APUNTES DE LA MATERIA DE PROGRAMACIÓN BÁSICA

FACULTAD DE CIENCIAS

Por

Héctor Eduardo Medellín Anaya

Tabla de Contenido

Introducción

Estos apuntes pretenden ser una guía práctica para los estudiantes del curso de computación de la carrera de Ingeniero Electrónico de la Facultad de Ciencias. Se incluyen numerosos programas tanto de ejemplos resueltos como de problemas propuestos. Los problemas fueron resueltos utilizando el compilador de C dev-C++ que puede obtenerse de forma gratuita en la dirección http://www.bloodshed.net/.

El primer capítulo contiene una breve historia de la computación, así como, una descripción de las componentes de un sistema de cómputo. También se hace una introducción a las bases numéricas.

El capítulo 2 trata de la resolución de problemas con la computadora. Se define el concepto de algoritmo y se estudian algunos ejemplos de algoritmos. Se define el lenguaje algorítmico que se utilizará a lo largo de todo el texto.

En el capítulo 3 se introduce a la escritura de programas en el lenguaje C. Se define el concepto de variable en programación y se revisan los tipos básicos de variables del lenguaje C. Se revisan las formas de asignar valores a variables, ya sea por asignación o mediante sentencias de entrada. Por último se estudia el uso de la biblioteca math.h que permite utilizar las funciones matemáticas más comunes y su aplicación en la evaluación fórmulas con funciones trigonométricas, exponenciales, logarítmicas, etc.

En el cuarto capítulo se revisan las sentencias de control de decisión. Se incluyen numerosos ejemplos de expresiones relacionales y expresiones lógicas. Con base en estas expresiones se construyen las condicionales `if` e `if-else`. Se estudian ejemplos del uso de la sentencia `switch` y del operador interrogación. Se incluyen ejemplos de anidamiento y validación de la entrada.

El quinto capítulo está dedicado a las sentencias de repetición. Se inicia con la descripción de la sentencia `while`, con ella se resuelven muchos ejemplos típicos. Se analizan ejemplos de ciclos controlados por centinela y ciclos anidados. Posteriormente, se revisa la sentencia `for`, que se utiliza, frecuentemente, en ciclos controlados por contador. Después se analiza la sentencia `do-while` y se dan ejemplos de sus aplicaciones más comunes como la validación de entrada.

El capítulo 6 es una introducción al uso de funciones en C. Se muestran ejemplos de funciones con parámetros que regresan valores numéricos. También se incluyen funciones sin parámetros y funciones de tipo `void`. Se analizan las reglas de alcance de las variables en C. Finalmente se estudian los parámetros por referencia y funciones recursivas.

En el capítulo 7 se revisan los arreglos en el lenguaje C. Se analizan ejemplos con arreglos de enteros de una dimensión. Se estudia la forma de pasar arreglos como parámetros a funciones y el uso de apuntadores. Se incluye el uso de arreglos de más de una dimensión.

En el capítulo 8 se estudia el uso de cadenas de caracteres y las bibliotecas que permiten su manipulación.

El capítulo 9 trata de estructuras y uniones. Se estudian numerosos ejemplos del uso de estructuras y la forma de pasar estructuras a funciones. El capítulo incluye ejemplos del uso de estructuras anidadas y arreglos de estructuras.

El capítulo 10 es una introducción al uso de archivos en C. Se estudia la forma de leer y escribir archivos secuenciales y archivos binarios. Se revisa el concepto de flujo en el manejo de archivos. Se revisa la forma de leer archivos de acceso aleatorio y sus aplicaciones.

Los apuntes incluyen numerosos ejemplos típicos de programación en cada uno de los capítulos así como de una colección muy amplia de problemas y proyectos de programación.

Capítulo 1. Introducción

1.1. Breve historia de la computación

El ábaco

Las primeras herramientas de cómputo se remontan a unos 2500 años atrás. Diversas formas de ábacos fueron utilizadas por los antiguos. Los más comunes en la actualidad son el ábaco chino, el japonés y el ruso. La siguiente figura muestra los tres tipos de ábacos representando al mismo número, 135,708.

Ábaco chino Ábaco japonés Ábaco ruso

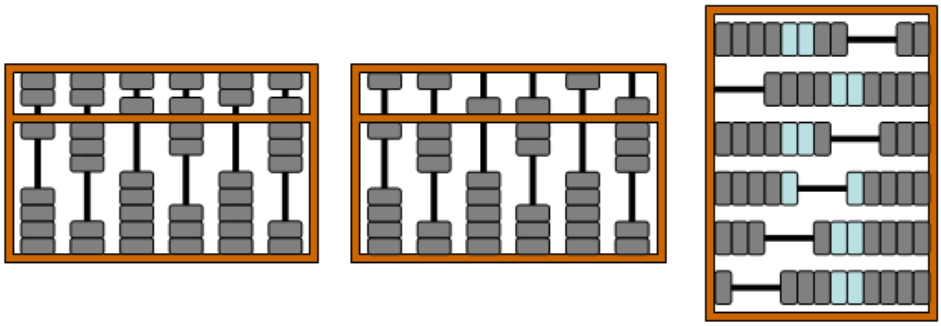


Figura 1.1. El ábaco.

Los logaritmos

En el siglo XVII John Napier inventó los logaritmos para simplificar las operaciones de multiplicación y división. Napier diseño unos palillos impresos con las tablas del 1 al 9 para simplificar los cálculos. Con estos palillos, llamados también estructuras de Napier se pueden realizar rápidamente operaciones de multiplicación mediante simples sumas de un solo dígito. Las estructuras pueden construirse de madera, metal, cartón, etc. Las estructuras de Napier se muestran en la siguiente figura.

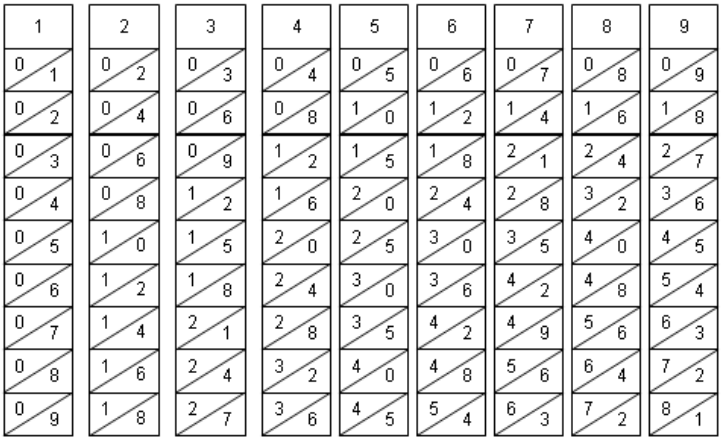


Figura 1.2. Estructuras de Napier.

La regla de cálculo

La regla de cálculo es una herramienta basada en los logaritmos. Está compuesta por dos reglas planas que se desplazan una respecto a la otra. Tiene un dispositivo indicador, llamado cursor, que permite seleccionar las cantidades a operar. Con la regla de cálculo es posible hacer operaciones de multiplicación y división de forma muy rápida con una precisión de 2 o 3 dígitos (4 en el mejor de los casos). La regla opera sumando o restando desplazamientos en escalas logarítmicas. La siguiente figura muestra una fotografía de una regla de cálculo.

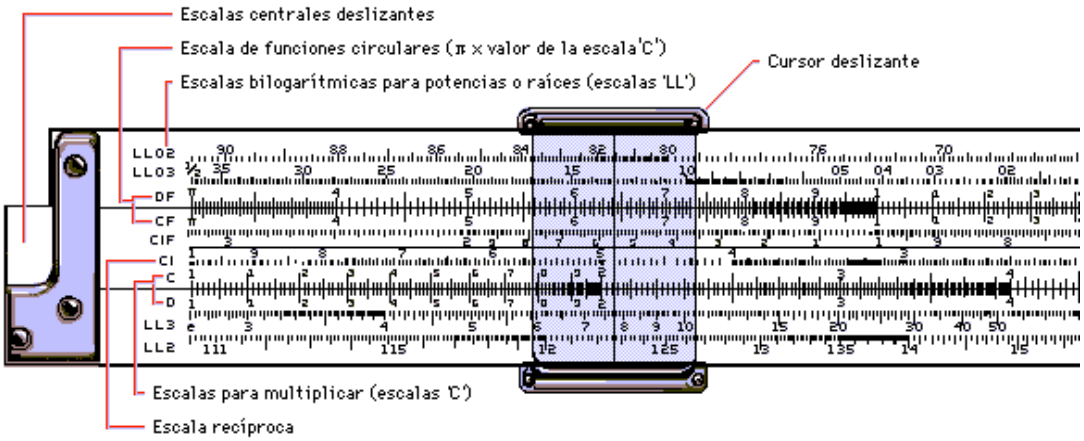


Figura 1.3. La regla de cálculo.

Algunas reglas poseen escalas para calcular funciones trigonométricas, exponenciales y logarítmicas. Téngase en cuenta que muchos de los cálculos que permitieron llevar el hombre a la luna fueron realizados utilizando reglas de cálculo.

Máquinas sumadoras y multiplicadoras

Pascal (1642) Calculadora Mecánica, realizaba sumas mediante ruedas dentadas. Leibniz (1690) la perfeccionó para realizar multiplicaciones. Joseph Marie Jacquard (1805) diseño un telar controlado por tarjetas perforadas. Charles Babbage y Augusta Ada Byron (1834) inventaron una serie de máquinas como la máquina diferencial, se les considera los padres de la computadora digital moderna. Herman Hollerith utilizó tarjetas perforadas para procesar el censo de 1890.

Primeras máquinas electrónicas

En el siglo XX los trabajos más relevantes son: entre 1939 y 1945 Alan Turing trabajo en el proyecto Colossus en Bletchley Park, Inglaterra. En 1945 se desarrolló el Calculador e Integrador Numérico Digital Electrónico (ENIAC) basado en trabajos de Atanasoff y Berry. Algunas características de la ENIAC eran

18,000 tubos de vacío
70,000 resistores
10,000 capacitores
150 kilovatios de potencia
15,000 metros cuadrados de superficie
30 toneladas de peso
trabajo 80223 horas
Se programaba mediante alambrado

Generaciones

Se pueden reconocer 5 generaciones de computadoras con base en la tecnología.

Primera Generación (1951-1958)

La primera generación se baso en tubos de vacío, tarjetas perforadas, tambores magnéticos y eran grandes y pesadas y consumían una gran cantidad de potencia eléctrica. Su costo era muy elevado para la época, alrededor de 10,000 dólares. La IBM 650 es un ejemplo de esta generación.

Segunda generación (1958 – 1964)

Uso de transistores que vinieron a sustituir a los bulbos. Memoria de núcleos de ferrita. Se desarrollaron lenguajes como FORTRAN y COBOL. Se crearon las mini computadoras y se inició el procesamiento remoto. Ejemplos de esta generación son la 5000 de Burroughs y la ATLAS de la Universidad de Manchester.

Tercera Generación (1964-1971)

Desarrollo de los circuitos integrados que permitieron compactar una gran cantidad de transistores. Los circuitos integrados recuerdan los datos, ya que almacenan la información como cargas eléctricas. Se desarrolla la multiprogramación. Se desarrolla la industria del software. Como ejemplos de esta generación están las minicomputadoras IBM 360 y DEC PDP-1.

Cuarta Generación (1971-1983)

Computadoras basadas en el microprocesador que integra en un solo chip toda la lógica de control de la computadora. Se desarrollan las tecnologías LSI y VLSI de circuitos integrados. La memoria se reemplaza por circuitos integrados. Aparecen las computadoras personales y las supercomputadoras.

Quinta Generación (1983 al presente)

Se expande el uso de las computadoras personales. Aparecen nuevos equipos personales como las Laptop, Palmtop, Netbook, etc. Se desarrolla el cómputo paralelo y distribuido. Aparece Internet. Se desarrolla la inteligencia artificial, la robótica, los sistemas expertos y las redes de comunicación.

Las computadoras tienen cada vez un uso más extendido. Actualmente encontramos computadores embebidos en todo tipo de aparatos, desde teléfonos celulares, licuadoras, reproductores de audio MP3, etc. La computación ha mejorado sustancialmente la vida humana y es indispensable para su funcionamiento y progreso. La programación es una disciplina entretenida y de alto nivel de abstracción.

1.2. Modelo de Von Neuman

El matemático Húngaro John Von Neuman definió la arquitectura de un sistema de cómputo moderno. En este modelo se almacenan los datos y los programas en la misma memoria a diferencia de los modelos anteriores. El modelo consta de cinco partes básicas: la **unidad de control**, que es la encargada de controlar todas las operaciones que realiza el sistema; la **unidad de memoria**, donde se almacenen los programas y datos que se utilizan durante los cómputos; la **unidad lógica y aritmética** que realiza las operaciones lógicas y aritméticas; los **dispositivos de entrada** se encargan de la introducción de datos y programas a la memoria y los **dispositivos de salida** que despliegan los resultados de los cómputos. La siguiente figura muestra el modelo básico de Von Neuman.

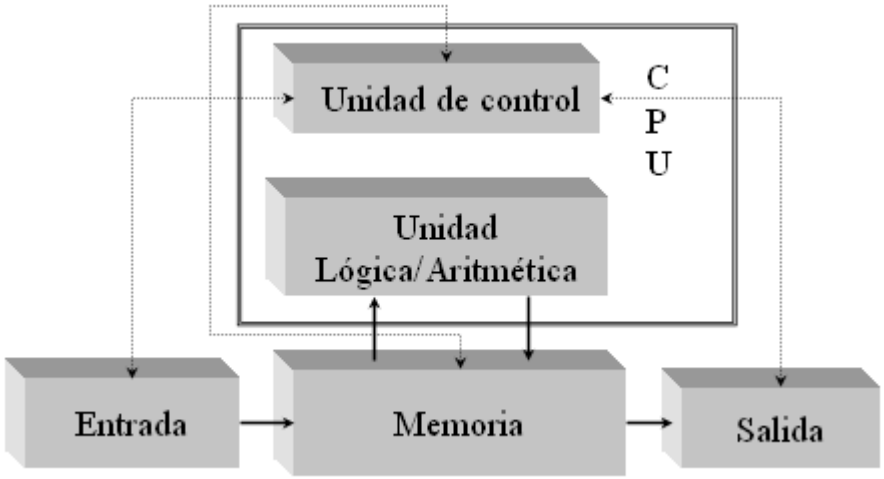


Figura 1.4. Modelo de John Von Neuman.

La línea encierra lo que se conoce ahora como **unidad central de proceso** que consta de la unión de la unidad de control y la unidad lógica y aritmética. Las flechas punteadas son líneas de control entre las diversas unidades y las flechas sólidas indican las rutas por las que se mueven los datos.

1.3. Bases numéricas

La base numérica que utilizamos normalmente es la base decimal. En esta base se utilizan 10 símbolos para representar los dígitos de los números: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 . Los números se representan mediante secuencias de estos dígitos. El valor real de un dígito depende de su posición dentro de la secuencia. El dígito en el extremo derecho del número representa las unidades, el siguiente a la izquierda, representa las decenas, el que le sigue las centenas y así sucesivamente. Por ejemplo el número 34564 realmente representa:

$34564 = 3 \times 10^4 + 4 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 4 \times 10^0$
$= 30000 + 4000 + 500 + 60 + 4$

Otras bases importantes en computación son la base 2, 8 y 16. En la base 2 solo se utilizan 2 dígitos: 0 y 1. Los números en base dos son secuencias de 1's y 0's. El sistema de base 2 se llama sistema binario.

Los dígitos de un número en el sistema binario representan el valor del dígito multiplicado por la potencia de 2 correspondiente. Al dígito en el extremo derecho le corresponde la potencia 0 de 2, al siguiente a la izquierda la potencia 1 de 2, y así sucesivamente. Los dígitos binarios reciben el nombre de *bit* (binary digit). Por ejemplo, el número 101001 representa

$101001 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 32 + 0 + 8 + 0 + 0 + 1 = 41$

Se acostumbra poner un subíndice para indicar la base cuando se usan bases diferentes, así

$101001_2 = 41_{10}$

El sistema en base 8, llamado sistema octal, consta de los dígitos: 0, 1, 2, 3, 4, 5, 6 y 7. En este sistema los números están basados en potencias de 8. El número 4526 representa al

$4526 = 4 \times 8^3 + 5 \times 8^2 + 2 \times 8^1 + 6 \times 8^0$
$= 4 \times 512 + 5 \times 64 + 2 \times 8 + 6$
$= 2048 + 320 + 16 + 6 = 2390$

$4526_8 = 2390_{10}$

En el sistema de base 16 se utilizan 16 símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. Este sistema se llama hexadecimal. El número A34E representa

$A34E = 10 \times 16^3 + 3 \times 16^2 + 4 \times 16^1 + 14 \times 16^0$
$= 10 \times 4096 + 3 \times 256 + 4 \times 16 + 14 = 40960 + 768 + 64 + 14 = 41806$

$A34E_{16} = 41806_{10}$

1.4. Conversión entre bases numéricas

La conversión de la base decimal a cualquier otra se realiza mediante el siguiente procedimiento: dividir el número entre la base deseada obteniendo el residuo, repetir el proceso dividiendo el cociente de la división hasta que el resultado de la división sea cero, el número convertido consta de los valores de los residuos, convertidos a la base nueva, comenzando de izquierda a derecha por el último residuo obtenido.

