres */

  for(t=0; t<80; t++) {
    car = getchar();
    switch(car) {
      case '\n': c[t] = '\0'; /* termina de leer la cadena */
                 return p;
      case '\b': if(t>0)
                 t--; /* se borra un caracter de la cadena leida */
                 break;
      default: c[t] = car;
    }
  }
  c[80] = '\0';
  return p;
}
```

En nuestra solución hemos simplificado mucho el tratamiento que hacemos a la hora de leer una cadena desde la entrada estandar. Una de las primeras limitaciones que hemos puesto a nuestro programa es que sólo podremos leer cadenas de 80 caracteres como máximo. En *C* no hay comprobación de rango en los arrays por tanto debemos siempre de tener mucho cuidado cuando pasamos arrays a las funciones o cuando les asignamos valores, siempre debemos de asegurarnos que hemos reservado memoria suficiente.

Como ejercicio adicional podemos modificar el programa anterior para que tratemos un mayor número de cosas que nos pueden suceder cuando leemos una cadena de caracteres desde el teclado.

6.2.3 Palíndromo, iterativamente.

Escribir una función que acepte una cadena de caracteres y devuelva si dicha cadena es, o no, un palíndromo. Una cadena (frase) es un palíndromo si se lee igual hacia delante que hacia atrás, carácter a carácter, ignorando los espacios en blanco. Por ejemplo:

```
"Dabale arroz a la zorra el abad"
```

o

```
"Reconocer"
```

La cadena de entrada se dará sin acentos, para suponer que las letras acentuadas y sin acentuar son la misma letra. Además, para ignorar la diferencia entre mayúsculas y minúsculas

se puede usar la función `toupper(c)`, que devuelve el carácter `c` en mayúsculas. También existe la función `tolower(c)` que convierte a minúsculas.

Escribir también un pequeño programa que pida una frase y devuelva si es o no palíndromo, usando la función anterior

Solución:

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

#define DIM 100

char palindromo (char *);

void main ()
{ char frase[DIM];

  puts ("\nEscribe la cadena:");
  gets (frase);

  if (palindromo (frase))
        puts ("\n- Es un palíndromo.");
  else  puts ("\n- NO es un palíndromo.");
}

/**************************************************************/
/* Objetivo: Averiguar si un string es Palíndromo.           */
/* Entrada : Cadena de caracteres.                            */
/* Salida  : 1 (verdad) si es Palíndromo, y 0 si no lo es.   */
/**************************************************************/
short palindromo (char *cadena)
{ int i=0,                 /* Contador que empieza por el Inicio de cadena */
      f=strlen(cadena)-1; /* Contador que empieza por el Final  de cadena */

  while (i<f) {
    while (cadena[i]==' ' && i<f) /* Saltamos espacios iniciales */
          i++;
    while (cadena[f]==' ' && i<f) /* Saltamos espacios finales   */
          f--;
    if (i<f) { /* Si no se han cruzado los índices */
       if (toupper(cadena[i]) != toupper(cadena[f]) )
          return 0;  /* Si llega hasta aquí NO es palíndromo */
       i++;
       f--;
    } /*if*/
  } /*while*/
```

```
  return 1; /* Si llega hasta aquí SI es palíndromo */
}
```

La función `strlen(cadena)` de `string.h` devuelve la longitud de la cadena.

Usamos dos índices `i` y `f` que comienzan apuntando por el inicio y por el final de la cadena respectivamente. Siempre que encontremos espacios en blanco (' ') al principio o al final, nos los saltamos. El algoritmo va comparando los caracteres en las posiciones `i` con los caracteres en las posiciones `f`. En el momento que encontremos una diferencia, la cadena no es palíndromo y devuelve 0. Si son iguales, incrementamos `i` y decrementamos `f`.

El algoritmo termina cuando encontremos una diferencia (devuelve 0) o cuando `i` sea mayor o igual que `f`, lo cual indica que ya hemos comparado la primera mitad de la cadena, con la segunda mitad, sin encontrar diferencias, por lo que sí es palíndromo (devuelve 1).

En cualquier momento, si no se cumple que `i<f` podemos abandonar el algoritmo sabiendo que sí es palíndromo. Por eso se incluye esta condición en cada paso.

6.2.4 Buscar subcadena.

Codificar una función que acepte dos cadenas de caracteres como parámetros, y que compruebe si la segunda cadena `s2` es una subcadena de la primera `s1`. El programa devolverá un entero que indicará la posición del primer carácter de la segunda cadena `s2` dentro de `s1`. Devolverá -1 si `s2` no está contenida en `s1`. Si existen varias ocurrencias de `s2` en `s1` se devolverá la posición de la primera de ellas, ignorando el resto.

La cabecera (o prototipo) de la función será:

```
int busca (char s1[], char s2[]);
```

Ejemplo:

```
  s1 = "Imagina que hay una GUERRA, y no vamos NADIE"
  s2 = "que"
```

La función devolverá el entero 8.

Otro ejemplo:

```
  s1 = "las lágrimas te impedirán ver las estrellas  -Tagore-"
  s2 = "im"
```

La función también devolverá el entero 8 (porque es la posición de la primera ocurrencia de las dos que hay).

Solución:

```
/************************************************************/
/* Objetivo: Devuelve la posición de la segunda cadena      */
/*           en la primera (de la primera ocurrencia).      */
```

```
/* Entrada : Dos cadenas de caracteres, terminadas en '\0'  */
/* Salida  : Posición de una en otra. -1 si la segunda      */
/*           no está incluida en la primera.                */
/************************************************************/
int busca (char c1[], char c2[])
{ int pos=0, /* Posición por la que empezamos a buscar en c1   */
      i;     /* Para seguir la cadena c2, y c1 a partir de pos */

  while (1) {
    for (i=0 ; c1[pos+i]!='\0' && c2[i]!='\0' ; i++)
        if (c1[pos+i]!=c2[i]) break;
    if (c2[i]     == '\0') return pos;
    if (c1[pos+i] == '\0') return -1;
    pos++;
  }
}
```

El bucle principal se repetirá hasta que salgamos de la función con un `return`. El bucle `for` mira si `c1` (empezando en `pos`) y `c2` son iguales. En cuanto encuentre una diferencia, se sale del bucle a través del `break`, incrementa `pos` y vuelve a empezar la comparación.

Del bucle `for` también nos podemos salir por que lleguemos al final de una de las dos cadenas. Si llegamos al final de `c2` (o si llegamos al final de ambas cadenas), hemos encontrado que `c2` está incluida en `c1` y devolvemos la posición `pos` de `c1` donde comienza dicha inclusión. Si, por el contrario, se sale del `for` por llegar al final de `c1`, es que no hemos encontrado que `c2` esté incluida en `c1` y entonces, devolvemos `-1`.

El bucle `for` también se podría haber codificado de la siguiente forma, pero de esta segunda forma queda más lioso:

```
for (i=0 ; c1[pos+i]!='\0' && c2[i]!='\0' && c1[pos+i]!=c2[i]; i++);
```

o equivalentemente, aprovechando que el 0 es falso para C, y también lo es el `'\0'`.

```
for (i=0 ; c1[pos+i] && c2[i] && c1[pos+i]!=c2[i]; i++);
```

En algunos compiladores, la función anterior puede ocasionar un aviso de posible error (*Warning*), que nos avise de que dicha función debería devolver un valor (entero, en este caso). Esto se produce debido a que ambos `return` están en el cuerpo de un `if` y el compilador no reconoce la estrategia del algoritmo.

6.2.5 Cadenas incluidas una en otra.

Diseñar una función en lenguaje C en la cual, dadas dos cadenas de caracteres, devuelva el siguiente resultado:

- 0 Ambas cadenas son iguales (ambas incluidas una en la otra)
- 1 Si la primera cadena está incluida en la segunda
- 2 Si la segunda está incluida en la primera
- 3 No sucede nada de lo anterior

Por ejemplo, si llamamos c1 a la primera cadena y c2 a la segunda, teniendo los siguientes valores:

```
c1 = "¿ Energía nuclear ?  No, gracias, que me enveneno "
c2 = "que me enveneno"
```

la función devolverá un 2, ya que c2 es una subcadena de c1.

Usando el ejercicio anterior es muy fácil codificar esta función:

Solución:

```
/***********************************************************/
/* Objetivo: Comprobar la mutua inclusión entre cadenas.   */
/* Entrada : Dos cadenas de caracteres, terminadas en '\0'. */
/* Salida  : 0  Son iguales (ambas incluidas una en otra).  */
/*           1  Primera cadena incluida en la segunda.      */
/*           2  Segunda cadena incluida en la primera.      */
/*           3  No sucede nada de lo anterior.              */
/***********************************************************/
char strsub (char c1[], char c2[])
{ int c2enc1=busca(c1,c2),
      c1enc2=busca(c2,c1);

  if (c2enc1 == -1)
     if (c1enc2 == -1)
        return 3; /* No hay inclusiones  */
     else return 1; /* c1 incluida en c2   */

  if (c1enc2 == -1)
     return 2;       /* c2 incluida en c1   */
  return 0;          /* c1 y c2 son iguales */
}
```

Se trata de aplicar la función busca(), de un ejercicio anterior, en ambos sentidos:
busca(c1,c2) y
busca(c2,c1),
y luego observar donde ha devuelto un valor -1 que nos indique que la segunda cadena no está incluida en la primera.

6.2.6 Obtener las Siglas de una cadena.

Escribir una función que reciba como parámetro de entrada una cadena de caracteres que representa el nombre de una entidad, y devuelva como resultado, en otra cadena, el acrónimo de la misma. Por ejemplo:

Entrada: "Tren articulado ligero Goicoechea Oriol."

Salida: "T.A.L.G.O."

La primera letra de cada palabra no tiene que estar en mayúsculas, por lo que podemos usar la función `toupper(c)`, que devuelve el carácter `c` en mayúsculas. Además, entre dos palabras puede haber uno o más espacios en blanco y al principio de la cadena puede haber o no espacios en blanco. Otro ejemplo:

Entrada: "   Este es un   homenaje   a   Ghandi."

Salida: "E.E.U.H.A.G."

Solución:

```
/*********************************************************/
/* Objetivo: Hallar las siglas de una cadena de caracteres. */
/* Entrada : Cadena de caracteres: c                       */
/* Salida  : Cadena de caracteres: siglas                  */
/* Requisitos: Requiere que en la cadena de salida exista  */
/* suficiente espacio reservado para almacenar el acrónimo. */
/*********************************************************/
void siglas (char c[], char siglas[])
{ int i=0,   /* Indice para seguir la cadena c      */
      s=0;   /* Indice para seguir la cadena siglas */

  for ( ; c[i]==' ' ; i++ );  /* Saltamos los blancos iniciales */
  if (c[i]!='\0') {
     siglas[s++]=toupper(c[i]);  /* Primera letra. */
     siglas[s++]='.';            /* Su punto.      */
     i++;
  }

  for ( ; c[i]!='\0' ; i++ )
      if (c[i]==' ') {
         for ( ; c[i]==' ' ; i++ ); /* Salta blancos entre palabras */
         if (c[i]!='\0') {
            siglas[s++]=toupper(c[i]);
            siglas[s++]='.';
         }
      }

  siglas[s]='\0'; /* Cerramos la cadena. */
}
```

Lo primero que hace el algoritmo es saltar los espacios en blanco iniciales, si los hay. Luego, si no se ha llegado al final de la cadena, ya se puede almacenar la primera letra del acrónimo, y su punto correspondiente.

A continuación, se entra en el bucle que va mirando carácter a carácter hasta encontrar un espacio en blanco. Entonces, se salta todos los espacios en blanco que hubiera y el siguiente carácter (si no es el `'\0'`), será otra letra del acrónimo.

No debemos olvidar cerrar la cadena de salida, que, como todas las cadenas deben terminar en '\0'.

6.2.7 Cuenta caracteres.

Programar una función que cuente el número de veces que aparece un determinado carácter en una cadena. La función aceptará como argumentos una cadena y un carácter y devolverá el número de ocurrencias del carácter en la cadena.

Solución:

```
/***********************************************************/
/* Objetivo: Cuenta el número de ocurrencias de un carácter */
/*           en una cadena.                                 */
/* Entrada : Una cadena de caracteres, y un carácter.       */
/* Salida  : Devuelve el número de veces que aparece el     */
/*           carácter en la cadena.                         */
/* Requisitos: Cadena terminada en '\0'.                    */
/***********************************************************/
int cuenta_char (char cad[], char caracter)
{ int pos=0,   /* Posición por la que vamos buscando en cad. */
      total=0; /* Total de ocurrencias encontradas.          */

  for ( ; cad[pos] ; pos++)
      if (cad[pos]==caracter)
         total++;

  return total;
}
```

6.2.8 Carácter más repetido en una cadena.

Codificar una función que devuelva el carácter más repetido de una cadena de caracteres que se le pase como único argumento.

Solución:

```
/*************************************************************/
/* Objetivo: Averiguar el carácter más repetido en una cadena. */
/* Entrada : Una cadena de caracteres.                          */
/* Salida  : Devuelve el carácter más repetido de la cadena.   */
/* Requisitos: Cadena terminada en '\0'.                        */
/*************************************************************/
char charmasrepe ( char cad[] )
{
  int pos=0,     /* Posición del carácter que vamos buscando.  */
      total=0,   /* Total de ocurrencias encontradas.           */
      i,         /* Indice para recorrer la cadena.             */
```

```
      maximo=0; /* Número de ocurrencias del carácter más repetido
                   por el momento. */

  char char_maximo=0; /* Carácter más repetido por el momento.*/

  for ( ; cad[pos] ; pos++) {
      for ( i=0, total=0 ; cad[i] ; i++)
          if ( cad[pos] == cad[i] )
             total++;

      if ( total > maximo) {
           maximo = total;
           char_maximo = cad[pos];
      }
  }

  return char_maximo;
}
```

6.2.9 Manipulación de cadenas con menú de opciones.

Leer dos cadenas de caracteres y mostrar en pantalla un menu con las opciones:

1. comparar cadenas.
2. copia la primera sobre la segunda.
3. concatena las dos cadenas.
4. determina la longitud.

Solución:

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#define TAM 40

/*****************************************************/
/* Objetivo: mostrar un menu de opciones en pantalla.*/
/*****************************************************/
void menu(void)
{
  clrscr();
  printf("\nOPCIONES");
  printf("\n1. Compara cadenas.");
  printf("\n2. Copia la primera sobre la segunda.");
  printf("\n3. Concatena las dos cadenas ");
  printf("\n4. Determina la longitud.");
```

```
  printf("\n0. Salir");
  printf("\nOpción : "),
}

void main(void)
{ char hello[ ]="Hola";
  char bye[ ]={'A','d','i','o','s','\0'};
  char op;
  char s1[TAM];
  char s2[TAM];
  char s3[TAM*2];

  clrscr();
  printf("\n%s amigo",hello);
  printf("\nIntroduzca una cadena: ");
  gets(s1);
  printf("\nIntroduzca otra cadena: ");
  gets(s2);
  menu();

  while ((op = getche()) != '0') {
        switch (op) {
          case '1' : if (!strcmp (s1,s2))
                        printf("\nSon iguales");
                     else
                        printf("\nSon diferentes");
                     break;
          case '2' : strcpy (s2,s1);
                     printf("\nLas cadenas son:");
                     printf("\ns1: %s",s1),
                     printf("\ns2: %s",s2);
                     break;
          case '3' : strcpy(s3,s1);
                     strcat(s3,s2);
                     printf("\nLa cadena suma es:");
                     printf("\ns3: %s",s3);
                     break;
          case '4' : printf("\nLa longitud de s1 es: %d", strlen(s1));
                     printf("\nLa longitud de s2 es: %d", strlen(s2));
                     printf("\nLa longitud de s3 es: %d", strlen(s3));
                     break;
          default  : printf("\n\nLo siento, no hago nada");
        }

    printf("\nPulse una tecla para seguir ...");
    getch();
    menu();
```

```
  }

  printf("\n%s amigo",bye);
}
```

Observe el tratamiento tan diverso que se hace de las cadenas en este programa y como se asignan sus valores.

6.2.10 Un double a cadena.

En la mayoría de las versiones de C hay funciones para convertir un número de tipo `double` a una cadena, como las funciones `fcvt()`, `gcvt()`, `ecvt()` y `sprintf()` disponibles en sistemas UNIX y sistemas para PC (como Turbo C). Aún así, se trata de construir otra función que efectúe esta operación, cuyo prototipo podría ser:

```
void conversion(double, char []);
```

Podemos suponer que existe una macro `MAX` definida que indica el máximo número de caracteres que podemos almacenar en la cadena de salida:

```
#define MAX 20
```

Podemos usar la función `ltoa(long int, char [], 10)` que nos devuelve un entero `long`, en base 10, en la cadena del segundo argumento. Además, podemos usar la función `floor(x)` que nos da el número entero más grande que sea menor que `x` y la función `ceil(x)` que nos da el número entero más pequeño que sea mayor que `x`, con `x` de tipo `double`.

Solución:

```
/************************************************************/
/* Objetivo: Convertir un double a cadena. de caracteres.   */
/* Entrada : Un double.                                     */
/* Salida  : Cadena de caracteres.                          */
/*           Devuelve: 0 si todo ha ido bien.               */
/*                     1 si se han perdido dígitos          */
/*                       en la parte entera.                */
/*                     2 si se han perdido dígitos          */
/*                       en la parte decimal.               */
/* Requisitos: Tener definida una macro MAX, con el máximo  */
/*             de caracteres reservados en la cadena.       */
/************************************************************/
short conversion(double x,char cadena[])
{ char inter[MAX]; /* Cadena intermedia */
  int posicion=0,i;

  if (x<0) { /* Signo */
     cadena[0]='-';
     posicion=1;
```

```
      x=(-1)*x; /* Pasamos a positivo */
   }

   ltoa((long int) floor(x),inter,10); /* Parte entera */
   for(i=0;(inter[i]!='\0') && (posicion<MAX-1);i++,posicion++)
      cadena[posicion]=inter[i];
   if (posicion>=(MAX-1)) {
      cadena[MAX-1]='\0';
      return 1; /* Hay más dígitos en la parte entera que se han PERDIDO. */
   }

   x=x-floor(x);
   if (x==0) {  /* No hay decimales */
      inicializar(cadena,posicion,MAX);
      return 0; /* Sin errores. */
   }
   cadena[posicion]='.';
   posicion++;

   while ((x!=0) && (posicion<MAX-1)) {
     x=x*10;                      /* Pasamos un decimal a la parte entera    */
     ltoa((long int) floor(x),inter,10); /* ...lo pasamos a carácter, y  */
     cadena[posicion]=inter[0];          /* lo introducimos en la cadena */
     posicion++;
     x=x-floor(x);             /* Le quitamos la parte entera, ya convertida */
   }

   if (posicion<MAX-1)
      inicializar(cadena,posicion,MAX);
   else {
      cadena[MAX-1]='\0';
      return 2; /* Hay más dígitos en la parte DECIMAL que se han PERDIDO. */
   }

return 0;
}

/************************************************************/
/* Objetivo: Inicializar una porción de una cadena a ' '.   */
/* Entrada : Una cadena y dos enteros (inicio y fin).       */
/* Salida  : La cadena con espacios entre inicio y fin.     */
/*           La cadena se cerrará con '\0' al final.        */
/************************************************************/
void inicializar (char *string,int ini,int tama)
{ for (;ini<tama;ini++)
       string[ini]=' ';
  string[tama-1]='\0';
```

```
}
```

La función anterior, en algunos sistemas puede ocasionar resultados inesperados, debido, principalmente, a errores de redondeo, inherentes de las máquinas digitales.

6.3 Funciones para manejo de arrays y matrices.

Un array es simplemente una colección de elementos del mismo tipo en el que sabemos a priori de cuantos elementos consta. Si cada elemento del array es de un tipo simple (entero, caracter, estructura...) tendremos un array unidimensional, también llamado vector.

Puede darse el caso de que cada elemento de un array sea, a su vez, otro array, teniendo así arrays bidimensionales, también llamados matrices. Así podemos consequir arrays de cualquier dimensión. No obstante manejar arrays de más de 3 dimensiones suele ser demasiado complicado y casi seguro merecerá la pena simplificar el problema.

En este apartado nos centraremos en cómo manejar eficientemente los arrays uni y bidimensionales, que son los más habituales.

6.3.1 Máximo de un vector con funciones.

Leer desde teclado un conjunto de 10 números enteros utilizando una función cuyo prototipo sea:

```
void leer_vector (int *vector);
```

y que encuentre el máximo de los valores del vector, con otra función cuyo prototipo sea:

```
void MaximoDelVector (int *vector, int *maximo);
```

El valor máximo se deberá imprimir desde la función `main()` como resultado final del programa.

Solución:

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

#define NUMEROS 10

void leer_vector ( int *vector );
void MaximoDelVector ( int *vector , int *maximo );

void main ()
{ int i,
      vector[NUMEROS],
      max;

  clrscr();
  leer_vector(vector);
```

```
  MaximoDelVector(vector , &max);
  printf ( "El máximo de los valores del vector es = %d", max);
}

/***************************************************/
/* Objetivo: leer un vector de enteros desde teclado.*/
/* Salida: vector (con los numeros leídos).        */
/***************************************************/
void leer_vector ( int *vector )
{
  int i;

  for ( i = 0; i < NUMEROS; i++ ) {
      printf ( "Introduzca número %d:", i ),
      scanf ( "%d", &vector[i]);
  }
}

/******************************************************/
/* Objetivo: hallar el máximo de los valores de un vector.*/
/* Requisitos: vector debe estar lleno completamente.    */
/* Entrada: vector.                                     */
/* Salida: maximo.                                      */
/******************************************************/
void MaximoDelVector ( int *vector , int *maximo )
{
  int i;

  *maximo = vector[0];

  for ( i = 1; i < NUMEROS; i++ )
      if ( *maximo < vector[i] )
         *maximo= vector[i];
}
```

6.3.2 Posición del mayor valor.

Programe una función para que devuelva la posición del mayor valor de un array de enteros. Su prototipo podría ser algo así:

```
unsigned Pos_mayorvalor (int [], unsigned);
```

La función aceptará el array y el tamaño del mismo. Leerá todos los elementos del array buscando el mayor. No devolverá el mayor elemento, sino la posición del mayor elemento dentro del array.

Solución:

```
/*************************************************************/
/* Objetivo: Averiguar la posición del mayor valor de un array */
/* Entrada : Un array y su tamaño.                            */
/* Salida  : Devuelve la posición del mayor valor.            */
/*************************************************************/
unsigned Pos_mayorvalor (int array[], unsigned tama)
{ int i         = 1,
      mayor     = array[0],
      pos_mayor = 0;

  for ( ; i < tama; i++)
      if (mayor < array[i]) {
       mayor = array[i];
       pos_mayor = i;
      }

  return pos_mayor;
}
```

6.3.3 Ordenación de un array por Selección.

Codifique una función en C, cuyo prototipo sea:

```
void Ordena_Array (int [], unsigned);
```

y que ordene el array de enteros del primer argumento, con tantos elementos como se le indique en el segundo argumento. El método de ordenación a emplear deberá usar 2 funciones de ejercicios anteriores: `swap()` y `Pos_mayorvalor()`.

El método es simple: Deberá intercambiar, con `swap()`, el elemento de la última posición con el elemento en la posición del mayor, la que devuelve `Pos_mayorvalor()`. Después, deberá intercambiar, el elemento de la penúltima posición, con el elemento de la posición del mayor, pero excluyendo el último elemento, y así sucesivamente.

En cada llamada a `Pos_mayorvalor()` se irá disminuyendo el número de elementos, y en cada llamada a `swap()` se irá intercambiando el mayor valor de ese rango con el valor de la posición del último elemento, de ese rango. El rango se irá disminuyendo hasta que quede un único elemento.

Solución:

```
/*************************************************************/
/* Objetivo: Ordenar un array de enteros.                    */
/* Entrada : Un array y su tamaño.                           */
/*************************************************************/
void Ordena_Array (int num[], unsigned tama)
{ int i = tama;
```

```
  for ( ; i > 1 ; i-- )
      swap ( &(num [ Pos_mayorvalor(num, i) ]), &(num [i-1]) );
}
```

6.3.4 Búsqueda Lineal en un array.

En un programa es muy frecuente tener que buscar un elemento concreto dentro de un conjunto de elementos, como puede ser un array, una lista... El método de **búsqueda lineal** consiste en ir mirando desde el primero hasta el último hasta que encontremos el elemento que buscamos.

Supóngase que se tiene un array de enteros. Se desea hacer una función en C que, dado este array, su dimensión, y un entero particular a buscar, nos devuelva la posición del entero buscado dentro del array. Si no existe ese entero devolverá -1. Su prototipo puede ser:

```
int BusquedaLin (int [], int, int);
```

La función resultante podría ser algo así:

Solución:

```
/***********************************************/
/* Objetivo: Buscar un elemento en un array,     */
/*           por el método de Búsqueda LINEAL.  */
/* Entrada : Un array de enteros, su tamaño y   */
/*           un entero a buscar.                */
/* Salida  : Devuelve la posición de un         */
/*           elemento que sea igual al entero.  */
/*           Si no existe, devuelve -1          */
/***********************************************/
int BusquedaLin ( int A[], int tama, int N)
{ int i=0;

  for ( ; i < tama ; i++ )
      if (A[i] == N)
         return i;

  return -1;
}
```

6.3.5 Búsqueda Binaria.

El método de búsqueda lineal, visto anteriormente, es válido, pero si tenemos arrays o listas muy grandes, el tiempo de acceso es tremendamente elevado. Supóngase el caso extremo de una lista con miles de elementos y que el buscado sea, precisamente, el último, o esté por el final.

Una solución a este problema es emplear un tiempo en ordenar de alguna forma la lista, de forma que podamos desarrollar un algoritmo más eficiente llamado **búsqueda binaria**. La búsqueda binaria requiere que el array o lista esté ordenado, pero para grandes cantidades de elementos, consigue reducir drásticamente el tiempo de búsqueda. Para muy pocos elementos

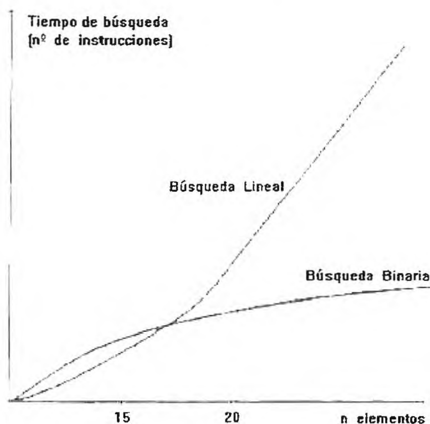


Figura 6.1: Tiempo de Acceso en la Búsqueda Lineal y Binaria.

(menos de 20) la búsqueda binaria es más lenta que la lineal, pero, como se trata de pocos elementos la diferencia no es significativa.

Codifique una función del mismo estilo que la anterior, pero usando el método de búsqueda binaria. La búsqueda Binaria consiste en localizar aproximadamente la posición media del array y comparar el elemento buscado con el elemento en dicha posición. Si el elemento buscado es menor, habrá que buscar en la primera mitad del array, y si es mayor buscaremos en la segunda mitad. El proceso se repite igualmente, mirando exclusivamente en la mitad seleccionada (mirando en la posición central de esa mitad...).

Tras las suficientes iteraciones, o encontramos el elemento buscado o encontramos el intervalo en el que debería estar y concluimos que no existe.

Una pega de este método con respecto a la búsqueda Lineal, se produce cuando existen elementos repetidos. En la búsqueda Lineal, estamos seguros de que encontramos el primer elemento de los que existan. En la binaria podemos encontrar cualquiera de ellos, aunque, como el array está ordenado, no será muy difícil encontrar el resto.

En la figura 6.1 puede observar la comparación de los tiempos de búsqueda de los métodos de búsqueda lineal y binaria.