Ejemplo: Convertir 346 de decimal a octal:

43	5	0
$8\overline{)346}$	$8\overline{)43}$	$8\overline{)5}$
2	3	5

El número convertido es 532. Para comprobar convertiremos a base 10:

$532_8 = 5 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 5 \times 64 + 24 + 2 = 320 + 26 = 346_{10}$

Ejemplo: Convertir 4560 de decimal a hexadecimal:

285	17	1	0
$16\overline{)4560}$	$16\overline{)285}$	$16\overline{)17}$	$16\overline{)1}$
0	13	1	1

El número convertido es 11D0. Para comprobar convertiremos a base 10:

$11D0_{16} = 1 \times 16^3 + 1 \times 16^2 + 13 \times 16^1 + 0 \times 16^0 = 4096 + 256 + 208 = 4560_{10}$

La conversión entre la base binaria y las bases octal y hexadecimal es especialmente sencilla ya que 8 y 16 son potencias de 2. Para convertir de binario a octal simplemente se agrupan los bits de 3 en 3 comenzando por la derecha, y a continuación se convierte cada terna en el dígito octal correspondiente. Ejemplo:

$11101001001_2 = 11\ 101\ 001\ 001 = 3\ 5\ 1\ 1 = 3511_8$

La conversión inversa es igualmente sencilla, ejemplo:

$34671_8 = 011\ 100\ 110\ 111\ 001 = 11100110111001_2$

Para convertir hacia hexadecimal se agrupan de 4 en cuatro y se convierte cada grupo en el dígito hexadecimal correspondiente, ejemplo:

$11110100001111_2 = 11\ 1101\ 0000\ 1111 = 3\ D\ 0\ F = 3D0F_{16}$

El proceso contrario se ilustra en el siguiente ejemplo.

$4FEA5_{16} = 0100\ 1111\ 1110\ 1010\ 0101\ 0001\ 0110 = 10011111101010010100010110_2$

Problemas propuestos

1. Convierta los números de la base indicada a base diez.
a. 11001_2 b. $637EE_{16}$ c. 7652_8 d. 11100111_2 e. 7652_{16} f. 3332111_{16}

2. Convierta de base diez a la base indicada
a. 23901 a base 16 b. 23901 a base 8 c. 23901 a base 2 d. 10101 a base 2

3. Convierta de binario a octal y hexadecimal.

a. 101010001100 b. 11101111011 c. 10001010101 d. 1011100111

4. Convierta de Hexadecimal y octal a binario.

a. 32158 b. 101018 c. EFEE16 d. 1010116

5. convierta a octal los siguientes números en hexadecimal.

a. EDF231 b. 33EEAA c. 56832 d. 4432CC

6. convierta a hexadecimal los siguientes números en octal.

a. 456312 b. 32456 c. 342245 d. 777111222

Capítulo 2. Resolución de problemas con la computadora

2.1. Algoritmos

Un algoritmo es la descripción detallada de los pasos necesarios para resolver un problema. Un algoritmo debe cumplir con tres características, a saber: los pasos deben ser simples y claros; el orden en que se ejecuten los pasos debe ser preciso; el algoritmo debe resolver el problema en un número de pasos finito. Los algoritmos deben ser especificados con instrucciones que puedan ser ejecutadas por alguna entidad. La entidad puede ser cualquiera que sea capas de seguir instrucciones como una persona, una computadora, un robot, o algo similar.

Un ejemplo de algoritmo es una receta para preparar café instantáneo. En este caso el algoritmo está dirigido a una persona, la cual preparará el café. Supondremos que se dispone de lo necesario para preparar el café, cosas tales como una tasa, un recipiente para calentar agua, estufa, cuchara, etc. El algoritmo puede ser el siguiente:

1. Ponga una tasa de agua en el recipiente para calentar sobre la estufa.
2. Caliente el agua hasta justo antes de hervir.
3. Vacíe el agua caliente en una tasa.
4. Agregue las cucharadas de café que desee.
5. Agregue las cucharadas de azúcar que desee.
6. revuelva hasta que los ingredientes se mezclen adecuadamente.

Note que el algoritmo anterior puede ser especificado con mayor detalle dependiendo de a que individuo esté dirigido. Por ejemplo, el paso 2 quizá requiera especificar de que forma se enciende la estufa; abriendo la llave del gas y presionando un botón o utilizando una cerilla para encenderla, etc. Hay que considerar todos los aspectos posibles cuando se trata de instruir a una computadora para resolver un problema.

Las computadoras pueden ejecutar solo un conjunto de instrucciones limitado. Los paso de los algoritmos debemos especificarlos utilizando solamente las instrucciones que la computadora sea capaz de ejecutar o alguna instrucción equivalente. Generalmente usamos instrucciones equivalentes a las que una computadora puede ejecutar debido a que los lenguajes de las computadoras se especifican con palabras en inglés o porque las instrucciones de la computadora son difíciles de entender para los humanos, tal es el caso del lenguaje C. A un lenguaje similar al lenguaje de la computadora le llamamos pseudo código.

2.2. Lenguaje algorítmico

Para resolver un problema con la computadora es necesario sobre todo entender bien en que consiste el problema. Una vez que el problema se entiende, se procede al diseño del algoritmo para resolverlo. Después de que el algoritmo ha sido escrito y probado en forma

de pseudo código, se puede proceder a traducirlo a algún lenguaje que la computadora entienda para que pueda se ejecutado por ésta.

Definiremos un lenguaje para expresar los algoritmos de una forma conveniente. El lenguaje algorítmico será cercano al lenguaje C para facilitar la traducción a programas ejecutables. Veamos el siguiente ejemplo:

Algoritmo Volumen de esfera. Este algoritmo calcula el volumen de una esfera dado su radio. Se utiliza la variable RADIO para representar el radio de la esfera y la variable VOLUMEN para representar su volumen. Suponemos ambas variables como números reales.

```
1. [Leer el radio de la esfera]
        Leer(RADIO)
2. [Calcular el volumen de la esfera]
        VOLUEN = 4*3.1416*RADIO^3
3. [Se imprimen el dato y el resultado]
        Escribe("El radio de la esfera es ", RADIO)
        Escribe("El volumen de la esfera es ", VOLÜMEN)
4. [Termina]
        Salir
```

El algoritmo se llama "volumen de esfera". Es conveniente siempre asignar un nombre a cada algoritmo para poder identificarlo claramente. Es conveniente hacer una breve descripción del propósito del algoritmo al inicio, también en esta sección se puede definir el tipo y propósito de cada una de las variables utilizadas. Cada paso del algoritmo esta numerado secuencialmente. Los comentarios los encerraremos en paréntesis cuadrados ([ ]). Estos comentarios solo sirven para aclarar cada uno de los pasos del algoritmo. Se ha utilizado la palabra "Salir" para indicar la terminación del algoritmo en la última línea. Otros podrán tener varios finales, esto implica que algunos finales no se encuentren en la última línea.

El paso 1 del algoritmo es una sentencia de entrada de datos, "Leer". Entre paréntesis encerraremos las variables que serán leídas, en este caso es solo la variable RADIO. El paso 2 contiene un enunciado de asignación. Este enunciado tiene por objeto establecer el valor de una variable, en este caso se calcula el valor del volumen de una esfera y se le asigna este valor a la variable VOLUMEN. El volumen se calcula con la expresión

$$Volumen = 4 \pi r^3$$

Vamos a escribir las expresiones lo más parecidas al lenguaje C, por eso hemos utilizado un asterisco "*" para indicar la operación de multiplicación y además escribiremos las potencias mediante el carácter "^". El paso 3 contiene dos sentencias de salida. Estas se utilizan para mandar letreros y valores de variables a la pantalla y así poder ver los resultados. La sentencia básica de salida es "Escribe" y hay que especificar dentro de paréntesis lo que se va a desplegar. En la primera sentencia de salida se desplegará "El radio de la esfera es" seguido del valor del radio, cada elemento de la sentencia se separa por una coma. La segunda sentencia de salida desplegará "El volumen de la esfera es"

seguido del valor calculado. El último paso del algoritmo es la sentencia de salida o terminación, esta sentencia indica la finalización del algoritmo, como se indicó anteriormente.

Es conveniente probar los algoritmos que desarrollemos para asegurar que estos hacen lo que se espera que hagan. Para probar un algoritmo ejecutamos cada uno de sus pasos en orden ascendente hasta llegar a la sentencia de Salida. Si encontramos alguna sentencia de entrada, suministraremos un valor para las variables que se introducirá.

Supongamos que deseamos calcular el volumen de una esfera con radio 4.5 cm. Entonces, este valor de 4.5 será la entrada para la variable RADIO. Es conveniente tabular los valores que adquieren las variables y las sentencias de salida en cada paso del algoritmo. Tendremos un renglón para cada paso. Las variables que no tengan un valor definido las indicaremos mediante un signo de interrogación.

En el paso 1 se le asigna 4.5 a RADIO. En el paso 2 se calcula el volumen de la esfera y se le asigna este valor a VOLUMEN. El paso 4 contiene las sentencias de salida. La tabla de prueba del algoritmo es la siguiente:

Paso	RADIO	VOLUMEN	Salida
1	4.5	?	
2	4.5	1145.11	
3	4.5	1145.11	El radio de la esfera es 4.5 El volumen de la esfera es 1145.11

Note les valores de las variables permanecen sin cambio una vez que son asignados o leídos.

1.3. Metodología de la resolución de problemas

Para poder resolver problemas con la computadora debemos seguir algunos pasos. Lo primero es tener muy claro el problema que se va a resolver, esto quiere decir que debe entenderse perfectamente el problema. Luego debemos diseñar un algoritmo que lo resuelve. Esta parte es a veces lo más complicado, sobre todo al principio, debido a que se carece de experiencia en el diseño de algoritmos. Sin embargo es la parte más importante, dado que si podemos diseñar un algoritmo, podremos convertir este algoritmo en un programa que pueda ejecutar la computadora.

Una vez que se tiene el algoritmo, se debe proceder a probar el algoritmo con datos válidos y de esta manera verificamos si el algoritmo resuelve o no el problema. El paso más sencillo es la traducción del algoritmo a un programa de computadora para que sea ejecutado por esta. Si nuestro algoritmo no resuelve el problema debemos modificarlo para que se obtenga la solución buscada. Este proceso puede repetirse hasta encontrar el algoritmo adecuado que nos lleva a la solución del problema.

Para problemas complicados es conveniente partir de un algoritmo especificado con pasos muy generales e ir refinándolo hasta encontrar la solución deseada.

2.3. Aplicaciones

En esta sección veremos algunos ejemplos de resolución de problemas mediante algoritmos. El primer problema es determinar el alcance y el tiempo de vuelo de un proyectil lanzado con una velocidad inicial v_0 y haciendo un ángulo de θ grados con la horizontal. Llamaremos al algoritmo "Tiro". Es conveniente leer el ángulo en grados ya que es más intuitivo para un estudiante de ciencias e ingeniería. Sin embargo, como veremos más adelante, las funciones que suministran los lenguajes de programación utilizan ángulos especificados en radianes. Debemos por tanto agregar una fórmula para convertir entre grados y radianes. Las fórmulas que usaremos son:

$rad = grados\frac{\pi}{180}$ para convertir entre grados y radianes.

$R = \frac{v_0^2 \text{sen} 2\theta}{g}$ máximo alcance de un proyectil, g es la aceleración de la gravedad.

$T = \frac{2v_0^2 \text{sen}\theta}{2}$ tiempo de vuelo.

Algoritmo Tiro. Este algoritmo calcula el alcance y el tiempo de vuelo de un proyectil. Usaremos la variable ANG para el ángulo leído y ANGRAD para el ángulo convertido a radianes. La velocidad inicial en m/s será almacenada en la variable VEL. El alcance se almacena en la variable R y el tiempo de vuelo en la variable T.

1. [leer datos de entrada]
   Leer(ANG,VEL)
2. [Convertir el ángulo a radianes]
   ANGRAD = ANG*3.1416/180.0
3. [Calcula alcance]
   R = VEL*VEL*SEN(2*ANGRAD)/9.8
4. [Calcula el tiempo de vuelo]
   T = 2*VEL*SEN(ANGRAD)/2
5. [Despliega resultados]
   Escribe ("Velocidad inicial ",VEL)
   Escribe ("Ángulo del tiro ",ANG)
   Escribe ("Alcance ",R)
   Escribe ("Tiempo de vuelo ",T)
6. [Fin]
   Salir

Se desea resolver un sistema de ecuaciones simultáneas de 2x2 de la forma

$a\,x + b\,y = c$
$d\,x + e\,y = f$

La solución de este sistema puede escribirse como

$$x = \frac{ce - bf}{ac - bd}$$

$$y = \frac{af - cd}{ac - bd}$$

Algoritmo Simultáneas. Resuelve un sistema de ecuaciones de 2x2. Los coeficientes son las variables A, B, C, D, E y F. Las soluciones se almacena en las variables X y Y.

1. [Leer coeficientes]
   Leer(A, B, C, D, E, F)
2. [Calcula la solución]
   X = (C*E – B*F)/(A*C – B*D)
   Y = (A*F – C*D)/(A*C – B*D)
3. [Imprime solución]
   Escribe (“x = ”, X)
   Escribe (“y = ”, Y)
4. [Fin]
   Salir

Problemas propuestos

1. El siguiente es el menú de un puesto de tacos. Escriba un algoritmo que lea el número de cada taco ordenado y calcule la cuenta total.

   Taco de bistec ($ 4)
   Taco de arrachera ($ 8)
   Taco al pastor ($ 6)
   Refresco ($ 8)
   Orden de cebollitas ($ 5)

2. Desarrolle algoritmos para realizar las siguientes conversiones.
a) Leer una cantidad en euros e imprimir el equivalente en dólares americanos.
b) Leer una cantidad en pesos mexicanos e imprimir el equivalente dólares y en euros.

Capítulo 3. Introducción al lenguaje de programación C

3.1. Partes de un programa en C

El lenguaje C ha demostrado ser un lenguaje conveniente para resolver todo tipo de problemas. Es un lenguaje de propósito general que igual se utilizó en sus inicios para escribir sistemas operativos como ahora en el desarrollo de juegos y aplicaciones científicas. El C es el lenguaje estándar en el desarrollo de aplicaciones en ingeniería y todo ingeniero que se precie debe ser capaz de leer y desarrollar programas en él.

Para entrar en materia escribiremos un primer programa en C. Escriba las siguientes líneas asegurándose que están idénticas a las que se muestran.

```
/*Primer programa en C */
#include <stdio.h>  /*biblioteca para entrada y salida*/
#include <conio.h>  /*biblioteca para la función getch*/

int main()/*aquí inicia el programa */
{
  printf("Hola mundo!.\n");/*sentencia de salida*/
  getch();
  return 0;/*terminación normal del programa*/
}/*fin del programa*/
```

La primera línea es un comentario. Los comentarios inician con “/*” y terminan con “*/”. Existe otra forma de escribir comentarios que veremos más adelante. La segunda línea sirve para incluir la biblioteca de entrada y salida. El lenguaje C no tiene instrucciones de entrada y salida propias, por tanto debemos siempre incluir alguna biblioteca para realizar estas operaciones.

La función `main` es la que contiene las sentencias ejecutables del programa. Todo programa en C debe tener una función `main`. La palabra **`int`** antes de `main` indica que la función regresa un valor entero. La palabra **`int`** es una palabra reservada, a lo largo del texto las palabras reservadas serán representadas con tipo negrita. Los paréntesis después de la palabra `main` se usan para definir los parámetros de la función, en este caso no tiene ningún parámetro. En algunas ocasiones los parámetros van a ser necesarios.

La llave que abre inicia un bloque de instrucciones. La primera instrucción es

```
  printf("Hola mundo!\n");/*sentencia de salida*/
```

Esta instrucción es una sentencia de salida. Note que se ha incluido un comentario para aclarar el significado de esta línea. La palabra `printf` se utiliza para desplegar letreros y valores de variables o expresiones. Esta palabra designa a una función encargada de dichas operaciones. Los paréntesis sirven para delimitar los elementos de salida. En este caso es un letrero el cual debe ir encerrado entre comillas, las comillas no se despliegan. En este caso

se desplegará la cadena `"Hola mundo!\n"`. La secuencia de caracteres \n indica que se mueva el cursor a la siguiente línea, a este tipo de controles se les llama secuencias de escape.

La sentencia `getch()`es una función que lee un carácter desde el teclado, esto permite que el programa se detenga hasta que el usuario presione una tecla. Por último, la sentencia `return 0` indica una terminación normal (sin errores) del programa. Note que todas las sentencias deben terminarse con un punto y coma, esto va a ser necesario en la mayoría de los casos de las sentencias en C.

La función `printf` imprime un letrero y deja el cursor en la posición que sigue al último carácter impreso. El siguiente ejemplo imprime una sola línea utilizando tres sentencias `printf`.

```
/*Ejemplo de varias sentencias prinf una sola línea de salida
*/
#include <stdio.h>  /*biblioteca para entrada y salida*/
#include <conio.h>  /*biblioteca para la función getch*/

int main(){
  printf("Esta es una");
  printf(" cadena impresa ");
  printf("en una sola línea\n");
  getch();
  return 0;
}
```

También pueden imprimirse varias líneas mediante una sola sentencia `printf`, por ejemplo:

```
/*Ejemplo de printf, una sentencia varias líneas*/
#include <stdio.h>
#include <conio.h>

int main(){
  printf("Línea 1\nEsta es la línea 2\ny está es la 3\n");
  getch();
  return 0;
}
```

Otra secuencia de escape para formatear la salida es `\t` (tabulador). Mediante el tabulador se pueden imprimir varias cadenas en columnas predeterminadas. Ejemplo:

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

int main(){
```

```
   printf("nombre\tdirección\tteléfono\n");
   printf("juan\tolmo 204 \t8-12-12-34\n");
   printf("maria\tpino 565 \t8-34-27-16\n");
   getch();
   return 0;
}
```

El ejemplo imprime la siguiente salida:

```
Nombre  dirección teléfono
Juan    olmo 204  8-12-12-34
Maria   pino 565  8-34-27-16
```

Problemas propuestos

3.1.1. Escriba un programa que despliegue su nombre, dirección y teléfono en tres líneas separadas.

3.1.2. Escriba un programa que muestre la siguiente salida utilizando cuatro sentencias `printf`, todas con cadenas no vacías.

```
Lenguaje de programación C. Primer curso de programación.
```

3.1.3. ¿Qué salida despliega el siguiente fragmento de programa?

```
printf(“\n\nEsta es una línea.\nEsta es otra línea.”);
printf(“ Esta es la continuación\n\n\nEsta es la última línea.”);
```

3.1.4. Escriba una sola sentencia `printf` que despliegue las siguientes líneas. Utilice secuencias de escape para tabuladores y alimentos de línea.

a)

```
Equipo        jj      jg      je      jp      pts
Cruz Azul     5       2       1       6       7
Guadalajara   6       4       2       0       14
```

b)

```
FACULTAD DE CIENCIAS

UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

Ingeniería electrónica            Lic. En Física
```

3.2. Compilación y ejecución de un programa

Antes de poder ejecutar programas debemos de disponer de algún compilador de C. En Internet se encuentran algunos compiladores de uso gratuito como Dev-C. Una vez que dispongamos del compilador, escribimos el programa en el editor de texto y lo guardamos con algún nombre. Algunos compiladores suponen que los programas tienen extensión ".c" o ".cpp" o algo parecido. Una vez guardado en un archivo podemos proceder a traducirlo a lenguaje de máquina para que pueda ser ejecutado. El proceso de traducción lo realiza el compilador, si no hay errores el compilador generará un archivo ejecutable que podemos correr.

Si el compilador encuentra cualquier tipo de error, enviará el mensaje adecuado, sobre todo si el compilador es del tipo de ambiente de trabajo como Dev-C. En todo caso, deberá corregir los errores encontrados y volver a compilar hasta que no haya errores.

El proceso de ejecución podemos separarlo en varias fases. La primera fase es la escritura del programa en el editor de texto. Luego tenemos la fase de compilación del programa, estas dos fases pueden repetirse en caso de que haya errores. La fase de compilación genera el programa ejecutable también llamado programa objeto. Por último está la fase de ejecución en la que el programa es ejecutado por la computadora, en este punto se introducen los datos de entrada, si los hay, y se despliega la salida que se haya especificado, también si la hay. También en la fase de ejecución se presentan errores, si los hay, estos errores se deberán quizá a que el programa no cumple con las especificaciones para resolver el problema dado.

Se deberá a proceder a corregir los errores que se presenten en la ejecución repitiendo todo el proceso de escritura-compilación-ejecución nuevamente.

3.3. Variables simples y asignación

Las variables permiten almacenar números de varias clases, así como otro tipo de datos. Por lo pronto exploraremos las variables que almacenan números enteros. Los nombres de las variables deben ser identificadores válidos. Un identificador es válido si es una secuencia de letras, dígitos y guiones bajos (_) que no comience con un dígito. El lenguaje C distingue entre letras mayúsculas y minúsculas, de tal manera que x2 es diferente de X2. Es conveniente utilizar palabras significativas para definir variables, de esta manera será mas sencillo la escritura y depuración de programas.

Todas las variables en C deben declararse antes de ser utilizadas. La declaración de las variables consiste en la definición de su tipo y nombre, opcionalmente se puede dar un valor inicial a la variable. La sintaxis es la siguiente:

```
Tipo nombre;
```

O

```
Tipo nombre = valor;
```

Se pueden declarar varias variables del mismo tipo simultáneamente. Ejemplos:

```
int suma;
```

Declara la variable suma de tipo entero.

```
int  cuenta, promedio, suma = 0;
```

Declara las variables siguientes: cuenta, promedio y suma de tipo entero y asigna 0 a la variable suma.

El lenguaje C suministra varios tipos de números enteros y reales que se utilizan con diferentes propósitos. La siguiente tabla resume estos tipos:

Tipo	Longitud	Rango
`unsigned char`	8 bits	0 a 255
`char`	8 bits	-128 a 127
`enum`	16 bits	-32,768 a 32,767
`unsigned int`	16 bits	0 a 4,294,967,295
`short int`	16 bits	-32,768 a 32,767
`int`	32 bits	-2,147,483,648 a 2,147,483,647
`unsigned long`	32 bits	0 a 4,294,967,295
`long`	32 bits	-2,147,483,648 a 2,147,483,647
`float`	32 bits	$3.4x10^{-38}$ a $3.4x10^{+38}$
`double`	64 bits	$1.7x10^{-308}$ a $1.7x10^{+308}$
`long double`	80 bits	$3.4x10^{-4932}$ a $1.1x10^{+4932}$

La asignación es la operación más elemental que puede aplicarse a una variable. El operador “=” se utiliza para esta propósito. El siguiente ejemplo asigna valores a dos variables enteras y calcula la división de los dos números. El operador de división es la diagonal normal “/”.

```
/*asignación de variables*/
#include <stdio.h>
#include <conio.h>

int main(){
  int divisor, dividendo, cociente;
  divisor = 13;
  dividendo = 45;
  cociente = dividendo / divisor;
  printf("dividendo: %d\n",dividendo);
  printf("divisor: %d\n",divisor);
  printf("cociente: %d\n",cociente);
  getch();
```

```
  return 0;
}
```

Escriba, compile y corra el programa. Note que el resultado de la división es 3 ya que el resultado de dividir dos números enteros es un número entero sin parte fraccionaria. En las sentencias de salida se utiliza un letrero que contiene el formato de las variables que se desplegaran. El formato para los números enteros decimales es "%d". Para cada variable o expresión que se desee desplegar debe utilizarse un una especificación de formato. La salida la podemos hacer con una sentencia `printf` como la siguiente:

```
printf("dividendo: %d\ndivisor: %d\ncociente: %d\n",
dividendo, divisor, cociente);
```

La operación de asignación puede realizarse entre variables de diferentes tipos. El compilador informará mediante una advertencia si se hace una asignación potencialmente peligrosa, tal como asignar a un entero un valor real (de punto flotante) ya que esto lleva a una posible pérdida de precisión. El siguiente ejemplo ilustra asignaciones entre variables de tipo `float` (punto flotante) y enteras. Note que los valores impresos por el programa son los últimos asignados a las variables. El valor de a debería ser 8500000000 pero tal número no cabe en 32 bits, por tanto se mostrará -2147483648.

```
/*Ejemplo de asignaciones*/
#include <stdio.h>
#include <conio.h>

int main(){
  int a,b,c;  /* 3 variables enteras */
  float x,y,z; /* 3 variables reales */
  a = 5;
  b = -2;
  c = 8;
  x = a;
  a = b;
  y = x;
  z = 8.5;
  x = z;
  a = 1e10*x;
  printf("a=%d\nb=%d\nc=%d\n",a,b,c);
  printf("x=%f\ny=%f\nz=%f\n",x,y,z);
  getch();
  return 0;
}
```

El formato de salida para variables de punto flotante es "%f". Una constante de punto flotante puede consistir de un a secuencia de dígitos decimales o también de dos secuencias de dígitos decimales separadas por un punto decimal. También podemos representar con constantes números muy grandes o pequeños utilizando multiplicadores por potencias de

10. El número 1e10 representa $1x10^{10}$. Note que el compilador envía un mensaje de advertencia (warning) por la asignación "`a = 1e10*x`" ya que se asigna un `float` a un entero. Esta advertencia puede ser ignorada por el momento.

Si bien el compilador no toma en cuenta el significado de los identificadores utilizados, es muy recomendable utilizar palabras con significado cuando los definamos para facilitar la lectura y comprensión de los programas. Por ejemplo si se va a representar mediante una variable los días trabajados, es mejor utilizar el identificador `dias_trabajados` o `diasTrabajados` que simplemente el identificador x.

Problemas propuestos

3.3.1. Defina identificadores con significado para las siguientes variables:
a. temperatura promedio b. grados por minuto c. segundos de retrazo
d. velocidad de despegue

3.3.2. ¿Que salida genera el siguiente fragmento de programa?

```
x = 5.7;
y = -12.5;
printf("\n\Valor de x = %f\nValor de y=%f",x,y);
```

3.3.3. ¿Que salida genera el siguiente fragmento de programa?

```
int a, b, c;
float t, w;
a = 20;
b = a;
t = b;
a = 12;
c = t;
w = 5.2;
printf("\na = %d, b= %d, w = %f\n",);
printf("\nc = %d, t = %f\n",);
```

3.4. Operadores aritméticos

La siguiente tabla resume los operadores aritméticos básicos de C.

Operación	Operador	Ejemplo
Suma	+	5 + 8
Resta	-	5 – 2
Multiplicación	*	4 * 8
División	/	2 / 7
Módulo	%	5 % 3

El operador módulo (%) calcula el residuo de la división. Este operador solo puede aplicarse a operandos de tipo entero, los tipos `float`, `double` y `long double` quedan excluidos.

$$\begin{array}{r} 37 \Leftarrow 645/17 \\ 17\overline{)645} \\ 16 \Leftarrow 645\%17 \end{array}$$

Los operadores tienen diferentes prioridades de evaluación cuando aparecen en la misma expresión. A estas prioridades se les llama reglas de precedencia y se resumen en la siguiente tabla:

Operador	Operación	Orden de evaluación
`( )`	Agrupa expresiones	Tiene la precedencia más alta, se evalúan los más internos primero.
`-, +`	Signo positivo o negativo	Operadores unarios
`*, /, %`	Operadores multiplicativos	Se evalúan de izquierda a derecha
`-, +`	Suma y resta	Operadores binarios. Se evalúan de izquierda a derecha

Los siguientes ejemplos muestran la conversión de algunas expresiones:

Expresión algebraica	Expresión en C
$a+b-\frac{d}{c}$	`a + b – d/c`
$a \bmod c + d$	`a % c + d`
$\frac{2a+b}{4c-f}$	`(2*a + b)/(4*c – f)`
$\frac{5(2+a)}{4bc}$	`5*(2+a)/(4*b*c)o` `5*(2+a)/4/b/c`

Es importante la inserción de paréntesis para garantizar que las expresiones estén correctamente escritas en C. En la última expresión se puede ahorrar un paréntesis dividiendo entre 4, luego entre b y luego entre c. El siguiente ejemplo calcula el área y volumen de una esfera.

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

/*calcula el área y el volumen de una esfera*/
int main(){
```

```
  float r,area,volumen;/*radio, área y volumen de la esfera*/
  r = 4.5;
  area = 4*3.1416*r*r;
  volumen = 4*3.1416*r*r*r/3;
  printf("el radio de la esfera es %f\n",r);
  printf("el área de la esfera es %f\n",area);
  printf("el volumen de la esfera es %f\n",volumen);
  getch();
}
```

Problemas propuestos

3.4.1. Escriba las siguientes expresiones algebraicas en C.

a) $a+b\dfrac{x+1}{x-1}$ b) $\dfrac{a}{b(2c+d)}$ c) $\dfrac{1}{1+\dfrac{1}{(3a-2b)}}$ d) $1+\dfrac{1}{1+\dfrac{1}{1+\dfrac{1}{x}}}$

3.4.2. Escriba las siguientes expresiones algebraicas en C utilizando un mínimo de paréntesis.

a) $\dfrac{(x+y)}{2abc}$ b) $(3a-4b)(5a-3)\dfrac{1}{2c(b-1)}$

3.4.3. Escriba un programa que calcule el área y el volumen de un cilindro de radio 3.3 unidades y altura 2.5 unidades. Las fórmulas para el área y el volumen de radio *r* y altura *h* son:

$area\ cilindro = 2\pi rh + \pi r^2$

$volumen\ cilindro = \pi r^2 h$

3.5. Entrada desde el teclado

Para la entrada de datos se utiliza la función `scanf` de la biblioteca stdio.h. La función lleva al menos dos argumentos, el primero es la cadena que especifica el formato de entrada y los demás son la lista de variables a introducir. Asegúrese de poner el formato adecuado a la variable que se va a leer, de no hacerlo así, el programa puede fallar. La sintaxis de `scanf` es:

```
scanf(cadena de formato, lista de variables);
```

Cada una de las variables de la lista de variables debe ir precedida del *operador de referencia* (&). Los formatos disponibles se muestran en la siguiente tabla.

tipo	Entrada	Tipo de argumento
c	**Carácter**: lee el siguiente carácter.	char *
d	**Entero con signo**: Entero decimal	int *

	posiblemente precedido por signo + o -.	
e, E, f, g, G	**Punto Flotante:** Número decimal con punto decimal, opcionalmente precedido por + o – y opcionalmente seguido de e o E y un número	float *
o	**Entero octal:** lee un número entero en octal	int *
s	**Cadena de caracteres:** Lee los caracteres tecleados hasta el siguiente blanco.	char *
u	**Entero sin signo:** Entero decimal sin signo.	unsigned int *
x, X	**Entero hexadecimal:** lee un número entero en hexadecimal	int *

Cuando se ejecuta la sentencia `scanf` se suspende la ejecución del programa hasta que se introduzcan las variables de la sentencia. Las variables deben separarse con espacios o alimentos de línea. El siguiente es una modificación del ejemplo de división visto anteriormente:

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

int main(){
   int divisor, dividendo, cociente;
   printf("Teclee el divisor: ");/*informa que se va a leer*/
   scanf("%d",&divisor);            /*Lee El divisor */
   printf("Teclee el dividendo: ");
   scanf("%d",&dividendo);          /*Lee el dividendo*/
   cociente = dividendo / divisor;
   printf("dividendo: %d\n",dividendo);
   printf("divisor: %d\n",divisor);
   printf("cociente: %d\n",cociente);
   getch();
   return 0;
}
```

Siempre que se va a leer un valor es muy importante exhibir un letrero que informe al usuario qué es lo que se va a leer, de no hacerlo así, no se sabría que está ocurriendo.

Veamos un programa para calcular el área de un triángulo, dadas las coordenadas de sus vértices. El área se calcula utilizando la siguiente fórmula:

$$area = \frac{1}{2}\left(x_1y_2 + x_2y_3 + x_2y_3 - x_1y_3 - x_2y_1 - x_3y_2\right)$$

En este caso las variables que deberán leerse son `x1`, `x2`, `x3`, `y1`, `y2` y `y3`. El área la almacenaremos en la variable `area`. El programa es el siguiente:

```
//Programa para calcular el área de un triángulo
#include <stdio.h>
```

```
#include <conio.h>

int main()
{
   float x1,x2,x3,y1,y2,y3;  // coordenadas del triángulo
   float area;   // área del triángulo
   printf("Teclee las coordenadas del primer punto: ");
   scanf("%f%f",&x1,&y1);
   printf("Teclee las coordenadas del segundo punto: ");
   scanf("%f%f",&x2,&y2);
   printf("Teclee las coordenadas del tercer punto: ");
   scanf("%f%f",&x3,&y3);

   area = (x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2.0;
   printf("área: %f\n",area);
   getch();
   return 0;
}
```

Muchos programas siguen un esquema simple que consiste en la lectura de algunas variables, la evaluación de expresiones y el despliegue de los resultados obtenidos. Para este tipo de problemas se puede seguir el siguiente algoritmo general.

1. Determinar las constantes, variables y su tipo
2. Escribir la declaración de variables
3. Escribir las sentencias de entrada
4. Escribir las expresiones para realizar los cálculos
5. Escribir las sentencias de salida
6. Probar con algunos valores de entrada

El siguiente ejemplo ilustra este algoritmo:

Escriba un programa que lea el radio de un círculo e imprima su perímetro y área. Defina el valor de π como 3.141592.

1. Determinar las constantes, variables y su tipo:

    Constates: π (3.141592) - número real
    Variables de entrada: radio – número real
    Variables de salida: area – número real, perimetro – número real

2. Escribir la declaración de variables

    Declaración de constante: `float pi = 3.141592;`
    Declaración de variables de entrada: `float radio;`
    Declaración de variables de salida: `float area, perimetro;`

3. Escribir las sentencias de entrada

```
printf(“Escriba el radio del círculo: “);
scanf(“%f”,radio);
```

4. Escribir las expresiones para realizar los cálculos

```
area = pi*radio*radio;
perimetro = 2*pi*radio;
```

5. Escribir las sentencias de salida

```
printf(“el área del círculo es: %f\n“,area);
printf(“el perímetro del círculo es: %f\n“,perimetro);
```

6. Probar con algunos valores de entrada

Para probar el algoritmo hay que simular su ejecución con valores de entrada aceptable. Para esto es conveniente hacer una tabla donde se especifique en cada paso los valores de las variables y los letreros y valores impresos en la pantalla. Hacemos una tabla con las siguientes columnas: paso, una columna para cada variable y salida.

Paso	radio	area	perimetro	SALIDA
3	5	-	-	“Escriba el radio del círculo: ”
4	5	78.54	31.42	-
5	5	78.54	31.42	“el área del círculo es: 78.54” “el perímetro del círculo es: 31.42”

En la tabla se muestra el ejemplo en que se ha tecleado como entrada un 5 para el valor del radio. Es muy conveniente probar el algoritmo para asegurar que resuelve el problema como realmente queremos. El programa completo es el siguiente:

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

main(){
    float pi = 3.1415926535; // valor de pi
    float radio; // radio del círculo, variable de entrada
    float area, perimetro; // área y perímetro del círculo
    printf("Escriba el radio del círculo: ");
    scanf("%f",&radio);
    area = pi*radio*radio;
    perimetro = 2*pi*radio;
    printf("el área del círculo es: %f\n",area);
    printf("el perímetro del círculo es: %f\n",perimetro);
    getch();
```

```
    return 0;
}
```

Se debe tener cuidado al escribir expresiones. Si las expresiones incluyen números de punto flotante, hay que asegurarse de que las constantes que aparezcan sean también de punto flotante. Por ejemplo 3/5*4.5 se evalúa como 0, ya que 3/5 da como resultado un cero. Lo correcto es 3.0/5*4.5 o 3/5.0*4.5 para obtener el valor correcto.

Problemas propuestos

3.5.1. Escriba un programa para calcular la magnitud de la fuerza entre dos cargas eléctricas dadas las coordenadas de las cargas y la magnitud de cada carga. La fuerza se calcula con la expresión siguiente donde $k = 8.99 \times 10^9$.

$$F = k\frac{q_1 q_2}{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Donde q_1 y q_2 son los valores de las cargas 8positivas o negativas), (x_1, y_1) y (x_2, y_2) son las coordenadas de q_1 y q_2 respectivamente. Deberá leer los valores de: x_1, y_1, x_2, y_2, q_1 y q_2.

3.5.2. Escriba un programa que lea un número entero representando segundos y despliegue el equivalente en días, horas, minutos y segundos. Por ejemplo: 34,622 segundos es equivalente a 0 días, 9 horas, 37 minutos, 2 segundos.

3.5.3. Escribir un programa para obtener la hipotenusa y los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

3.5.4. La famosa ecuación de Eisntein para la conversión de una masa m en energía viene dada por la fórmula $E = mc^2$, donde c es la velocidad de la luz igual a 2.997925×10^{10}cm/s. Escribir un programa que lea una masa en gramos y obtenga la cantidad de energía en Joules producida de acuerdo con la ecuación de Einstein.

3.6. Biblioteca matemática `math.h`

La biblioteca `math.h` contiene las declaraciones de las funciones trigonométricas, exponenciales, etc. Un breve listado es el siguiente, en el apéndice se da un listado más completo:

Función	Significado
`ceil(x)`	redondea al entero más pequeño no menor que x.
`cos(x)`	coseno de x.
`exp(x)`	e^x
`fabs(x)`	valor absoluto de x.
`floor(x)`	redondea al entero más grande no mayor que x.
`log(x)`	logaritmo natural de x.

`log10(x)`	logaritmo base 10 de x.
`pow(x,y)`	x^y.
`sin(x)`	seno de x.
`sqrt(x)`	raíz cuadrada de x.
`tan(x)`	tangente de x.

El siguiente ejemplo calcula el área de un triángulo conocidas las longitudes de los lados *a*, *b*, *c* utilizando la fórmula:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

Donde *s* es el semiperímetro del triángulo, o sea,

$$s = \frac{a+b+c}{2}$$

1. Determinar las constantes, variables de entrada y de salida y su tipo

    Variables de entrada: a, b, c de tipo real
    Variable intermedia: s de tipo real
    Variable de salida: areaTriangulo de tipo real

2. Escribir la declaración de variables

```
float a, b, c;/* lados del triángulo */
float s;      /*semiperímetro */
float areaTriangulo;   /* área del triángulo */
```

3. Escribir las sentencias de entrada

```
printf("Teclee las longitudes de los lados: ");
scanf(“%f%f%f”,&a,&b,&c);   /* lee los lados */
```

4. Escribir las expresiones para realizar los cálculos

```
s = (a + b + c)/2.0;
areaTriangulo = sqrt(s*(s-a)*(s-b)*(s-c));
```

5. Escribir las sentencias de salida

```
printf("área: %f\n",areaTriangulo);
```

6. Probar con algunos valores de entrada

Paso	a	b	c	s	areaTriangulo	Salida
3	3	4	5	-	-	“las longitudes de los

						lados:”
4	3	4	5	6	$\sqrt{6 \cdot 3 \cdot 2 \cdot 1} = 6$	
5	3	4	5	6	6	“área: 6”

El programa completo es el siguiente:

```
//Programa para calcular el área de un triángulo
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main()
{
   float a,b,c;  /* lados del triángulo */
   float s;      /*semiperímetro */
   float areaTriangulo;   /* área del triángulo*/
   printf("Teclee las longitudes de los lados: ");
   scanf("%f%f%f",&a,&b,&c);   /* lee los lados */
   s = (a + b + c)/2.0;
   areaTriangulo = sqrt(s*(s-a)*(s-b)*(s-c));
   printf("área: %f\n",areaTriangulo);
   getch();
   return 0;
}
```

El siguiente programa calcula la altura máxima, el alcance y el tiempo de vuelo de un proyectil en tiro parabólico. Se dan como datos de entrada la velocidad inicial del proyectil y el ángulo que hace la velocidad inicial con el eje x. Sea h la altura máxima, R el alcance y T el tiempo de vuelo, estos se calculan con las siguientes fórmulas:

$$h = \frac{v_0^2 sen^2\theta}{2g} \qquad R = \frac{v_0^2 sen2\theta}{g} \qquad T = \frac{2v_0 sen\theta}{g}$$

Constantes:
PI – 3.14159265
g - 9.81

Los datos de entrada son la velocidad inicial y el ángulo del disparo:
v0 – velocidad inicial (tipo float)
ang – ángulo del disparo (tipo float)

Los datos de salida son:
h – altura máxima (tipo float)
R – alcance máximo (tipo float)
T – tiempo de vuelo (tipo float)

La velocidad inicial la supondremos en m/s. El ángulo del tiro lo supondremos en grados. Las funciones trigonométricas suponen el argumento en radianes, es necesario convertir el ángulo de grados a radianes por esta razón.

Algoritmo Tiro parabólico. Calcula la altura máxima, el alcance y el tiempo de vuelo de un proyectil en tiro parabólico.

1. [Leer datos]
   Leer(v0, ang)
2. [Convertir a radianes]
   ang = ang*PI/180
3. [Calcular resultados]
   h = v0*v0*sin(ang)*sin(ang)/2/g;
   R = v0*v0*sin(2*ang)/g;
   T = 2*v0*sin(ang)/g;
4. [Desplegar resultados]
   Escribe(h, R, T)
5. [Terminar]
   Salir

El programa completo es el siguiente:

```
/*Calcula alcance, altura máxima y tiempo de vuelo en un tiro
parabólico*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

main(){
  float PI = 3.14159265;
  float g = 9.81;
  float v0,ang;/* datos de entrada*/
  float h,R,T; /* datos de salida*/
  /*Leer datos*/
  printf("TIRO PARABOLICO\n");
  printf("Teclee velocidad inicial (en m/s): ");
  scanf("%f",&v0);
  printf("Teclee angulo del tiro (en grados): ");
  scanf("%f",&ang);
  /*Convertir a radianes*/
  ang = ang*PI/180;
  /*Calcular resultados*/
  h = v0*v0*sin(ang)*sin(ang)/2/g;
  R = v0*v0*sin(2*ang)/g;
  T = 2*v0*sin(ang)/g;
  /*Desplegar resultados*/
  printf("altura maxima: %f m\n",h);
  printf("alcance maximo: %f m\n",R);
```

```
  printf("tiempo de vuelo: %f s\n",T);
  getch();
  /*Terminar*/
  return 0;
}
```

Problemas propuestos

3.6.1. Dadas las siguientes variables en C escriba una sentencia para leer sus valores desde el teclado en el orden que usted quiera.

```
float a,b;
int m,n;
```

3.6.2. Suponga que un usuario escribe las siguientes líneas ¿Cuál sentencia en C procesan la respuesta del usuario sin errores?

```
3.4 5 6.4

scanf(“%d%f%d”,&a, &b, &c);
scanf(“%d%f%f”,&a, &b, &c);
scanf(“%f%d%f”,&a, &b, &c);
```

3.6.3. Escriba un programa convierta una velocidad en metros por segundo a kilómetros por hora.

3.6.4. Escriba un programa que lea una cantidad de tiempo en minutos y la convierta a horas, días, semanas y meses.

3.6.5. Escriba un programa que lea una distancia en años luz y la convierta en kilómetros, millas y parsecs (1 milla = 1.609 km, 1 parsec = 150,000,000 km)

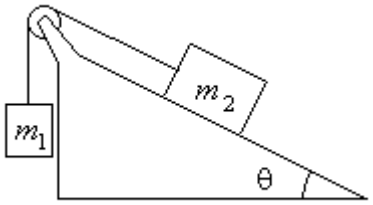
3.6.6. Escriba un programa que convierta la altura de una persona de metros a pies y pulgadas. Despliegue el resultado separando pies y pulgadas, ejemplo: 1.78 m = 5 pies 10”, aproximadamente.

3.6.7. Escriba un programa para calcular la distancia entre dos puntos en el plano. Los puntos están especificados por sus coordenadas (x, y) leyendo los valores de x y y para los dos puntos. La distancia se calcula utilizando la expresión siguiente

$$\text{distancia} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3.6.8. La aceleración de dos cuerpos de masa m_1 y m_2, unidos por una cuerda en un plano inclinado como se muestra en la figura, está dada por la fórmula:

$$a = \frac{m_2 sen\theta - m_1}{m_1 + m_2} g$$



Donde g es la aceleración de la gravedad (9.8 m/s^2). Escriba un programa que lea los valores de las masas m_1 y m_2 y el ángulo θ (en grados) del plano inclinado y calcule la aceleración de los cuerpos.

3.6.9. La fórmula de Lorentz para la contracción de la longitud de un cuerpo en función de su velocidad está dada por $L = \sqrt{1 - v^2/c^2}\,L'$, donde L' es su longitud en reposo y c es la velocidad de la luz (2.9979 x 10^8 m/s). Escriba un programa que lea la velocidad del objeto y calcule el porcentaje de contracción de acuerdo a la fórmula de Lorentz.

3.6.10. Escriba un programa que determine la velocidad inicial con que debe lanzarse un proyectil para alcanzar una altura h si se lanza con un ángulo θ, estos datos dados por el usuario.

3.6.11. Usando la fórmula de Einstein del problema 3.5.5 escriba un programa que lea la cantidad de energía utilizada en algún proceso y calcule la cantidad de masa que se convirtió en energía.

Capítulo 4. Control condicional

El control condicional permite alterar la secuencia de ejecución de las sentencias de un programa. De esta manera se puede ejecutar o no una o más sentencias dependiendo del valor de alguna expresión de control. En este capítulo revisaremos las formas de tomar decisiones en un programa.

4.1. Operadores relacionales

Los operadores relacionales permiten comparar dos números o expresiones. Estos operadores generan solo dos valores: 0 o 1. En el lenguaje C el valor 1 representa el valor verdadero y 0 representa el valor falso. Todos los operadores relacionales son operadores binarios, es decir, se aplican a dos operandos. La siguiente tabla resume los operadores relacionales:

Operando	Significado
`==`	Igualdad, 1 si los dos operandos son iguales, 0 en otro caso.
`>`	Mayor que, 1 si el primer operando es mayor que el segundo, 0 en otro caso
`<`	Menor que, 1 si el primer operando es menor que el segundo, 0 en otro caso
`>=`	Mayor o igual a, 1 si el primer operando es mayor o igual al el segundo, 0 en otro caso
`<=`	Menor o igual a, 1 si el primer operando es menor o igual al el segundo, 0 en otro caso
`!=`	Diferente a, 1 si el primer operando es diferente al segundo, 0 en otro caso

Algunos ejemplos de comparaciones son los siguientes:

```
1 > 2           =    0          falso
3 < 5           =    1          verdadero
(7 – 4) == 3    =    1          verdadero
17 >= (5 + 12)  =    1           verdadero
i = 3; j = 7;
i * j != 21     =    0          falso
float a=0.1;
(3*a–0.3)==0    =    0          falso (OJO con los reales)
3 > 1 > 0       =    1           verdadero
```

Note que si probamos igualdad de números reales, el resultado de la comparación puede ser no esperado, esto es debido a que la representación de un número real no es exacta. Se pueden utilizar expresiones relacionales para seleccionar entre dos o más expresiones a evaluar. Por ejemplo:

```
(3*a + b)*(a<0) + (5*a – b)*(a>=0)
```

Será igual a `3*a+ b` si `a` es menor que cero, e igual a `5*a – b` si `a` es mayor o igual a cero. Por ejemplo se `a = –3` y `b= 4`, la expresión se evaluara como:

```
(3*(–3)+4)*(–3<0)+(5*(–3)–4)*(–3>=0)=(–9+4)*(1)+(–15–4)*(0)=
–5
```

Si `a = 2` y `b = 4`, el valor será

```
(3*(2)+4)*(2<0)+(5*(2)–4)*(2>=0)=(6+4)*(0)+(10–4)*(1)=6
```

Los operadores relacionales tienen una prioridad de evaluación inferior a los operadores aritméticos. La expresión 3 + 7 >= 5 – 4 se evalúa como 10 >= 1.

Ejemplo de conversión con expresiones relacionales

El siguiente programa convierte una temperatura leída desde el teclado de la escala Celsius a la Farenheit y viceversa. Se le solicita al usuario el tipo de conversión que desea realizar:

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

main(){
  float tempOriginal,tempConvertida;
  int opcion;
  printf("Teclee temperatura: ");
  scanf("%f",&tempOriginal);
  printf("Tipo de conversión (1 - C a F, 2 - F a C): ");
  scanf("%d",&opcion);  //solicita tipo de conversión
  tempConvertida = (opcion==1)*(9*tempOriginal/5+32)+
                   (opcion==2)*(5.0/9*(tempOriginal-32));
  printf("Valor convertido: %f\n",tempConvertida);
  getch();
}
```

La expresión `opcion==`1 es verdadera, es decir, vale 1 si la variable `opcion` vale 1 y la expresión `opcion==`2 vale 1 si `opcion` es igual a 2. De la manera mostrada podemos seleccionar entre una serie de expresiones. Por ejemplo, si deseamos convertir de centígrados a kelvin, agregamos el término siguiente a la expresión.