Solución:

```
/************************************************/
/* Objetivo: Buscar un elemento en un array,    */
/*           por el método de Búsqueda BINARIA. */
/* Entrada : Un array de enteros, su tamaño y   */
/*           un entero a buscar.                */
/* Salida  : Devuelve la posición de un         */
/*           elemento que sea igual al entero.  */
/*           Si no existe, devuelve -1          */
/* Requisitos: El array debe estar ordenado de  */
/*           menor a mayor.                     */
```

```
/************************************************/
int BusquedaBin ( int A[], int tama, int N)
{ int inf = 0,      /* Posición inferior del rango en el que buscar */
      sup = tama-1, /* Posición superior */
      medio;        /* Posición intermedia */

  while (inf <= sup) {
    medio = (inf + sup ) / 2;
    if (N < A[medio])
      sup = medio - 1;
    else
      if (N > A[medio])
        inf = medio + 1;
      else
        return medio; /* Encontrado    */
  }
  return -1;            /* NO Encontrado */
}
```

6.3.6 Cambio en monedas.

Existen multitud de máquinas expendedoras de los más diversos productos, desde golosinas hasta billetes de metro o autobús. Estas máquinas, reciben un importe en metálico, en billetes o monedas, y deben cobrar el importe y efectuar la devolución del dinero restante.

Deseamos hacer una función en C, que calcule el tipo y cantidad de moneda fraccionaria a devolver, pero, entregando siempre el menor número de monedas posible. Además, debemos tener en cuenta que la cantidad de monedas/billetes puede variar, por lo que deberemos hacer una función que no haya que modificarla por cada cambio que realice la Fábrica Nacional de Moneda y Timbre. Otro punto importante es que, en determinados casos, puede no disponerse de un tipo de moneda en particular, por lo que la devolución no podrá contener ese tipo de moneda.

La función tendrá 4 argumentos:

1. Un array de tipo `unsigned` con tantos elementos como moneda fraccionaria exista (monedas o billetes). El array contendrá de mayor a menor el valor de cada moneda. Si una moneda se ha agotado, y no se puede usar para dar el cambio, se pueden hacer dos cosas:

   - Quitar dicha moneda, como si no existiera y correr las siguientes monedas una posición.
   - Poner, en vez de su valor real, el máximo número que admita el tipo de dato (en este caso `unsigned`).

   Un ejemplo de inicialización de este array sería el siguiente, suponiendo que tenemos 9 tipos de monedas/billetes distintos, y disponemos de todos ellos.

   ```
   unsigned monedas[] = { 10000, 5000, 2000, 1000, 500, 100, 25, 5, 1 };
   ```

La última moneda debe ser siempre 1, y no debe faltar para poder devolver cantidades impares. En realidad, se puede suprimir, si, por ejemplo, sólo cobramos y aceptamos de entrada cantidades múltiplos de 5.

2. Un **unsigned** que indique el número total de monedas de las que se dispone. Dicho de otro modo, el tamaño del array anterior.

3. Un **unsigned** que contenga la cantidad total a devolver. Dicha cantidad será la que tendremos que dividir en monedas/billetes, de forma que se entreguen el menor número de monedas/billetes posible.

4. Por último, un array de **unsigned** con tantos elementos como el array de monedas. En este array la función devolverá, la cantidad de monedas a devolver de cada tipo. La declaración, sería del tipo:

```
unsigned cambio [9];
```

Por tanto, la llamada a la función sería de la siguiente forma:

```
devolucion (monedas, 9, cantidad, cambio);
```

Tras la llamada, una forma de mostrar el resultado sería:

```
for ( i = 0 ; c < 9 ; c++ )
  printf ("--> De %u: %u\n", monedas[c], cambio[c] );
```

Observe que el array de salida `cambio` puede tener inicialmente cualesquiera valores en sus elementos.

Solución:

```
/**************************************************************/
/* Objetivo: Obtener el número y tipo de monedas que hay que */
/*           devolver, en la compra de un objeto.            */
/* Entrada : - Un array de enteros positivos, que nos indica */
/*           las distintas monedas/billetes de las que      */
/*           disponemos para dar el cambio.                 */
/*           - Número de monedas/billetes. Tamaño del array. */
/*           - Un entero positivo que indica la cantidad que */
/*           hay que devolver: Total a dividir en monedas.   */
/* Salida  : - Un array de enteros positivos que nos indica  */
/*           la cantidad de monedas que hay que devolver, en */
/*           el mismo orden que en el array de monedas.      */
/* Requisitos: - El array de monedas debe estar ordenado de  */
/*             mayor a menor.                                */
/*             - En el array de salida (cambio) debe tener   */
/*             tantas posiciones como monedas existan.       */
/**************************************************************/
void devolucion (unsigned monedas[], unsigned tama, unsigned cantidad,
```

```
                    unsigned cambio[] )
{ unsigned pos = 0;

  for ( ; pos < tama ; pos++)     /* Inicializamos cambio */
      cambio[pos] = 0;

  pos = 0;

  while (cantidad > 0)
    if (cantidad >= monedas[pos]) {
       cantidad -= monedas[pos];
       cambio[pos]++;
    }
    else
       pos++;
}
```

El algoritmo anterior, consta de un bucle que termina cuando la cantidad a devolver sea cero. Mientras sea mayor que cero, comprueba si la cantidad es mayor que la moneda más grande que tenemos (posición 0 del array `monedas`). Si es mayor, se le quita dicha cantidad, y se incrementa el array `cambio` en la moneda correspondiente en una unidad. En este momento, podríamos ya entregar al cliente una moneda de ese tipo. Si la cantidad a devolver es menor que la moneda en cuestión, pasamos a la siguiente moneda más pequeña (incrementando la posición `pos`).

6.3.7 Relleno especial de matriz.

Rellenar (asignar valores) una matriz bidimensional `A` de 3 filas y 4 columnas, de la forma siguiente:

- Se debe leer un valor por teclado (`semilla_relleno`) que puede ser real.
- Para cada elemento de cada fila (de la primera a la tercera) su contenido debe ser doble del inmediatamente anterior, comenzando con la asignación que hace corresponder al primer elemento de la matriz (fila primera, columna primera) el valor `semilla_relleno`.

Esto deberá hacerse con una función cuyo prototipo sea:

```
void relleno_especial (float a[][NUM_COLUMNAS], float semilla_relleno);
```

El resultado debe imprimirse con una función:

```
void imprime_matriz_especial(float a[][NUM_COLUMNAS]);
```

Se definirán sendas macros `NUM_FILAS` y `NUM_COLUMNAS` que indiquen el número de filas y columnas respectivamente de la matriz.

Solución:

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

#define NUM_FILAS 3
#define NUM_COLUMNAS 3

/**************************************************************************/
/* Objetivo: realiza en relleno de una matriz a partir de un valor semilla*/
/* Entrada: a (matriz), semilla_relleno.                                  */
/* Salida: a (matriz modificada).                                         */
/**************************************************************************/
void relleno_especial(float a[NUM_FILAS][NUM_COLUMNAS], float semilla_relleno)
{
  int i,j;
  float contador=1;

  for (i=0;i<NUM_FILAS;i++)
      for (j=0;j<NUM_COLUMNAS;j++) {
          a[i][j]=semilla_relleno*contador;
          contador*=2;
      }
}

/*******************************************************/
/* Objetivo: mostrar una matriz en pantalla.           */
/* Requisitos: a (matriz) debe estar llena completamente. */
/* Entrada: a (matriz)                                 */
/*******************************************************/
void imprime_matriz_especial(float a[NUM_FILAS][NUM_COLUMNAS])
{
  int i,j;

  clrscr();
  printf("\n\n\n\n\n\n\t\tLA MATRIZ VALE: \n\n\n\t\t");
  for (i=0;i<NUM_FILAS;i++) {
      for (j=0;j<NUM_COLUMNAS;j++)
          printf("\t%g",a[i][j]);
      printf("\n\t\t");
  }
}

void main(void)
{ float mat[NUM_FILAS][NUM_COLUMNAS], semilla_relleno;

  clrscr();
```

```
  printf("\n\n\n\tPor favor introduzca el valor de semilla_relleno: ");
  scanf("%f",&semilla_relleno);
  relleno_especial(mat, semilla_relleno);
  imprime_matriz_especial(mat);
}
```

6.3.8 Producto especial de vectores.

Introducir dos vectores reales de tres componentes desde teclado (utilizando arrays) y realizar el producto componente a componente (mostrando posteriormente el resultado), utilizando una función cuyo prototipo sea:

```
void prodvect (float *a, float *b, float *c);
```

Solución:

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

#define TAM 3

/****************************************************************/
/* Objetivo: producto vectorial de dos vectores.                */
/* Requisitos: a,b (vectores) deben estar llenos completamente */
/* Entrada: a,b (vectores).                                     */
/* Salida: c (vector resultado).                                */
/****************************************************************/
void prodvect ( float *a , float *b , float *c )
{ int i;

  for ( i = 0; i < TAM; i++ ) {
      printf( "Introduzca componente %d del vector x: ", i + 1 );
      scanf("%f",&a[i]);
  }

  for ( i = 0; i < TAM; i++ ) {
      printf("Introduzca componente %d del vector y: ", i + 1 );
      scanf("%f",&b[i]);
  }

  for ( i = 0; i < TAM; i++ ) {
      c[i] = a[i] * b[i];
      printf ( "\nProducto de componentes %d: %g", i+1 , c[i]);
  }
}

void main (void)
{ float x[TAM];
```

```
  float y[TAM];
  float z[TAM];
  float solucion = 0.0;
  int i;

  clrscr();
  prodvect( x , y , z );
  printf( "\n\n( %g , %g , %g )", x[0] , x[1] , x[2] );
  printf( " x ( %g , %g , %g )", y[0] , y[1] , y[2] );

  for ( i = 0; i < TAM; i++ )
      solucion = z [i] + solucion;

  printf( " = %g", solucion );
}
```

6.3.9 Pares en un intervalo.

Leer desde teclado un número entero (`num`) y almacene en un vector los números pares entre 1 y ese número mediante una función cuyo prototipo sea:

```
void calcular_pares (int num, int pares[]);
```

El resultado se deberá imprimir mediante una función cuyo prototipo sea:

```
void imprimir_pares (int pares[]);
```

Solución:

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

#define  TAM MAXINT/2
/*MAXINT está definido en <values.h> con el valor del mayor numero entero posible*/

void calcular_pares ( int num , int *pares );
void imprimir_pares ( int *pares );

void main ()
{ int pares [TAM];
  int i,num;

  for (i=0;i<TAM;i++) pares[i]=-1;

  clrscr ();
  printf( "Introduzca número entero: " );
  scanf( "%d", &num );
```

```
  calcular_pares( num , pares );
  imprimir_pares( pares );
}

/*************************************************************************/
/* Objetivo: calcular los números pares entre desde 1 a un número determinado*/
/* Requisitos: num >=2                                                   */
/* Entrada: num (el número)                                              */
/* Salida: pares (vector que contiene los pares)                         */
/*************************************************************************/
void calcular_pares ( int num , int *pares )
{ int i;

  for ( i = 1; 2 * i <= num; i++ )
      pares[i] = 2 * i;
}

/**************************************************/
/* Objetivo: mostrar un vector de enteros en pantalla*/
/* Requisitos: pares debe estar lleno completamente */
/* Entrada: pares (vector de números).              */
/**************************************************/
void imprimir_pares ( int *pares )
{
  int i ;

  printf ( "Los pares son: \n ");
  for ( i = 1; ((i <= TAM) && (pares[i]!=-1)); i++ )
      printf ( "%d\t", pares [ i ] ),
}
```

6.3.10 Media y varianza con funciones.

Leer 10 números reales desde el teclado introduciéndolos en un array, y calcular la media y la varianza de estos, definiendo y utilizando las funciones cuyos prototipos son los siguientes:

```
float varianza (int v[]);
float media (int *v);
```

Recuerde que es lo mismo declarar un argumento como `int v[]` que como `int *v` En ambos casos lo que se declara es un puntero a entero.

Solución:

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

```
#include <conio.h>

#define DIMENSION 10

float media(int v[]);
float varianza(int v[]);

void main(void)
{ int i,
      vector[DIMENSION];

  clrscr();
  for (i=0;i<DIMENSION;i++){
      printf("Introduzca valor [%d]: ", i+1);
      scanf("%d", &vector[i]);
  }

  clrscr();
  printf("La media de estos %d números es: %5.1f\n",
         DIMENSION, media(vector));
  printf("La varianza de estos %d números es: %5.1f\n",
         DIMENSION, varianza(vector));
}

/**********************************************************************/
/* Objetivo: hallar la media de los valores de un vector de enteros.*/
/* Requisitos: v debe estar lleno completamente.                    */
/* Entrada: v (vector de enteros).                                  */
/* Salida: con return (media aritmética).                           */
/**********************************************************************/
float media(int v[])
{
  int y, acum = 0;

  for (i=0;i<DIMENSION;i++)
      acum += *v++; /*equivale a  acum=acum+*v;  v++;*/

  return (float)acum/DIMENSION;
}

/***********************************************************************/
/* Objetivo: hallar la varianza de los valores de un vector de enteros*/
/* Requisitos: v debe estar lleno completamente                       */
/* Entrada: v (vector de enteros)                                     */
/* Salida: con return (varianza)                                      */
/***********************************************************************/
float varianza(int v[])
```

```
{
  int i;
  float acum = 0.0, nmedia;

  nmedia = media(v);
  for (i=0;i<DIMENSION;i++){
      acum += ((nmedia - (float) *v) * (nmedia - (float) *v)) *
                  (float)1/DIMENSION;
      ++v;
  }

  return acum;
}
```

En las funciones se accede al vector v a partir del puntero al comienzo (que es el nombre del vector), incrementandose para apuntar a cada una de las casillas siguientes. El contenido de una casilla es *v, ya que v apunta a ella.

6.3.11 Media y desviaciones de un vector con funciones.

Leer desde teclado un conjunto de 10 números enteros y almacenarlos en un vector. Calcular la media con una función cuyo prototipo sea:

```
float media(void);
```

y que almacene en otro vector las diferencias entre cada uno de los valores originales y la media (desviaciones), por medio de una función cuyo prototipo sea:

```
void desviaciones(void);
```

El vector de diferencias se deberá imprimir al final del programa utilizando una función cuyo prototipo sea:

```
void imprimir_desviaciones(void);
```

Solución:

```
/* Como las funciones no reciben explicitamente ningun valor es necesario
definir los vectores como globales */

/* Como la función que calcula las desviaciones necesita conocer el valor
de la media y tampoco la recibe como par metro, es tambi,n necesario
definir la variable como global */

#include <stdio.h>
#define TAM 10
```

```
/* definiciones de variables globales */
int vector[TAM];
float desviacion[TAM];
float media_valores;

/************************************************************************/
/* Objetivo: Hallar la media.                                           */
/* Requisitos: El vector de entrada a la función es una variable global */
/*             vector debe estar lleno completamente.                   */
/* Entrada: Vector (global).                                            */
/* Salida: Devuelve el valor de la media.                               */
/************************************************************************/
float media (void)
{
  float med=0.0;
  int i,sum=0;

  for (i=0;i<TAM;i++)
      sum = sum + vector[i];

  med = (float)sum / (float)TAM;
  return med;
}

/***********************************************************************/
/* Objetivo: hallar las diferencias entre los elementos de un vector   */
/*           y su media.                                               */
/* Requisitos: vector ,media_valores y desviacion globales             */
/*             vector debe estar lleno completamente.                  */
/* Entrada: vector y media_valores son globales.                       */
/* Salida: desviacion (vector con las diferencias).                    */
/***********************************************************************/
void desviaciones(void)
{
  int i;
  for (i=0;i<TAM;i++)
      desviacion[i]=vector[i]-media_valores;
}

/******************************************************/
/* Objetivo: mostrar un vector de enteros en pantalla */
/* Requisitos: vector debe estar lleno completamente  */
/* Entrada: desviacion (global).                      */
/******************************************************/
void imprimir_desviaciones (void)
{
  int i;
```

```
  printf("\nEl vector de desviaciones es:\n\n");
  for (i=0;i<TAM;i++)
      printf("%f \t",desviacion[i]);
}

void main (void)
{ int i;

  printf("\nIntroduce un vector de enteros");
  for (i=0;i<TAM;i++) {
      printf("\nElemento %d: ",i);
      scanf("%d",&vector[i]);
  }

  media_valores=media();
  desviaciones();
  imprimir_desviaciones();
}
```

Observe que en este ejercicio hemos utilizado variables globales. Esta no es una técnica de programación buena, pero se ha utilizado aquí para mostrar otra forma de hacer las cosas. En general se deben usar parámetros para pasar valores a las funciones, de forma que se puedan ver las funciones como cajas negras que reciben valores y devuelven valores y en las que todos estos valores están declarados en sus argumentos.

6.3.12 Lista de direcciones.

Supongamos la siguiente estructura de datos (array):

```
struct dir
{
  char nombre[30];
  char calle[40];
  char ciudad[20];
  char provincia[3];
  unsigned long int codigo;
} info_dir[MAX];
```

Realizar un sencillo programa que sea capaz de manejar una lista de estas estructuras, es decir, que sea capaz de tener las siguientes funciones:

- Añadir una dirección a la lista.
- Borrar una dirección de la lista.
- Listar las direcciones que tenemos.

Para que el programa no sea demasiado complejo suponer que la lista es un array de estructuras. La dimensión del array es lo que en un principio debemos de fijar.

Solución:

```
/*****************************************************************/
/* Objetivo: Un sencillo programa de lista de correo que ilustra */
/*           el uso de arrays de estructuras.                    */
/* Entrada:  Un registro de dirección.                           */
/* Salida:   Registros de dirección.                             */
/*****************************************************************/
#include "stdio.h"
#include "stdlib.h"

#define MAX 100

struct dir
{
  char nombre[30];
  char calle[40];
  char ciudad[20];
  char provincia[3];
  unsigned long int codigo;
} info_dir[MAX];

void inic_lista(void);
void intro(void);
void borrar(void);
void listar(void);
int menu(void);
int busca_libre(void);

main(void)
{ char opcion;

  inic_lista(); /* inicializa el array de estructuras */
  for(;;) {
    opcion = menu();

    switch(opcion) {
      case 1: intro();
              break;
      case 2: borrar();
              break;
      case 3: listar();
              break;
      case 4: exit(0);
    }
  }
}
```

```
/* Inicializa la lista */
void inic_lista(void)
{ register int t; /* Variable almacenada en un registro del procesador */

  for (t=0; t<MAX; t++)
       info_dir[t].nombre[0] = '\0';
  /* Suponemos que si no hay nombre es que la ficha esta vacía */
}

/* Seleccionar una operación mediante menu */
menu(void)
{ char s[80];
  int c;

  printf("1. Introducir una ficha de dirección\n");
  printf("2. Borrar una ficha de dirección\n");
  printf("3. Listar las direcciones\n");
  printf("4. Salir\n");

  do {
    printf("\nIntroduzca su opción: ");
    gets(s);
    c = atoi(s);
  } while(c<0 || c>4);
  return c;
}

/* Introducir direcciones en el array */
void intro(void)
{ int sitio;
  char s[80];

  sitio = busca_libre(); /* Localizar una ficha disponible */
  if(sitio==-1) {
    printf("\nLista llena");
    exit(1);
  }

  /* rellenar la ficha con los valores apropiados */
  printf("Introduzca nombre: ");
  gets(info_dir[sitio].nombre);
  printf("Introduzca calle: ");
  gets(info_dir[sitio].calle);
  printf("Introduzca ciudad: ");
  gets(info_dir[sitio].ciudad);
  printf("Introduzca provincia: ");
```

```
  gets(info_dir[sitio].provincia);
  printf("Introduzca código: ");
  gets(s);
  info_dir[sitio].codigo = strtoul(s, '\0', 10);
}

/* Busca un sitio no usado en el array de direcciones */
busca_libre(void)
{ register int t;

  for(t=0; info_dir[t].nombre[0] && t<MAX; t++);
  if(t==MAX)
    return -1; /* no hay sitio libre */
  return t;
}

/* Eliminar una dirección del array */
void borrar(void)
{ register int sitio;
  char s[80];

  printf("Introduzca número de registro: ");
  gets(s);
  sitio = atoi(s);
  if(sitio>=0 && sitio < MAX)
  /* Si la ficha existe se elimina el nombre para hacerla disponible */
    info_dir[sitio].nombre[0] = '\0';
}

/* Mostrar la lista de direcciones en pantalla */
void listar(void)
{
  register int t;

  for(t=0; t<MAX; t++)
    if(info_dir[t].nombre[0]) {
      printf("%s\n", info_dir[t].nombre),
      printf("%s\n", info_dir[t].calle);
      printf("%s\n", info_dir[t].ciudad);
      printf("%s\n", info_dir[t].provincia);
      printf("%lu\n", info_dir[t].codigo);
    }

  printf("\n\n");
}
```

En el programa se han utilizado variables registro, cuando nosotros ponemos la siguiente sentencia `register int` estamos intentando que el programa reserve un registro del procesador para poder usarlo como una variable. La razón de utilizar registro en vez de zonas de memoria es para economizar tiempo. Si no hay registros disponible se reserva la zona de memoria adecuada como normalmente. Esta practica puede ser recomendable cuando estamos tratando con varibles temporales de tipo entero y que vamos a utilizar un número de veces muy elevado. En los demás casos es suficiente con reservarla de la forma abitual.

Como en casos anteriores otros posibles programas sería realizar el mismo problema pero utilizando una estructura dinámica en vez de un array para almacenar las direcciones.

6.3.13 Búsqueda de personas en un vector de estructuras.

Almacenar en un vector, leyendo desde teclado, una lista de registros de tipo **persona**, con dos campos:

1. nombre: alfanumérico de longitud máxima 30
2. edad: entero

Una vez hecho ésto, el programa deberá pedir un nombre y mostrar la edad correspondiente a esa persona, utilizando una función que devuelva la edad, si coincide el nombre solicitado con el nombre del registro, cuyo prototipo sea:

```
int busca_nombre (persona x[], char *nombre);
```

La función anterior devolverá –1 si no encuentra a la persona en el array.

Solución:

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

#define MAXLIST 3
#define MAXNOM (30+1)

typedef struct {
                  char nombre[MAXNOM];
                  int edad;
                } persona;

/*********************************************************************/
/* Objetivo: leer de teclado los valores de nombre y edad           */
/*           para meterlos en un vector de elementos de tipo persona.*/
/* Salida: p (vector de elemento de tipo persona).                   */
/*********************************************************************/
void leer_nombres(persona *p)
```

```
{
  int i;
  char edad[50];

  for (i=0;i<MAXLIST;i++) {
      printf("\n\nIntroduzca el nombre [%d]: ",i);
      gets(p[i].nombre);
      printf("\nIntroduzca la edad: ");
      gets(edad);
      p[i].edad=atoi(edad);
  }
}

/*****************************************************************/
/* Objetivo: buscar en un vector de elementos persona la edad   */
/*           de alguna a partir de su nombre.                    */
/* Requisitos: x (debe estar lleno completamente).               */
/* Entrada: x (vector de persona), nombre (a buscar).            */
/* Salida: con return la edad de esa persona (-1 si no existe). */
/*****************************************************************/
int busca_nombre(persona *x, char *nombre)
{
  int i, res=-1;

  for (i=0;i<MAXLIST;i++)
      if (strcmp(x[i].nombre,nombre) == 0) {
         res=x[i].edad;
         break;
      }

  return res;
}

void main()
{ int cont=0, edad;
  persona vec_per[MAXLIST];
  char nombre[MAXNOM];

  clrscr();
  leer_nombres(vec_per);
  printf("\nEntrada de datos terminada.");
  printf("\n\nIntroduzca un nombre para saber su edad: ");
  gets(nombre);
  edad=busca_nombre(vec_per,nombre);

  if (edad != -1)
     printf("\nLa edad de %s es %d",nombre,edad);
```

```
  else
     printf("\nLo siento ese nombre no esta en la lista.");

  printf("\n\n\nPulse una tecla para salir...");
  getch();
}
```

6.3.14 Copia de Matrices.

Implementar una función que nos permita copiar todos los elementos de una matriz de enteros en otra matriz. Para simplificar supondremos que las matrices son cuadradas de dimensión `DIM`, siendo `DIM` una macro definida con `#define`, por ejemplo de la siguiente forma:

```
#define DIM 5       /* Dimensión de la matriz cuadrada */
```

Una matriz es cuadrada cuando el número de filas coincide con el número de columnas.

El prototipo de la función tendrá dos argumentos de tipo array bidimensional de enteros:

```
void MatrizCopia (int origen[][], int destino[][]);
```

donde `origen` es la matriz original y `destino` es la matriz donde se copiarán los elementos de la matriz `origen`.

En la función supondremos que ambas matrices están bien declaradas y que está definida la contaste simbólica `DIM`.

Solución:

```
/*****************************************************/
/* Objetivo: Copiar una matriz cuadrada de enteros   */
/*           en otra, de dimensión DIMxDIM.          */
/* Entrada : Matriz origen de enteros: DIMxDIM       */
/* Salida  : Matriz destino de enteros: DIMxDIM      */
/* Requisitos: Que exista definida una macro DIM con */
/*             la dimensión de la matriz.            */
/*             Que las matrices origen y destino     */
/*             sean de enteros y de dimensión DIMxDIM */
/*****************************************************/
void MatrizCopia (int origen[][DIM], int destino[][DIM])
{ int i, j;              /* Variables índice para filas y columnas    */

  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
        destino[i][j] = origen[i][j];
}
```

Para copiar todos los elementos de una matriz en otra, tenemos que recorrer todos los elementos de la matriz. Como la matriz es bidimensional, necesitamos dos bucles anidados, uno para moverse por las filas (`i`) y otro para las columnas (`j`). Lo único que se hace es recorrer la matriz elemento a elemento y copiándolo en la matriz `destino` en la misma posición.

Es obvio, que para copiar una matriz en otra, igual que para copiar arrays de cualquier dimensión, no basta con asignar los identificadores (`destino = origen`), ya que de esta forma estamos copiando los punteros y no los elementos. De esta forma no tendremos 2 matrices iguales sino que tendremos la misma matriz. Tendremos los mismos elementos en memoria referenciados con 2 nombres distintos.

6.3.15 Matriz Traspuesta.

Implementar una función que calcule la traspuesta de una matriz. Supondremos que la matriz es una matriz de enteros cuadrada de dimensión `DIM`, siendo esta una constante simbólica que supondremos ya definida.

El prototipo de la función será:

```
void MatrizTraspuesta (int M[][]);
```

donde `M` es la matriz de entrada y la matriz donde se devolverá su traspuesta.

Solución:

```
/*****************************************************/
/* Objetivo: Hallar la traspuesta de una matriz de    */
/*           de enteros de dimensión DIMxDIM.         */
/* Entrada : Matriz cuadrada de enteros: DIMxDIM      */
/* Salida  : Matriz traspuesta a la de entrada en el  */
/*           mismo array de entrada.                  */
/* Requisitos: Que exista definida una macro DIM con  */
/*             la dimensión de la matriz.             */
/*             Que la matriz de entrada sea cuadrada  */
/*             de dimensión DIMxDIM.                  */
/*****************************************************/
void MatrizTraspuesta (int M[][DIM])
{ int i, j,            /* Variables índice para filas y columnas     */
      filas = DIM-1,   /* Filas a comparar (la última no hace falta) */
      cambio;          /* Para hacer el intercambio                  */

  for (i=0; i<filas; i++)
    for (j=i+1; j<DIM; j++)
      if (M[i][j] != M[j][i]) {
         cambio  = M[i][j];
         M[i][j] = M[j][i];
         M[j][i] = cambio;
      }
}
```

Lo que hacemos es intercambiar los elementos: El elemento de la fila `i` y columna `j` por el elemento de la fila `j` y columna `i`. Esto lo haremos recorriendo la matriz por filas. Por eso, cuando lleguemos a la última fila no habrá que hacer ningún cambio, ya que todos los elementos de esa fila ya se habrán intercambiado previamente. De ahí que en el primer `for`

tenga por condición `i<filas`, siendo `filas` el número de filas a comparar, que será el número de filas total, menos 1 (`DIM-1`).