```
(opcion==3)*(tempOriginal+273.16)
```

Problemas propuestos

4.1.1. Escriba una sentencia para convertir de centímetros a pulgadas si una variable `opcion` vale 1 y convertir de pulgadas a centímetros si `opcion` vale 2.

4.1.2. Modifique el programa de conversión de temperaturas para incluir conversiones de centígrados a Kelvin y Kelvin a centígrados.

4.1.3. Escriba un programa para convertir cm a pulgadas, pulgadas a cm, pies a cm y cm a pies, utilizando solamente expresiones relacionales (1 pulgada = 2.54 cm, 1 pie = 12 pulgadas).

4.2. Sentencia `if`

La sentencia `if` permite decidir ejecutar una o más instrucciones dependiendo del valor de una expresión. La sintaxis de la sentencia `if` es:

```
if( condición )
   instrucción o bloque;
```

Un bloque está formado por una serie de instrucciones encerrado entre llaves {}. La condición es cualquier expresión que genere un valor numérico real o entero. La instrucción o bloque se ejecutará si la condición toma un valor diferente de cero. En el lenguaje algorítmico usaremos la construcción siguiente.

SI condición ENTONCES
    sentencias

El programa siguiente determina si un número tecleado es positivo o negativo:

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

main(){
  float numero;
  printf("Teclee un número: ");
  scanf(“%d”,&numero);
  if(numero>=0)
    printf("número positivo\n");
  getch();
}
```

Para determinare si un número es divisible entre otro verificamos si el residuo de la división es cero. El siguiente programa verifica si un número es divisible entre 7 o no:

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

main(){
  int numero;
```

```
  printf("Teclee un número: ");
  scanf(“%d”,&numero);
  if(numero%7==0)
    printf(“%d es divisible entre 7\n”,numero);
  getch();
}
```

Supongamos el siguiente problema. Se realiza un examen con N preguntas y se desea saber si un alumno aprobó o no el examen con base en las respuestas correctas que obtuvo.

Diseñaremos un algoritmo para resolver este problema. El algoritmo en primara aproximación realizara los siguientes pasos.

1. Leer el total de preguntas
2. Leer el total de aciertos
3. Determinar si aprobó o reprobó

Necesitaremos dos variables enteras, una para el total de preguntas y otra para los aciertos. Sea N el total de preguntas y A el total de aciertos. Supondremos que para aprobar se requiere de un 0.6 de respuestas correctas para aprobar. También requeriremos de una variable real FRACCION para almacenar la fracción de respuestas correctas. Una versión más detallada del algoritmo es la siguiente.

Algoritmo Aprobado. Este algoritmo determina si un alumno aprobó un examen, el usuario teclea el número de preguntas y el total de aciertos. Se imprimirá el letrero de aprobado si el cociente de aciertos es mayor o igual a 0.6. En N se almacena el total de preguntas y en A el número de aciertos, FRACION almacena el cociente de A y N.

```
1. [Leer el total de preguntas]
        Leer(N)
2. [Leer el número de aciertos]
        Leer(A)
3. [Calcular la fracción de respuestas correctas]
        FRACCION = A/N
4. [Aprobó?]
        SI FRACCION>=0.6 ENTONCES
                Escribe(“Aprobado”)
5. [Termina]
        Salida
```

Hay que tener mucho cuidado al traducir el programa a C. La instrucción FRACCION = A/N puede dar como resultado que siempre obtengamos un valor igual a 0, esto debido a que al dividir dos números enteros el resultado es un número entero y el único resultado entero que podemos obtener es 0 debido a que N es mayor que A. Para evitar esto, usaremos un cambio de tipo (typecast) que convierta a números reales los valores enteros y haga la división correctamente. El programa es el siguiente.

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

main(){
  int n,a;
  float fraccion;
  /*Leer el total de preguntas*/
  printf("Escriba el total de preguntas: ");
  scanf("%d",&n);
  /*Leer el número de aciertos*/
  printf("Escriba el total de aciertos: ");
  scanf("%d",&a);
  /*Calcular la fracción de respuestas correctas*/
  fraccion = (float)a/n;
  /*Aprobó? */
  if(fraccion>=0.6)
    printf("Aprobado con %f porciento",fraccion*100);
  getch();
}
```

La sintaxis para cambiar de tipo es la siguiente:

```
(tipo nuevo) expresión;
```

Problemas propuestos

4.2.1. Escriba un programa que lea dos números enteros luego decida si el primero es divisible entre el segundo e informe el resultado.

4.2.2. Escriba un programa que lea tres números reales y decida si forman un triángulo. Suponga que los números se leen de mayor a menor.

4.2.3. Escriba un programa que lea tres números reales representando los lados de un triángulo y decida si el triángulo es rectángulo o no. Suponga que los números se leen de mayor a menor.

4.2.4. Escriba un programa para sumar dos cantidades en horas, minutos y segundos. Imprima los resultados en formato normalizado, es decir, hh:mm.ss, donde 0<=mm<=59 y 0<=ss<=59.

4.3. Sentencia if completa

La sentencia if completa consta de una parte que se ejecuta cuando la condición es 1 y otra cuando es 0. La sintaxis es:

```
if( condición )
```

```
   instrucción o bloque;
else
   instrucción o bloque;
```

En el lenguaje algorítmico usaremos la construcción siguiente.

SI condición ENTONCES
    sentencias
SINO
    sentencias

El siguiente ejemplo es un programa que determina si dos rectas se intersecan o no, y si se intersecan determina las coordenadas (x, y) de la intersección. Suponga que las rectas se representan mediante una ecuación de la forma $y = mx + b$. Supondremos que el usuario introduce los valores de m y b para cada recta. Para que dos rectas se intercepten es necesario que las pendientes sean diferentes, usaremos esta condición para resolver el problema.

Variables necesarias: todas de tipo real.
pendiente1, ordenada1 – pendiente y ordenada al origen de la primera recta.
pendiente2, ordenada2 – pendiente y ordenada al origen de la segunda recta.
x, y – coordenadas x,y de la intersección.

Algoritmo Intersección. Determina si dos rectas representadas mediante la expresión y = mx + b se intersecan. Para que esto suceda las pendientes deben ser diferentes. Las pendientes son pendiente1 y pendiente2 y las ordenadas son ordenada1 y ordenada2.

1. [Leer datos de las dos rectas]
    Leer(pendiente1, ordenada1)
    Leer(pendiente2, ordenada2)
2. [Determina intersección, si existe]
    SI pendiente1 diferente de pendiente2 ENTONCES
        x = (ordennada2 – ordenada1)/(pendiente2 - pendiente1)
        y = pendiente1 * x + ordenada1
    SINO
        Escribe(“Las rectas no se intersecan”)
3. [Termina]
    Salir

El programa es el siguiente:

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

int main()
{
  float pendiente1, pendiente2, ordenada1 ,ordenada2,x,y;
```

```
  /*Leer datos de las dos rectas*/
  printf("teclee la pendiente y ordenada al origen 1: ");
  scanf("%f%f",&pendiente1,&ordenada1);
  printf("teclee la pendiente y ordenada al origen 2: ");
  scanf("%f%f",&pendiente2,&ordenada2);
  /*Determina intersección, si existe*/
  if(pendiente1 != pendiente2){
    x = (ordenada2 - ordenada1)/( pendiente1- pendiente2);
    y = pendiente1*x+ ordenada1;
    printf("Las rectas se interceptan en: %f, %f",x,y);
  }
  else
    printf("Las rectas no se interceptan...");
  getch();
}
```

Suponga que deseamos saber cual es el mayor de dos números. Una forma de resolver este problema es utilizar una construcción if-else. El algoritmo es el siguiente.

Algoritmo Mayor. Despliega el mayor de dos números introducidos por el usuario. Uno de los números es A y el otro B.

1. [Leer los valores]
   Leer(A, B)
2. [despliega el mayor]
   SI A>B ENTONCES
      Escribe(A)
   SINO
      Escribe(B)
3. [Termina]
   Salir

El programa es el siguiente.

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

main(){
  int a,b;
  /*Leer los valores*/
  printf("Introduzca dos enteros: ");
  scanf("%d %d",&a,&b);
  /*despliega el mayor*/
  if(a>b)
    printf("%d",a);
  else
    printf("%d",b);
```

```
  getch();
}
```

Esto mismo puede hacerse con una expresión que incluya expresiones relacionales si cambiamos el paso 2 por

2. [Despliega el mayor]
    Escribe((A>B)*A+(B>A)*B)

Note que el algoritmo con esta modificación imprimirá 0 cuando se den dos valores iguales. Para corregir este caso hay que poner el operador >= en alguna de las dos expresiones relacionales. Note que el algoritmo solo funciona para un lenguaje como C en el que los valores booleanos son los enteros 0 y 1, en otros lenguajes los valores booleanos pueden ser tipos predefinidos no enteros o pueden ser enteros con diferentes valores para verdadero y falso, como -1 y 0 o cualquier otro valor. Escriba el siguiente programa y pruébelo.

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

main(){
  int a,b;
  printf("Introduzca dos enteros: ");
  scanf("%d %d",&a,&b);
  printf("%d",(a>b)*a+(b>a)*b);
  getch();
}
```

Problemas propuestos

4.3.1. Escriba un programa que lea un número e imprima el letrero “número positivo” si el número leído es positivo y el letrero “número negativo” si no lo es.

4.3.2. Escriba un programa que una opción para convertir de cm a pulgadas, de pulgadas a cm, luego lea la cantidad a convertir y despliegue el resultado.

4.3.3. Escriba un programa que lea tres números y diga cuales son pares y cuales impares.

4.3.4. Escriba un programa para encontrar la recta perpendicular a una recta dada en un punto específico. Suponga que la recta dada tiene pendiente m y ordenada al origen b, y el punto tiene coordenadas x,y. Considere el caso en que la pendiente dada es cero.

4.3.5. En una elección el candidato que obtiene más votos que la suma de votos de los demás gana la elección, sino se hace una segunda ronda de elecciones. Desarrolle un algoritmo que lea el número de votos de una elección entre 4 candidatos y determine si habrá o no una segunda vuelta. Suponga que el número de votos se leen ordenadamente de mayor a menor.

4.4. Operadores lógicos

Los operadores lógicos nos permiten construir condiciones más complejas. Existen cuatro operadores lógicos en C. Las siguientes son sus tablas de verdad.

Operador O lógico (`||`):

Expresión1	Expresión2	Expresión1 `||` Expresión2
0	0	0
0	1	1
1	0	1
1	1	1

Operador Y lógico (`&&`):

Expresión1	Expresión2	Expresión1 && Expresión2
0	0	0
0	1	0
1	0	0
1	1	1

Operador O exclusivo (`^` este operador funciona a nivel de bit):

Expresión1	Expresión2	Expresión1 ^ Expresión2
0	0	0
0	1	1
1	0	1
1	1	0

Operador complemento lógico (`!`)

Expresión	!Expresión
0	1
1	0

La siguiente expresión es verdadera para valores de la variable `x` mayores que cero y menores que 20:

```
x > 0 && x <20
```

En el siguiente fragmento se despliegan solo los múltiplos de 7 o cuadrados perfectos

```
if(num % 7 == 0 || sqrt(num)-floor(sqrt(num))==0)
     printf(“%d\n”,num);
```

La parte `num%7==0` determina si el número es múltiplo de 7, y la parte `sqrt(num)-floor(sqrt(num)` resta la parte entera de la raíz cuadrada de `num` de la raíz cuadrada de `num`, si estas son iguales, el resultado será cero siendo `num` un cuadrado perfecto, de otra forma el resultado será distinto de cero.

Los operadores lógicos tienen menor precedencia que los operadores relacionales. Si existe alguna duda del orde en que se evaluará una expresión, es necesario que se utilicen paréntesis. El siguiente ejemplo imprimirá dos unos.

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

main(){
  int a,b,c,d;
  a = 5; b = 3; c = -2;
  d = a+b>c && b>c;
  printf("%d\n",d);
  d = !a==b-c || b>c;
  printf("%d\n",d);
  getch();
}
```

Determinación del mayor de tres números

Para determinar el mayor de tres números podemos utilizar condiciones compuestas. Para esto debemos comparar, con una sentencia **`if`**, el primero de los números con los otros dos. Después repetir la primera sentencia para el segundo y tercer número. Suponga que `a` es el primer número, `b` el segundo y `c` el tercero. El código en C es el siguiente.

```
if(a>b&&a>c)
  printf("%d es el mayor\n",a);
if(b>a&&b>c)
  printf("%d es el mayor\n",b);
if(c>b&&c>a)
  printf("%d es el mayor\n",c);
```

Note que este código supone que los números son todos diferentes, si no lo son, no se imprimirá ningún valor. La solución se puede mejorar utilizando otra variable y cambiando > por >=. Veamos el código:

```
if(a>=b&&a>=c)
  max = a;
if(b>=a&&b>=c)
  max = b;
if(c>=b&&c>=a)
```

```
  max = c;
printf(“%d es el mayor\n”,max);
```

Este código funciona en todos los casos.

Una aplicación muy común de las condiciones compuestas es para probar si un valor se encuentra en un intervalo determinado. Suponga que deseamos saber si la variable `edad` se encuentra entre 12 y 18. Una forma de probar es mediante la siguiente sentencia:

```
if(edad>=12 && edad<=18)
```

La siguiente sentencia verifica si un valor `x` es menor que 10 o mayor que 50 y además si es un número par:

```
if(x<10 && x>50 && x%2==0)
```

Las condiciones compuestas pueden llegar a ser bastante complejas. Es conveniente escribir expresiones que sean fáciles de entender para mantener la legibilidad de los programas y facilitar su depuración. En la siguiente sección estudiaremos otra forma de tomar decisiones complejas mediante el uso de anidamiento.

Problemas propuestos

4.4.1. Suponga que `a = 3`, `b = 5` y `c = -3` ¿cuáles de las siguientes condiciones son verdaderas?

a) `3*a <= 2*b || 2*b != 10`
b) `5*a–2*b–c>=10&&!(b<c)`
c) `(2*b+3*a>20)&&(c<0)`
d) `!(a+c<=0)||(2*a+b==8)`
e) `!(4*a+c<=20)||(3*b+5>21)`

4.4.2. Escriba sentencias que sean verdaderas para:
x = 1, 2, 4, 8
x = 7, 14, 21, …, 49
x = 1, 2, 3, 6, 7, 8, 9
x = …, -5, -3, -1, 0, 2, 4, 6, …

4.4.3. Escriba un programa que lea e imprima el mayor y el menor de tres números. Suponga que se leen en desorden. Si se introducen valores repetidos, deberá imprimir el mayor y menor una sola vez. Utilice condiciones compuestas.

4.4.4. Escriba un programa que lea un número y determine si el número es un cuadrado perfecto o un cubo perfecto.

4.4.5. Escriba un programa que lea tres números *a*, *b* y *c* y determine si los números satisfacen las siguientes expresiones: $a^3 - 4b^4 + 5c^2 < 300$ y $4a^2 - 3b^2 + c > 120$.

4.4.6. Dada la ecuación de una cónica $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ el discriminante permite saber de que tipo de cónica se trata de acuerdo con la siguiente tabla

discr	genero	Radicando	Especie de curva
< 0	elipse	trinomio de raíces reales	Elipse real
		trinomio con raíz doble	Elipse degenerada
		trinomio de raíces complejas	Elipse imaginaria
= 0	parábola	binomio o monomio con término en x	Parábola real
		sin término en x	Parábola que ha degenerado en:
		N>0	Dos rectas paralelas
		N = 0	Dos rectas coincidentes
		N < 0	Parábola imaginaria
> 0	hipérbola	trinomio de raíces reales	Hipérbola que corta el diámetro
		trinomio con raíz doble	Hipérbola que ha degenerado en 2 rectas que se cortan
		trinomio de raíces complejas	Hipérbola que no corta el diámetro

Donde discr = B*B–4*A*C y N es el término cuadrático $(B^2 - 4AC)x^2+2(BE–2CD)x+E^2–4CF$. Escriba un programa que dados los coeficientes A, B, C, D, E y F determine de que tipo de cónica se trata. Utilice condiciones compuestas.

4.4.7. En una elección el candidato que obtiene más votos que la suma de votos de los demás gana la elección, sino se hace una segunda ronda de elecciones. Desarrolle un algoritmo que lea el número de votos de una elección entre 4 candidatos y determine si habrá o no una segunda vuelta. El número de votos se leerán en desorden.