Además, para cada fila no es necesario mirar todas las columnas. Ejemplo: Si estamos operando en la segunda fila, fila número 1, el elemento (1,0) no hay que tocarlo ya, porque ya se intercambió con el elemento (0,1) cuando operamos en la primera fila (fila 0). El siguiente elemento de esa fila, el (1,1), tampoco hay que intercambiarlo consigo mismo (sería absurdo). Por tanto, tendremos que empezar a intercambiar el elemento (1,2) con el (2,1), el (1,3) con el (3,1) y así sucesivamente.

Generalizando, si estamos en la fila `i`, tendremos que empezar a intercambiar elementos a partir de la columna `i+1`. Este requisito está expresado en el segundo `for`.

6.3.16 Comparar dos matrices.

Programar una función que nos permita averiguar si dos matrices cuadradas de enteros son iguales. El prototipo de la función será:

```
int MatrizIguales (int A[][], int B[][]);
```

donde `A` y `B` serán las dos matrices a comparar. Supondremos que ambas matrices son de dimensión `DIMxDIM`, donde `DIM` será una constante simbólica que supondremos que ya está definida previamente.

La función devolverá un valor entero:

`MatrizIguales (A,B)`	Relación entre `A` y `B`
0	Ambas matrices son distintas.
1	Ambas matrices son iguales.

Solución:

```
/*****************************************************/
/* Objetivo: Comparar si son iguales dos matrices     */
/*           cuadradas de enteros, de tamaño DIMxDIM. */
/* Entrada : Dos matrices de enteros: DIMxDIM         */
/* Salida  : Devuelve: 1 Si son iguales.              */
/*                     0 Si son distintas.            */
/* Requisitos: Que exista definida una macro DIM con  */
/*             la dimensión de la matriz.             */
/*             Que las matrices a comparar sean       */
/*             de enteros y de dimensión DIMxDIM      */
/*****************************************************/
int MatrizIguales (int A[][DIM], int B[][DIM])
{ int i, j;            /* Variables índice para filas y columnas     */

  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
        if (A[i][j] != B[i][j])
           return 0;   /* Son distintas */
```

```
  return 1; /* Son iguales */
}
```

6.3.17 Comprobar si una matriz es simétrica.

Hacer una función que compruebe si una matriz es simétrica. Una matriz es simétrica si coincide con su traspuesta. El prototipo de la función será:

```
int MatrizSimetrica (int [][]);
```

La función devuelve un valor entero:

0 La matriz NO es simétrica.
1 La matriz SI es simétrica.

Obviamente, para que una matriz sea simétrica debe ser cuadrada. Supondremos que es una matriz cuadrada de enteros de dimensión `DIM`, siendo `DIM` una constante simbólica que supondremos que ya está definida previamente.

Para este ejercicio proponemos 3 soluciones, siendo al última la más eficiente.

Solución 1:

Para esta primera solución vamos a usar las funciones creadas anteriormente.

```
/*****************************************************/
/* Objetivo: Averiguar si una matriz cuadrada DIMxDIM */
/*           de enteros es Simétrica.                */
/* Entrada : Matriz cuadrada de enteros: DIMxDIM     */
/* Salida  : Devuelve: 1 si la matriz es Simétrica.  */
/*                     0 si no lo es.                */
/* Requisitos: Que exista definida una macro DIM con */
/*             la dimensión de la matriz.            */
/*             Que la matriz de entrada sea cuadrada */
/*             de tamaño DIMxDIM.                    */
/*****************************************************/
int MatrizSimetrica (int M[][DIM])
{ int T[DIM][DIM];

  MatrizCopia (M,T);
  MatrizTraspuesta (T);
  return MatrizIguales (M,T);
}
```

La explicación es simple: Primero saco una copia de la matriz `M` en la matriz `T` (que es una matriz local a la función), luego se halla su traspuesta en la matriz `T` y por último se compara si `M` es igual a su traspuesta `T`. Si son iguales es que la matriz es simétrica y si son distintas es que no lo es.

Solución 2:

La solución anterior es válida, pero requiere hacer 3 llamadas a función y cada una de esas funciones tiene a su vez dos bucles anidados. Se puede resolver el mismo problema con sólo dos bucles anidados:

```
/*****************************************************/
/* Objetivo: Averiguar si una matriz cuadrada DIMxDIM */
/*           de enteros es Simétrica.                 */
/* Entrada : Matriz cuadrada de enteros: DIMxDIM      */
/* Salida  : Devuelve: 1 si la matriz es Simétrica.   */
/*                     0 si no lo es.                 */
/* Requisitos: Que exista definida una macro DIM con  */
/*             la dimensión de la matriz.             */
/*             Que la matriz de entrada sea cuadrada  */
/*             de tamaño DIMxDIM.                     */
/*****************************************************/
int MatrizSimetrica (int M[][DIM])
{ int i, j;

  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
      if (M[i][j] != M[j][i])
         return 0;  /* No Simétrica */

  return 1; /* Matriz Simétrica */
}
```

La función recorre todos los elementos de la matriz y va comparando cada elemento con su traspuesto, cambiando filas por columnas. Esto es, compara el elemento (i,j) con el (j,i). En el momento que encuentra dos elementos que son distintos, la matriz ya no es simétrica y termina devolviendo cero. Si finaliza los dos bucles sin devolver cero es que todos los elementos son iguales, por lo que la matriz es simétrica y devuelve uno.

Solución 3:

La función anterior es perfectamente válida pero efectúa comparaciones inútiles. Por ejemplo, es absurdo comprobar si el elemento (0,0) coincide con su traspuesto, ya que este pertenece a la diagonal principal de la matriz. Además, si comparamos por filas, cuando lleguemos a la fila `i`, no es necesario que comparemos los elementos situados en las columnas anteriores a la `i`, ya que dichos elementos ya han sido comparados previamente. Esto también ocurría en el ejercicio **Matriz Traspuesta** anterior.

Por ejemplo, al llegar a la fila 3, ya hemos comparado los elementos de las filas 0, 1 y 2. Por tanto, no tenemos que comparar los elementos (3,0), (3,1) y (3,2), que se compararon cuando se procesaron los elementos (0,3), (1,3) y (2,3) respectivamente.

Por tanto, en la fila `i` empezaremos a comparar a partir de los elementos situados en la columna `i+1`.

Además, la última fila no es necesario procesarla, ya que al llegar a esta ya se habrán comparado todos sus elementos con otros de otras filas previas.

Teniendo en cuenta todo esto, la función, de forma óptima, quedaría de la siguiente forma:

```
/*****************************************************/
/* Objetivo: Averiguar si una matriz cuadrada DIMxDIM */
/*           de enteros es Simétrica.                 */
/* Entrada : Matriz cuadrada de enteros: DIMxDIM      */
/* Salida  : Devuelve: 1 si la matriz es Simétrica.   */
/*                     0 si no lo es.                 */
/* Requisitos: Que exista definida una macro DIM con  */
/*             la dimensión de la matriz.             */
/*             Que la matriz de entrada sea cuadrada  */
/*             de tamaño DIMxDIM.                     */
/*****************************************************/
int MatrizSimetrica (int M[][DIM])
{ int i, j,            /* Variables índice para filas y columnas     */
      filas = DIM-1;   /* Filas a comparar (la última no hace falta) */

  for (i=0; i<filas; i++)
    for (j=i+1; j<DIM; j++)    /* Empezamos por la columna siguiente a la fila */
      if (M[i][j] != M[j][i]) /* Cambiamos filas por columnas y comparamos */
         return 0;  /* No Simétrica */

  return 1; /* Matriz Simétrica */
}
```

6.3.18 Multiplicación de matrices NxK por KxM.

Realizar una función que multiplique dos matrices de enteros de dimensión NxK y KxM respectivamente. La función debe permitir que se puedan multiplicar dos matrices **para cualquier valor de N, K y M**. Se puede establecer un límite máximo en K y M.

Por la forma de almacenamiento de las matrices en memoria del C, en una función tenemos que declarar forzosamente el tamaño de la segunda dimensión (columnas) de una matriz bidimensional. Por tanto, podemos establecer que nuestra función multiplique matrices NxK por KxM, con K y M menores que 20 (por ejemplo). No es necesario limitar la N.

El C, almacena los elementos de un array en posiciones contiguas en memoria. Así, un array bidimensional (o matriz), es como un array de arrays, esto es, un array en el que cada elemento es de tipo array. Por tanto, se almacenarán en posiciones contiguas en memoria todos los elementos, de tipo array.

Así, si tenemos un array de dimensión 4x3:

```
int A[4][3];
```

Sus elementos, son nombrados como se ve en la siguiente tabla:

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

Como todos los elementos se almacenan de forma contigua en memoria, también podemos acceder a todos los elementos, como si fueran de la fila 0:

A[0][0]	A[0][1]	A[0][2]
A[0][3]	A[4][1]	A[0][5]
A[0][6]	A[0][7]	A[0][8]
A[0][9]	A[0][10]	A[0][11]

En general, podemos asegurar que el elemento `A[x][y]` es el mismo elemento que el denotado por `A[0][y+filasA*x]`.

Esa forma de acceder a los elementos de una matriz es importante, ya que en nuestra función tendremos que declarar una matriz de más columnas (de 20), y si accedemos normalmente, estaremos accediendo a elementos que no existen en nuestra matriz original.

Codificar también un pequeño programa principal que pruebe la función anterior

Solución:

```
#include <stdio.h>

void multimatriz ( int A[][], int B[][], int C[][],
                   int N, int K, int M );

void main()
{ int A[4][3] = { 10, 3, 5,
                  1, 0, 2,
                  7, 2, 3,
                  5, 6, 7 },
```

```
        B[3][5] = { 0, 1, 2, 3, 4,
                    1, 2, 3, 4, 5,
                    1, 1, 1, 1, 1 },
        C[4][5],
        i;

    multimatriz (A,B,C,4,3,5);

    for (i=0; i<4; i++)
        printf ("\n    %3i %3i %3i %3i %3i  ",
                C[i][0], C[i][1], C[i][2], C[i][3], C[i][4] );
    puts("\n\n*** FIN ***");
}

/************************************************/
/* Objetivo: Multiplicar 2 matrices de enteros. */
/* Entrada : Matriz A, de dimensión NxK y       */
/*           matriz B, de dimensión KxM.        */
/* Salida  : Matriz C, de dimensión NxM.        */
/* Requisitos: K y M menores de 20.             */
/************************************************/
void multimatriz ( int A[][20], int B[][20], int C[][20],
                   int N, int K, int M )
{
    int i=0,j,  /* Indices para seguir todos los elementos de C    */
        z,      /* Para movernos por la fila i, columna j de A y B */
        despA, despC;


    for ( ; i<N ; i++ ) {                /* Filas de C   */
        despA = K * i;
        despC = M * i;

        for ( j=0 ; j<M ; j++ ) {     /* Columnas de C */
            C[0][j+despC]=0;
            for ( z=0 ; z<K ; z++ )
                C[0][j+despC] += A[0][z+despA] * B[0][j+M*z],
            /* En general: A[x][y] == A[0][y+filasA*x] */
        }
    }
}
```

6.3.19 Juego de las 3 en Raya (tic-tac-toe).

Como ejercicio podemos realizar un programa que simule el juedo de las tres en raya. El programa debe de hacer que un jugador humano juegue contra el ordenador.

La versión de las 3 en raya que vamos a utilizar es la más simple, se van colocando fichas

de forma alternativa hasta que un jugador gane o no se puedan poner más fichas en el tablero.

El alumno tiene que hacer una función para simular el movimiento de las fichas del ordenador. De esta función depende que el ordenador juegue bien o mal a este juego.

Solución:

```
/*************************************************************/
/*                                                           */
/* Objetivo: Realizar el juego de las 3 en raya siendo uno de */
/*           los jugadores el ordenador                       */
/*                                                           */
/*************************************************************/

#include <stdio.h>
#include <stdlib.h>

char tablero[3][3]; /* Tablero que vamos ha utilizar para el juego */

char comprobar(void);
void inic_tablero(void);
void obt_mov_jugador(void);
void obt_mov_ordenador(void);
void mostrar_tablero(void);

void main(void)
{ char hecho; /* Variable que controla cuando el juego a terminado */

  printf("Este es el juego de las tres en raya.\n");
  printf("Se trata de una partida contra el ordenador.\n");

  hecho = ' ';

  inic_tablero(); /* Situamos el tablero para empezar a jugar */

  do { /* Bucle que simula el juego de las 3 en raya */
    mostrar_tablero();
    obt_mov_jugador();
    hecho = comprobar(); /* ver si gana */
    if ( hecho != ' ') break; /* ganador */
    obt_mov_ordenador();
    hecho = comprobar(); /* ver si gana */
  } while (hecho == ' ');

  if (hecho == 'X')
    printf("Ha ganado!\n"); /* Gana el jugador */
  else
    printf("Yo gano!!!!\n"); /* Gana el ordenador */
```

```
  mostrar_tablero(); /* mostrar las posiciones finales */
}

/* Inicializar el tablero de juego */
void inic_tablero(void)
{ int i,j;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      tablero[i][j] = ' ';
}

/* Obtener un movimiento del juegador */
void obt_mov_jugador(void)
{ int x,y;

  printf("Introduzca las coordenadas de su ficha X: ");
  scanf("%d%d",&x,&y);

  x--;y--; /* coordenadas en el tablero */

  if(tablero[x][y] != ' ') {
    printf("Movimiento ilegal, prueba de nuevo.\n");
    obt_mov_jugador();
  }
  else
    tablero[x][y] = 'X';
}

/* Obtener un movimiento del ordenador */
/* Cambiando esta funcion hacemos que el juego del */
/* ordenador sea más agrasivo o menos agresivo */
void obt_mov_ordenador(void)
{ int i,j;

  for(i=0; i<3; i++) {
    for(j=0;j<3;j++)
      if(tablero[i][j]== ' ') break;
        if(tablero[i][j]== ' ') break;
  }
  /* El ordenador busca una posición libre */

  if(i*j==9) {
    printf("tablas\n");
    exit(0);
  }
```

```
  else
    tablero[i][j] = 'O';
}

/* Mostrar el tablero en la pantalla. */
void mostrar_tablero(void)
{ int t;

  for(t=0; t<3; t++) {
    printf(" %c | %c | %c ",tablero[t][0],
                            tablero[t][1], tablero[t][2]);
    if(t!=2)
     printf("\n---|---|---\n");
  }
  printf("\n");
}

/* Ver si hay un ganador devolviendo la ficha del mismo */
char comprobar(void)
{ int i;

  for(i=0;i<3;i++) /* comprobar filas */
    if(tablero[i][0]==tablero[i][1] &&
       tablero[i][0]==tablero[i][2])
     return tablero[i][0];

  for(i=0;i<3;i++) /* comprobar columnas */
    if(tablero[0][i]==tablero[1][i] &&
       tablero[0][i]==tablero[2][i])
     return tablero[0][i];

  /* Comprobar diagonales */
  if(tablero[0][0]==tablero[1][1] &&
       tablero[1][1]==tablero[2][2])
     return tablero[0][0];

  if(tablero[0][2]==tablero[1][1] &&
       tablero[1][1]==tablero[2][0])
     return tablero[i][0];

  return ' ';
}
```

Otras modificaciones al programa que se pueden hacer son las siguientes:

1. Cambiar la función de juego del ordenador para que este siga una estrategia más agresiva.
2. Cambiar el programa para que jueguen 2 personas.

3. Modificar el programa para admitir todas las posibilidades, es decir, que juegue 2 personas, una persona y el ordenador o sólo el ordenador.

6.4 Funciones más complejas.

En este apartado hemos incorporado unos cuantos ejercicios que nos parecen muy importantes y que manejan algunas cosas tan importantes como operadores a nivel de bits y punteros al tipo nulo (`void *`).

6.4.1 Visualizar la representación en binario de un entero.

Escribir una función que escriba en la salida estándar la representación en binario de un número entero que se le dé como único argumento.

Solución:

```
/***********************************************/
/* Objetivo: Escribir en la Salida Estándar     */
/*           el valor en binario de un entero.  */
/* Entrada : Un entero.                         */
/* Salida  : Su valor en binario, por stdout .  */
/***********************************************/
void int2bin (int numero)
{ unsigned i = 0,
           TotalBits = 8 * sizeof (int),       /* 8 bits por byte */
           UltimoBit = 1 << (TotalBits-1);     /* Ultimo bit a 1  */

  printf("\n- El numero %i en binario es: ", numero);
  for ( ; i < TotalBits ; i++)
    printf("%i", ( (numero << i) &  UltimoBit) ? 1 : 0 );

  puts ("");
}
```

Para facilitar las operaciones, lo que hacemos es operar a nivel de bits. Suponemos que un byte son 8 bits y en `TotalBits` almacenamos el total de bits que se usan para representar un entero. En la variable `UltimoBit` almacenamos el número que resulta de correr hacia la izquierda el número 1, tantas veces como `TotalBits-1`. Con eso conseguimos el número que tiene un 1 seguido de todo ceros. Si suponemos que un entero utiliza 2 bytes, el entero 1 estará representado en binario por:

0000 0000 0000 0001

y el número `UltimoBit` estará representado en binario por el resultado de desplazar esos bits hacia la izquierda 15 posiciones:

1000 0000 0000 0000

En el bucle, lo que hacemos es hacer una operación AND, a nivel de bits (operador &), entre el número UltimoBit y el número en cuestión. Pero en cada iteración vamos desplazando los bits del número en cuestión una posición a la izquierda, de forma que en cada iteración vamos averiguando si es 1 ó 0 un dígito, empezando por el más a la izquierda (el más significativo).

De hecho, esa operación AND a nivel de bits lo que hace es decir si el número en cuestión desplazado i posiciones a la izquierda, tiene o no tiene el bit más significativo a 1. Usando el operador ?: decidimos si se imprime un 1 o un 0. Observe que esta función imprime los bits tal y como estén en el número en cuestión. Así, si el número es negativo dará la representación de dicho número negativo en nuestro ordenador, que habitualmente será en complemento a 2.

6.4.2 Conversión de base general.

Hacer una función que sirva para cambiar de base un número entero positivo. Supondremos que la base será como máximo 16 (hexadecimal).

La función printf() tiene varios modificadores para cambiar de base (%x, %o,...), y usando la variante sprintf() podemos conseguir el resultado en una cadena, pero el objetivo de nuestra función será poder pasar a cualquier base menor o igual que 16. El prototipo de nuestra función podría ser:

```
char *cambiobase (unsigned num, char cad[], unsigned base);
```

donde num es el número a cambiar de base, cab es la cadena de salida resultante y base es la base a la que vamos a cambiar el número.

Solución:

```
/***********************************************************/
/* Objetivo: Convertir un número entero positivo en base 10 */
/*           a otra base <= 16.                             */
/* Entrada : Número y base.                                 */
/* Salida  : Cadena de caracteres con el número en la nueva */
/*           base. Devuelve el puntero a esta cadena.       */
/* Requisitos: cad debe tener suficiente espacio reservado. */
/*             base debe ser <= 16                          */
/***********************************************************/
char *cambiobase (unsigned num, char cad[], unsigned base)
{ unsigned i, j;
  char c_aux;

  /* Hallamos los restos sucesivos: */
  for (j = 0; num >= base; j++) {
      cad[j] = num % base;
      num    = num / base;
  }
  cad [j] = num;
  cad [j+1] = '\0';
```

```
    /* Cambiamos los números por caracteres: */
    for (i=0; i<=j; i++)   /* j = longitud de cad */
        if (cad [i] < 10)
            cad [i] = cad [i] + '0';  /* Caracteres del '0' al '9'   */
        else switch (cad [i]) {       /* Del 10 al 15 (de 'A' a 'F') */
               case 10: cad [i] ='A'; break;
               case 11: cad [i] ='B'; break;
               case 12: cad [i] ='C'; break;
               case 13: cad [i] ='D'; break;
               case 14: cad [i] ='E'; break;
               case 15: cad [i] ='F'; break;
             }

     /* Le damos la vuelta a la cadena: */
     for (j = 0, i--; j < i ; j++, i--) {
         c_aux  = cad[j]; /* Intercambio primeros por últimos */
         cad[j] = cad[i];
         cad[i] = c_aux;
     }

     return cad;
}
```

Veamos cómo se comporta el anterior algoritmo con un número como 431 y una base como 16:

num	cad[0]	cad[1]	cad[2]	i	j	Nota
431						431>base
26	15				0	26>base
1	15	10			1	No 1>base
1	15	10	1		2	
	'F'	10	1	0	2	0<2
	'F'	'A'	1	1	2	1<2
	'F'	'A'	'1'	2	2	No 2<2
	'1'	'A'	'F'	2	0	Cambio (0,2)
	'1'	'A'	'F'	1	1	No j<i

Observe en la tabla anterior como distinguimos entre la representación numérica y la representación de caracteres. Estos últimos están entre comillas simples.

Finalmente obtendríamos en `cad` el valor `"1AF"` que es el número `431` en hexadecimal.

6.4.3 Intercambio entre valores de cualquier tipo.

Realizar una función de intercambio del valor de dos variables sean del tipo que sean.

Para esto usaremos punteros al tipo nulo (`void *`).

Solución:

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

void intercambiar(void *x, void *y, size_t tam);

void main(void)
{ float a,b;
  int c,d;

  clrscr();
  printf("\nIntroduce un número (float):");
  scanf("%f",&a);
  printf("\nIntroduce otro:");
  scanf("%f",&b);
  printf("\nLos números son %f y %f",a,b);

  intercambiar(&a,&b,sizeof(a));

  printf("\nLos números intercambiados son %f y %f",a,b);
  printf ("\n\nPulsa una tecla...");
  getch();

  clrscr();
  printf("\nIntroduce un número (int):"),
  scanf("%d",&c);
  printf("\nIntroduce otro:");
  scanf("%d",&d);
  printf("\nLos números son %d y %d",c,d);
  intercambiar(&c,&d,sizeof(c));
  printf("\nLos números intercambiados son %d y %d",c,d);
  printf ("\n\nPulsa una tecla...");
  getch();
}

/***************************************************************************/
/* Objetivo: intercambiar el contenido de dos variables de cualquier tipo*/
/* Entrada: x,y (variables a intercambiar),                               */
/*          tam (numero de bytes que ocupan las variables).               */
/* Salida: x,y (intercambiadas).                                          */
/***************************************************************************/
void intercambiar(void *x, void *y, size_t tam)
{
  char inter;
  int i;
```

```
  for (i=0;i<tam;i++) {
       inter              =  ( (char *)x )[i];
       ( (char *)x )[i]   =  ( (char *)y )[i];
       ( (char *)y )[i]   =  inter;
  }
}
```

6.4.4 Asignación de memoria dinámica.

Con este ejercicio se pretende esclarecer las bases más básicas del manejo de memoria dinámica.

Hacer un programa que declare inicialmente un puntero a carácter y que posteriormente se le asigne memoria dinámica suficiente para almacenar la cadena `"Aquí hay 22 caracteres"`. Tras asignarle la cadena se deberá mostrar la longitud de la misma con la función `strlen()` (de `string.h`).

Suponga a continuación que desea añadirle un punto final a la cadena. Añádaselo y muestre la longitud de la cadena modificada. Tenga en cuenta que cuando hace la asignación de memoria inicial no sabe que posteriormente tendrá que añadirle un nuevo carácter a la cadena, por lo que deberá reasignar nueva memoria al puntero de la cadena.

Solución:

```
/*************************************************************/
/* Objetivo: Ejemplo de asignación, reasignación y liberación */
/*           de memoria dinámica a una cadena de caracteres.  */
/*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{ char *cadena;

  cadena = (char *) malloc (23);
  if (!cadena) {
     puts ("ERROR: No hay suficiente memoria libre.");
     exit (1);
  }

  strcpy (cadena, "Aquí hay 22 caracteres");
  printf ("\n- Longitud inicial : %u.", strlen(cadena) );

  cadena = realloc (cadena, 24);
  if (!cadena) {
     puts ("ERROR: No hay suficiente memoria libre.");
     exit (1);
  }
```

```
    strcat (cadena, ".");
    printf ("\n- Longitud final   : %u.", strlen(cadena) );

    free(cadena);
    puts("\n  ***** FIN *****");
}
```

La asignación de memoria dinámica es una técnica muy útil y muy usada en programas medianamente complejos. Consiste en asignar memoria a nuestro programa justo cuando este la necesita y justo la cantidad que necesite en ese momento. En muchos programas, no se puede saber cuanta memoria vamos a necesitar, por lo que no podemos declarar tantas variables como necesitemos. Es necesario que el programa en tiempo de ejecución (cuando se está ejecutando) solicite al sistema operativo que el conceda memoria libre. Si no hay suficiente memoria, el sistema operativo informará al programa y éste deberá efectuar lo que considere oportuno (abortar la ejecución, indicar que no ha podido efectuar dicha operación...). Si no se consigue toda la memoria que el programa necesita no se debe usar dicha memoria como si hubiera sido asignada. Es totalmente imprescindible para escribir un buen programa que se compruebe si hemos conseguido o no asignar toda la memoria que el programa necesita.

El ANSI **C** incorpora la definición de algunas funciones para el manejo de memoria dinámica.

Como las cadenas de caracteres siempre terminan en el carácter `'\0'` tenemos que intentar asignar inicialmente 23 bytes con la función `malloc()`. Se podría también haber usado `calloc()`. Ambas funciones devuelven un puntero nulo (`void *`), por lo que usamos un molde (*casting*) para variar el tipo de la asignación a puntero a carácter (`char *`).

Posteriormente, con `realloc()` se intenta asignar un byte más a la cadena. Con `strcat()` se concatena el punto a la cadena y por último, con `free()` liberamos toda la memoria dinámica asignada al puntero. En algunos sistemas no es necesario liberar la memoria al final del programa pero es una muy buena costumbre liberar la memoria cuando deja de ser útil.

Cada vez que se asigna o reasigna memoria dinámica se debe comprobar si dicha operación ha finalizado correctamente. Cuando no se ha podido asignar memoria dinámica se devuelve un puntero nulo (`NULL`). En caso de no haber podido asignar toda la memoria que necesitamos en este programa, se da un mensaje de error y se aborta la ejecución del programa devolviendo un código de error 1.

Hemos supuesto que las funciones de asignación de memoria dinámica están declaradas en `stdlib.h`, pero en algunos sistemas están en `malloc.h`. Las funciones `malloc()`, `calloc()`, `realloc()` y `free()` pertenecen al estándar ANSI **C**.

6.4.5 Punteros a Funciones: Algo muy útil.

Codificar un programa en lenguaje C que lea un número real (`double`) de la entrada estándar y a continuación muestre un menú de opciones como el siguiente:

1. Logaritmo decimal.
2. Logaritmo neperiano.
3. Suelo del número (redondeo inferior, truncamiento).
4. Techo del número (redondeo superior).

5. Exponencial del número.
6. Cuadrado del número.
7. Salir.

El programa leerá una opción (tipo `unsigned`) del 1 al 7 y efectuará dicha operación al número que capturó anteriormente. Se pueden usar las funciones de la biblioteca matemática: `log10()`, `log()`, `floor()`, `ceil()` y `exp()`.

En ningún momento se deben usar las estructuras de control de Selección. Esto es, no se puede usar nunca en el programa ni `if` ni `switch`.

PISTA:

Un puntero a función es un puntero a la dirección de memoria donde comienza dicha función. Teniendo ese puntero se puede ejecutar (llamar) a dicha función. Es similar a lo que ocurre con los arrays: Igual que el identificador (nombre) de un array es el puntero al primer elemento del array, el identificador de una función es la dirección de la función.

Un puntero a una función que devuelve un entero se declara de esta peculiar forma:

```
int (*puf) ();
```

Los últimos paréntesis indican que se trata de una función, y los primeros nos indican que es un puntero a función. Si no se pusieran esos paréntesis:

```
int *puf ();
```

estaríamos declarando una función que devuelve un puntero a entero y no un puntero a función. Por tanto, `puf`, declarado de la primera forma, es un puntero a una función que devuelve un entero (entre los paréntesis de la función se pueden indicar el tipo de sus argumentos).

Darle valor a un puntero a función es tan fácil como asignarle el valor de la dirección de memoria de una función que ya exista. Así, si suponemos que tenemos una función `MCD()` que devuelve un entero, la siguiente asignación es correcta:

```
puf = MCD;
```

Tras esta asignación podemos llamar a la función `MCD()` a través del puntero:

```
printf ("Resultado %i.", (*puf) (5,8) );
```

donde:

- `puf` es el puntero a la función que deseamos ejecutar.

- `*puf` es la función en sí, ya que usamos el operador `*` (de indirección).
- `(*puf)` se pone entre paréntesis para indicar que el operador `*` va asociado a `puf`. Esos paréntesis son totalmente necesarios. De esta forma queda claro que los argumentos de la función `(5,8)` se refieren a la función `(*puf)`.

Cuando tenemos que dependiendo del valor de una variable se ejecuta una u otra función de una lista de opciones, se puede decidir qué función ejecutar mediante `if` anidados o con un gran `switch`. Pero si esta operación se realiza muchas veces esas dos formas son bastante poco eficientes. Lo mejor es tener un array de punteros a funciones.

Así, con el índice del array podremos referirnos directamente a la función que deseemos y no habrá necesidad de usar `if` o `switch`.

El siguiente programa tiene un array de tantos punteros a función como opciones tiene el menú (6). El programa leerá una opción y llamará a la función que está apuntada por el elemento del array que está en esa posición. Por ejemplo, si el programa lee que se desea ejecutar la opción 6 (cuadrado del número) deberá ejecutar la función que está apuntada por el sexto elemento, que corresponde a la posición 5 del array (recuerde que los arrays en C comienzan en la posición cero)

Solución:

```
/*******************************************************/
/* Objetivo: Aplicar diversas operaciones matemáticas   */
/*           a un número double, elijiendo la operación */
/*           de un menú y sin usar ni  if  ni  switch.  */
/*******************************************************/
#include <stdio.h>
#include <math.h>

#define MAXOPCION 6  /* Número de opciones en el menú. */

typedef double (*puntfunc) (double);    /* Definición del tipo puntfunc,
                                           que es un puntero a función. */

double cuadrado (double);  /* Devuelve el cuadrado de su único argumento. */
unsigned menu (void);      /* Muestra menú y devuelve la opción a ejecutar. */

void main ()
{ double num;
  unsigned opcion;
  puntfunc funciones [MAXOPCION] = { /* Array de punteros a funciones */
           log10,    /* Opción 1 (posición 0 del array) */
           log,      /*        2 */
           floor,    /*        3 */
           ceil,     /*        4 */
           exp,      /*        5 */
           cuadrado  /*        6 */
  };
```

```
  puts   ("****** OPERACIONES a un NUMERO ******\n");
  printf ("- Deme el número: ");
  scanf  ("%lf", &num);

  opcion = menu();  /* Leemos la opción del menú a ejecutar */

  while (opcion<=MAXOPCION) { /* Mientras sea una opción válida.. */
    printf ("\n\t* Opción %u aplicada al número %lf :\n", opcion, num);

    printf ("\t* SOLUCION: %lf",  (*funciones[opcion-1]) (num)  );

    opcion = menu();
  }

  puts ("***** FIN (THE END) *****");
}

/*************************************************/
/* Objetivo: Mostrar un menú de opciones y capturar  */
/*           la opción que desee ejecutar el usuario */
/* Salida   : Devuelve la opción que pulse el usuario */
/*************************************************/
unsigned menu (void)
{ unsigned op;

  puts ("\n\n\t******    Menú de Opciones    ******\n");
  puts ("\t 1. Logaritmo decimal del número.");
  puts ("\t 2. Logaritmo neperiano del número.");
  puts ("\t 3. Suelo del número (redondeo inferior, truncamiento).");
  puts ("\t 4. Techo del número (redondeo superior)."),
  puts ("\t 5. Exponencial del número.");
  puts ("\t 6. Cuadrado del número.");
  puts ("\n\t 7. Salir.");
  printf ("\n - Elija opción: ");
  scanf ("%u", &op);
  return op;
}

/*************************************************/
/* Objetivo: Calcular el cuadrado de un número.      */
/* Entradas: Número double.                          */
/* Salidas : Devuelve el cuadrado de la entrada.     */
/***********************************************Pp*/
double cuadrado (double valor)
{ return (valor * valor);
}
```

Con typedef definimos el tipo puntfunc que es un puntero a función que devuelve un double y que tiene como argumento otro double.

Ya en la función principal, se declara un array de elementos de tipo puntfunc y se le dá valor a dichos elementos directamente. Observe que log10, log, floor, ceil, exp y cuadrado son funciones que existen y que se han declarado previamente en el programa (unas en math.h y otras directamente en este módulo). Fíjese que hemos asignado los elementos al array puntfunc en el mismo orden que aparecen en nuestro menú.

Hay que hacer incapié en la simplicidad del bucle while. Cuando en la variable opcion tenemos la opción que deseamos ejecutar del menú, tan sólo tenemos que hacer una llamada a la función que está apuntada por el elemento del array funciones que está en la posición opcion-1 (se resta 1 porque los arrays en C empiezan en la posición cero). Esta llamada es:

```
(*funciones[opcion-1]) (num)
```

donde:

- opcion-1 es la posición del puntero a la función que deseamos ejecutar en el array funciones.
- funciones[opcion-1] es el puntero a la función que deseamos ejecutar.
- *funciones[opcion-1] es la función que deseamos ejecutar.
- num es el único argumento que tiene la función (se pone entre paréntesis como en cualquier otra llamada a función).
- (*funciones[opcion-1]) se pone entre paréntesis para indicar que el operador * va asociado a funciones[opcion-1] y no a otra cosa. Esos paréntesis son totalmente necesarios.
- la llamada a la función devuelve un valor de tipo double que será imprimido por la función printf() usando el código de formato %lf (*long float*).

Capítulo 7

Manejo y Control de Ficheros.

Naturalmente, no podía faltar un apartado que manejara este peculiar tipo de dato.

Es muy normal que, trabajando con un programa, queramos salvar nuestro trabajo en un disco u otro dispositivo para recuperarlo cuando sea preciso. Imagine por ejemplo que tiene un programa de diseño y desea guardar el diseño de un objeto particular (para usarlo, modificarlo... más tarde), o si en un juego deseamos guardar los nombres de los jugadores que han conseguido mejor puntuación. Para hacer eso se requieren manejar ficheros. Un fichero es un conjunto de información con un nombre determinado con el que podemos referirnos a él y con el que lo conoce el Sistema Operativo que usemos.

En C, el tipo de dato fichero es de tipo puntero a `FILE`. El contenido de `FILE` puede variar de un sistema operativo a otro, pero eso no debe importar al programador, pues este usa el tipo `FILE` sin importarle lo que este contenga.

Para manejar ficheros, primero hay que abrirlos, con la función `fopen()`, luego leer o escribir en ellos y por último cerrarlos con `fclose()`.

A continuación presentamos una serie de ejercicios básicos para el manejo de ficheros que utilizan las funciones más habituales. En el último tema se propone un ejercicio más complejo (una Agenda) que maneja ficheros.

7.1 Crear un fichero.

Hacer un programa que lea de la entrada estandar y que guarde todas las pulsaciones en un fichero que previamente se lo hemos pasado como un parámeto en la linea de comando del S.O (Sistema Operativo). El programa leerá caracteres, almacenándolos en el fichero, hasta que se pulse el carácter $.

Este es un ejemplo sencillo de como crear un fichero de texto.

Solución:

```
/************************************************************************/
/* Objetivo: Lee una serie de pulsaciones de teclas y las guarda en  */
/*           un fichero.                                              */
/* Entrada:  Pulsaciones de teclas y un nombre de fichero.            */
/* Salida :  Un fichero donde se almacenan                            */
/************************************************************************/
```

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

void main(int argc, char *argv[])
{
  FILE *fp;
  char car;

  if (argc!=2) {
    printf("Olvidó introducir el nombre del archivo\n");
    exit(1);
  }

  if((fp=fopen(argv[1],"w"))==NULL) {
    printf("No se puede abrir el archivo\n");
    exit(1);
  }

  do{                /* Bucle que lee un carácter hasta que se */
    car = getchar(); /* introcude el carácter $ */
    putc(car,fp);
  } while (car!='$');

  fclose(fp);
}
```

No hacemos ningún tratamiento especial a las pulsaciones de teclas que leemos. Una modificación al programa sería ver si se ha pulsado una tecla especial (como la de borrar) y hacer las acciones pertinentes a ello.

7.2 Mostrar un fichero: `type`.

Hacer un programa que simule la actuación de la orden del sistema operativo `type`, es decir, presentar un fichero de texto por la salida estandar. El nombre del fichero de texto será introducido como primer argumento en la línea de comandos del S.O.

Solución:

```
/*****************************************************************/
/* Objetivo: Lee un fichero y lo muestra en la pantalla          */
/* Entrada:  Nombre de fichero.                                  */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{ FILE *fp;
```

```
  char car;

  if (argc!=2) {
    printf("Olvidó introducir el nombre del archivo\n");
    exit(1);
  }

  if((fp=fopen(argv[1],"r"))==NULL) { /* Abrir para leer */
    printf("No se puede abrir el archivo\n");
    exit(1);
  }
  car = getc(fp); /* lee un carácter */

  while(car!=EOF) { /* mientras no fin de fichero */
    putchar(car);  /* lo muestra en pantalla */
    car = getc(fp);
  }

  fclose(fp);
}
```

7.3 Copiar de Ficheros: copy.

Realizar un programa que copie dos ficheros. El nombre de los ficheros debe de ser suministrado al programa por medio de la línea de comandos del SO.

El programa debe de hacer algo similar a lo que se realiza cuando utilizamos la orden copy de un S.O. tipo DOS (orden cp de UNIX).

Solución:

```
/*********************************************************************/
/* Objetivo: Copia de un fichero sobre otro                          */
/* Entrada:  Nombre de fichero origen y destino.                     */
/* Salida:   Copia del fichero origen en el destino                  */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  FILE *origen, *destino;
  char car;

  if (argc!=3) {
    printf("Olvidó introducir el nombre del archivo\
origen, destino o ambos.\n");
    exit(1);
```

```
  }
  if((origen=fopen(argv[1],"rb"))==NULL) ´/* Abrir para leer byte a byte */
    printf("No se puede abrir el archivo origen\n");
    exit(1);
  }

  if((destino=fopen(argv[2],"wb"))==NULL) { /* Abrir para escribir
                                               byte a byte (en binario) */
    printf("No se puede abrir el archivo destino\n");
    exit(1);
  }

  /* Copia del fichero origen sobre el destino */
  while(!feof(origen)) { /* Mientras no sea el final de fichero copiar */
    car = getc(origen); /* Se lee byte a byte */
    if(!feof(origen))
        putc(car,destino);
  }

  fcloseall(); /* Cierra todos los ficheros abiertos */
}
```

7.4 Borrar de un fichero: del.

Realizar un programa que simule la orden de sistema operativo (DOS) `del`. El nombre del fichero que deseamos eliminar a de pasarse al programa desde la línea de comandos del sistema operativo.

Solución

```
/*******************************************************************/
/* Objetivo: Borrado de un fichero                                 */
/* Entrada:  Nombre de fichero.                                    */
/*******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

main(int argc, char *argv[])
{ char cad[3];

  if(argc!=2) {
    printf("Uso: borra <nombre_fichero>\n");
    exit(1);
  }

  printf("Borrar %s? (S/N): ", argv[1]);
```

```
  gets(cad);

  if(toupper(*cad)=='S')
    if(remove(argv[1])) {
       printf("No se ha podido borrar el fichero\n");
       exit(1);
    }

  return 0; /* Se indica que la operación terminó bien */
}
```

7.5 Concatenación de ficheros.

Programa para concatenar dos ficheros de texto. El programa deberá leer de la entrada estándar los nombres de los dos ficheros. El fichero resultado se llamará SALIDA.NEW y contendrá el primer fichero y a continuación el segundo fichero.

Solución:

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

void main(void)
{
  FILE *f1,*f2,*fsal; /* Declaración de los 3 descriptores de ficheros */
  int letra;
  char nombre[100],nombre2[100];

  clrscr();
  printf("\nPrograma que concatena dos ficherós de texto");
  printf("\n\nNOMBRE DEL FICHERO 1:");
  gets(nombre);
  printf("\n\nNOMBRE DEL FICHERO 2:");
  gets(nombre2);

  if ( (f1=fopen(nombre,"rt")) == NULL ) {
     printf("\nERROR AL ABRIR");
     exit(1);
  }

  if ( (f2=fopen(nombre2,"rt")) == NULL ) {
     printf("\nERROR AL ABRIR");
     exit(1);
  }

  if ( (fsal=fopen("salida.new","wt")) == NULL ) {
     printf("\nERROR AL ABRIR");
```

```
      exit(1);
   }

   while (!(feof(f1)&&feof(f2))) {
      if (!feof(f1)) {
         letra=fgetc(f1);
         putc(letra,fsal),
      }
      else {
         letra=fgetc(f2);
         putc(letra,fsal);
      }
   }

   fclose(f1);
   fclose(f2);
   fclose(fsal);

}
```

La utilidad de este programa la suelen incorporar la mayoría de los sistemas operativos con algunas órdenes. En MS-DOS se puede hacer con la orden `copy` (con el argumento +) o usando la orden `type` con redirecciones (>>). En UNIX existe un comando, casi explícitamente para eso, llamado `cat` (*concat*) que hay que usarlo redireccionando su salida estándar.

Obsérvese que al final del programa se cierran todos los ficheros utilizados. Un fichero se debe cerrar siempre que no se vaya a necesitar más. Al final del programa no es totalmente imprescindible, pues por defecto se cierran, pero es recomendable.

Cuando ocurre un error salimos con la función `exit()` que cierra automáticamente todos los fichero que estén abiertos.

7.6 Eliminación de tabuladores en un fichero.

Lo que debemos de hacer es lo que se conoce como un filtro, es decir, tomamos como entrada un fichero de texto y hacemos ciertas transformaciones en el mismo. En este caso buscamos el carácter especial de tabulación y lo sustituimos por un número determinado de caracteres blancos.

Se debe de dar en la línea de comandos el nombre del fichero a filtrar y otro para el fichero filtrado. La cantidad de espacios por los que vamos a sustituir un tabulador se debe de fijar antes de empezar el programa.

Otra cosa que debemos de hacer es que el programa sea capaz de tratar los errores que se produzcan en el transcurso de la operación de filtrado.

Solución:

```
/*****************************************************************/
/* Objetivo: Toma como entrada un fichero y substituye las       */
/*           tabulaciones por espacios y realiza comprobación de */
```

```
/*            errores.                                                */
/* Entrada:   Nombre de fichero origen y destino (de texto).          */
/* Salida:    Fichero filrado de tabulaciones                         */
/**********************************************************************/
#include <stdio.h>
#include <stdlib.h>

#define TAB_LONG 8
#define ORIGEN 0
#define DESTINO 1

void err(int e);

void main(int argc, char *argv[])
{ FILE *origen, *destino;
  int tab, i;
  char car;

  if (argc!=3) {
    printf("Uso: quitatab <origen> <destino>\n");
    exit(1);
  }
  if((origen=fopen(argv[1],"rb"))==NULL) {
    printf("No se puede abrir el archivo %s\n", argv[1]);
    exit(1);
  }
  if((destino=fopen(argv[2],"wb"))==NULL) {
    printf("No se puede abrir el archivo %s\n", argv[2]);
    exit(1);
   }

  tab = 0;
  do {
    car = getc(origen);
    if(ferror(origen))
       err(ORIGEN);

    if(car=='\t') { /* Si se encuentra un tabulador */
                    /* Introduce el número apropiado de espacios */
      for(i=tab; i<8; i++) {
         putc(' ',destino);
         if(ferror(destino))
         err(DESTINO);
      }
      tab = 0;
    }
    else {
```

```
      putc(car,destino);
      if(ferror(destino))
        err(DESTINO);
      tab++;
      if(tab==TAB_LONG)
         tab = 0; /* Se ha llegado al límite del tabulador y se pasa al
                     siguiente salto de tabulador */
      if(car=='\n' || car=='\r')
        tab = 0; /* Hay un salto de l¡nea y se empieza por el primer
                    tabulador */
    }
  } while(!feof(origen));

  fcloseall(); /* Cierra todos los ficheros abiertos */
}

/*
Función que realiza el tratamiento del error segun un parámetro de entrada.
*/
void err(int e)
{ if(e==ORIGEN)
    printf("Error en el fichero origen\n");
  else
    printf("Error en el fichero destino\n");
  exit(1);
}
```

7.7 Cambio de un carácter por otro en un fichero.

Programa para cambiar en un fichero un carácter por otro. El fichero resultado se llamará `SALIDA.NEW` y el de entrada se leerá de la entrada estándar. El programa leerá el nombre de fichero de entrada, el carácter a cambiar y el carácter por el cual se va a cambiar.

Solución:

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

void main(void)
{ FILE *f,*f2;
  int letra,lnew,lold;
  char nombre[100];

  clrscr();
  printf("\nPROGRAMA QUE CAMBIA UN CARACTER POR OTRO EN UN  FICHERO");
  printf("\n\nNOMBRE DEL FICHERO:");
  gets(nombre);
```

```
  printf("\nLetra a cambiar: ");
  lold=getche();
  printf("\nPor la letra: ");
  lnew=getche();

  if ( (f=fopen(nombre,"rt")) == NULL ) {
     printf("\nERROR AL ABRIR");
     exit(1);
  }

  if ( (f2=fopen("salida.new","wt")) == NULL ) {
     printf("\nERROR AL ABRIR");
     exit(1);
  }

  while (!feof(f)) {
    letra=fgetc(f);
    if (letra==lold)
       putc(lnew,f2);
    else
       putc(letra,f2);
  }
}
```

7.8 Lectura de datos desde fichero.

Leer desde un fichero un conjunto de coordenadas de puntos tridimensionales y almacenarlos en un vector, teniendo en cuenta que cada trio de valores es una línea del fichero, y que cada casilla del vector debe contener un registro con los campos x,y,z de un punto.

Solución:

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

typedef struct {  double x;
                  double y;
                  double z;
} punto;

void main (void)
{ FILE *fp;
  char nom_fich[40];
  char s1[10],s2[10],s3[10];
  punto p[50];
  int i;
```

```
  printf("\nIntroduzca el nombre del fichero a leer: ");
  gets(nom_fich);

  if ((fp=fopen(nom_fich,"r"))==NULL) {
     printf("\nLo siento, no se puede abrir");
     exit(1);
  }

  i=0;
  while (!feof(fp)) {
    fscanf(fp,"%s%s%s\n",s1,s2,s3);
    p[i].x=atof(s1);
    p[i].y=atof(s2);
    p[i].z=atof(s3);
    printf("\n%lf %lf %lf",p[i].x,p[i].y,p[i].z);
    i++;
  }

  fclose(fp);
}
```

7.9 Cálculo del centro de masas con datos desde fichero.

Leer desde un fichero un conjunto de coordenadas de puntos tridimensionales y almacenarlos en un vector, teniendo en cuenta que cada trio de valores es una línea del fichero, y que cada casilla del vector debe contener un registro con los campos x,y,z de un punto. Posteriormente se hallará el centro de masas del sistema de partículas que determinan los puntos.

Solución:

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

#define DATOS "datos "
#define TAM 20

typedef struct
{ double masa;
          double x;
          double y;
} PUNTO_BI;

void ini_punto(PUNTO_BI puntos[TAM]);
PUNTO_BI cdm_bi(PUNTO_BI puntos[TAM]);
void leer_datos(PUNTO_BI *puntos, char *nombre);
```

```
void mostrar_datos(PUNTO_BI punto);

/*COMIENZO: main()*/
void main(void)
{
  PUNTO_BI puntos[TAM],cdm;
  ini_punto(puntos);
  leer_datos(puntos,DATOS);
  cdm = cdm_bi(puntos);
  mostrar_datos(cdm);
}

/*********************************************************************/
/* Objetivo: inicializa un vector de elementos de tipo PUNTO_B       */
/* Salida: puntos (vector de PUNTO_B)                                */
/* Observaciones: la inicialización consiste el poner el campo masa a -1*/
/*********************************************************************/
void ini_punto(PUNTO_BI puntos[TAM])
{ int i;

  for (i=0;i<TAM;i++)
      puntos[i].masa=-1;
}

/**********************************************************************/
/* Objetivo: leer desde un fichero los datos de de un vector de PUNTO_BI */
/* Requisitos: existencia del fichero de datos en el que              */
/*             en cada línea se encuantren las corrdenadas de un punto  */
/*             El fichero no debe contener más líneas que el tamaño del */
/*             vector.                                                */
/* Entrada: nombre (cadena de caracteres con el nombre del fichero).   */
/* Salida: puntos (vector de PUNTO_BI).                               */
/* Observaciones: el fichero puede contener menos líneas              */
/*                que el tamaño del vector.                           */
/**********************************************************************/
void leer_datos(PUNTO_BI *puntos, char *nombre)
{
  FILE *f;
  int i;
  char cad1[20],cad2[20],cad3[20];
  f=fopen(nombre,"r");
  i=0;

  while(!feof(f)) {
    fscanf(f,"%s%s%s\n",cad1,cad2,cad3);
    puntos[i].masa=atof(cad1);
    puntos[i].x=atof(cad2);
```

```
    puntos[i].y=atof(cad3);
    i++;
  }

  fclose(f);
}

/**************************************************************/
/* Objetivo: hallar el centro de masas de vector de PUNTO_BI */
/* Entrada: puntos (vector de PUNTO_BI).                      */
/* Salida: con return el centro de masas de tipo PUNTO_BI    */
/**************************************************************/
PUNTO_BI cdm_bi(PUNTO_BI puntos[TAM])
{
  int i;
  PUNTO_BI resul={0,0,0};

  for (i=0 ; puntos[i].masa!=-1 ; i++) {
      resul.masa += puntos[i].masa;
      resul.x += puntos[i].x*puntos[i].masa;
      resul.y += puntos[i].y*puntos[i].masa;
  }

  resul.x = resul.x/resul.masa;
  resul.y = resul.y/resul.masa;
  return resul;
};

/********************************************************************/
/* Objetivo: mostrar el contenido de una variable de tipo PUNTO_BI */
/* Entrada: punto (de tipo PUNTO_BI).                               */
/********************************************************************/
void mostrar_datos(PUNTO_BI punto)
{
  clrscr();
  printf("\n\n\n\n\n\n\t\tEl centro de masas es:\n");
  printf("\n\n\t\t\tMASA\t = \t%lf",punto.masa);
  printf("\n\n\t\t\tX\t = \t%lf",punto.x);
  printf("\n\n\t\t\tY\t = \t%lf",punto.y);
}
```

Este ejemplo nos sirve para aprender de qué forma puede una función retornar una serie de valores. Esto ese puede conseguir diseñando una función para que devuelva una estructura (o de otra forma, utilizando varios parámetros por referencia).

En general cuando se plantea la necesidad de utilizar una función se debe tener muy claro qué datos se le deben proporcionar como entrada, y qué datos debe la función devolver como salida. Los datos de entrada sólo pueden ser pasados a la función como parámetros, en

la llamada. Los datos de salida pueden salir de la función como parámetros de salida, por referencia, o como un dato devuelto con `return`, teniendo en cuenta para esto último, que si se quieren retornar varios datos la función debe retornar un dato de tipo estructura que los contenga todos.

El fichero fuente se encuentra dividido en varias partes:

1. includes.
2. defines.
3. definiciones de nuevos tipos.
4. declaración de prototipos de funciones.
5. definición de la función principal `main()`.
6. definición de las demás funciones.

7.10 Tamaño de un fichero

Programar una función que nos devuelva el tamaño (en bytes) de un fichero que se le pase como su único argumento. Lo que le pasaremos a la función será el puntero, de tipo (`FILE *`), por lo que el fichero ya debe estar abierto anteriormente.

Tras la llamada a la función, el apuntador que nos indica por donde vamos leyendo o escribiendo en el fichero debe permanecer en la misma posición que antes de la llamada.

Solución:

```
/****************************************************/
/* Objetivo: Averiguar el tamaño de un fichero.     */
/* Entrada : Fichero (fich).                        */
/* Salida  : Devuelve el tamaño en bytes de fich.   */
/* Requisitos: Fichero YA abierto.                  */
/****************************************************/
long BytesFichero (FILE *fich)
{ long NumBytes,  /* Tamaño del fichero. */
       posicion;  /* Posición anterior de fich, que no modificamos. */

  posicion = ftell (fich);      /* Posición anterior.  */
  fseek (fich, 0L, SEEK_END);  /* Nos vamos al final. */
  NumBytes = ftell (fich);      /* La posición es el tamaño del fichero. */
  fseek (fich, posicion, SEEK_SET);  /* Reponemos la posición anterior. */

  return NumBytes;
}
```

La función `ftell(f)` que nos devuelve (en un `long`) la posición actual del apuntador del fichero `f`. La función `fseek(f, offset, inicio)` cambia la posición del apuntador del fichero `f`, a un desplazamiento `offset` (de tipo `long`) a partir de la posición indicada en `inicio` (de

tipo `int`). En `inicio` se pueden usar unas macros que nos indican a partir de donde se mide el desplazamiento `offset`:

Macro	Número	A partir de
SEEK_SET	0	Comienzo del fichero.
SEEK_CUR	1	Posición actual del apuntador.
SEEK_END	2	Final del Fichero.

En la función, primero guardamos la posición actual del apuntador del fichero, en la variable `posicion`. Luego, nos vamos al final del fichero con la función `fseek()`, usando `0L` (cero como `long`) como `offset` y tras esto, en la posición del apuntador tenemos el tamaño del fichero, que lo guardamos en la variable `NumBytes`. Finalmente, reponemos la posición del apuntador en el fichero `f`, mediante otra llamada a la función `fseek()`, usando la posición que guardamos al principio.

7.11 Inversión de un fichero.

Codificar una función que nos escriba un fichero en otro pero de forma invertida, es decir, que empiece escribiendo el segundo fichero con el final del primer fichero. La función aceptará los ficheros ya abiertos (`FILE *`), el primero en modo lectura y el segundo en modo escritura.

Solución:

```
/*****************************************************/
/* Objetivo: Escribir un fichero de forma inversa.  */
/* Entrada : Fichero a invertir (ent).              */
/* Salida  : Fichero invertido (sal).               */
/* Requisitos: Ficheros YA abiertos en modo lectura */
/*             y escritura respectivamente.         */
/*****************************************************/
void invierte (FILE *ent, FILE *sal)
{ long int posicion;
  char x;

  fseek(ent,0L,SEEK_END);

  for (posicion=ftell(ent)-1 ; posicion >= 0; posicion--) {
     fseek(ent,posicion,SEEK_SET);
     x=fgetc(ent);
     fputc(x,sal);
  }
}
```

La explicación es sencilla: Primeramente, posicionamos el apuntador del fichero al final de este, y luego, en el `for`, leemos esa posición y la vamos decrementando hasta que llegamos al inicio del fichero, hasta que `(posicion==0)`

En cada iteración, leemos un carácter (x) del fichero de entrada `ent` y lo escribimos en el fichero de salida `sal`. Se podía haber abreviado usando `fputc(fgetc(ent),sal);` sin tener que usar el carácter x intermedio.

7.12 Ordenar un fichero de palabras (en memoria).

Codificar una función que ordene un fichero de texto, con palabras y lo escriba, ordenadamente en otro fichero. La función aceptará dos punteros a `FILE`, por lo que ambos ficheros deben estar previamente abiertos en modo lectura y escritura respectivamente.

Se debe usar el algoritmo de ordenación por BURBUJA, ya visto en un ejercicio anterior pero para enteros. Podemos establecer que como máximo, ordenaremos las 100 primeras palabras del primer fichero de entrada. Además, supondremos que cada palabra tiene un máximo de 50 caracteres. En el fichero, la separación entre palabras será por algún carácter separador: espacios, tabuladores, retornos de carro...

La ordenación de las palabras se realizará en memoria, no directamente en disco. En otro ejercicio se verá una ordenación directamente desde el disco.

En el fichero de entrada, el carácter `EOF` (*End Of File*) que nos indica que se ha terminado el fichero, debe aparecer al final de la última palabra.

En el fichero de salida las palabras se escribirán separadas por un espacio.

Solución:

```
/************************************************************/
/* Objetivo: Ordenar un fichero con palabras.               */
/* Entrada : Un fichero de texto como origen.               */
/* Salida  : Un fichero destino, con las palabras ordenadas. */
/* Requisitos: - Deben estar YA abiertos en modo lectura y  */
/*               escritura respectivamente.                 */
/*             - Máximo 100 palabras a ordenar.             */
/*             - Máximo  50 caracteres por palabra.         */
/*             - Carácter EOF al final de la última palabra. */
/************************************************************/
void fburbuja (FILE *origen, FILE *destino)
{ int pos,w,v=1,N;
  char array[100][51];
  char aux[51];

  /* Pasamos el fichero a un array: */
  for (N=0; !feof(origen) && N<100; N++)
      fscanf(origen, "%s", array[N]);

  /* Ordenamos el array de N elementos por BURBUJA: */
  do { w=pos=0;
       do { if ( strcmp(array[pos],array[pos+1]) > 0 ) {
                 strcpy (aux, array[pos]);     /*Intercambio*/
                 strcpy (array[pos], array[pos+1]);
                 strcpy (array[pos+1], aux);
```

```
                w=1;  /* w=1 --> Array NO ordenado aún. */
            }
            pos++;
        }
        while ( pos < N-v );

        v++;
    }
    while (w);

  /* Pasamos el array al fichero destino: */
  for (pos=0; pos<N; pos++)
      fprintf(destino, "%s ", array[pos]);
}
```

Lo primero que hacemos es leer las palabras del fichero en un array de cadenas de caracteres. Como máximo leemos 100 palabras. Obsérvese, que si el carácter `EOF` no está al final de la última palabra, al leer esta no detectaremos el final del fichero y volveremos a leer otra palabra más, por lo que nos incluirá en el array una palabra extra de *basura*.

Después, con el algoritmo Burbuja ordenamos el array. Para una explicación más detallada del algoritmo Burbuja, véase un ejercicio anterior sobre enteros. Tras el algoritmo Burbuja, copiamos, el array ordenado en el fichero de salida.

Es importante remarcar que al copiar las cadenas de caracteres, usamos la función `strcpy()`, y no una asignación simple. Si usáramos:

```
aux = array[pos];    /* ¡¡Incorrecto!! */
```

lo que estaríamos haciendo es copiar un puntero en otro, pero no la cadena de caracteres. Por eso es necesario usar esa función que nos copia toda la cadena de caracteres del segundo al primer argumento.

Para la comparación de cadenas, usamos la función `strcmp()` que nos devuelve un número positivo si la primera cadena es mayor que la segunda y negativo en caso inverso. Si son iguales, devuelve cero.

Naturalmente, esto hace una ordenación en el código que usemos, que será normalmente el `ASCII`. En este código, las mayúsculas están antes que las minúsculas, por lo que la palabra `"HOLA"` estaría antes que `"hola"`, y antes que `"Hay"`.

7.13 Ordenacion de fichas en un archivo.

Suponemos que tenemos un archivo en el que se almacenan una serie de fichas que tienen la siguiente estructura:

```
struct direc {
  char nombre[30];
  char calle[40];
  char ciudad[20];
  char provincia[2];
```

```
  char codigo[10];
} dinfo;
```

Se debe de realizar un programa que las ordene. La ordenación debe de llevarse sobre el archivo, es decir, no se debe de leer las fichas en un array y luego ordenar el mismo.

Para que la ordenación en el fichero sea sencilla se considera que el mismo es de acceso directo, en el caso que se tratase de un fichero con acceso secuencial la ordenación debería de hacerse en memoria para que la operación no se complique demasiado.

Para la ordenación de la estructura se va a considerar como campo clave el `codigo`. Una vez que tenemos un campo clave la ordenación se realiza de la misma forma en que ordenamos números, sólo hay que tener en cuenta que lo que tenemos es una estructura compleja y no un dato simple.

Como en otros programas el fichero a ordenar se le suministra al programa por medio de la línea de comandos del sistema operativo.

Solución

```
/*********************************************************************/
/* Objetivo: El programa ordena de forma rápida una estructura de      */
/*           direcciones que se encuentra almacenada en un fichero    */
/*           de acceso directo.                                        */
/* Entrada:  Nombre del fichero.                                       */
/* Salida:   El fichero ordenado.                                      */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_ELEMENT 100 /* Este es un número arbitrario que se debe de
                           determinar dinámicamente para cada lista */
struct direc {
  char nombre[30];
  char calle[40];
  char ciudad[20];
  char provincia[2];
  char codigo[10];
} dinfo;

void rapida_disco(FILE *fp, int cont);
void or_disco(FILE *fp, int izq, int der);
void cambiar_campos(FILE *fp, long i, long j);
char *obtener_codigo(FILE *fp, long rec);

void main(int argc, char *argv[])
{ FILE *fp;

  if(argc!=2) {
```

```
    printf("Uso: ordena <nombre_fichero>\n");
    exit(1);
  }

  if((fp=fopen(argv[1],"rb+"))==NULL) {
    printf("Imposible abrir archivo para lectura/escritura\n");
    exit(1);
  }

  rapida_disco(fp,NUM_ELEMENT);
  fclose(fp);
  printf("Lista ordenada\n");
}

/* Una ordenación rápida para archivos */
void rapida_disco(FILE *fp, int cont)
{
  or_disco(fp, 0, cont-1);
}

void or_disco(FILE *fp, int izq, int der)
{ long int i,j;
  char x[100];

  i = izq;
  j = der;

  strcpy(x, obtener_codigo(fp,(long)(i+j)/2));
  /* Obtener código medio */
  /* La ordenación se realiza por medio de intercambio */
  do {
    while(strcmp(obtener_codigo(fp,i),x)<0 && i<der) i++;
    while(strcmp(obtener_codigo(fp,j),x)>0 && j>izq) j--;

    if(i<=j) {
      cambiar_campos(fp,i,j);
      i++;
      j--;
    }
  } while(i<=j);

  if(izq<j)
    or_disco(fp,izq,(int)j);
  if(i<der)
    or_disco(fp, (int)i,der);
}
```

```
void cambiar_campos(FILE *fp, long i, long j)
{ char a[sizeof(dinfo)], b[sizeof(dinfo)];

  /* primero lee los registros i y j */
  fseek(fp, sizeof(dinfo)*i, 0);
  fread(a, sizeof(dinfo), 1, fp);

  fseek(fp, sizeof(dinfo)*j, 0);
  fread(b, sizeof(dinfo), 1, fp);

  /* después los escribe en orden contrario */
  fseek(fp, sizeof(dinfo)*j, 0);
  fwrite(a, sizeof(dinfo), 1, fp);

  fseek(fp, sizeof(dinfo)*i, 0);
  fread(b, sizeof(dinfo), 1, fp);
}

/* Devolver un puntero al código postal */
char *obtener_codigo(FILE *fp, long rec)
{ struct direc *p;

  p = &dinfo;

  fseek(fp, rec*sizeof(dinfo), 0);
  fread(p, sizeof(dinfo), 1, fp);

  return dinfo.codigo;
}
```

El método de ordenación que hemos utilizado es uno de los métodos de ordenación rápida. Se podría haber elegido cualquier otro método para la ordenación. El método de ordenación consiste en elegir un elemento llamado `comparando`. Luego se debe dividir la lista en dos, en una primera parte aquellos elementos que sean menores o iguales al valor de partición y en el otro lado los mayores. Este metodo se repite con las dos mitades y asi hasta que no se puede hacer más. Una vez que se ha terminado tenemos toda la lista ordenada.

Otra consideración que se debe de hacer es que una vez que sabemos el tamaño de la estructura que compone la ficha, sabemos cuanto ocupa en disco, por lo que podremos acceder a cada una de las fichas de forma independiente como si se tratasen de elementos almacenados en un array en memoria. Sólo hay que cambiar las operaciones que debemos de hacer, es decir, debemos de leer o escribir en disco en vez de realizar simples asignaciones de variables en memoria.

Capítulo 8

Conceptos básicos de Sonidos y Gráficos.

Este es un tema muy importante al que le damos muy poca importancia. La explicación es fácil: Siempre hemos pretendido, en la medida de lo posible, ajustarnos al estándar del lenguaje ANSI C, y para este tema, el ANSI C no tiene apenas estándares ya que tanto las funciones gráficas como las de sonido dependen mucho de la arquitectura particular de la máquina que se utilice. Nosotros, para este tema, supondremos una arquitectura PC.

No queremos ni tan siquiera insinuar que estos temas carezcan de importancia, sino todo lo contrario. Las nuevas tecnologías en programación y las cada vez mayores exigencias de los usuarios y del mercado, están exigiendo cada vez más a los programadores, que sus creaciones sean de las llamadas Multimedia. Simplificando un poco, podemos decir que la Multimedia es la ciencia que se encarga de englobar y sincronizar muchos recursos en un programa u ordenador. Así, un buen programa multimedia será capaz de visualizar gráficos, vídeo, animaciones, incorporar sonidos y música... y todo ello, a ser posible, de forma interactiva, de manera que el usuario pueda *navegar* fácilmente por el programa minimizando los tiempos de aprendizaje y amenizando el uso del programa en cuestión.

Obviamente, para conseguir todo eso necesitaremos un Sistema Operativo que libere al programador de la tarea que supone programar para distintos tipos de tarjetas de sonido, vídeo... En este reducido tema nos centraremos en las generalidades más básicas de estas técnicas bajo un S.O. tan precario como el DOS.

Nos gustaría añadir, que aunque con el Lenguaje de Programación **C**, y mejor aún, con su hermano mayor **C++**, se pueden programar todo tipo de aplicaciones, existen otros lenguajes, los llamados Lenguajes de Autor, que simplifican en gran medida la creación de programas Multimedia. Incluso, la mayoría de estos Lenguajes de Autor permiten hacer llamadas a bibliotecas de funciones que podemos haberlas programado en C. Ejemplo de esto son las DLL (*Dinamic Link Library*, Bibliotecas de enlace dinámico) que utiliza Windows.

8.1 Noche de Paz.

Hacer un programa que interprete el famoso villancico de Franz Grüber, Noche de Paz, a una sola voz. Para ello usará el pequeño altavoz que se incorpora de serie en un PC.

El programa recibirá un argumento en la línea de comandos, que será un entero positivo que indique la duración de la nota negra. Si no se especifica este valor, tomará uno por defecto

(por ejemplo 1100) De esta forma podremos controlar la velocidad de ejecución de la obra.

En el Apéndice A se muestra una tabla con las frecuencias de las 8 octavas de notas más usuales.

Solución:

```
/************************************************/
/* Objetivo: Interpretar el famoso villancico   */
/*           Noche de Paz, por el pequeño       */
/*           altavoz (speaker) de un PC.        */
/*           Funciona sólo en sistemas bajo     */
/*           un S.O. de tipo DOS.               */
/* Entrada : El programa recibe como argumento */
/*           un entero positivo que representa */
/*           la duración, en milisegundos,  de */
/*           la nota negra. Si no se especifica */
/*           este valor en la línea de comandos */
/*           tomará uno por defecto.            */
/************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

/* Definición de las Frecuencias de las Notas */
#define C5  524  /* DO en la escala 5 */
#define D5  587  /* RE  */
#define E5  659  /* MI  */
#define F5  698  /* FA  */
#define G5  784  /* SOL */
#define A5  880  /* LA  */
#define B5  988  /* SI  */
#define C6 1047  /* DO en la escala 6 */
#define D6 1175  /* RE  */
#define E6 1329  /* MI  */
#define F6 1397  /* FA  */

#define NEGRA 1100  /* Nota Negra (en milisegundos), por defecto     */
                    /* A partir de aquí, calculamos las demás notas. */

void Noche_De_Paz (unsigned);
void toca (unsigned, unsigned);

void main (int argc, char *argv[])
{ int negra;

  if (argc < 2)      /* No hay argumentos en la línea de comandos */
     Noche_De_Paz (NEGRA);
```

```
  else {
     negra = atoi (argv[1]);
     if (negra > 0)
        Noche_De_Paz (negra);
     else {
        puts("\n* NOCHE de PAZ :");
        puts("\t  - Debe especificar un número mayor que cero.");
        puts("\t  - Dicho número será la duración de la nota Negra.");
     }
  }
} /* main */

/************************************************/
/* Objetivo: Interpretar el famoso villancico   */
/*           Noche de Paz, por el pequeño       */
/*           altavoz (speaker) de un PC.        */
/* Entrada: Duración de la nota negra en mlsgs. */
/************************************************/
void Noche_De_Paz (unsigned negra)
{ int i;
  unsigned blanca          = negra * 2,
           blanca_puntillo = negra * 3,
           negra_puntillo  = negra + negra/2,
           corchea         = negra / 2;

  for ( i=0; i<2; i++) {  /* Repetir 2 veces */
      toca (G5,negra_puntillo);
      toca (A5,corchea);
      toca (G5,negra);
      toca (E5,blanca_puntillo);
  }

  toca (D6,blanca);
  nosound(); /* Por ser dos notas iguales seguidas  */
  toca (D6,negra);
  toca (B5,blanca_puntillo);
  toca (C6,blanca);
  nosound();
  toca (C6,negra);
  toca (G5,blanca_puntillo);

  for ( i=0; i<2; i++) {  /* Repetir 2 veces */
      toca (A5,blanca); nosound();
      toca (A5,negra);
      toca (C6,negra_puntillo);
      toca (B5,corchea);
      toca (A5,negra);
```

```
      toca (G5,negra_puntillo);
      toca (A5,corchea);
      toca (G5,negra);
      toca (E5,blanca_puntillo);
  }

  toca (D6,blanca); nosound();
  toca (D6,negra);
  toca (F6,negra_puntillo);
  toca (D6,corchea);
  toca (B5,negra);
  toca (C6,blanca_puntillo);
  toca (E6,blanca_puntillo);

  toca (C6,negra);
  toca (G5,negra);
  toca (E5,negra);
  toca (G5,negra);
  toca (F5,negra);
  toca (D5,negra);
  toca (C5,blanca_puntillo); /* ligadura */
  toca (C5,negra);

  nosound();
}

/**************************************************/
/* Objetivo: Toca una nota y espera una duración */
/* Entrada : La nota, en herzios y la duración   */
/*           en milisegundos.                    */
/* Salida  : Sonido por el altavoz de un PC.     */
/* Requisitos: La función no apaga la nota, por  */
/*           lo que se deberá usar la función    */
/*           nosound() posteriormente.           */
/**************************************************/
void toca ( unsigned nota, unsigned duracion)
{ sound (nota);
  delay (duracion);
}
```

Hay que indicar que, en su especificación, la función `delay()` recibe un argumento que son los milisegundos que debe esperar. Sin embargo, hemos comprobado que esto es falso y que el tiempo que espera depende de la velocidad del ordenador en el que se ejecuta. Existe otra función de similares características: `sleep()`

Figura 8.1: Símbolo de la PAZ.

8.2 Símbolo de la Paz.

Hacer un programa que dibuje en la pantalla el Símbolo de la Paz. Dicho símbolo se puede ver en la figura 8.1.

Solución:

```
/*****************************************/
/* Objetivo: Programa para la PAZ mundial */
/*           Bueno... por desgracia, sólo */
/*           dibuja el símbolo de la PAZ. */
/*****************************************/
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>

void Inicializar(void);  /* Inicializa el modo gráfico */

void main()
{ int CentroX, CentroY,  /* Centro de la Pantalla */
      MaxX, MaxY,        /* Valores máximos en la pantalla */
      Color;             /* Color a usar */

  Inicializar();

  CentroX = (MaxX=getmaxx())/2,
  CentroY = (MaxY=getmaxy())/2;
  Color=getmaxcolor()-4;
  setcolor(Color);
  setbkcolor(1);        /* Fondo oscuro, pero no negro */
```

```
  circle (CentroX,CentroY,CentroY-20);
  circle (CentroX,CentroY,CentroY-70);
  setfillstyle(SOLID_FILL,Color);
  floodfill(CentroX,21,Color);

  line(CentroX-25,70,CentroX-25,MaxY-70);
  line(CentroX+25,70,CentroX+25,MaxY-70);
  floodfill(CentroX,CentroY,Color);

  line(CentroX+25,CentroY+25,CentroX+120,CentroY+120);
  line(CentroX+25,CentroY-35,CentroX+145,CentroY+145-60);
  floodfill(CentroX+26,CentroY,Color);

  line(CentroX-25,CentroY+25,CentroX-120,CentroY+120);
  line(CentroX-25,CentroY-35,CentroX-145,CentroY+145-60);
  floodfill(CentroX-26,CentroY,Color);
  getch(); /* Espera a que se pulse una tecla */

  setcolor(getcolor()-1);
  settextstyle(GOTHIC_FONT,HORIZ_DIR,7);
  outtextxy(15,5,"Amor");
  getch();
  outtextxy(MaxX-110,5,"y");
  getch();
  outtextxy(40,MaxY-99,"Paz");

  setcolor(getcolor()-3);
  settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
  outtextxy(MaxX-110,MaxY-30,"Pp (95)");

  getch();
  closegraph(); /* Cerramos el modo gráfico */
}

/***************************************************/
/* Objetivo: Inicializa el modo gráfico y avisa de */
/*           los posibles errores.                 */
/***************************************************/
void Inicializar(void)
{ int GraphDriver, GraphMode, ErrorCode;

  GraphDriver = DETECT;  /* Request auto-detection */
  initgraph( &GraphDriver, &GraphMode, "" );
  ErrorCode = graphresult(); /* Read result of initialization*/
  if( ErrorCode != grOk ){ /* Error occured during init */
    printf("\n- ERROR en el sistema Gráfico: %s\n",
```

```
grapherrormsg( ErrorCode ) );
    exit(1);
  }

}
```

Capítulo 9

Utilidades y programas más complejos.

Este capítulo pretende dar una visión más completa y general de lo que es un programa en C. Hemos tratado de buscar unos ejercicios fáciles pero significativos, que muestran gran parte de la potencia de este lenguaje.

Además, como primer ejercicio de este capítulo incorporamos cómo hacer una biblioteca de funciones para manejar el ratón. Con esto se pueden mejorar los ejercicios siguientes para que soporten este útil periférico.

9.1 Control del ratón para DOS.

En este ejercicio trataremos de explicar los principios básicos para el tratamiento y manejo del ratón, un periférico muy útil, en un sistema PC basado en la familia de procesadores x86. El objetivo será construir una biblioteca de funciones para controlar el roedor desde un programa en C.

Primeramente, repasemos algunos conceptos básicos. Para que el ratón funcione es necesario primero que esté conectado a un puerto serie del ordenador (`COMi`, usualmente `COM1` o `COM2`), y luego, tener cargado en memoria un controlador (*driver*) o programa residente que se encargue de las tareas de más bajo nivel del ratón, como son el movimiento del cursor del ratón, y el control de los puertos series. El controlador del ratón usa dos interrupciones, una hardware y otra software. La primera se encarga de la gestión del puerto serie y se encargará de avisar al procesador cuando ocurra algún evento (movimiento del ratón o pulsación de algún botón). Acto seguido, se genera una interrupción software, la 33h (33 en hexadecimal, o `0x33`) que será la que actualize el estado del ratón, y la posición del cursor. Para ello, tendrá que borrar el cursor en la posición anterior, reponer lo que allí había, y dibujarlo en la posición actual.

Todo eso es tarea del *driver* y es transparente al programador. El programador sólo tendrá que vigilar el estado del ratón: Posición en pantalla y estado de sus botones. Esto se hace usando una interrupción software.

Una interrupción software es un programa que puede efectuar distintas acciones dependiendo del servicio que se solicite de dicha interrupción. Para obtener información sobre el estado del ratón, tenemos que ejecutar (o generar) la interrupción 33h. El servicio requerido, de esta interrupción, se almacena en el registro AX del procesador, y en los registros BX, CX

Figura 9.1: El ratón, un periférico muy importante...

y DX se almacenarán los parámetros requeridos antes de la interrupción y servirán también para devolvernos en ellos los valores deseados.

El prototipo de la función que usaremos para generar la interrupción, es el siguiente:

```
int86 ( int Num_Interrupcion,
        union REGS *entrada,
        union REGS *salida );
```

En `entrada` se almacenan los valores de los registros de entrada, y en `salida` se obtendrán los valores devueltos. Hay que tener en cuenta que la unión `REGS` está definida como sigue:

```
union REGS {
   struct WORDREGS x;
   struct BYTEREGS h;
};

struct WORDREGS {
   unsigned int ax, bx, cx, dx;
   unsigned int si, di, cflag, flags;
};

struct BYTEREGS {
   unsigned char al, ah, bl, bh;
```

Servicio	Utilidad	Entrada	Salida
0	Inicializar Ratón	AX = 0	AX = 0 ⟶ Si no hay ratón. BX = 0 ⟶ Tiene 1 ó más de 3 botones. BX = 2 ⟶ 2 botones compatibles Microsoft. BX = 3 ⟶ 3 botones del estándar Logitech. BX = 65535 ⟶ 2 botones no comp. Microso
1	Mostrar Cursor	AX = 1	
2	Ocultar Cursor	AX = 2	
3	Ver estado	AX = 3	CX ⟶ Posición Horizontal. BX&1 = 1 ⟶ Botón izdo. pulsado. BX&2 = 2 ⟶ Botón dcho. pulsado.
4	Posicionar Cursor	AX = 4 CX = Coordenada X DX = Coordenada Y	
7	Limitar espacio en X	AX = 7 CX = Valor mínimo de X DX = Valor máximo de X	
8	Limitar espacio en Y	AX = 8 CX = Valor mínimo de Y DX = Valor máximo de Y	

Tabla 9.1: Servicios de la interrupción 33h.

```
    unsigned char cl, ch, dl, dh;
};
```

Antes de empezar a trabajar con el ratón hay que inicializarlo y verificar que el ratón está conectado, con el servicio 0 de la interrupción 33h. Para mostrar el cursor se usará el servicio 1, y para ocultarlo, se usará el servicio 2.

Si deseamos hacer alguna operación con la pantalla, como imprimirla o hacer volcados de RAM a RAM de video, debemos ocultar antes el ratón para que no cause errores. Estos errores se producen al modificar el estado de la pantalla en la posición del ratón y luego, al mover el ratón, se restaurará, lo que allí había antes del cambio (deshaciendo ese cambio en ese trozo de pantalla).

Podemos mantener una variable global `visualizado` que nos indique si el cursor está activo u oculto, de forma que no sea necesario activarlo, si ya está activo, u ocultarlo si ya está oculto.

Para ver el estado del ratón, se usa el servicio 3, que nos dará en CX su posición horizontal, en DX su posición vertical. Las coordenadas devueltas serán relativas a una pantalla VGA (640x480), aunque estemos en modo texto. En BX nos dará el estado de los botones: Si está a 1 el bit de orden cero (el más bajo), es que está pulsado el botón izquierdo, y si está a 1 el bit de orden 1, estará pulsado el botón derecho. De esta forma también podremos saber si están pulsados los 2 botones.

Para posicionar el ratón en una posición determinada se usará el servicio 4, y para delimitar el espacio de movimiento del ratón se usará el servicio 7 para la coordenada X y el servicio 8 para la Y.

En la tabla 9.1 se muestran los servicios mencionados con más detalle.

Por tanto, para manejar el ratón desde un programa en C, sólo habrá que implementar unas funciones que nos permitan ejecutar todas las acciones mencionadas. Podría ser algo, como lo siguiente:

Solución:

```
/*******************************************/
/* Biblioteca de funciones para manejo del */
/*                 R A T O N               */
/*******************************************/
#include <dos.h>

char visualizado; /* Indica si el ratón está o no visualizado */

/****************************************************/
/* Objetivo: Inicializar el ratón.                  */
/* Devuelve el Número de botones que tiene:         */
/*      -1  -->  ERROR: No hay ratón o controlador. */
/*       0  -->  tiene 1 ó más de 3 botones.        */
/*       2  -->  2 botones compatibles Microsoft.   */
/*       3  -->  3 botones del estándar Logitech.   */
/*   65535  -->  2 botones no compatibles Microsoft. */
/****************************************************/
int inicializa_raton (void)
```

```
{ union REGS registros; /* Registros del procesador */

  registros.x.ax = 0;   /* Servicio a ejecutar de la interrupción 33h */
  int86(0x33,&registros,&registros); /* Ejecución de esa interrupción */

  if (!registros.x.ax)
     return -1;

  visualizado = 0;
  return registros.x.bx;
}

/****************************************************/
/* Objetivo: Visualiza el ratón si está oculto.     */
/* Salida  : Pone la variable 'visualizado' a 1.    */
/****************************************************/
void visualiza_raton (void)
{ union REGS registros;

  if (!visualizado) {
     registros.x.ax = 1;
     int86(0x33,&registros,&registros);
     visualizado = 1;
  }
}

/****************************************************/
/* Objetivo: Oculta el ratón si está visualizado.   */
/* Salida  : Pone la variable 'visualizado' a 0.    */
/****************************************************/
void oculta_raton (void)
{ union REGS registros;

  if (visualizado) {
     registros.x.ax = 2;
     int86(0x33,&registros,&registros);
     visualizado = 0;
  }
}

/****************************************************/
/* Objetivo: Obtener la posición del ratón y        */
/*           el estado de sus botones.              */
/* Salida  : Valor de la posición x e y en sus args. */
/*           Devuelve el estado de sus botones:     */
/*    Bit más bajo    = 1  --> Pulsando botón izdo. */
/*    Sig. bit + bajo = 1  --> Pulsando botón dcho. */
```

```
/*    Ambos bits      = 1  --> Ambos botones están  */
/*                         pulsados simultáneamente. */
/***************************************************/
int lee_raton (int *x, int *y)
{ union REGS registros;

  registros.x.ax = 3;
  int86(0x33,&registros,&registros);

  *x = registros.x.cx;
  *y = registros.x.dx;
  return registros.x.bx;
}

/***************************************************/
/* Objetivo: Poner el cursor del ratón en una       */
/*           posición (x,y) concreta.               */
/* Entrada : Dos enteros: (x,y).                    */
/***************************************************/
void pon_raton (int x, int y)
{ union REGS registros;

  registros.x.ax = 4;
  registros.x.cx = x;
  registros.x.dx = y;
  int86(0x33,&registros,&registros);
}

/***************************************************/
/* Objetivo: Establece una ventana o recuadro, donde */
/*           se puede mover el ratón.               */
/* Entrada : Cuatro enteros: Esquinas superior-izda  */
/*           e inferior-dcha del recuadro deseado.   */
/***************************************************/
void limites_raton (int izda, int arriba, int dcha, int abajo)
{ union REGS registros;

  registros.x.ax = 7; /* Coordenada X */
  registros.x.cx = izda;
  registros.x.dx = dcha;
  int86(0x33,&registros,&registros);

  registros.x.ax = 8; /* Coordenada Y */
  registros.x.cx = arriba;
  registros.x.dx = abajo;
  int86(0x33,&registros,&registros);
}
```

```
/***************************************************/
/* Objetivo: Lee si está pulsado el botón izquierdo  */
/*           del ratón.                              */
/* Salida   : 0 si NO está pulsado. Otro valor si    */
/*           está pulsado el botón izquierdo.        */
/* También se puede saber con la función lee_raton() */
/***************************************************/
char boton_izdo (void)
{ union REGS registros;

  registros.x.ax = 3;
  int86(0x33,&registros,&registros);

  return registros.x.bx & 1; /* Operación booleana a nivel de bits */
}

/***************************************************/
/* Objetivo: Lee si está pulsado el botón derecho    */
/*           del ratón.                              */
/* Salida   : 0 si NO está pulsado. Otro valor si    */
/*           está pulsado el botón derecho.          */
/* También se puede saber con la función lee_raton() */
/***************************************************/
char boton_dcho (void)
{ union REGS registros;

  registros.x.ax = 3;
  int86(0x33,&registros,&registros);

  return registros.x.bx & 2;
}

/***************************************************/
/* Objetivo: Lee si están pulsados los botones       */
/*           izquierdo y derecho del ratón.          */
/* Salida   : 0 si NO están pulsados. Otro valor si  */
/*           están pulsados ambos botones.           */
/* También se puede saber con la función lee_raton() */
/***************************************************/
char dos_botones (void)
{ union REGS registros;

  registros.x.ax = 3;
  int86(0x33,&registros,&registros);

  return (registros.x.bx & 1) && (registros.x.bx & 2);
```

```
}
```

Podemos observar que las funciones `boton_izdo()`, `boton_dcho()` y `dos_botones()` no son realmente necesarias, ya que se obtiene esa información con la función `lee_raton()`. Se podrían incluir también funciones que esperaran a que se pulsara uno o ambos botones. Eso se deja como ejercicio propuesto para el lector.