4.5. Anidamiento de sentencias `if`

Una sentencia if puede tener otra sentencia if como su sentencia verdadera o falsa. Esto da como resultado que un if dependa de otro if. A esta situación se le llama anidamiento. Considere el siguiente ejemplo:

```
if(a > 5)
  if(a < 10)
    printf("%d\n",a);
```

El valor de `a` se imprimirá solo si es mayor que 5 y es menor que 10, es decir, para `a` = 6, 7, 8 y 9. Note que lo mismo se puede obtener para la siguiente sentencia compuesta:

```
if(a > 5 && a < 10)
  printf(“%d\n”,a);
```

El problema de determinar el mayor tres números puede resolverse utilizando anidamiento. Una solución es la siguiente:

```
if(a>b)
  if(a>c)
    printf(“%d es el mayor\n”,a);
  else
    printf(“%d es el mayor\n”,c);
else
  if(b>c)
    printf(“%d es el mayor\n”,b);
  else
    printf(“%d es el mayor\n”,c);
```

El siguiente fragmento despliega el mayor utilizando condiciones compuestas y anidamiento:

```
if(a >= b && a >= c)
  printf(“%d\n”,a);
else
  if(b >= a && b >= c)
    printf(“%d\n”,b);
  else
    if(c >= a && c >= b)
      printf(“%d\n”,c);
```

Es importante el formatear el texto para que el programa sea legible. El siguiente es un listado equivalente al anterior pero sin el sangrado adecuado. Note que es difícil establecer el anidamiento. El compilador de C ignora el formateo del código pero es difícil para una persona leer código mal formateado.

```
if(a >= b && a >= c)
printf(“%d\n”,a);
else
if(b >= a && b >= c)
printf(“%d\n”,b);
else
if(c >= a && c >= b)
printf(“%d\n”,c);
```

Solución de una ecuación cuadrática

Mediante sentencias `if` podemos resolver fácilmente una ecuación cuadrática. Recuerde que la ecuación cuadrática de la forma

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

puede tener tres tipos de soluciones: dos raíces reales, una raíz real positiva y dos raíces complejas conjugadas. La naturaleza de las soluciones depende del valor del discriminante de la ecuación

$$d = b^2 - 4ac$$

El siguiente algoritmo Cuadrática es una posible solución al problema.

Algoritmo Cuadrática. Algoritmo para resolver una ecuación cuadrática con coeficientes reales. Los coeficientes son a, b y c. Las raices reales son x1 y x2. La parte real es parteReal y la parte imaginaria es parteImaginaria.

```
1. [Leer coeficientes]
        Leer(a, b, c)
2. [Calcular el discriminante]
        d = b*b-4*a*c
3. [Determinar tipo de solución]
        SI d>0 ENTONCES
                x1 = (-b+sqrt(d))/2/a
                x2 = (-b-sqrt(d))/2/a
                Escribe(x1, x2)
        SINO
                SI d=0 ENTONCES
                        x1 = -b/2/a
                        Escribe(x1)
                SINO
                        parteReal = -b/2/a
                        parteImaginaria = -sqrt(-d)/2/a
                        Escribe(parteReal, parteImaginaria
4. [Termina]
        Salir
```

El programa en C es el siguiente.

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

main(){
  float a,b,c,d,x1,x2,ParteReal,ParteImaginaria;
  printf("Programa para resolver una cuadratica\n");
  /*Leer coeficientes*/
  printf("Dame los coeficientes de la ecuacion (a,b,c): ");
  scanf("%f %f %f",&a,&b,&c);
  d = b*b-4*a*c;
```

```
  /*Determinar tipo de solución*/
  if( d>0 ){
    printf("Dos raices reales diferentes.\n");
    x1 = (-b+sqrt(d))/2/a;
    x2 = (-b-sqrt(d))/2/a;
    printf("x1= %.3f, x2= %.3f\n",x1,x2);
  }else if( d==0 ){
    printf("Una raiz real repetida.\n");
    x1 = -b/2/a;
    printf("x1= %.3f\n",x1);
  }else{
    printf("Raices complejas.\n");
    ParteReal = -b/2/a;
    ParteImaginaria = sqrt(-d)/2/a;
    printf("Parte Real=%.3f Parte Imaginaria= %.3f",
ParteReal, ParteImaginaria);
  }
  getch();
}
```

Ejemplo de dos vehículos

Dos vehículos se mueven a diferentes velocidades en t = 0 con aceleraciones constantes y diferentes y se encuentran en diferente posición. Haga un programa que determine en que tiempos y posiciones ambos vehículos coinciden.

Sea x_{01}, v_1 y a_1 la posición inicial, la velocidad y la aceleración con que se mueve el primer vehículo y x_{02}, v_2 y a_2 la posición inicial, la velocidad y la aceleración en que se mueve el segundo vehículo, respectivamente. Las ecuaciones de movimiento de cada cuerpo son:

$$x_1 = x_{01} + v_1 t + 0.5 a_1 t^2$$

$$x_2 = x_{02} + v_2 t + 0.5 a_2 t^2$$

Los tiempos pedidos se encuentran cuando $x_1 = x_2$ o

$$x_{01} + v_1 t + 0.5 a_1 t^2 = x_{02} + v_2 t + 0.5 a_2 t^2$$

Simplificando se llega a

$$0.5 (a_1 - a_2) t^2 + (v_1 - v_2) t + x_{01} - x_{02} = 0$$

Esta es una ecuación cuadrática de la forma

$$a x^2 + b x + c = 0$$

Donde:

$a = 0.5\ (a_1 - a_2)$ $b = (v_1 - v_2)$ $c = x_{01} - x_{02}$

Cuya solución es:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Esta tiene solución cuando $b^2 - 4\ a\ c >= 0$. Esto significa que si la ecuación tiene alguna solución real y positiva, los vehículos se encontrarán en el futuro, en caso contrario, los vehículos no se encontrarán. El algoritmo para resolver el problema es:

Algoritmo Vehículos. Determina el tiempo en que dos vehículos que se mueven con aceleración constante se intersecan. Solo nos interesan las soluciones con tiempos mayores que cero.

```
1. [Leer los valores del primer vehículo]
        Leer(x01, v1, a1)
2. [Leer los valores del segundo vehículo]
        Leer(x02, v2, a2)
3. [Calcular los coeficientes a, b y c]
        a = 0.5 * (a1 – a2)
        b = (v1 – v2)
        c =  x01 – x02
4. [Calcular el valor del discriminante]
        d = b*b – 4*a*c
5. [Determina si hay solución y la calcula]
        SI el discriminante es mayor o igual a cero ENTONCES
                t1 = (-b – sqrt(d)/2/a
                t2 = (-b + sqrt(d)/2/a
                SI t1>0 ENTONCES
                        pos1 = x01+v1*t1+0.5*a1*t1*t1;
                        Escribe(t1, pos1)
                FINSI
                SI t2>0 ENTONCES
                        pos2 = x01+v1*t2+0.5*a1*t2*t2;
                        Escribe(t1, pos1)
                FINSI
        SINO
                Escribe("No se encuentran")
        FINSI
6. [Terminar]
        Salir
```

En este ejemplo tenemos un SI-ENTONCES anidado. Si la ecuación cuadrática para calcular el tiempo tiene una solución real, entonces se calculan las soluciones, y si la

solución para el tiempo es positiva, se calcula la posición y se despliega. Suponemos que solo nos interesan los tiempos que son mayores que cero, un tiempo menor que cero indica que dadas las condiciones del problema el encuentro ocurrió en el pasado. El programa completo es el siguiente:

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

main(){
  //variables de entrada
  float x01,v1,a1,x02,v2,a2;
  //coeficientes de la cuadrática y discriminante
  float a,b,c,d;
  //tiempos y posiciones
  float t1,t2,pos1,pos2;
  //lectura de datos
  printf("ENCUENTRO DE DOS VEHICULOS\n");
  /*Leer los valores del primer vehículo*/
   printf("teclee la posición, velocidad y aceleración del
coche 1: ");
  scanf("%f%f%f",&x01,&v1,&a1);
  /*Leer los valores del segundo vehículo*/
   printf("teclee la posición, velocidad y aceleración del
coche 2: ");
  scanf("%f%f%f",&x02,&v2,&a2);
  printf("t01 = %f, x01 = %f, a1 = %f\n",x01,v1,a1);
  printf("t02 = %f, x02 = %f, a2 = %f\n",x02,v2,a2);
  /*Calcular los coeficientes a, b y c*/
  a = 0.5*(a1-a2);
  b = v1-v2;
  c = x01-x02;
  d = b*b - 4*a*c;
  /*Determina si hay solución y la calcula*/
  if(d>=0){
    t1 = (-b+sqrt(d))/2/a;
    t2 = (-b-sqrt(d))/2/a;
    if(t1>0){
      pos1 = x01+v1*t1+0.5*a1*t1*t1;
      printf("se encontrarán en t=%f s y x=%f m\n",t1,pos1);
    }
    if(t2>0){
      pos2 = x01+v1*t2+0.5*a1*t2*t2;
      printf("se encontrarán en t=%f s y x= %f m\n",t2,pos2);
    }
  }else
    printf("NO se encuentran\n");
```

```
  getch();
  return 0;
}
```

Problemas propuestos

4.5.1. Rescriba las siguientes sentencias de **if** anidados sin usar ningún anidamiento:

a)
```
if ( a < c )
  if ( b < c )
    x = y;
  else
    x = z;
else
  x = z;
```

b)
```
if ( a < b )
  if ( c >= b )
    x = z;
  else
    x = y;
else
  x = z;
```

4.5.2. Escriba las siguientes sentencias con condiciones compuestas utilizando sentencias if anidadas.

a)
```
if(a!=7&&b<16) x=3;
  if(a!=7&&b>=16) z=12;
  if(a==7) y = 6;
```

b)
```
if(x>5&&y<=10&&z>1)a=3;
  if(x>5&&y>10&&z>1)b=a;
  if(x>5&&y>10&&z<=1)a=5;
```

4.5.3. Escriba un programa que lea las edades de tres personas y despliegue los nombres de los tres en orden del más viejo al más joven. Suponga que los nombres son Juan, Pedro y Raúl. Utilice solo sentencias if anidadas con condiciones simples.

4.5.4. Una cierta función $f(x)$ está definida como sigue:

$$f(x)=\begin{cases} 3x^2-4x, si\ 0\le x<5 \\ 8x-x/5+15, si\ 5\le x<10 \\ x^2-17, si\ 10\le x<20 \end{cases}$$

Escriba un programa que lea un valor de x y calcule el valor de la función. Si $0 \le x < 20$, despliegue el valor calculado sino despliegue un valor de 0.

4.5.5. Escriba un programa que lea dos fechas representadas cada una de ellas como tres números enteros, uno para el día, otro para el mes y otro para el año, y calcule el número de años completos transcurridos entre la primera y la segunda fecha.

4.5.6. Un cajero automático dispone de billetes de 500, 200, 100, 50 y 20. Desarrolle un programa que lea una cantidad y exhiba el número de billetes de cada denominación para entregar ese importe.

4.5.7. Dada la ecuación de una cónica $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ el discriminante permite saber de que tipo de cónica se trata de acuerdo con la siguiente tabla

discr	genero	Radicando	Especie de curva
< 0	elipse	trinomio de raíces reales	Elipse real
		trinomio con raíz doble	Elipse degenerada
		trinomio de raíces complejas	Elipse imaginaria
= 0	parábola	binomio o monomio con término en x	Parábola real
		sin término en x	Parábola que ha degenerado en:
		N>0	Dos rectas paralelas
		N = 0	Dos rectas coincidentes
		N < 0	Parábola imaginaria
> 0	hipérbola	trinomio de raíces reales	Hipérbola que corta el diámetro
		trinomio con raíz doble	Hipérbola que ha degenerado en 2 rectas que se cortan
		trinomio de raíces complejas	Hipérbola que no corta el diámetro

Donde discr = B*B - 4*A*C y N es el término cuadrático $(B^2 - 4AC)x^2+2(BE-2CD)x+E^2-4CF$. Escriba un programa que dados los coeficientes A, B, C, D, E y F determine de que tipo de cónica se trata. Utilice sentencias anidadas.

4.6. Sentencia `switch`

Otra sentencia de decisión es la sentencia switch (selección). Ésta permite seleccionar entre una serie de alternativas posibles. La sentencia es controlada por una expresión de tipo entero. La sintaxis es:

```
switch (<expresión>){
  case <constante>:[case <constante>:]<sentencias>;break;
  ...
  [default:<sentencias>;break;]
}
```

Para los algoritmos usaremos la construcción siguiente

```
SELECCIONA OPCION expresión
        Constante: sentencias
        Constante: sentencias
SINO
        sentencias
```

Cada caso se precede por la palabra reservada **`case`** y una constante numérica, al final debe colocarse la sentencia **`break`** para que el control pase al final de la sentencia **`switch`**, de no hacerlo así, se ejecutará la siguiente sentencia **`case`**. La parte **`default`** es opcional y se ejecuta en caso de que no se cumpla ningún caso.

El siguiente programa utiliza una sentencia **switch** para desplegar un día de la semana, si el usuario escribe un número no válido, se desplegará el letrero “día no válido”.

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

int main(){
  int dia;
  printf("teclee el número del día: ");
  scanf("%d",&dia);
  switch(dia){
    case 1:printf("Lunes");break;
    case 2:printf("Martes");break;
    case 3:printf("Miércoles");break;
    case 4:printf("Jueves");break;
    case 5:printf("Viernes");break;
    case 6:printf("Sabado");break;
    case 7:printf("Domingo");break;
    default: printf("día no válido");
  }
  getch();
}
```

La sentencia switch es muy adecuada para manejo de varias opciones de menú. El siguiente ejemplo muestra un menú de opciones, espera a que el usuario seleccione alguna opción e imprime un mensaje adecuado a la opción seleccionada:

```
/* Ejemplo de menú */
#include <stdio.h>
#include <conio.h>

main(){
  int a;
  printf("\t\tMenu");
  printf("\n-----------------------------------");
  printf("\n1 - Microsoft Word");
  printf("\n2 - Yahoo messenger");
  printf("\n3 - AutoCAD");
  printf("\n4 - Java Games");
  printf("\n-----------------------------------");
  printf("\nIngrese numero de su preferencia: ");
  scanf("%d",&a); // leer entrada
  switch (a){
    case 1: // si entrada es 1
      printf("\nPersonal Computer Software\n");break;
    case 2: // si entrada es 2
```

```
      printf("\nWeb based Software\n");break;
    case 3: // si entrada es 3
      printf("\nScientific Software\n");break;
    case 4: // si entrada es 4
      printf("\nEmbedded Software\n");break;
   default:  printf("\nEntrada incorrecta\n");
  }
  getch();
}
```

Formateo de la salida

Para terminar esta sección revisaremos algunas opciones para formatear la salida de valores numéricos. Para esto utilizaremos modificadores de ancho y número de decimales en los formatos de salida. Para establecer el ancho de un campo usamos una constante entera antes del carácter de formato. Por ejemplo: `%5d`, indica un campo de anchura cinco para un número entero. Para los números de punto flotante se utiliza un modificador para el número de decimales después del punto. Por ejemplo, `%5.2f` indica un campo de anchura cinco y con dos decimales después del punto.

```
//ejemplos de formatos
#include <stdio.h>
#include <conio.h>

main(){
  int a=8,b=56,c=135;
  float x=12.41232,y=4.8967,z=1.60823e-19;
  double xd=12.41232,yd=4.89678876,zd=1.60823e-19;
  printf("%8d%8d%8d\n",a,b,c);
  printf("%15f%15f%15f\n",x,y,z);
  printf("%15.3f%15.3f%15.3f\n",x,y,z);
  printf("%15f%15f%15f\n",xd,yd,zd);
  printf("%15.3f%15.3f%15.3f\n",xd,yd,zd);
  getch();
}
```

Note que si no se define el número de dígitos después del punto, se imprimen seis decimales.

Ejemplo: Sueldo en una empresa

Una empresa paga por hora a sus empleados de acuerdo a su categoría según la siguiente tabla:

Categoria Pago x hora

1 26.90
2 24.30
3 21.50

Además si el empleado trabaja más de 150 horas mensuales tiene una bonificación del 5% de sueldo. El algoritmo es el siguiente. Se ingresara el nombre del empleado, seguido de su categoría y al final el número de horas trabajadas.

Algoritmo Empleados. Determina el sueldo de un trabajador dependiendo de su categoría y el número de horas trabajadas en un mes. Los empleados se reconocen por su número de empleado. La categoría es un entero de 1 a 3. La variable pagoHora almacena el pago por hora del trabajador y sueldo almacena el sueldo total.

1. [Leer datos]
    Leer(numeroEmpleado, categoría, horas)
2. [Determinar sueldo en base a su categoría]
    SELECCIONA OPCION categoría
        1: pagoHora = 26.9
        2: pagoHora = 24.3
        3: pagoHora = 21.5
3. [Calcula sueldo base]
    sueldo = pagoHora*horas
4. [Determinar monto de bonificación]
    SI horas>150 ENTONCES
        bonificacion = sueldo*1.05
    SINO
        bonificacion = 0
5. [Calcula sueldo total]
    sueldo = sueldo+bonificacion
6. [Imprimir número, categoría y sueldo total]
    Escribe(numeroEmpleado, categoría, suedo)
7.[Terminar]
    Salir

La salida se presentará en forma tabular haciendo uso de secuencias de escape.