A continuación se expone un programa de prueba de las anteriores funciones:

```
/**********************************************************/
/* Objetivo: Programa de prueba de las rutinas de manejo  */
/*           y control del roedor.                        */
/**********************************************************/
#include<stdio.h>
#include<conio.h>

main ()
{ int x,y;

  if (inicializa_raton() == -1) {
     puts ("Error al inicializar ratón");
     exit (-1);
  }

  visualiza_raton();
  limites_raton(50,50, 150,150);
  pon_raton(100,100);
  clrscr();

  do { lee_raton (&x,&y);
       gotoxy(1,1);printf ("---> (%i,%i); \t",x,y);
       gotoxy(1,2);
       if (boton_izdo() && !boton_dcho())
          puts("- Pulsado izdoooooo!!!!!!!");
       if (boton_dcho() && !boton_izdo())
          puts("- Pulsado Dchoooooo.......");
       if (dos_botones())
          puts("- Pulsados 2 botonesssssss");
  } while (!kbhit());

  oculta_raton();
}
```

9.2 Una calculadora sencilla.

Implementar un programa calculadora, que permita hacer múltiples operaciones aritméticas. Nuestra calculadora será capaz de realizar las siguientes operaciones:

- Operadores binarios:

Operador	Tecla
Suma	+
Resta	-
Multiplicación	*
División	/
Módulo (resto)	M
Potencia (n-ésima)	E
Raíz (n-ésima)	R
Tanto por ciento	%
Porcentaje	P

- Operadores unarios:

Operador	Tecla
Logaritmo Neperiano	N
Logaritmo Decimal	L
Exponencial	X
Cambio de Signo	I

El objetivo es que el programa se comporte (en lo fundamental) como una calculadora convencional, en el modo de pedir los datos y de ir mostrando los resultados. Se prestará más atención a dicho comportamiento que a la presentación gráfica de la calculadora. La tecla de = se sustituirá por `RETURN` (`INTRO`). Además, incorporará teclas para anular el último dígito tecleado (*backspace*), para anular el operando completo (tecla `ESC`) y, por supuesto, para salir del programa (tecla `S`).

- Ejemplos: Si se pulsa la secuencia de teclas de la izquierda, seguidas de `RETURN`, se debe producir el resultado de la derecha:

`1 + 2`	`3`
`1 + 2 - 5`	`-2`
`5 M 4`	`1`
`200 % 10`	`20`
`10 P 200`	`5`

Para facilitar más el trabajo debe permitir trabajar en mayúsculas o minúsculas y usará el tipo `double` para poder tratar con decimales.

Para no complicar más el programa, no hemos incluido operaciones de cambio de base (decimal, hexadecimal, octal, binario...), ni operaciones trigonométricas. Además, tampoco hemos incluido la típica memoria temporal que incorporan la mayoría de las calculadoras (las teclas `MC`, `M+` y `M-`) ni la operación sucesiva manteniendo constante el segundo operando, al pulsar reiteradamente `RETURN` (=) después de una operación binaria. Dejamos como ejercicio incluir todas estas funcionalidades y otras que se puedan añadir.

Solución:

```
/*************************************************/
/* Objetivo: Simular el comportamiento de una    */
```

```
/*           calculadora convencional.          */
/* Entrada : Los operandos y los operadores se  */
/*           introducirán por teclado.          */
/* Salida  : El resultado de las operaciones y  */
/*           un mensaje de estado, por pantalla.*/
/***********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <string.h>
#include <conio.h>

#define ESC '\x1B'    /* Carácter ESCape (27) */
#define MAX 50        /* Máximo número de dígitos */
#define MAXMSG 70     /* Máximo número de caracteres en el msg de estado */
#define DECIMALES 10 /* Máximo número de decimales a obtener */
#define ENTER 13      /* Carácter '\n' */

#define BINARIOS "+-*/MER%P" /* 9 Operadores Binarios */
#define UNARIOS  "XNLI"      /* 4 Operadores Unarios  */


void ayuda(void); /* Muestra la ayuda por pantalla */
void dibujo(void);
void inicializar (char *,int,int);
char estado1 (void);
char estado2 (void),
char estado3 (void),
char estado4 (void);
double operar (double, double, char);
double operar_unario (double, char);
void prepara_unario (char);
void conversion (double, char []);
void copia (char [], char[]);


/* Variables Globales: Se usan en casi todas las funciones y siempre
                       tienen el mismo significado.
                       Esto, simplifica los pasos de parámetros. */
char op1[MAX],      /* Operandos */
     op2[MAX],
     haypunto1 = 0, /* Si hay punto decimal en los respectivos operandos */
     haypunto2 = 0,
     msg[MAXMSG],   /* Mensaje de estado */
     opdor = 0;     /* Indica si hay o no OPERADOR y cual es */
int  posop1=0, posop2=0; /* Indica la posición actual de ese OPERANDO */
```

```
/*********************- PRINCIPAL -**********************/
void main (int argc, char *argv[])
{ char NoSalir = 1,  /* Vale 1 mientras no se quiera salir del programa */
       estado  = 1,  /* Estado inicial del autómata de estados */
       c;            /* Carácter leído actual */

  if (argc>1)
     if (argv[1][1] == '?')
        ayuda();
     else {
        cprintf("\n\n* Programa CALCULADORA:");
        cprintf("\n  Teclee:  CALC /?   Muestra una ayuda.");
        cprintf("\n           CALC      Ejecuta la Calculadora.\n");
     }
  else {
     inicializar(op1,0,MAX);
     inicializar(op2,0,MAX);
     inicializar(msg,0,MAXMSG);
     copia (msg,"***** Introduzca la operación *****");
     clrscr();
     dibujo();
     do {
       gotoxy(25,3);cputs(op1);
       gotoxy(25,5);putch(opdor);
       gotoxy(25,7);cputs(op2);
       gotoxy(10,21);cputs(msg);

       switch (estado) {
         case 1: estado = estado1();
                 break;
         case 2: estado = estado2();
                 break;
         case 3: estado = estado3();
                 break;
         case 4: estado = estado4();
                 break;

         case 10: case 20: case 30: case 40: /* Salir ? */
                  inicializar(msg,0,MAXMSG);
                  gotoxy(10,21); cputs(msg);
                  gotoxy(10,21); cprintf("* Pulse otra vez 'S' para Salir,");
                  gotoxy(10,22); cprintf("  u otra tecla para continuar.");
                  c=toupper(getch());
                  if (c == 'S') NoSalir = 0;
                  else          estado  = estado / 10;
```

```
                clrscr();
                dibujo();
                break;

        } /*switch*/
      } while (NoSalir);

      clrscr();
      cputs ("*** FIN CALCULADORA ***");
  } /*else*/

} /* main */

/***********************************************/
/* Objetivo: Muestra una lacónica ayuda, por    */
/*           pantalla.                          */
/***********************************************/
void ayuda (void)
{ cputs("\n * CALCULADORA para DOS de fácil y rápido uso.");
  cputs("\n   Introduzca las operaciones como en una calculadora normal,");
  cputs("   pero en esta Calculadora podrá ver en todo momento, los operandos");
  cputs("   y el operador que actuará sobre ellos.");
  cputs("\n   Para Salir deberá pulsar dos veces la tecla 'S'.");
  cputs("   Para Cancelar los operandos, pulse ESC.");
  cputs("   Para Borrar el último dígito pulse Retroceso <- (DEL).");
  cputs("\n   Operadores posibles: Suma (+), Resta (-), Multiplicación (*),");
  cputs("   División (/), Módulo (M), Potencia (E), Raíz enésima (R),");
  cputs("   Porcentaje (%), Proporción (P), Exponencial (X),\n");
  cputs("   Invertir signo (I),");
  cputs("   Logaritmo decimal (L) y Logaritmo Neperiano (N).\n");
}

/************************************************/
/* Objetivo: Muestra el dibujo de la calculadora */
/*           y todos los operadores disponibles. */
/************************************************/
void dibujo (void)
{ gotoxy(10,3); cprintf("- Operando 1:");
  gotoxy(10,5); cprintf("- Operador  :");
  gotoxy(10,7); cprintf("- Operando 2:");
  gotoxy(1,9);  cprintf("* OPERADORES:");
  gotoxy(1,11);
  cprintf("BINARIOS: [+]  Suma.  [*]  Multiplicación.  \
[E]  Elevado (potencia).");
  gotoxy(10,12);
  cprintf(" [-]  Resta. [/]  División.        \
[R]  Raíz enésima (Ej.: 16R4=2).");
```

```
  gotoxy(10,13);
  cprintf(" [M]  Módulo (resto) de los valores enteros."),
  gotoxy(10,14);
  cprintf(" [%]  Porcentaje (Ej.: 10%200=20)  \
[P]  Proporción (Ej.: 10P200=5%).");
  gotoxy(1,16);
  cprintf("UNARIOS : [X]  Exponencial                     [I]  Invertir Signo.");
  gotoxy(10,17);
  cprintf(" [N]  Logaritmo Neperiano.         [L]  Log. base 10.");
  gotoxy(10,19);cprintf("Pulse 'S' para Salir, y 'ESC' para anular.");
}

/**********************************************/
/* Objetivo: Poner espacios en una porción de  */
/*           una cadena de caracteres.         */
/* Entrada : Rango de la cadena que queremos   */
/*           inicializar a espacios: ini y fin. */
/* Salida  : Cadena con espacios entre las po-  */
/*           siciones ini y fin-2.             */
/*           La longitud de la cadena es fin-2. */
/**********************************************/
void inicializar (char *string, int ini, int fin)
{ for (; ini<fin; ini++)
      string[ini] = ' ';
  string[fin-1] = '\0';
}

/**********************************************/
/* Objetivo: Aceptar los primeros caracteres   */
/*           del operando1 (op1).              */
/*           Se puede anular op1 pulsando ESC.  */
/* Entrada : Lee un carácter de teclado (c).   */
/* Salida  : Modifica: op1, posop1, haypunto y  */
/*           msg.                              */
/*           Devuelve el siguiente estado.     */
/**********************************************/
char estado1 (void)
{ char c;

  do { c = toupper(getch()); /* Pasamos a mayúscula */

       switch (c) {
         case 'S': /* Salir ? */
                   return 10;

         case '\b':
         case ESC: inicializar(op1,0,MAX);
```

```
                    posop1 = 0;
                    copia (msg,"***** Comience la operación *****");
                    return 1;

          case '-': op1[0] = '-';
                    posop1 = 1;
                    copia (msg,
                    "***** Operando 1 NEGATIVO: Pulse ESC para anularlo *****");
                    return 1;

          case '.': haypunto1 = 1;
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
                    op1[posop1] = c;
                    posop1++;
                    copia (msg,
               "***** Introduciendo Operando 1: Pulse ESC para anularlo *****");
                    return 2;
        }

  } while (1);
}

/**********************************************/
/* Objetivo: Aceptar el resto de caracteres     */
/*           del operando1, hasta que se intro- */
/*           duzca un operador válido (o ESC).  */
/*           Si el operador es unario se opera, */
/*           y si es binario se va al estado 3  */
/*           para leer el operando2.            */
/*           Se puede anular op1 pulsando ESC.  */
/* Entrada : Lee un carácter de teclado (c).    */
/* Salida  : Modifica: op1, posop1, haypunto1 y */
/*           opdor y msg.                       */
/*           Devuelve el siguiente estado.      */
/**********************************************/
char estado2 (void)
{ char c;

  do { c = toupper(getch()); /* Pasamos a mayúscula */

       switch (c) {
         case 'S': /* Salir ? */
                   return 20;

         case ESC: inicializar(op1,0,MAX);
                   posop1 = haypunto1 = 0;
```

```
          copia (msg,
          "***** Anulado operando 1: Inicie operación *****");
          return 1;

case '.': if (haypunto1)  /* Si ya hay punto, ni caso */
             break;
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
          if (posop1 >= (MAX-1)) {
             copia (msg,"***** ¡ No admito más números ! *****");
             msg[MAXMSG-2] = '\a'; /* beep */
             op1[MAX-1]    = '\0';
          }
          else {
            if (c == '.') haypunto1 = 1;
            op1[posop1] = c; /* Más dígitos de op1 */
            posop1++;
          }
          return 2;

case '\b': /* Borrar último dígito introducido en op1, si existe */
           if (posop1 > 1) {
              posop1--;
              if (op1[posop1] == '.') haypunto1=0;
              op1[posop1] = ' ';
              copia (msg,"***** Borrando Operando 1 *****");
              return 2;
           }
           else {
              op1[0] = ' ';
              posop1 = haypunto1 = 0;
              copia (msg,
                "***** Operando 1 totalmente borrado *****");
              msg[MAXMSG-2] = '\a'; /* beep */
              return 1;
           }

case '-': case '+': case '*': case '/': /* OP. BINARIOS */
case 'M': case '%': case 'E': case 'R':
case 'P': opdor = c,
          copia (msg,
 "***** Ahora, puede cambiar el operador dado, si lo desea *****");
          return 3;

 case 'L': case 'N': case 'X': case 'I': /* OP  UNARIOS */
           prepara_unario (c);
```

```
                return 3;
      }

  } while (1);
}

/***************************************************/
/* Objetivo: Aceptar el primer dígito del operando2  */
/*           o cambiar el operador introducido antes.*/
/*           Si el operador es unario se opera,      */
/*           y si es binario se va al estado 3       */
/*           para leer el operando2.                 */
/*           Se puede anular op1, pulsando ESC.      */
/* Entrada : Lee un carácter de teclado (c).         */
/* Salida  : Modifica: op1, posop1, haypunto1, opdor,*/
/*                     op2, posop2, haypunto2 y msg. */
/*           Devuelve el siguiente estado.           */
/***************************************************/
char estado3 (void)
{ char c;

  do { c = toupper(getch()); /* Pasamos a mayúscula */

       switch (c) {
         case 'S': /* Salir ? */
                   return 30;

         case ESC: inicializar(op1,0,MAX);
                   posop1 = haypunto1 = 0;
                   opdor  = 0; /* Se borra también el operador */
                   copia (msg,
               "***** Anulado operando 1 y Operador: Inicie operación *****");
                   return 1;

         case '.': haypunto2=1;
         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':

                   if (strchr(UNARIOS,opdor)!=NULL) /* Si opdor es unario, */
                      return 3;          /* ignoramos el segundo operando. */

                   op2[posop2] = c;  /* Este es el primer dígito de op2 */
                   posop2++;
                   copia (msg,
             "***** Introduciendo Operando 2: Pulse ESC para anularlo *****");
                   return 4;
```

```
        case '-': case '+': case '*': case '/': /* OP. BINARIOS */
        case 'M': case '%': case 'E': case 'R':
        case 'P':
                  opdor = c; /* Modificamos el operador */
                  copia (msg,"***** Operador Actual: \'");
                  msg[24]=opdor;
                  msg[25]='\0';
                  strcat(msg,"\' *****");
                  inicializar(msg,32,MAXMSG);
                  return 3;

        case 'L': case 'N': case 'X': case 'I': /* OP. UNARIOS */
                  prepara_unario (c);
                  return 3;
      }

  } while (1);
}

/****************************************************/
/* Objetivo: Aceptar resto de dígitos de operando2.  */
/*           Se puede anular op2 (con ESC) y volver  */
/*           a introducirlo, o ver el resultado de   */
/*           la operación binaria (pulsando Enter    */
/*           o un operador binario).                 */
/* Entrada : Lee un carácter de teclado (c).         */
/* Salida  : Modifica: op2, posop2, haypunto2,       */
/*                     opdor y msg.                  */
/*           Devuelve el siguiente estado.           */
/****************************************************/
char estado4 (void)
{ double x,y;
  char c;

  do { c=toupper(getch()); /* Pasamos a mayúscula */

      switch (c) {
        case 'S': /* Salir ? */
                  return 40;

        case ESC: /* Pulsando un ESC se anula el op2, y podemos cambiar
                     (o no) el operador.
                     Pulsando otra vez ESC, se anula op1 en el estado3 */
                  inicializar (op2,0,MAX);
                  posop2 = haypunto2 = 0;
                  copia (msg,
        "***** Anulado operando 2. Pulse ESC Para anular también el 1 *****");
```

```
          return 3;

case '.': if (haypunto2) /* Si ya tiene punto, ni caso */
             break;
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
          if (posop2 >= (MAX-1)) {
             copia (msg,"***** ¡ No admito más números ! *****");
             msg[MAXMSG-2] = '\a'; /* beep */
             op2[MAX-1]    = '\0';
          }
          else {
            if (c == '.') haypunto2=1;
            op2[posop2] = c; /* Más dígitos de op2 */
            posop2++;
          }
          return 4;

case '\b': /* Borrar último dígito introducido en op2, si existe */
           if (posop2 > 1) {
              posop2--;
              if (op2[posop2] == '.') haypunto2 = 0;
              copia (msg,"***** Borrando Operando 2 *****");
              op2[posop2] = ' ';
              return 4;
           }
           else { /* Si se borra totalmente, debe ir al estado 3 */
             op2[0] = ' ';
             posop2 = haypunto2 = 0;
             copia (msg,
               "***** Operando 2 totalmente borrado *****");
             msg[MAXMSG-2] = '\a'; /* beep */
             return 3;
           }

case ENTER: c=opdor;  /* Guardamos el operador anterior */
case '-': case '+': case '*': case '/': case 'M':
case '%': case 'E': case 'P': case 'R':
          inicializar (op1,posop1,MAX);
          x = atof(op1);
          inicializar (op2,posop2,MAX);
          y = atof(op2);
          if (opdor=='/' && y==0) {  /* División por 0 */
             copia (msg,
               "***** DIVISIÓN por 0: Resultado infinito *****");
             msg[MAXMSG-2] = '\a'; /* beep */
          }
```

```
                    else if ( (opdor=='R') &&
                              (x<0)        &&
                              (int)y==y    &&
                              !((int)y%2) ) {
                            copia (msg,
            "***** ERROR: No existe una Raíz par de un número negativo *****");
                            msg[MAXMSG-2] = '\a'; /* beep */
                    }
                    else {
                      x = operar(x,y,opdor);
                      copia (msg,
                        "***** El RESULTADO está en el Operando 1 *****");
                      conversion (x,op1);
                      posop1 = strlen(op1);
                    }
                    inicializar (op2,0,MAX);
                    posop2 = haypunto2 = 0;
                    opdor  = c;
                    return 3;
        }

  } while (1);
}

/****************************************************/
/* Objetivo: Realizar la operación con un operador   */
/*           binario.                                */
/* Entrada : Dos operandos (x,y) y un operador.      */
/* Salida  : Devuelve el resultado de la operación.  */
/****************************************************/
double operar (double x, double y, char operador)
{ switch (operador) {
    case '+': return x+y;               /* Suma           */
    case '-': return x-y;               /* Resta          */
    case '*': return x*y;               /* Multiplicación */
    case '/': return x/y;               /* División       */
    case 'M': return (int)x % (int)y;   /* Módulo         */
    case 'E': return pow(x,y);          /* Potencia       */
    case 'R': return pow(x,1/y);        /* Raíz y-ésima   */
    case '%': return (x*y)/100;         /* Tanto por cien */
    case 'P': return (100*x)/y;         /* Porcentaje     */
  }
}

/****************************************************/
/* Objetivo: Realizar la operación con un operador   */
/*           unario.                                 */
```

```
/* Entrada : Un operando (x) y un operador unario.   */
/* Salida  : Devuelve el resultado de la operación.  */
/**************************************************/
double operar_unario (double x, char operador)
{ switch (operador) {
    case 'L': return log10(x);        /* Logaritmo en base 10 */
    case 'N': return log(x);          /* Logaritmo Neperiano  */
    case 'X': return exp(x);          /* Exponencial          */
    case 'I': return (-1)*x;          /* Invertir Signo       */
  }
}

/**************************************************/
/* Objetivo: Ver si es posible realizar la operación */
/*           con un operador unario. En caso afirma- */
/*           tivo, prepara dicha operación.          */
/* Entrada : Un operador (c).                        */
/* Salida  : Modifica: op1, posop1, opdor y msg.     */
/**************************************************/
/* Prepara la operación unaria, si es posible */
void prepara_unario (char c)
{ double x;

  inicializar (op1, posop1, MAX);
  x = atof(op1);
  if ((c=='L' || c=='N')  &&  x < 0) {
     copia (msg,
       "***** ERROR: Logaritmo no definido para números negativos *****");
     msg[MAXMSG-2] = '\a'; /* beep */
  }
  else {
    x = operar_unario (x,c);
    copia (msg,"***** El RESULTADO está en el Operando 1 *****");
    conversion (x,op1);
    posop1 = strlen(op1);
    opdor  = c;
  }
}

/**************************************************/
/* Objetivo: Convierte un double a cadena de máximo  */
/*           MAX caracteres.                         */
/* Entrada : Un double (x)                           */
/* Salida  : Una cadena con los dígitos de x.        */
/**************************************************/
void conversion (double x, char cadena[])
{ char inter[MAX];
```

```
  int posicion = 0, i;

  if (x<0) { /* Signo */
     cadena[0] = '-';
     posicion  = 1;
     x = (-1)*x; /* Pasamos a positivo */
  }

  /* PARTE ENTERA */
  ltoa ((long int) floor(x),inter,10); /* Parte entera */
  for(i=0; (inter[i]!='\0') && (posicion<MAX-1); i++,posicion++)
     cadena[posicion] = inter[i];

  if (posicion >= (MAX-1)) {
     copia (msg,
     "***** Hay más dígitos en la parte entera que se han PERDIDO *****");
     cadena[MAX-1] = '\0';
     return;
  }

  x = x-floor(x);
  if (x == 0) {  /* No hay decimales */
     inicializar (cadena,posicion,MAX);
     return;
  }
  cadena[posicion] = '.';
  posicion++;

  /* PARTE DECIMAL */
  i = 0; /* Número de decimales, por el momento */
  while ((x>0.00001) && (posicion<MAX-1) && i<DECIMALES) {
    x = x*10;                   /* Pasamos un decimal a la parte entera... */
    ltoa((long int) floor(x),inter,10); /* ...lo pasamos a carácter, y  */
    cadena[posicion] = inter[0];        /* lo introducimos en la cadena */
    posicion++;
    i++;
    x = x-floor(x);
  }

  if (posicion < MAX-1)
     inicializar (cadena,posicion,MAX);
  else {
     copia (msg,
     "***** Hay más dígitos en la parte DECIMAL que se han PERDIDO *****");
     cadena[MAX-1] = '\0';
  }
}
```

```
/**************************************************/
/* Objetivo: Copia c2 en c1 y rellena c1 con blancos.*/
/* Entrada : La cadena c2.                        */
/* Salida  : Cadena c1 con los primeros caracteres */
/*           como c2 y el resto a espacios.       */
/**************************************************/
void copia (char c1[], char c2[])
{ int i, tama = strlen(c2);

  strcpy (c1,c2);
  for (i=tama; i<MAXMSG; i++)
      c1[i] = ' ';
  c1[MAXMSG-1] = '\0';
}
```

Con el programa anterior se cumplen los objetivos marcados inicialmente. No obstante, como cualquier programa, es susceptible de ser mejorado y ampliado. Lo que pretendíamos es dar una visión de un modo de resolución de problemas característico, y no era nuestro objetivo hacer una calculadora con muchas operaciones, que nos permitiera mucha precisión y que fuera muy vistosa gráficamente (aunque se ha cuidado la presentación del texto en pantalla).

El método empleado es el conocido como Autómata de Estados. Nuestro autómata cuenta con 8 estados. Inicialmente estamos en un estado (1) y se produce una transición a otro estado cuando se produce un evento determinado. En nuestro caso, cada vez que se pulsa una tecla en el teclado, se realiza alguna acción y se va a un estado determinado. Cada estado se preocupará de una parte del problema. De los 8 estados que tenemos, los numerados del 1 al 4 son los más importantes. Explicaremos a continuación su cometido:

- **Estado 1:** Es el estado inicial de nuestro sistema. En él se encuentra al empezar el programa, y se encarga de tomar el primer dígito del primer operando (operando 1). Si el primer operando es negativo se debe preceder del signo `'-'`. Podemos introducir (pulsar) uno de los siguientes caracteres (si pulsamos cualquier otro estando en este estado, será ignorado) y según el que sea, se producirán las siguientes acciones:
  - El signo menos: pone el signo al operando 1 en la primera posición y seguiremos estando en el estado 1.
  - Tecla de Escape (`ESC`) o de retroceso (borrado, *del* o *backspace*): Se anula totalmente el operando 1 y se vuelve a empezar.
  - Un dígito o un punto decimal: Se introduce el dígito en el operando 1 y se pasa al estado 2.

- **Estado 2:** En este estado se termina de aceptar el primer operando. Como ya tenemos al menos un dígito del operando 1 (tomado en el estado 1), ya podemos aceptar un operador. Los caracteres aceptados en este estado son:
  - Tecla de Escape (`ESC`): Anula el operando 1 y vuelve al estado 1.

  - Tecla de retroceso: Se anula el último dígito tecleado en el operando 1. Esta función no la tienen la mayoría de las calculadoras convencionales, y es bastante útil.
  - Un dígito o un punto decimal: Se almacena en el operando 1. Naturalmente, tendremos que controlar si ya se ha introducido un punto decimal, para no poder poner más de uno. Eso se hace con la variable `haypunto1`. También controlamos que no se puedan introducir más dígitos que los que acepte el operando 1.
  - Un operador binario: Una vez que se ha pulsado un operador quiere decir que el operando 1 ya está completado. Guardamos el operador y vamos al estado 3.
  - Un operador unario: Si tenemos el operando 1 y un operador unario, lo que hay que hacer es aplicar dicho operador. El resultado se guardará en el operando 1 y se pasa al estado 3.

- **Estado 3:** En este estado se puede aceptar un operador y tomar el primer dígito del segundo operando (operando 2), igual que en el estado 1 lo aceptábamos del operando 1. Los caracteres aceptados en este estado son:

  - Tecla de Escape (`ESC`): Anula el operando 1 y vuelve al estado 1.
  - Un dígito o un punto decimal: Se introduce el dígito en el operando 2 y se pasa al estado 4.
  - Un operador binario: Se acepta dicho operador, cambiando el operador anterior.
  - Un operador unario: Se aplica dicho operador al operando 1. El resultado se guardará en el operando 1 y se pasa al estado 3.

- **Estado 4:** En este estado se aceptan el resto de dígitos del operando 2. Los caracteres aceptados en este estado son:

  - Tecla de Escape (`ESC`): Anula el operando 2 y vuelve al estado 3, donde podremos cambiar el operando 2 y hasta el operador.
  - Un dígito o un punto decimal: Se introduce el dígito en el operando 2.
  - Tecla de retroceso: Anula el último dígito tecleado del operando 2 (si se borran todos los dígitos, iremos al estado 3.
  - Un operador binario: Se opera ambos operandos con el operador que tuviéramos antes, el resultado se almacena en el operando 1, y establece el operador nuevo pulsado como actual. Se vuelve al estado 3 donde podremos empezar a aceptar el nuevo operando 2.
  - Tecla `RETURN` o `ENTER`: En este caso operamos igual que antes pero no cambiamos el operador actual. Así, por ejemplo, si necesitamos efectuar muchas sumas sucesivas podemos teclear tras cada número, esta tecla o la tecla de suma.