```
//Salida en forma tabular
Printf("\nnúmero\tCateg\tHoras\tPago/Hr\tBoni.\tsueldo\n%d\t
%d\t%d\t%f\t%f\t
%f",numeroEmpleado,categoria,horas,pagoHora,bonificacion,suel
do);
```

La salida es con el siguiente formato:

```
Número    Categ     Horas     Pago/Hr   Boni.     sueldo
345       2         130       24.3      0         3159
```

El programa completo es el siguiente:

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

main(){
  int numeroEmpleado; //nombre del empleado
  int categoria; //categoría 1, 2, 3
  int horas; //horas trabajadas
  float pagoHora, sueldo, bonificacion;
  /*Leer datos*/
  printf("numero del empleado: ");
  scanf("%d",&numeroEmpleado);
  printf("categoría (1, 2 o 3): ");
  scanf("%d",&categoria);
  printf("horas trabajadas: ");
  scanf("%d",&horas);
  /*Determinar sueldo en base a su categoría*/
  switch(categoria){
    case 1:pagoHora = 26.90;break;
    case 2:pagoHora = 24.30;break;
    case 3:pagoHora = 21.50;break;
  }
  /*Calcula sueldo base*/
  sueldo = horas*pagoHora;
  /*Determinar monto de bonificación*/
  if(horas>150)
    bonificacion = 0.05*sueldo;
  else
    bonificacion = 0.0;
  /*calcula sueldo total*/
  sueldo = sueldo+bonificacion;
  /*Salida en forma tabular*/
printf("\nnúmero\tCateg\tHoras\tPago/Hr\tBoni.\tsueldo\n%d\t
%d\t%d\t%5.1f\t%5.1f\t%5.1f\n",  numeroEmpleado,  categoria,
horas, pagoHora, bonificacion, sueldo);
  getch();
}
```

Calificación de un alumno

Deseamos hacer un programa que imprima un letrero dependiendo de la calificación obtenida en un examen. La calificación de el examen puede ser un valor entero de 0 a 10. Si la calificación esta entre 0 y 5 (inclusive), se imprimirá el letrero de “reprobado”, si es 6, se imprimirá “suficiente”, si es 7, imprimirá “bien”, si es 8, imprimirá “muy bien”, si es 9, imprimirá “excelente” y si es 10, imprimirá “sobresaliente”. El algoritmo es el siguiente.

Algoritmo Letreros. Imprime el letrero correspondiente a una calificación.

1. [Leer calificación]
    Leer(calificacion)
2. [Imprime letrero]
    SELECIONA OPCION calificacion
        0,1,2,3,4,5: Escribe(“reprobado”)
        6: Escribe (“suficiente”)
        7: Escribe (“bien”)
        8: Escribe (“muy bien”)
        9: Escribe (“excelente”)
        7: Escribe (“sobre saliente”)
    SINO
        Escribe(“Calificación no válida”)
3. [Termina]
    Salir

El programa en C es el siguiente. Note como escribir la sentencia `switch`. En este caso hay un caso de la sentencia `switch` que se ejecuta para 0, 1, 2, 3, 4 y 5. En este caso debemos poner cada uno de los valores, no hay forma de poner un intervalo como de 0 a 5.

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

main(){
  int calificacion;
  /*Leer calificación*/
  printf("Escriba la calificacion: ");
  scanf("%d",&calificacion);
  /*Imprime letrero*/
  switch(calificacion){
    case 0:case 1:case 2:case 3:case 4:case 5:
           printf("reprobado\n");break;
    case 6:printf("suficiente\n");break;
    case 7:printf("bien\n");break;
    case 8:printf("muy bien\n");break;
    case 9:printf("excelente\n");break;
    case 10:printf("sobresaliente\n");break;
    default:printf("calificacion no valida\n");
  }
  getch();
}
```

Problemas propuestos

4.6.1. Una empresa paga a sus empleados de acuerdo a su categoría. Además si un empleado tiene 60 o más años recibe un bono del 5%. Los sueldos por categoría se muestran en le siguiente tabla:

Categoría	Sueldo
1	$7,000
2	$7,800
3	$8,500
4	$9,000

Haga un programa que lea la categoría y edad del empleado y determine el monto de su sueldo.

4.6.2. Escriba un programa que despliegue los letreros de la siguiente tabla dependiendo del valor de una variable num.

Valor	Letrero
negativo	“número negativo”
0	“cero”
2, 4, 6, 8	“número par menor que 10”
1, 3, 5, 7, 9	“número impar menor que 10”
10	“diez”
Mayor que 10	“mayor que 10”

4.6.3. Escriba un programa que lea el número de mes y el día y calcule el número de días transcurridos desde el 1º de enero hasta el día especificado.

4.6.4. Escriba un programa que utilice un menú para conversión de una temperatura entre las escalas Fahrenheit, Celsius y Kelvin. Ponga todas las posibles conversiones entre las tres.

4.6.5. Diseñe un algoritmo y escriba el programa en C para calcular la distancia entre dos puntos especificados en coordenadas rectangulares o polares o cualquier mezcla de estas coordenadas. Deberá utilizar un menú para seleccionar el tipo de datos que se introducirá. Lea los ángulos en grados y convierta a radianes.

4.7. Operador interrogación

Existe un forma alternativa de sentencia `if` mediante el operador interrogación (?:). Existen dos formas, una que regresa un valor y otra equivalente a la sentencia `if-else`. La sintaxis es:

Condición? operando1: operando2;

o

Condición? instrucción1: instrucción2;

En el primer caso la instrucción regresa operando1 si se cumple la condición, sino regresa el operando2. En el segundo caso se ejecutará la instrucción1 si se cumple la condición, sino se ejecutará la instrucción2.

Ejemplos:

```
printf(cal>=60?“Aprobado”:“Reprobado”);
```

Esta sentencia imprimirá “Aprobado” se cal es mayor o igual a 60 e imprimirá “Reprobado” si es menor que 60. Note que los paréntesis son necesarios dado que el operador ?: tiene la menor precedencia.

También es válido:

```
cal>=60?printf(“Aprobado”):printf(“Reprobado”);
```

Sentencia **if**	sentencia ?
`if(a>5) x = 6;else x = 7;`	`a>5 ? x=6: x=7;`

`if(x>8 && x<12)`
`  a = 2*b+c;`
`else`
`  a = 3*b+c;`
	`(x>8 && x<12) ?`
	`a=2*b+c:a=3*b+c;`

`if(a>b)`
`  if(b>c)`
`  x = 5*a+c;`
`  else`
`  x = 6*a+b;`
`else`
`  x = 8*a+4*b;`
	`(a>b)?(b>c?`
	`x=5*a+c:x=6*a+b;):x=8*a+4*b;`

Note que el uso de el operador ? es a veces difícil de descifrar.

Problemas propuestos

4.7.1. Traducir a sentencias **if**

```
y = (a>b)?3*a+b:2*a+b;

z =a>=2*x?(a<6)?4*a:5*a:3*a+1;
```

4.7.2. ¿Qué valores se imprimen en el siguiente fragmento de código?

```
int a=8,b=9,x=5,y,z;
y = (a>b)?3*a+b:2*a+b;
z =a<=2*x?(a<6)?4*a:5*a:3*a+1;
printf("y = %f z = %f\n",y,z);
```

4.7.3. Traduzca las sentencias if a sentencias con el operador ?:.

```
if(x>8) y = 23;else z = -5;

if(x>=8 && x!=12)
  z = sqrt(6*x);
else
  if(x>0)
    z = sin(3*x);
  else
    z = sin(5*x);
```

4.7.4. Traduzca la siguiente sentencia con el operador ? a sentencias if

```
x = (y>5) ? 2*y+1:(z<10) ? 3*y+2:5*z+1;
```

4.7.5. Las fechas de los signos zodiacales se muestran en la siguiente tabla. Escriba un programa que lea el día y mes de nacimiento y despliegue el signo zodiacal correspondiente. Utilice sentencias con el operador ?.

22 dic al 20 ene: CAPRICORNIO
21 ene al 19 feb: ACUARIO
20 feb al 20 mar: PISCIS
21 mar al 20 abr: ARIES
21 abr al 21 may: TAURO
22 may al 21 jun: GEMINIS
22 jun al 22 jul: CANCER
23 jul al 23 ago: LEO
24 ago al 23 sep: VIRGO
24 sep al 23 oct: LIBRA
24 oct al 22 nov: ESCORPION
24 nov al 21 dic: SAGITARIO

Capítulo 5. Instrucciones de repetición

5.1. Motivación

Suponga que se desea sumar una lista de 20 números y obtener el promedio. Sin estructuras de repetición habría que escribir 20 sentencias de entrada. Algo como:

```
int n,suma = 0;
printf( “tecle n: “);
scanf(“%d”,n);
suma = suma + n;
```

Repetir las tres últimas sentencias 19 veces, o bien definir veinte variables diferentes y escribir un código como el siguiente.

```
int n1,n2, … ,n20, suma = 0;
printf( “teclee los 20 valores “;
scanf(“%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d
%d”,n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,14,n15,n16,n17
,n18,n19,n20);
suma = n1 + n2 + … + n20;
```

Ambas soluciones son posibles. Sin embargo si el número de valores que se deben sumar es diferente, habrá que modificar cualquiera de ellos para obtener una solución adecuada. Además si el número de valores a sumar es muy grande, programa se hace prohibitivamente grande. Por lo anterior vemos que debe existir otra alternativa para resolver este tipo de problemas. La solución nos la dan las instrucciones de repetición. Primero revisaremos la sentencia `while`.

5.2. Sentencia `while`

La sentencia `while` es una estructura de control que permite repetir una o más instrucciones mientras se cumpla una condición lógica. La sintaxis es:

```
while(condición)
  sentencia o bloque;
```

Si la condición se cumple se ejecutan las sentencias del bloque y se regresa el flujo de control a evaluar nuevamente la condición. El proceso se repite hasta que la condición sea falsa. El ciclo puede ejecutarse 0 veces si la condición no se cumple al entrar en él. En el lenguaje algorítmico usaremos la siguiente construcción. Utilizaremos un sangrado del texto de las sentencias para indicar las instrucciones internas a cada ciclo.

1. [inicio del ciclo]
   MIENTRAS condición HACER
2. [Cuerpo del ciclo]

instrucciones que se repetirán
3. [Fin del ciclo]
FINMIENTRAS

El programa para sumas 20 números es el siguiente.

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

int main(){
    float suma = 0.0, num;
    int contador = 0;

    while(contador < 20){
        printf("Teclee un número:");
        scanf("%f",&num);
        suma = suma + num;
        contador = contador + 1;
    }
    printf("\nLa suma de los 20 números es: %5.0f\n",suma);
    getch();
}
```

En esta solución solo tuvimos que utilizar 3 variables para sumar 20 valores. Fácilmente puede modificarse para sumar otra cantidad de valores. La condición que controla la sentencia **`while`** es `contador<20`. Cuando se ejecuta por primera vez la variable contador vale 0 y la condición es verdadera. Dentro de las instrucciones del ciclo la variable contador se incrementa mediante la sentencia `contador = contador + 1`, eventualmente llegará a valer 20 y en ese momento la condición dejará de ser verdadera y por lo tanto el ciclo dejará de ejecutarse.

5.3. Ciclos controlados por centinela

Si no se conoce de antemano el número de datos, se utiliza un valor de entrada especial como una bandera o centinela para terminar la entrada de datos. Como ejemplo supongamos que deseamos calcular el promedio de una serie de valores positivos. Podemos usar como valor centinela un valor negativo. Los ciclos controlados por centinela requieran que se haga una lectura antes de entrar al ciclo y otra dentro del ciclo cuando utilizamos un ciclo **`while`**.

Algoritmo Centinela. Calcula el promedio de una serie de valores reales. Utiliza un centinela para terminar la entrada de datos.
1. [Solicitar un nuevo valor para promediar]
Leer(num)
2. [Inicia el lazo de lectura]
MIENTRAS num diferente de -1 HACER

3. [Acumular suma e incrementar contador]
suma = suma + num
contador = contador+1
4. [Solicitar nuevo valor para promediar]
Leer(num)
5. [Fin ciclo]
FINMIENTRAS
6 [verificar si se leyeron valores]
SI contador > 0 ENTONCES
promedio = suma/contador
Escribe(promedio)
SINO
Escribe("NO se teclearon valores")
7. [Termina]
Salir

El programa completo es el siguiente:

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

int main(){
  float suma = 0.0, num, promedio;
  int contador = 0;
  /*Solicitar un nuevo valor para promediar*/
  printf("Teclee un número (-1 = fin):");
  scanf("%f",&num);
  /*Inicia el lazo de lectura*/
  while(num != -1){
    /*Acumular suma e incrementar contador*/
    suma = suma + num;
    contador = contador + 1;
    /*Solicitar un nuevo valor para promediar*/
    printf("Teclee un número (-1 = fin):");
    scanf("%f",&num);
  /*Fin ciclo*/
  }
  /*verificar si se leyeron valores*/
  if(contador>0){
    promedio = suma/contador;
    printf("\nEl promedio es: %5.2f",promedio);
  }
  else
    printf("\nNo se teclearon valores\n");
  getch();
}
```

La sentencia `suma = suma+num`, agrega al valor actual de la variable `suma` el valor de la variable `num`, de esta manera se va acumulando el valor total de los números leídos. Similarmente la sentencia `contado = contador+1`, agrega 1 al valor actual de la variable `contador`. Note que hay dos lugares donde se lee la variable `num`.

Se desea un programa para leer las calificaciones de un grupo de alumnos y calcular el promedio general, así como el número de alumnos aprobados y reprobados. No conocemos el número de alumno, por tanto es adecuado el esquema del centinela. Necesitaremos las siguientes variables:

contador – contador de calificaciones
aprobados – contador de aprobados
reprobados – contador de reprobados
calificacion – calificación
suma – suma de calificaciones
promedio – promedio de calificaciones

Algoritmo Alumnos aprobados. Lee las calificaciones de un grupo de alumnos y calcula el promedio general del grupo. Además se cuenta cuantos alumnos han aprobado. Supondremos que las calificaciones tienen valores numéricos de 0 a 10.

```
1. [Leer siguiente calificación]
        Leer(calificación)
2. [Ciclo de lectura por centinela]
        MIENTRAS calificación diferente de -1 HACER
3. [acumular suma]
                suma = suma+calificacion
4. [Cuenta los alumnos aprobados y reprobados]
                SI calificación>= 6 ENTONCES
                        aprobados = aprobados+1
                SINO
                        reprobados = reprobados+1
5. [Incrementar total de alumnos]
                total = total+1
6. [Leer siguiente calificación]
                Leer(calificación)
7. [Fin ciclo]
        FINMIENTRAS
8. [calcular el promedio e imprimir]
        SI total>0 ENTONCES
                promedio = suma/total
                Escribe(promedio)
                Escribe(aprobados)
                Escribe(reprobados)
        SINO
                Escribe("NO se introdujeron datos")
9. [Termina]
        Salir
```

La operación de incremento es muy común sobre todo en el uso dentro de ciclos, es por eso que en C existen operadores para incrementar o decrementar una variable entera. El operador de incremento consiste de dos signos "+" antes o después de la variable. Más adelante veremos la diferencia entre colocar el operador antes o después de la variable. El programa es el siguiente. Se utiliza el operador ++ para las operaciones de incremento.

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

main(){
  int total=0, aprobados=0, reprobados=0;
  float calificacion, suma=0,promedio;
  /*Leer siguiente calificación*/
  printf("teclee calificacion (-1 para terminar):");
  scanf("%f",&calificacion);
  /*Ciclo de lectura por centinela */
  while(calificacion != -1){
    /*acumular suma*/
    suma = suma+calificacion;
    /*Cuenta los alumnos aprobados y reprobados*/
    if(calificacion >= 6.0)
      aprobados++;
    else
      reprobados++;
    /*Incrementar total de alumnos*/
    total++;
    /*Leer siguiente calificación*/
    printf( "teclee calificacion (-1 para terminar):");
    scanf("%f",&calificacion);
  /*Fin ciclo*/
  }
  if(total>0){
    /*calcular el promedio e imprimir*/
    promedio = suma/total;
    printf( "Promedio: %.3f\n",promedio);
    printf( "aprobados: %d\n ",aprobados);
    printf( "reprobados: %d\n ",reprobados);
  }
  else
    printf( "no se introdujeron datos.");
  getch();
}
```

El esquema del centinela puede usarse para controlar la ejecución de un programa. Anteriormente vimos un ejemplo de tiro parabólico. Suponga que deseamos resolver ese problema para varios valores de entrada. El algoritmo sería el siguiente.