- **Estados 10, 20, 30 y 40:** A estos estados sólo se va si se pulsa la tecla de salida (`'S'`). En dichos estado confirmamos si realmente se quiere salir. Si se vuelve a pulsar la tecla de salida, nos salimos del programa, y si no, volveremos al estado en el que nos encontrábamos. Para saber ese estado basta dividir por 10 el estado actual.

En nuestra calculadora, el operador activado se visualiza en pantalla cosa que no ocurre en la mayoría de las calculadoras convencionales. No se permiten números muy grandes y no permite mucha precisión por problemas de redondeo inherentes de estas máquinas digitales. Para conseguir mayor precisión y poder manejar números tan grandes como queramos, tendríamos que haber empleado estructuras de datos dinámicas, que son más complejas (listas...) y que excedía del objetivo inicial de este ejercicio.

Hemos usado las funciones `cprintf()`, `cputs()` y `putch()` en vez de `printf()`, `puts()` y `putchar()` porque estas escriben directamente en pantalla y no en la salida estándar. De esta forma el programa seguirá funcionando incluso si accidentalmente se redirige su salida estándar.

En las funciones `operar()` y `operar_unario()` es muy posible que el compilador nos avise de un posible error (un *warning*), al no devolver ningún valor. Sin embargo, la función sí que devuelve un valor. La explicación es que al estar todos los `return` en lugares de ejecución condicional, el compilador no es capaz de deducir que en nuestro algoritmo siempre se ejecutara una de estas partes condicionales.

9.3 Intérprete interactivo de una Máquina de Pila (calculadora).

Codificar un programa en Lenguaje C que actúe como un pequeño intérprete interactivo de una Máquina de Pila para poder efectuar cálculos matemáticos.

Una **pila** es una estructura de datos en la que se almacenan y recuperan elementos de uno en uno, de manera que el único elemento que podemos recuperar es el último elemento que se almacenó y al recuperar un elemento este se elimina de la pila (se desapila).

En síntesis es como una *pila* de platos colocados uno sobre otro. Al colocar un plato (almacenar un elemento) lo situaremos en la cima de la pila (arriba del todo) y al retirar un plato (recuperar un elemento) se recupera el situado en la cima de la pila (que fue el último en colocarse).

Una pila tiene, por tanto, dos operaciones básicas: Apilar y desapilar. En la primera se situa un elemento en la cima de la pila y en la segunda se retira el elemento de la cima de la pila viendo así su valor.

Nuestra estructura de datos pila puede ser un simple array de números reales (de tipo `float`) y un entero (puntero de pila, SP, *stack pointer*) que indique cual es el elemento de la cima de la pila. Inicialmente la pila estará vacía y el puntero de pila valdrá -1. Cuando introduzcamos un número en la pila (apilar), el puntero de pila lo incrementamos en una unidad pasando a valer 0 y en esa posición del array asignaremos el número que deseamos introducir en la pila. Así, si el puntero de pila vale n indicará que en esa posición del array está el último elemento de la pila y que por tanto hay n+1 elementos apilados (desde la posición 0 del array hasta la n).

Nuestra Máquina de Pila servirá para efectuar operaciones matemáticas como una calculadora convencional con memoria. La principal ventaja es que podremos almacenar en la memoria tantos elementos como el tamaño máximo de la pila. Podemos establecer un tamaño máximo para la pila (de 100 elementos por ejemplo).

El usuario irá tecleando las operaciones que desee efectuar sobre la pila y el programa irá ejecutándolas de forma interactiva. Las operaciones que se pueden realizar sobre los elementos (números reales) de la pila son de 3 tipos:

1. Operaciones con la pila:

   `SCN` Esta instrucción leerá un dato de la entrada estándar (teclado) y lo introducirá en la cima de la pila. Como inicialmente la pila estará vacía, esta instrucción será siempre la primera instrucción que deberá ejecutar el usuario.

   `PTR` Muestra el dato situado en la cima de la pila, pero no lo desapila. Esto es, el dato seguirá estando en su posición.

   `DEL` Elimina el dato situado en la cima de la pila (lo desapila).

   `NUM` Muestra el número de elementos que tiene la pila en ese momento.

   `VER` o `SEE` Mostrará el estado global de la pila. Esto es, mostrará cada uno de los elementos apilados y su posición respectiva dentro de la pila.

   `HLP` Mostrará una pequeña ayuda del programa con un resumen de los comandos de los que se disponen.

   `END` Termina la ejecución del programa y vuelve al Sistema Operativo.

2. Operaciones UNARIAS con el elemento en la cima de la pila:

   `INV` Calcula el inverso del elemento situado en la cima de la pila. El proceso es simple: Primero desapilamos un elemento `n` (el de la cima de la pila), luego calculamos su inverso `1/n`, y por último, introducimos en la pila este resultado. El efecto es como si sustituyéramos el elemento situado en la cima de la pila por su inverso.

   `OPS` Calcula el opuesto del elemento situado en la cima de la pila (`-n`). Funciona de similar manera que el operador anterior.

   `EXP` Calcula la exponencial del elemento situado en la cima de la pila (e^{n}).

   `SQR` Calcula la raíz cuadrada ($\sqrt{n}$).

   `FAC` Calcula el factorial de la parte entera del elemento situado en la cima de la pila. Téngase en cuenta que sólo existe el factorial de los números naturales (enteros positivos).

   `DUP` Duplica el elemento situado en la cima de la pila. Esto es, desapila un elemento de la pila y lo vuelve a apilar 2 veces.

3. Operaciones BINARIAS con los dos elementos de la cima de la pila:

   `ADD` Calcula la suma de los 2 elementos situados en la cima de la pila. Esto es, desapila dos elementos y apila la suma de ambos. Es como sustituir ambos elementos por su suma. Naturalmente, tras esta operación el número de elementos de la pila habrá disminuido en una unidad.

   `SUB` Calcula la resta de los 2 elementos de la cima de la pila. El primero en desapilar menos el siguiente en desapilar.

   `MUL` Calcula la multiplicación de los 2 elementos situados en la cima de la pila.

   `DIV` Divide el elemento que primero desapilamos por el segundo elemento en desapilar.

   `POW` Calcula la potencia: El primer elemento en desapilar elevado al segundo elemento en desapilar.

	SCN	DUP	FAC	ADD
		3	6	
	3	3	3	9

Figura 9.2: Ejemplo de cómo afectan diversos comandos a la pila de la Máquina de Pila.

Por ejemplo, en la figura 9.2 podemos ver como se apilan y desapilan números en cada operación. Inicialmente suponemos que la pila está vacía (pila más a la izquierda). Siguiendo con el ejemplo, si el usuario ejecuta la instrucción `SCN` el programa el pedirá que número desea introducir en la pila. En el ejemplo suponemos que introduce el número 3. Después, si ejecuta el comando `DUP` se duplicará este elemento, obteniendo en la pila dos elementos iguales. Al ejecutar un comando unario como `FAC` se desapila el 3 de la cima, se le halla el factorial (6) y se apila el resultado. Si aplicamos un operador binario comoo `ADD` se desapilarán dos elementos (6 y 3), y se apilará la suma de ambos (9). A continuación podemos ejecutar el comando `PTR` para ver el resultado.

Mas ejemplos:

- Teorema de pitágoras: Para aplicar este famoso teorema habría que ejecutar las siguientes instrucciones en ese orden:

    1. `SCN` (Captura el primer cateto y lo mete en la pila)
    2. `DUP`
    3. `MUL` (Calcula el cuadrado del primer cateto)
    4. `SCN` (Captura el segundo cateto)
    5. `DUP`
    6. `MUL` (Calcula el cuadrado del segundo cateto)
    7. `ADD` (Suma ambos cuadrados)
    8. `SQR` (Raiz cuadrada de la suma: Hipotenusa)
    9. `PTR` (Muestra el resultado)
    10. `END` (Termina la ejecución)

- Area de un círculo: Para aplicar la fórmula πr^2 se le debe dar al principio la constante π (con los decimales que se precisen) y el radio del círculo:

    1. `SCN` (Captura el valor de π: 3.14159265359)
    2. `SCN` (Captura el radio r)
    3. `DUP`
    4. `MUL` (Calcula el cuadrado del radio)
    5. `MUL` (Multiplica por π)
    6. `PTR` (Muestra el resultado)

7. `END` (Termina la ejecución)

- Perímetro de una circunferencia ($2\pi\mathtt{r}$):

  1. `SCN` (Captura el valor de π: 3.14159265359)
  2. `SCN` (Captura el radio `r`)
  3. `SCN` (Captura el valor de la constante 2)
  4. `MUL`
  5. `MUL`
  6. `PTR` (Muestra el resultado)
  7. `END` (Termina la ejecución)

- Volumen de una esfera ($\frac{4}{3}\pi\mathtt{r}^3$):

  1. `SCN` (Captura el valor de π: 3.14159265359)
  2. `SCN` (Captura el valor de la constante 3)
  3. `SCN` (Captura el valor de la constante 4)
  4. `DIV`
  5. `SCN` (Captura el radio `r`)
  6. `DUP`
  7. `DUP`
  8. `MUL`
  9. `MUL`
  10. `MUL`
  11. `MUL`
  12. `PTR` (Muestra el resultado)
  13. `END` (Termina la ejecución)

- Area de un triángulo: La fórmula es $\frac{\mathtt{bh}}{2}$, donde `b` es la longitud de la base y `h` es la altura del triángulo:

  1. `SCN` (Captura el valor de la constante 2)
  2. `SCN` (Captura el radio `b`)
  3. `SCN` (Captura el radio `h`)
  4. `MUL`
  5. `DIV`
  6. `PTR` (Muestra el resultado)
  7. `END` (Termina la ejecución)

Durante la utilización de nuestra peculiar calculadora se pueden producir algunos errores que debemos tener en cuenta y dar un aviso indicándo lo sucedido. Entre ellos, los más importantes son:

- Si el usuario teclea una instrucción que no existe: Deberá indicársele que existe la instrucción HLP para obtener una breve ayuda.
- Si intentamos ejecutar un operador unario y la pila está vacía.
- Si intentamos ejecutar un operador binario y la pila tiene menos de 2 elementos.
- División por cero.
- Raiz cuadrada de un número negativo.
- Factorial de un número negativo. El factorial de un número no entero (con decimales) también podría dar error, pero se ha supuesto que siempre se desea hacer el factorial de la parte entera del número en cuestión.

Solución:

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

#define MAX_ELEMENTS 99 /* Máximo número de elementos en la pila menos 1 */

/* Cabeceras de las funciones de operaciones */
void Suma(void);          /*ADD*/
void Resta(void);         /*SUB*/
void Multiplica(void);    /*MUL*/
void Divide(void);        /*DIV*/
void Potencia(void);      /*POW*/
void Opuesto(void);       /*OPS*/
void Raiz(void);          /*SQR*/
void Invierte(void);      /*INV*/
void Exponencial(void);   /*EXP*/
void Factorial(void);     /*FAC*/
void Help(void);          /*HLP*/

/* Definición de la estructura de datos: PILA. */
float pila[MAX_ELEMENTS + 1]; /* Pila de datos */
int cima = -1;                /* Cima de la pila: Inicialmente vacia (-1) */

/* Cabeceras (prototipos) de las PRIMITIVAS de la estructura de datos PILA */
void apila (float);
float desapila(void);
unsigned pila_vacia(void);
unsigned pila_llena(void);
unsigned num_elementos(void);
void Ver_Pila(void);

/*********************************************************/
```

```
/*                    FUNCION  PRINCIPAL                 */
/* Objetivo: Intérprete de un lenguaje para una máquina */
/*           de PILA.                                   */
/*******************************************************/
void main()
{ char instruccion[20]=""; /* Instrucción dada por el usuario. */
  float dato;

  puts("\n-- Intérprete de una máquina de PILA --");
  puts("--    Pulse HLP para ayuda (HeLP)    --\n");

  while (stricmp(instruccion,"END")) { /* Mientras no sea END... */
    printf("* Instrucción ? ");
    scanf("%s", instruccion); /* Lee de teclado la instrucción a ejecutar */
    instruccion[3] = '\0';    /* Tomamos sólo los 3 primeros caracteres   */

    if (!stricmp(instruccion,"PTR")) /* Escribe la cima sin desapilarlo */
       if (pila_vacia())
          puts("- ERROR: Pila vacía.");
       else {
          printf("- Cima de la Pila: %f\n", dato=desapila());
          apila(dato);
       }

    else if (!stricmp(instruccion,"SCN")) /* Apilar nuevo dato */
          if (pila_llena())
             puts("- ERROR: Pila llena (no apilar más datos).");
          else {
             printf("- Elemento a apilar: ");
             scanf("%f",&dato); /* Lee un dato de la Entrada Estándar */
             apila(dato);       /* ... y lo introduce en la pila      */
          }

    else if (!stricmp(instruccion,"DUP")) /* Duplicar cima de la pila */
          if (pila_llena())
             puts("- ERROR: Pila llena (no apilar más datos).");
          else {
             dato = desapila(); /* Desapilamos el dato y... */
             apila(dato);       /* lo apilamos 2 veces      */
             apila(dato);
          }

    else if (!stricmp(instruccion,"NUM")) /* Número de elementos apilados */
          printf("- Número de elementos apilados: %u\n",num_elementos());

    else if (!stricmp(instruccion,"DEL")) /* Borra cima de la pila */
          if (pila_vacia())
```

```
            puts("- ERROR: Pila VACIA.");
          else desapila();

    else if (!stricmp(instruccion,"ADD")) /* Sumar */
          Suma();
    else if (!stricmp(instruccion,"SUB")) /* Restar */
          Resta();
    else if (!stricmp(instruccion,"MUL")) /* Multiplicar */
          Multiplica();
    else if (!stricmp(instruccion,"DIV")) /* Dividir */
          Divide();
    else if (!stricmp(instruccion,"POW")) /* Potencia */
          Potencia();
    else if (!stricmp(instruccion,"OPS")) /* Opuesto (-n) */
          Opuesto();
    else if (!stricmp(instruccion,"SQR")) /* Raiz Cuadrada */
          Raiz();
    else if (!stricmp(instruccion,"INV")) /* Inverso (1/n) */
          Invierte();
    else if (!stricmp(instruccion,"EXP")) /* Exponencial (e^n) */
          Exponencial();
    else if (!stricmp(instruccion,"FAC")) /* Factorial (n!) */
          Factorial();
    else if (!stricmp(instruccion,"VER") || !stricmp(instruccion,"SEE"))
          Ver_Pila();                     /* Ver el estado de la Pila   */
    else if (!stricmp(instruccion,"HLP")) /* Ayuda, resumen de comandos */
          Help();
    else
      puts("***** Comando NO existe (use HLP para Ayuda) *****");
  } /*while*/

} /*main*/

/*******************************************************/
/* Objetivo: Saca dos elementos de la pila, los suma e  */
/*           introduce en la pila el resultado.         */
/*******************************************************/
void Suma()
{ float x,y;

  if (num_elementos() < 2)
     puts("- ERROR: No hay suficientes datos ");
  else {
     x = desapila();
     y = desapila();
     apila (x+y);
  }
```

```
}

/*****************************************************/
/* Objetivo: Saca dos elementos de la pila, los resta e */
/*           introduce en la pila el resultado.        */
/*****************************************************/
void Resta()
{ float x,y;

  if (num_elementos() < 2)
     puts("- ERROR: No hay suficientes datos.");
  else {
     x = desapila();
     y = desapila();
     apila (x-y);
  }
}

/*****************************************************/
/* Objetivo: Saca 2 números de la pila, los multiplica  */
/*           e introduce en la pila el resultado.       */
/*****************************************************/
void Multiplica()
{ float x,y;

  if (num_elementos() < 2)
     puts("- ERROR: No hay suficientes datos.");
  else {
     x = desapila();
     y = desapila();
     apila (x*y);
  }
}

/*****************************************************/
/* Objetivo: Saca dos elementos de la pila, los divide  */
/*           e introduce en la pila el resultado.       */
/*****************************************************/
void Divide()
{ float x,y;

  if (num_elementos() < 2)
     puts("- ERROR: No hay suficientes datos.");
  else {
     x = desapila();
     y = desapila();
     if (y)
```

```
         apila (x/y);
      else {
         puts("- ERROR: División por cero.");
         apila(y);
         apila(x);
      }
   }
}

/*********************************************************/
/* Objetivo: Saca 2 números de la pila, eleva el primero*/
/*           al segundo e introduce el resultado.       */
/*********************************************************/
void Potencia()
{ float x,y;

  if (num_elementos() < 2)
     puts("- ERROR: No hay suficientes datos.");
  else {
     x = desapila();
     y = desapila();
     apila ((float)pow(x,y));
  }
}

/*********************************************************/
/* Objetivo: Saca 1 elemento de la pila, le cambia el    */
/*           signo e introduce el resultado.             */
/*********************************************************/
void Opuesto()
{ if (pila_vacia())
     puts("- ERROR: Pila VACIA.");
  else
     apila ((-1)*desapila());
}

/*********************************************************/
/* Objetivo: Saca 1 elemento de la pila, calcula su      */
/*           raiz cuadrada e introduce el resultado.     */
/*********************************************************/
void Raiz()
{ float x;

  if (pila_vacia())
     puts("- ERROR: Pila VACIA.");
  else {
     x=desapila();
```

```
      if (x>=0)
         apila ((float)sqrt(x)),
      else {
         puts("- ERROR: No existe la raiz de un número negativo.");
         apila(x);
      }
  }
}

/*****************************************************/
/* Objetivo: Saca 1 elemento de la pila, calcula su   */
/*           inverso e introduce el resultado.         */
/*****************************************************/
void Invierte()
{ if (pila_vacia())
      puts("- ERROR: Pila VACIA.");
  else
      apila (1/desapila());
}

/*****************************************************/
/* Objetivo: Saca 1 número de la pila, calcula su     */
/*           exponencial e introduce el resultado.     */
/*****************************************************/
void Exponencial()
{ if (pila_vacia())
      puts("- ERROR: Pila VACIA.");
  else
      apila ((float)exp((float)desapila()));
}

/*****************************************************/
/* Objetivo: Saca 1 elemento de la pila, calcula su   */
/*           factorial e introduce el resultado.       */
/*****************************************************/
void Factorial()
{ long int x;
  float resultado=1;

  if (pila_vacia())
     puts("- ERROR: Pila VACIA.");
  else {
     x= (long int) desapila();
     if (x>=0) {
        while ((x!=1) && (x!=0))
            resultado *= x--;
        apila (resultado);
```

```
    }
    else {
       puts("- ERROR: No existe el factorial de un número negativo.");
       apila(x);
    }
  }
}

/******************************************************/
/* Objetivo: Saca por la Salida Estándar un resumen de  */
/*           los comandos disponibles.                  */
/******************************************************/
void Help()
{ puts("\t\t***** AYUDA: Resumen de Comandos de la PILA *****\n");
  puts("\tEste es un pequeño e interactivo intérprete para una máquina de");
  puts("PILA. El objetivo es manejar los elementos de la PILA con opera-");
  puts("ciones matemáticas (similar a una calculadora con memoria).");
  puts("\tLas operaciones siempre actúan sacando los elementos situados");
  puts("en la cima de la PILA, operándolos e introduciendo el resultado.");
  puts("\t* Operaciones BINARIAS:");
  puts("\t\tADD\tSuma.\t\t\tSUB\tResta.");
  puts("\t\tMUL\tMultiplica.\t\tDIV\tDivide.");
  puts("\t\tPOW\tPotencia.");
  puts("\t* Operaciones UNARIAS:");
  puts("\t\tINV\tInverso (1/n).\t\tSQR\tRaiz Cuadrada.");
  puts("\t\tOPS\tOpuesto (-n).\t\tFAC\tFactorial parte entera.");
  puts("\t\tEXP\tExponencial.\t\tDUP\tDuplica la cima.");
  puts("\t* Operaciones con la PILA:");
  puts("\t\tPTR\tMuestra el dato de la cima, sin desapilarlo.");
  puts("\t\tDEL\tElimina el dato de la cima (lo desapila).");
  puts("\t\tSCN\tCaptura un dato de teclado y lo apila.");
  puts("\t\tVER\tMuestra el estado global de la Pila (también SEE).");
  puts("\t\tNUM\tMuestra el número de elementos que tiene la pila.");
  puts("\t\tHLP\tMuestra esta pantalla de ayuda.");
  puts("\t\tEND\tTermina la ejecución del programa."),
}

/******************************************************/
/*   P R I M I T I V A S   para manejo de la   P I L A */
/******************************************************/

/******************************************************/
/* Objetivo: Apila el dato de su único argumento, colo- */
/*           cándolo en la cima de la pila.             */
/* Entrada : Dato a apilar, de tipo float               */
/* Requisitos: Se supone que la pila no está llena.     */
/******************************************************/
```

```
void apila (float dato)
{ pila[++cima] = dato;
}

/*******************************************************/
/* Objetivo: Desapila un dato de la pila.             */
/* Salida  : Devuelve el dato situado en la cima.     */
/* Requisitos: Se supone que la pila no está vacía.   */
/*******************************************************/
float desapila()
{ return pila[cima--];
}

/*******************************************************/
/* Objetivo: Ver si la pila está o no vacía.          */
/* Salida  : Devuelve: 1 (true) si está vacía y       */
/*                     0 (false) si no está vacía.    */
/*******************************************************/
unsigned pila_vacia()
{ return (cima == -1);
}

/*******************************************************/
/* Objetivo: Ver si la pila está o no llena.          */
/* Salida  : Devuelve: 1 (true) si está llena y       */
/*                     0 (false) si no está llena.    */
/*******************************************************/
unsigned pila_llena()
{ return (cima == MAX_ELEMENTS);
}

/*******************************************************/
/* Objetivo: Calcular el número de elementos de la pila */
/* Salida  : Devuelve el número de elementos.         */
/*******************************************************/
unsigned num_elementos()
{ return cima+1;
}

/*******************************************************/
/* Objetivo: Ver el estado global de la pila.         */
/* Salida  : Imprime en la Salida Estándar un listado */
/*           con todos los elementos de la pila, desde */
/*           la cima de la pila hasta el último.      */
/*******************************************************/
void Ver_Pila()
{ int i;
```

```
  if (pila_vacia())
     puts("- Pila VACIA.");
  else {
     puts("  Posicion  Elemento");
     puts(" --------------------");
     for (i=cima;i>=0;i--)
         printf("%7i  %11.5f\n",i+1,pila[i]);
  }
}
```

Una consideración muy importante a la hora de implementar un programa que utilize tipos de datos abstractos (pilas, colas, árboles, grafos...) es separar claramente la implementación de las funciones que accedan a estas estructuras. Estas funciones se llaman primitivas, y un programa debe acceder a estas estructuras *sólo y exclusivamente a través de sus primitivas*.

De esta forma, en cualquier momento podremos cambiar la representación de la pila y sólo tendremos que modificar las funciones primitivas. El resto del programa seguirá funcionando igual de bien.

Así, para nuestra pila hemos considerado las siguientes primitivas: `apila(dato)` que introduce el `dato` en la pila, `desapila()` que desapila un dato y lo devuelve, `pila_vacia()` que será verdad si la pila está vacía, `pila_llena()` que será verdad si la pila está completamente llena, `num_elementos()` que devuelve el número de elementos apilados en la pila y `Ver_Pila()` que muestra uno a uno todos los elementos de la pila y sus posiciones.

Obsérvese que se han considerado como primitivas todas las funciones que acceden a la representación de la pila, es decir, que acceden directamente al array de la pila y al puntero de pila (`cima`). El programa principal (`main`) *y cualquier otra función del programa no deben* NUNCA acceder a la representación de la pila. Las primitivas deben ser la única forma de comunicar el programa con el tipo de dato abstracto (pila, cola, árbol, grafo...).

Desde un punto de vista estricto, la función `Ver_Pila()` no debería ser una primitiva de una pila, pues en una pila sólo podemos ver el elemento de la cima y no el resto, pero se ha incluido aquí para facilitar el manejo de la Máquina de Pila.

Hemos situado la declaración de la pila como variable global para simplificar el paso de argumentos. En programas más complejos, la definición del tipo pila y sus primitivas deben declararse y definirse en un módulo aparte y en cada primitiva se debe pasar un argumento que indique a qué pila nos referimos. Téngase en cuenta que un programa puede necesitar manejar varias pilas y debe hacerlo con las mismas primitivas para cada una. No sería lógico tener que *implementar unas primitivas distintas para cada pila que necesitemos en un programa*.

Para simplificar no hemos tenido en cuenta la posibilidad de que se produzca un error por desbordamiento (*overflow*) al efectuar una operación cuyo resultado sea excesivamente grande.

Además de estas operaciones se pueden incluir otras que expandan la utilidad de nuestra Máquina de Pila, como son: Parte entera de la cima de la pila, operaciones trigonométricas (seno, coseno, tangente...), operaciones de cambio de base (binaria, octal, hexadecimal...), logaritmos en distintas bases (neperiano, decimal...), resolución de ecuaciones del tipo $ax^2 + bx + c = 0$ suponiendo que los coeficientes `a`, `b` y `c` están situados juntos y en la cima de la pila y muchas operaciones más.

9.4 Una simple Agenda.

Hacer un programa que nos permita mantener en un fichero una Agenda con los datos de nuestros amigos. De cada amigo mantendremos información sobre su nombre, dirección y teléfono. Además, podremos almacenar unas notas para cada uno, para poder guardar otros datos de interés. También permitirá almacenar un entero por cada amigo, como un código numérico para cualquier utilidad.

Nuestro programa permitirá, como mínimo, las siguientes operaciones:

1. Insertar datos en la Agenda: Crea el fichero de agenda, si no existe uno.
2. Leer datos secuencialmente: Desde el primero hasta el último.
3. Buscar por nombre: Buscar los datos de un amigo sabiendo el nombre.

Se deja como ejercicio incluirle más operaciones a nuestra maravillosa agenda. Como por ejemplo: Modificar datos del fichero, Borrar datos, Poder usar distintos nombres de fichero para distintas agendas, Ordenar datos por algún campo, en otro fichero, Búsquedas por una palabra del nombre o de la dirección, Búsquedas sin tener en cuenta mayúsculas/minúsculas ni acentos, incluir una ayuda...

Solución:

```
/*********************************************************/
/* Objetivo: Programa Agenda, para almacenar datos de      */
/*           personas (nombre, dirección y teléfono).      */
/*           Además permitirá almacenar unas notas para    */
/*           cada persona y un número (código).            */
/*********************************************************/
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define MAXCADENA 50
#define MAXTLFNO  20
#define ESC       '\x1B'

void menu (void);
void Insertar_datos (void);
void Leer_datos (void);
void Buscar_datos (void);

typedef struct { char nombre [MAXCADENA],
                      direccion [MAXCADENA],
                      tlfno [MAXTLFNO],
                      notas [MAXCADENA];
```

```
                int  codigo;
               } amigo;

char mensaje[60],
     blancos[] = "                                                   ";

void main()
{ char opcion;

  do { menu ();
       gotoxy (32,14);
       opcion = toupper(getch());

       switch (opcion) {
         case '1':
         case 'I': Insertar_datos ();
                   break;
         case '2':
         case 'L': Leer_datos ();
                   break;

         case '3':
         case 'B': Buscar_datos ();
                   break;
         case '4':
         case 'S'
         case ESC: gotoxy(10,14);
                   printf (" --> Pulse [4], [S] ó [ESC] para Salir: ");
                   opcion = toupper(getch());
       }

  } while (opcion != 'S' && opcion != '4' && opcion != ESC);

  clrscr();
  puts ("***** ***** FIN programa AGENDA ***** *****");
}

/************************************************************/
/* Objetivo: Mostrar el menú principal de la Agenda.        */
/************************************************************/
void menu (void)
{ clrscr();

  puts ("                    Menú de AGENDA");
  puts ("                   ----------------\n");
  puts ("\n          1. Insertar datos en la Agenda.");
  puts ("\n          2. Leer datos secuencialmente.");
```

```
  puts ("\n          3. Buscar por nombre.");
  puts ("\n          4. Salir.");

  printf ("\n\n          * Pulsa una opción: ");
  printf ("\n\n          %s", mensaje);
}

/*************************************************************/
/* Objetivo: Insertar datos en el fichero de Agenda.        */
/* Entrada : Leerá los datos de teclado.                     */
/* Salida  : Los escribirá en un fichero.                    */
/* Observaciones: Esta función abre el fichero.              */
/*************************************************************/
void Insertar_datos (void)
{ amigo colega;
  char sino,
       codigo[7];
  int i;
  FILE *fich;

  for (i=16; i<=20; i++) {          /* Borrar con espacios */
      gotoxy (16,i); printf ("%s", blancos);
  }

  gotoxy (1,16); printf ("  - NOMBRE   : "); gets (colega.nombre);
  gotoxy (1,17); printf ("  - DIRECCIÓN: "); gets (colega.direccion);
  gotoxy (1,18); printf ("  - TELÉFONO : "); gets (colega.tlfno);
  gotoxy (1,19); printf ("  - NOTAS    : "); gets (colega.notas);
  gotoxy (1,20); printf ("  - Código   : "); gets (codigo),
  colega.codigo = atoi (codigo);
  gotoxy (1,20); printf ("  - Código   : %i                          \
                              ", colega.codigo);

  gotoxy (1,22);printf ("  * Grabar estos datos (S/N): ");
  do
    sino = toupper(getch());
  while (sino!='S' && sino!='N');

  if (sino=='N') {
     strcpy (mensaje, "***** Datos NO grabados *****");
     return;
  }

  if ((fich = fopen ("agenda.dat","a")) == NULL) {
     strcpy (mensaje, "***** Imposible abrir fichero de datos *****");
     return;
  }
```

```
  fprintf (fich, "%s\n%s\n%s\n%i%s\n", colega.nombre,
                          colega.direccion,
                          colega.tlfno,
                          colega.codigo,
                          colega.notas);
  fclose (fich);
  strcpy (mensaje, "***** Datos grabados *****");
}

/************************************************************/
/* Objetivo: Leer datos del fichero de Agenda.              */
/* Entrada : Leerá los datos del fichero.                   */
/* Salida  : Los escribirá en pantalla.                     */
/* Observaciones: Esta función abre el fichero              */
/************************************************************/
void Leer_datos (void)
{ amigo colega;
  int numero = 0, i;
  char masmenos  = 0;
  FILE *fich;

  if ((fich = fopen ("agenda.dat","r")) == NULL) {
     strcpy (mensaje, "***** Fichero no existe o no encontrado *****");
     return;
  }

  fgets (colega.nombre, MAXCADENA, fich);

  while (!feof(fich) && masmenos!=ESC) {
    fgets (colega.direccion, MAXCADENA, fich);
    fgets (colega.tlfno, MAXTLFNO, fich);
    fscanf (fich, "%i", &colega.codigo);
    fgets (colega.notas, MAXCADENA, fich);
    numero++;

    for (i=16; i<=20; i++) {
     gotoxy (16,i); printf ("%s", blancos);
    }

    gotoxy (1,16); printf ("  - NOMBRE    : %s", colega.nombre);
    gotoxy (1,17); printf ("  - DIRECCIÓN: %s", colega.direccion);
    gotoxy (1,18); printf ("  - TELÉFONO : %s", colega.tlfno);
    gotoxy (1,19); printf ("  - NOTAS     : %s", colega.notas);
    gotoxy (1,20); printf ("  - Código    : %i         - Número: %i",
               colega.codigo, numero);
```

```
    gotoxy (1,22);
    printf ("  [INTRO] Leer siguiente.  [ESC]  Volver al menu principal.");

    do
      masmenos = getch();
    while (masmenos!='\r' && masmenos!='+' && masmenos!=ESC);

    if (masmenos!=ESC)
       fgets (colega.nombre, MAXCADENA, fich);

  } /* while */

  if (feof(fich)) {
     gotoxy (1,22);
     printf ("  *****   NO HAY MAS REGISTROS  (Total = %i)   *****        ",
            numero);
     getch();
  }

  fclose (fich);
  strcpy (mensaje, " ");
}

/*************************************************************/
/* Objetivo: Busca datos del fichero de Agenda.             */
/* Entrada : El nombre de la persona a buscar.              */
/* Salida  : Escribirá los datos de dicha persona.          */
/* Observaciones: Esta función abre el fichero.             */
/*************************************************************/
void Buscar_datos (void)
{ amigo colega;
  int numero = 0, i,
      encontrados = 0;
  char masmenos  = 0,
       nombre [MAXCADENA];
  FILE *fich;


  if ((fich = fopen ("agenda.dat","r")) == NULL) {
     strcpy (mensaje, "***** Fichero no existe o no encontrado *****");
     return;
  }

  fgets (colega.nombre, MAXCADENA, fich);

  gotoxy (1,16); printf ("%s", blancos);
  flushall();
```

```
gotoxy (1,16); printf ("  - Nombre a buscar: "); gets (nombre);

i = strlen (nombre); /* Ponemos el formato de los datos leídos de fich */
nombre [i]   = '\n';
nombre [i+1] = '\0';

while (!feof(fich) && masmenos!=ESC) {
  fgets (colega.direccion, MAXCADENA, fich);
  fgets (colega.tlfno, MAXTLFNO, fich);
  fscanf (fich, "%i", &colega.codigo);
  fgets (colega.notas, MAXCADENA, fich);
  numero++;

  if (!strcmp(colega.nombre, nombre)) {
     encontrados++;

     for (i=16; i<=20; i++) {
      gotoxy (16,i); printf ("%s", blancos);
     }

     gotoxy (1,16); printf ("  - NOMBRE   : %s", colega.nombre);
     gotoxy (1,17); printf ("  - DIRECCIÓN: %s", colega.direccion);
     gotoxy (1,18); printf ("  - TELÉFONO : %s", colega.tlfno);
     gotoxy (1,19); printf ("  - NOTAS    : %s", colega.notas);
     gotoxy (1,20); printf ("  - Código   : %i         - Número: %i",
                  colega.codigo, numero);
     printf("      - Encontrado no.: %i", encontrados);

     gotoxy (1,22); printf
     ("  [INTRO] Leer siguiente.  [ESC]  Volver al menu principal.");

     do
    masmenos = getch();
     while (masmenos!='\r' && masmenos!='+' && masmenos!=ESC);
  }

  if (masmenos!=ESC)
     fgets (colega.nombre, MAXCADENA, fich);

} /* while */

if (feof(fich)) {
   gotoxy (1,22); printf
   ("  *****   TOTAL ENCONTRADOS: %i de %i   *****                 ",
   encontrados, numero);
   getch();
}
```

```
  fclose (fich);
  strcpy (mensaje, " ");
}
```

Apéndice A

Tablas de consulta para programadores en C.

En este apéndice pretendemos dar un elenco de tablas de consulta para que, sobre todo los programadores principiantes en lenguaje C, puedan consultar algunas de las características más básicas e importantes de este potente lenguaje.

A.1 Tipos de datos del estándar ANSI C.

Estos son los tipos de datos básicos del estándar ANSI C. Naturalmente, esto puede variar de una máquina a otra o incluso de un compilador a otro, pero creemos que los cambios nunca serán sustanciales.

Las longitudes (en bits) y los rangos de cada tipo de dato asumen palabras de 16 bits y bytes de 8 bits. Igualmente, se supone el tipo de representación interna más habitual, la representación en complemento a dos (de los números negativos).

Tipo	N° bits	Rango
`char`	8	Caracteres ASCII (similar a `unsigned char`)
`unsigned char`	8	0 a 255
`signed char`	8	–128 a 127
`int`	16	-32768 a 32767
`unsigned int`	16	0 a 65535
`signed int`	16	Igual que `int`
`short int`	8	–128 a 127
`unsigned short int`	8	0 a 255
`signed short int`	8	Igual que `short int`
`long int`	32	–2147483648 a 2147483649
`unsigned long int`	32	0 a 42949667296
`signed long int`	32	Igual que `long int`
`float`	32	6 dígitos de precisión aproximadamente
`double`	64	12 dígitos de precisión aproximadamente
`long double`	128	24 dígitos de precisión aproximadamente.

A.2 Caracteres de escape.

Existen caracteres que son imposibles de imprimir por lo que se debe usar otra forma para representarlos. Para ellos se usa el carácter de escape barra invertida (\) seguido de una letra que indica el tipo de carácter que representa. En la siguiente tabla se muestran los códigos de estas constantes de tipo carácter que necesitan esta representación especial:

Carácter	ASCII	Acción
`\n`	10	nueva-línea (newline)
`\t`	9	tabulador horizontal (tab)
`\b`	8	retroceso (back space)
`\r`	13	retorno de carro (carriage return)
`\f`	12	salto de página (impresora)
`\\`	92	barra-atrás (back slash)
`\'`	39	apóstrofo
`\"`	34	comillas
`\a`	7	alerta (pitido)
`\0`	0	Carácter nulo. Fin de cadena de caracteres
`\o`	7	Constante octal
`\x`	7	Constante hexadecimal
`\v`	7	Tabulación vertical

A.3 Operadores aritméticos.

A continuación se muestran los operadores aritméticos que se pueden usar en C. Hay que recordar que los operadores son sensibles al tipo de dato al cual se aplican. Es decir, por ejemplo la división 5/2 dará un resultado entero ya que ambos operandos son enteros por lo que el resultado será 2. Igualmente, 5.0/2 dará el resultado real 2.5. También es válido 5./2 ó 5/2.0 o cualquier otra expresión en la que un operando sea real (usando moldes...).

Observe que el C no tiene operador de potenciación contrariamente a lo que ocurre con otros lenguajes. Para ello se debe usar una función que normalmente está ya incluida en las bibliotecas estándares del sistema: Función `pow()` de `math.h`.

Operador	Acción
-	Resta o Sustración(también el menos unario)
+	Suma o Adición
*	Multiplicación o Producto
/	División o Partición
%	Resto de una división o Módulo
--	Decremento (pre o postdecremento)
++	Incremento (pre o postdecremento)

A.4 Operadores relacionales.

Los operadores relacionales son aquellos que nos permiten establecer una relación entre dos valores comparando ambos. El valor de estas expresiones será siempre 1 (verdad) ó 0 (falso) dependiendo de la comparación establecida entre dichos valores.

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

A.5 Operadores lógicos.

Con los operadore lógicos podemos conectar expresiones con operadores relacionales u otro tipo de operadores pudiendo expresar condiciones más complejas.

<table>
<tr><th>Operador</th><th>Acción</th></tr>
<tr><td>&&</td><td>Y lógico (AND)</td></tr>
<tr><td>||</td><td>O lógico (OR)</td></tr>
<tr><td>!</td><td>NO lógico o negación (NOT)</td></tr>
</table>

A.6 Operadores a nivel de bits.

Al contrario que muchos otros lenguajes de programación, el C soporta unos operadores a nivel de bits que le dota de una potencia similar al lenguaje ensamblador. No obstante hay que ser muy cauto cuando se usan estos operadores pues pueden fácilmente confundir al programador.

<table>
<tr><th>Operador</th><th>Acción</th></tr>
<tr><td>&</td><td>Operación Y bit a bit (AND)</td></tr>
<tr><td>|</td><td>Operación O bit a bit (OR)</td></tr>
<tr><td>~</td><td>Complemento a 1 (NOT)</td></tr>
<tr><td>^</td><td>O exclusivo (XOR)</td></tr>
<tr><td>>></td><td>Desplazamiento de bits a la derecha</td></tr>
<tr><td><<</td><td>Desplazamiento de bits a la izquierda</td></tr>
</table>

A.7 Precedencia de operadores.

Cuando en una expresión aplicamos varios operadores es fundamental saber en qué orden se van a aplicar para obtener el resultado final. Este orden lo marca la precedencia que tenga cada operador implicado. A mayor precedencia antes se aplicará dicho operador.

En caso de duda o en caso de quedar la expresión algo complicada de entender es aconsejable usar paréntesis, los cuales pueden cambiar el orden de evaluación de la expresión: Lo que va entre paréntesis se evalúa primero.

<table>
<tr><td>Mayor</td><td>( ) [ ] -></td><td>Paréntesis, corchetes de array y flecha</td></tr>
<tr><td></td><td>! ~ ++ -- - (tipo) & * sizeof</td><td>Negación lógica, complemento a 1,
incremento, decremento, menos unario (signo),
moldes, dirección, indirección y tamaño</td></tr>
<tr><td></td><td>* / %</td><td>Producto, división y módulo</td></tr>
<tr><td></td><td>+ -</td><td>Suma y resta</td></tr>
<tr><td></td><td><< >></td><td>Desplazamiento a izquierda y derecha</td></tr>
<tr><td></td><td>< <= > >=</td><td>Menor, menor o igual, mayor y mayor o igual</td></tr>
<tr><td></td><td>== !=</td><td>Igual y distinto</td></tr>
<tr><td></td><td>&</td><td>Operador Y a nivel de bits</td></tr>
<tr><td></td><td>^</td><td>Operador XOR (O exclusivo) a nivel de bits</td></tr>
<tr><td></td><td>|</td><td>Operador O a nivel de bits</td></tr>
<tr><td></td><td>&&</td><td>Y lógico</td></tr>
<tr><td></td><td>||</td><td>O lógico</td></tr>
<tr><td></td><td>?</td><td>Operador condicional</td></tr>
<tr><td></td><td>= += -= *= /=</td><td>Asignación y asignación con operaciones</td></tr>
<tr><td>Menor</td><td>,</td><td>Operador coma</td></tr>
</table>

A.8 Formatos para imprimir con printf()

La función `printf()` es una de las funciones de salida más usuales. Su objetivo es imprimir algo en la salida estándar con un formato determinado (de ahí la `f` del nombre de esta función). El formato es especificado con unos códigos especiales que empiezan con el carácter %. En la siguiente tabla se presentan estos códigos, su finalidad y qué tipo de argumento requieren dentro de los argumentos de `printf()`.

%(Carácter tipo)	Tipo de Argumento	Formato de Salida
%d	Entero (int)	Entero decimal, con signo
%i	Entero	Entero decimal, con signo
%u	Entero	Entero decimal, sin signo
%o	Entero	Entero octal, sin signo
%x	Entero	Entero hexadecimal, sin signo (a, b, c, d, e, f)
%X	Entero	Entero hexadecimal, sin signo (A, B, C, D, E, F)
%f	Real (coma flotante)	Valor con signo: [-]dddd.dddd
%e	Real	Valor con signo (Notación científica) [-]d.dddde[+|-]ddd
%g	Real	Valor con signo, en forma e o f, según en el valor dado y su precisión
%E	Real	Igual que e, pero usa E como exponente
%G	Real	Igual que g, pero usa E como exponente
%c	Carácter (char)	Un sólo carácter
%s	Puntero a cadena	Imprime caracteres hasta un terminador
%%	Ninguno	Imprime el carácter '%'
%n	Puntero a int	Almacena en el entero los caracteres escritos hasta el %n
%p	Puntero	Imprime el formato de puntero (XXXX:YYYY)

A.9 Formatos para leer con scanf()

La función **scanf()** es una de las funciones de entrada o lectura más usuales. Su objetivo es leer algo en la entrada estándar con un formato determinado (de ahí la f del nombre de esta función). El formato es especificado con unos códigos especiales que empiezan con el carácter %. En la siguiente tabla se presentan estos códigos, su finalidad y qué tipo de puntero requieren dentro de los argumentos de **scanf()**

%(Carácter tipo)	Tipo de Argumento: Puntero a	Entrada esperada
%d	Entero (int)	Entero decimal
%D	Entero	Entero decimal
%o	Entero	Entero octal
%O	Entero long	Entero octal
%i	Entero	Entero decimal, octal o hexadecimal
%I	Entero long	Entero decimal, octal o hexadecimal
%u	unsigned int	Entero decimal, sin signo
%U	unsigned long	Entero decimal, sin signo
%x	Entero	Entero hexadecimal
%X	Entero long	Entero hexadecimal
%e	Real	Real (en coma flotante)
%E	Real	Real (en coma flotante)
%f	Real	Real (en coma flotante)
%c	Carácter	Un carácter
%s	*string* de caracteres	Cadena de caracteres
%P	Puntero	En forma hexadecimal (XXXX:YYYY)

A.10 Frecuencias de las notas musicales.

Para ejecutar sonidos, a veces es necesario saber las frecuencias de las distintas notas musicales. En la siguiente tabla presentamos las frecuencias de las 8 octavas más usuales.

Recordamos algo trivial para todo los músicos pero que puede no recordarse por los que no lo son: A es la nota La, B la nota Si, C la nota Do, D la nota Re, E la nota Mi, F la nota Fa y G la nota Sol. Además, el símbolo # indica sostenido que consiste en elevar un semitono la nota a la que afecta. Entre cada nota consecutiva hay un tono menos entre Mi y Fa y menos entre Si y Do que hay un semitono. Por esto Mi sostenido es igual que Fa y Si sostenido es igual que Do.

Nota	Frecuencia	Nota	Frecuencia	Nota	Frecuencia	Nota	Frecuencia
C0	16.35	C2	65.41	C4	261.63	C6	1046.50
C#0	17.32	C#2	69.30	C#4	277.18	C#6	1108.73
D0	18.35	D2	73.42	D4	293.66	D6	1174.66
D#0	19.45	D#2	77.78	D#4	311.13	D#6	1244.51
E0	20.60	E2	82.41	E4	329.63	E6	1328.51
F0	21.83	F2	87.31	F4	349.23	F6	1396.91
F#0	23.12	F#2	92.50	F#4	369.99	F#6	1482.00
G0	24.50	G2	98.00	G4	392.00	G6	1567.98
G#0	25.96	G#2	103.83	G#4	415.30	G#6	1661.22
A0	27.50	A2	110.00	A4	440.00	A6	1760.00
A#0	29.14	A#2	116.54	A#4	446.16	A#6	1864.66
B0	30.87	B2	123.47	B4	493.88	B6	1975.53
C1	32.70	C3	130.81	C5	523.55	C7	2093.00
C#1	34.65	C#3	138.59	C#5	554.37	C#7	2217.46
D1	36.71	D3	146.83	D5	587.33	D7	2349.32
D#1	38.89	D#3	155.56	D#5	622.25	D#7	2489.02
E1	41.20	E3	164.81	E5	659.26	E7	2637.02
F1	43.65	F3	174.61	F5	698.46	F7	2793.83
F#1	46.25	F#3	185.00	F#5	739.99	F#7	2959.96
G1	49.00	G3	196.00	G5	783.99	G7	3135.96
G#1	51.91	G#3	207.65	G#5	830.61	G#7	3322.44
A1	55.00	A3	220.00	A5	880.00	A7	3520.00
A#1	58.27	A#3	233.08	A#5	932.33	A#7	3729.31
B1	61.74	B3	246.94	B5	987.77	B7	3951.07
						C8	4186.01

Apéndice B

Reglas de Oro de la Programación

1. El programador debe siempre tener en cuenta que el usuario siempre se hace el inútil (a veces, además, lo es).
2. El usuario nunca se lee las ayudas *on line.*
3. El programa debe incluir ayuda *on line* para que el programador se las lea al usuario cuando este lo solicite.
4. El usuario nunca sabe lo que quiere hasta que ve lo que le dan.
5. El usuario y el programador siempre hablan distinto idioma.
6. Hacer un programa es fácil. Lo difícil es que funcione correctamente.
7. Modificar un programa es fácil. Lo difícil es modificarlo correctamente.
8. Detectar fallos en un programa es fácil. Lo difícil es detectarlos todos.
9. Para que un programa funcione correctamente, no basta que lo haga 1 vez.
10. Para que un programa funcione correctamente, no basta que lo haga 100.000 veces.
11. Primer enunciado de la Ley Universal de la Programación: Todo programa siempre es posible mejorarlo —léase también ampliarlo/corregirlo—.
12. Segundo enunciado de la Ley Universal de la Programación: Un programa no se termina nunca (y cuando se está programándolo, parece interminable).
13. Las *Normas de Estilo* deben cumplirse durante la programación, y no al terminar el programa, en la revisión final.
14. Si un programa no está bien indentado, no es programa sino jeroglífico.
15. Lo más difícil de un programa es elegir los nombres de los identificadores (constantes, variables, funciones, procedimientos y tipos de datos), para que estos sean significativos.
16. Usar identificadores no significativos ayuda a aumentar la complejidad de nuestro jeroglífico.

17. Usar constantes *a pelo* en nuestro programa (sin asociarlas a identificadores: **#define**...), ayuda a aumentar la complejidad de nuestro jeroglífico.

18. Nuestro programa debe incluir suficientes comentarios.

19.
```
if (numero_comentarios != suficiente)
   GOTO Regla_Anterior;
```

20. ¡ La sentencia GOTO no debe aparecer en un programa !

21. Un jeroglífico NO debe estar bien estructurado y modularizado. Debe contener todo en la función principal, sin que existan otras funciones (procedimientos) que puedan clarificar el programa.

22. Poner a varios programadores a trabajar sobre el mismo programa no les convierte en un equipo.

23. En un equipo de 2 programadores, siempre ocurre alguna de las siguientes situaciones:
    - Uno lo hace todo y el otro sólo pone su nombre.
    - Ambos lo hacen todo y obtienen 2 programas distintos, imposible de unificarlos.
    - Deciden separarse por problemas de entendimiento y al final el programa no se termina.

24. Ley de Brooks: Meter más gente en un proyecto de Software retrasado, lo retrasa aún más.

25. No existe relación entre el tamaño del error y los problemas que causa.

26. La documentación es el aceite de ricino de los programadores. Los jefes suponen que si tanto la odian los programadores, algo tendrá de bueno.

27. Una vez comprobado concienzudamente que todo funciona correctamente y todo está bien integrado, aún quedan al menos cuatro meses más de trabajo para que lo comprobado sea casi-cierto.

28. La generación de números aleatorios es demasiado importante para abandonarla al azar.

29. Para descubrir los fallos de un programa probado exhaustivamente, hay que dárselo a probar al usuario más novato.

30. La torpeza del usuario es directamente proporcional al número de fallos que descubre en nuestro programa.

31. Los fallos del programa atraen al usuario más intensamente si es antes de pagar al programador.

32. Si un programa no funciona, seguramente es que necesitas un ordenador unas 100.000 pts. más potente.

33. La programación sirve para solucionar problemas, no para crearlos... bueno... esta mejor la quito.

33. Aunque suene paradójico, la programación NO tiene, ni debe tener *Reglas de Oro*.

Apéndice C

Bibliografía

C.1 Programación general

1. J. Galindo G., A. Yañez E., P.J. Sánchez S., M.J. del Jesus D., J.M. Rodríguez C., J.J. Aguilera G., A. Sánchez N. y J.F. Argudo A.
   Fundamentos Informáticos.
   Servicio de Publicaciones de la Universidad de Cádiz (1996)
2. J. Galindo G., A. Yañez E., P. Sánchez S., M.J. del Jesus D., J.M. Rodríguez C., J.J. Aguilera G., A. Sánchez N. y J.F. Argudo A.
   Apuntes de Informática.
   Universidad de Cádiz.
   S. Rafael (1995).
3. Pedro L. Galindo Riaño.
   Estructuras de Datos.
   Dpto. Lenguajes y Sistemas Informáticos, Universidad de Cádiz.
   S. Rafael, S.L. (1994).
4. Luis Joyanes Aguilar.
   Fundamentos de Programación. Algoritmos y estructuras de datos.
   McGraw-Hill (1988).
5. Niklaus Wirth.
   Algoritmos + Estructuras de Datos = Programas.
   Ed. del Castillo (1980).
6. T.G. Lewis, M.Z. Smith.
   Estructuras de Datos. Programación y aplicaciones.
   Paraninfo (1985).

7 J.Biondi, G. Clavel.
Introducción a la Programación.
Masson (1988).

8. P.C. Scholl, J.P. Peyrin.
Esquemas Algorítmicos Fundamentales.
Masson (1991).

9. Milton Rosenstein.
Estructuras de Datos. Un enfoque práctico.
Anaya Multimedia (1990).

10. José Luis Balcázar.
Programación Metódica.
McGraw-Hill (1993).

11. Robert W. Sebesta.
Concepts of Programming Languages.
The Benjamin/cummings Publishing Company, Inc. (1989)

12. David Gries.
The Science of programming.
Springer-Verlag (1989).

C.2 Programación en C básica

1. J.M. Rodríguez Corral y J. Galindo Gómez.
Aprendiendo C.
Servicio de Publicaciones de la Universidad de Cádiz (1996).

2. Herbert Shildt.
C: Manual de bolsillo.
McGraw-Hill (1992).

3. Herbert Shildt.
C: Manual de referencia.
McGraw-Hill (1990).

4. Hancock, Krieger.
Introducción al Lenguaje C.
McGraw-Hill.

5. Kelley y Pohl.
   Lenguaje C. Introducción a la Programación.
   Addison-Wesley Iberoamericana (1987).

6. Augie Hansen.
   ¡Aprenda C ya!
   Microsoft Press.
   Anaya Multimedia (1990).

7 Jean-Michel Drappier y Anne Mauffrey.
   Programación Práctica del Lenguaje C.
   Ed. Técnicas Rede, S.A. (1983).

8. Philippe Dax.
   Lenguaje C.
   Paraninfo (1988).

9. B. Costales.
   Introducción al Lenguaje C.
   Colección Ciencia Informática.
   Ed. Gustavo Gili, S.A. (1985).

10. Jack Purdum.
    C Guía de Programación.
    Ed. Díaz de Santos, S.A. (1983).

11. Kris Jansa.
    Lenguaje C: Biblioteca de funciones.
    Osborne/McGraw-Hill.

12. P.J. Plauger.
    The Standard C Library
    Prentice-Hall (1992).

13. Kernighan, Ritchie.
    El lenguaje de programación C.
    Prentice Hall.

14. Herbert Shildt.
    Teach yourself C.
    Osborne/McGraw-Hill (1990).

15. M. Waite, S. Prata, D. Martin.
    Programación en C. Introducción y Conceptos Avanzados.
    Segunda edición, Julio 1988.

16. Delores M. Etter.
    Introduction to ANSI C for Engineers and Scientists.
    Prentice Hall (1996).

17. Kamal B. Rojiani.
    Programming in C for Engineers.
    Prentice Hall (1996).

18. Richard Johnsonbaugh y Martin Kalin.
    C for Scientists and Engineers.
    Prentice Hall (1996).

C.3 Programación C avanzada

1. Herbert Shildt.
   C: Guía para usuarios expertos.
   McGraw-Hill (1991).

2. A.M. Tenenbaum, Y. Langsam, M.A. Augenstein.
   Estructuras de Datos en C.
   Prentice-Hall (1993).

3. Kennet A. Barclay.
   C: Problem solving and programming.
   (1989).

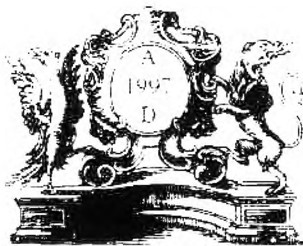
4. Ira Pohl.
   C++ for C programmers
   The Benjamin/cummings Publishing Company, Inc. (1994).

5. Ben Ezzell.
   Programación de Gráficos en Turbo C++.
   Addison-Wesley/Diaz de Santos.

6. Foley, Van Dam, Feiner, Hughes y Phillips.
   Introducción a la graficación por Computador
   Addison-Wesley (1996)

7. Francisco Charte.
Programación en Windows Multimedia.
Anaya Multimedia.

8. Mark L. Murphy.
C/C++ Software Quality Tools.
Prentice Hall (1996).



Se terminó de componer este libro
en el Servicio de Autoedición e Impresión
de la Universidad de Cádiz,
el día 15 de noviembre,
festividad de San Alberto Magno,
gigante de la ciencia de su tiempo
y del que pudo decirse
Mundo luxisti, quia totum scivile scisti
Murió a los 85 años de su edad,
tal día de 1280.

SERVICIO·DE·PUBLICACIONES
UNIVERSIDAD·DE·CADIZ
1997