1. [preguntar si desea resolver un problema de tiro]
   Leer(continuar)
2. [repetir mientras continuar es 1]
   MIENTRAS continuar es 1 HACER
3. [Problema del tiro parabólico]
   Insertar aquí el algoritmo Tiro parabólico
4. [preguntar si desea resolver un problema de tiro]
   Leer(continuar)
5. [fin de ciclo]
   FINMIENTRAS
6. [Terminar]
   Salir

El programa completo queda como sigue.

```
/*Calcula alcance, altura máxima y tiempo de vuelo en un tiro
parabólico*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

main(){
  float PI = 3.14159265;
  float g = 9.81;
  float v0,ang;/* datos de entrada*/
  float h,R,T; /* datos de salida*/
  int continuar;
  /*preguntar si desea resolver un problema de tiro*/
  printf(“Desea resolver un problema de tiro parabolico? ”);
  scanf(“%d”,&continuar);
  /*repetir mientras continuar es 1*/
  while(continuar){
    /*Problema del tiro parabólico*/
    printf("TIRO PARABOLICO\n");
    printf("Teclee velocidad inicial (en m/s): ");
    scanf("%f",&v0);
    printf("Teclee angulo del tiro (en grados): ");
    scanf("%f",&ang);
    ang = ang*PI/180; /* conversión a radianes*/
    h = v0*v0*sin(ang)*sin(ang)/2/g;
    R = v0*v0*sin(2*ang)/g;
    T = 2*v0*sin(ang)/g;
    printf("altura maxima: %f m\n",h);
    printf("alcance maximo: %f m\n",R);
    printf("tiempo de vuelo: %f s\n",T);
    /*preguntar si desea resolver un problema de tiro*/
```

```
    printf("Otro problema de tiro parabolico? ");
    scanf("%d",&continuar);
  /*fin de ciclo*/
  }
  getch();
  return 0;
}
```

Note que la condición del lazo es solo el valor de la variable continuar. Cualquier valor diferente de cero provocará que el ciclo se repita. El programa se ejecutará hasta que el usuario responda con un 0.

Otro ejemplo de terminación de un ciclo es el siguiente. Dado un número entero positivo se dividirá entre dos si es par o sino se multiplicará por 3 y se le sumará 1. El proceso se repetirá hasta que el resultado obtenido sea igual a 1. Por ejemplo, el número 6 genera la secuencia: 6, 3, 10, 5, 16, 8, 4, 2, 1. El algoritmo es el siguiente.

Algoritmo Secuencia de enteros. Imprime la secuencia generada a partir de un número entero positivo al que se le divide entre 2 si es par y se multiplica por 3 y se le agrega 1 si es impar.

1. [Leer número]
   Leer(num)
2. [Hacer ciclo mientras num diferente de 1]
   MIENTRAS num diferente de 1 HACER
4. [Imprimir num]
   Escribe(num)
3. [decisión de que hacer]
   SI num es par ENTONCES
      num = num/2
   SINO
      num = 3*num+1
   FINSI
5. [Fin del ciclo]
   FINMIENTRAS
6. [Terminar]
   Salir

La traducción a C es bastante simple.

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

main(){
  int i;
  /*Leer número*/
  printf("Teclede numero: ");
  scanf("%d",&i);
```

```
  /*Hacer ciclo mientras num diferente de 1*/
  while(i>1){
    /*Imprimir num*/
    printf("%d ",i);
    /*decisión de que hacer*/
    if(i%2==0)
      i = i/2;
    else
      i = 3*i+1;
  /*Fin del ciclo*/
  }
  printf("%d ",i);
  getch();
}
```

Problemas propuestos

5.3.1. Escriba programa con un ciclo controlado por centinela para leer una serie de números hasta que el usuario teclee el número 9999. Dentro del ciclo cuente los múltiplos de 2, de 3 y de 5. Imprima cuantos múltiplos se teclearon en cada caso.

5.3.2. Diseñe un algoritmo para imprimir los cuadrados y cubos de los primeros 20 números enteros.

5.3.3. Escriba un programa que calcule la suma de cada tercer entero, comenzando por i = 2 (es decir suma de 2 + 5 + 8 + 11 + …) para todos los valores de i menores que 100.

5.3.4. Escriba un programa que calcule la suma de cada tercer entero, comenzando por i = 2 para todos los valores de i menores que 100. Calcular la suma de los enteros generados que sean divisibles por 5. Utilizar dos métodos distintos para comprobar la divisibilidad:
a) Utilizar el operador ?:
b) Utilizar una instrucción if.

5.3.5. Modifique el algoritmo “Secuencia de enteros” para que cuente el número de elementos que tiene la secuencia de números generada. Escriba el programa correspondiente.

5.3.6. Describir la salida que generará los siguientes programas en C:

```
#include <stdio.h>
#include <conio.h>
main(){
  int i=0,x=0;
  while(i<20){
    if(i%5==0){
      x  = x + i;
```

```
      printf("%d ",x);
    }
    i = i+1;
  }
  printf("\nx = %d ",x);
  getch();
}
```

```
#include <stdio.h>
#include <conio.h>
main(){
  int i=1,x=0;
  while(i<10){
    if(i%2==1)
      x  = x + i;
    else
      x  = x - 1;
    printf("%d ",x);
    i = i+1;
  }
  printf("\nx = %d ",x);
  getch();
}
```

5.3.7. Desarrolle un algoritmo para calcular el promedio ponderado de n números, utilizando la fórmula

$x_{\text{media}} = f_1x_1 + f_2x_2 + \ldots f_nx_n$

donde las *f* son los pesos de cada número y cumplen con:

$0 \leq f_i < 1$ y $f_1 + f_2 + ... + f_n = 1$

5.4. Ciclos anidados

Muchos procesos requieren ciclos que incluyen ciclos internos. A estos ciclos se les llama ciclos anidados. Por ejemplo, suponga que deseamos imprimir una tabla de multiplicar de los números entre 1 y 10. Para esto requerimos imprimir los múltiplos de 1 en el primer renglón, luego los múltiplos de 2 en el segundo renglón, y así sucesivamente. El siguiente código imprime la tabla de multiplicar:

```
numero = 1;
while(numero<=10){
  factor = 1;
  while(factor<=10){
    printf( “&4d”,numero*factor);
    factor++;
```

```
  }
  printf(“\n”);
  numero++;
}
```

El siguiente ejemplo imprime un rectángulo de asteriscos, el usuario escribe el ancho y el alto del rectángulo, se validan las dimensiones para ancho y alto del rectángulo. Este problema puede resolverse fácilmente mediante un ciclo dentro de otro. El algoritmo es

Algoritmo Recuadro. Dibuja en la pantalla de salida un rectángulo utilizando el carácter “*”. Se utilizan ciclos anidados, el exterior controlado por la variable r y el interior por la variable c. Las dimensiones del rectángulo son ancho y alto.

1. [Leer dimensiones]
   Leer(ancho,alto)
2. [Inicia ciclo externo]
   r = 0
3. [Repetir alto veces]
   MIENTRAS r < alto HACER
4. [Inicia ciclo interno]
   c = 0
5. [Repetir ancho veces]
   MIENTRAS c<ancho HACER
6. [Imprime un asterisco sin cambiar de renglón]
   Escribe(“*”)
   c = c+1
7. [Fin del ciclo interno]
   FINMIENTRAS
8. [Cambiar de línea e incrementa el contador de renglones]
   Escribe()
   r = r+1
9. [Fin del ciclo exterior]
   FINMIENTRAS
10. [Termina]
    Salir

El programa completo es el siguiente. Se ha utilizado un sangrado en el texto del programa para clarificar el anidamiento de los ciclos. Es importante que mantenga un formato adecuado del texto en programas con ciclos anidados, el no hacerlo puede hacer muy confuso el texto del programa.

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

main(){
  int ancho, alto, c, r;
  /*Leer dimensiones*/
  printf("ancho del rectangulo: ");
```

```
  scanf("%d",&ancho);
  printf("alto del rectangulo: ");
  scanf("%d",&alto);
  /*Inicia ciclo externo*/
  r = 0;
  /*Repetir alto veces*/
  while(r<alto){
    /*Inicia ciclo interno*/
    c = 0;
    /*Repetir ancho veces*/
    while(c <ancho){
      printf("*");
      c++;
    /*Fin del ciclo interno*/
    }
        /*Cambiar de línea e incrementa el contador de
renglones*/
    printf("\n");
    r++;
    }
  /*Fin del ciclo exterior*/
  }
  printf("\n");
  getch();
}
```

Números Primos

Un número es primo si solo es divisible entre el mismo y la unidad. Desarrollemos un algoritmo para determinar si un número es primo o no. Para determinar si es primo debemos revisar si es divisible entre 2, 3, 4, .. hasta un valor igual a su raíz cuadrada, ya que un número solo puede tener un divisor menor o igual a su raíz cuadrada. El algoritmo es el siguiente.

Algoritmo Primo. Determina si un número leído desde el teclado es primo o no. Usa el entero divisible para controlar el lazo.

1. [Leer un número]
   Leer(numero)
2. [Calcula el límite superior]
   raiz = sqrt(numero)
3. [Inicialmente suponemos que es primo]
   divisible = falso
   divisor = 2
4. [Repetir mientras no sea divisible y divisor menor que raiz]
   MIENTRAS no divisible y divisor menor o igual que raiz HACER
5. [Determina si es divisible, sino siguiente divisor]
   SI residuo de numero/divisor es cero ENTONCES

divisible = verdadero
SINO
divisor = divisor + 1
6. [Fin del ciclo]
FINMIENTRAS
7. [es primo?]
SI divisible ENTONCES
Escribe(“NO es primo”)
SINO
Escribe(“SI es primo”)
8. [Terminar]
Salir

El programa es el siguiente.

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

main(){
  int numero,divisor,divisible,raiz;
  /*Leer un número*/
  printf("Teclee un numero: ");
  scanf("%d",&numero);
  /*Calcula el límite superior*/
  raiz = (int)sqrt(numero);
  /*Inicialmente suponemos que es primo*/
  divisible = 0;
  divisor = 2;
   /*Repetir mientras no sea divisible y divisor menor que
raiz*/
  while(!divisible && divisor<=raiz)
    /*Determina si es divisible, sino siguiente divisor*/
    if(numero%divisor==0)
      divisible = 1;
    else
      divisor++;
  /*es primo?*/
  if(divisible)
    printf("%d NO es primo",numero);
  else
    printf("%d SI es primo",numero);
  getch();
}
```

Ahora consideremos el problema de determinar todos los números primos entre 1 y un valor dado. Este problema implica el uso de ciclos anidados. El algoritmo es el siguiente.

Algoritmo Primos. Determina los números primos entre 1 y un número dado.
1. [Leer el número límite]
    Leer(limite)
2. [Inicia control del ciclo]
    numero = 2
3. [Lazo exterior, analiza cada número de 1 a limite]
    MIENTRAS numero<limite HACER
4. [Algoritmo Primo, pasos 2 al 6]
5. [Si es primo, desplegarlo]
        SI no divisible ENTONCES
            Escribe(numero)
6. [siguiente numero]
        numero = numero+1
6. [Fin del ciclo exterior]
    FINMIENTRAS
7. [Termina]
    Salir

El programa que incluye ambos ciclos es el siguiente.

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

main(){
  int limite,numero,divisor,divisible,raiz;
  printf("Teclee un numero: ");
  scanf("%d",&limite);
  numero = 2;
  while(numero<limite){
    raiz = (int)sqrt(numero);
    divisible = 0;
    divisor = 2;
    while(!divisible && divisor<=raiz)
      if(numero%divisor==0)
        divisible = 1;
      else
        divisor++;
    if(!divisible)
      printf("%d ",numero);
    numero++;
  }
  getch();
}
```

Problemas propuestos

5.4.1. Escriba un programa para dibujar un patrón de tablero de ajedrez de tamaño 2 a 20 como se muestra:

```
* * * *
 * * * *
* * * *
 * * * *
```

Utilice ciclos anidados y sentencias de salida que impriman un asterisco y espacio en blanco.

5.4.2. Escriba un programa que despliegue la siguiente figura. El tamaño de la figura podrá ser de 3 a 10 y deberá leerse desde el teclado.

```
1
12
123
1234
12345
1234
123
12
1
```

5.4.3. Escriba un programa para generar la siguiente pirámide de dígitos:

```
          1
         232
        34543
       4567654
      567898765
     67890109876
    7890123210987
   890123454321098
  90123456765432109
 0123456789876543210
```

No escriba solamente diez cadenas de caracteres. Consiga una fórmula para generar los dígitos correspondientes para cada línea.

5.4.4. Un número es perfecto si es igual a la suma de sus divisores, exceptuando a si mismo. Por ejemplo: 6 es perfecto porque es igual a 1+2+3. Diseñe un algoritmo y haga el programa correspondiente para encontrar todos los números perfectos entre 1 y 10000.

5.4.5. Diseñe un algoritmo para calcular y tabular los valores de la función

$$f(x,y) = \frac{x^2 - y^2}{x^2 + y^2}$$

para x = 2, 4, 6, 8, y y = 6, 9, 12, 15, 18, 21.

5.4.6. Diseñe un algoritmo para calcular el número de puntos con coordenadas enteras contenidos en una elipse centrada en el origen con semieje mayor A y semieje menor B. Los valores de A y B deberán ser leídos desde el teclado.

5.5. Operadores de asignación

En C existen operadores para abreviar las operaciones de asignación. Por ejemplo: c = c + 3 puede escribirse como c += 3. En general

variable = variable operador expresión

es equivalente a

variable operador= expresión

Ojo

```
a *= c + d
```

equivale a

```
a = a*(c + d)  no   a = a*c + d
```

Algunos operadores de asignación son los siguientes:

```
a = a + 3;          a += 3
c = 2*c;            c *= 2;
x = x – z;          x –= z;
s = s*b – s*c;      s *= b – c;
d = d/(f + 5);      d /= f + 5;
r = r % 5;          r %= 5;
```

5.6. Operadores de incremento y decremento

Existen cuatro operadores de incremento y decremento. El operador de preincremento (++variable), el operador de posincremento (variable++), el operador de predecremento (--variable) y el operador de posdecremento (variable--). Los operadores de preincremento y predecremento se evalúan con la máxima prioridad cuando aparecen dentro de una expresión y los operadores de posincremento y posdecremento se evalúan con la prioridad

más baja. El operador de preincremento `++a` es equivalente a `a = a + 1` o `a += 1`. Similarmente el operador de predecremento `–a` es equivalente a `a = a - 1` o `a -= 1`.

Consideremos el siguiente ejemplo de uso de los operadores de pre incremento y pos incremento.

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

main(){
  int a,b,c;
  a = 5;
  printf("%d\n",a);
  b = a++;
  printf("%d %d\n",a,b);
  c = ++a+b++;
  printf("%d %d %d\n",a,b,c);
  getch();
}
```

La primera línea imprime el valor asignado a a, o sea 5. Luego se asigna este valor a b y después se incrementa le valor de a y se imprime 6 y 5. Por último, se incrementa el valor de a, luego se suma este valor a b y se le asigna a c, y al último se incrementa el valor de b, por tanto se imprime 7 para el valor de a, 6 correspondiente a b y 7+5=12, correspondiente a c. Por lo tanto se imprimirá:

5
6 5
7 6 12

Problemas propuestos

5.6.1. ¿Que valores se imprimen en el siguiente fragmento de programa?

```
  int a = 8, b,c;
  b = ++a;
  printf("%d %d\n",a,b);
  c = 2*a++-++b;
  printf("%d %d %d\n",a,b,c);
```

5.6.2. ¿Cuál es la salida de los siguientes ciclos?

```
i = 0;
while(i < 10)
  printf(“%d%10d”,2*i,3*i*i);
  i += 3;
}
```

```
i = 5;
while(i > -20)
  printf("%d%10d",2*i,2*i*i);
  i -= 3;
}
```

5.6.3. Escriba un programa para encontrar todos los divisores de un número, Ejemplo, los divisores de 246 son: 2, 3, 6, 41, 82, 123.

5.7. Ciclo `for`

Una forma conveniente de escribir ciclos con contador es utilizando la sentencia `for`. Esta sentencia permite definir un contador, iniciar el contador, incrementarlo y establecer la condición para el límite final, todo en una sola instrucción. La sintaxis del ciclo `for` es:

```
for(expresión1; expresión2; expresión3)
  instrucción o boque;
```

Esta sentencia es equivalente a una construcción `while` como la siguiente:

```
expresion1;
while(expresion2){
  instrucción;
  expresion3;
}
```

Generalmente el significado de las expresiones es el siguiente:

expresion1 = sentencia de iniciación
expresion2 = condición de terminación
expresion3 = sentencia de incremento

La sentencia de iniciación solo se ejecuta una vez al iniciar el ciclo, la condición de terminación se evalúa en cada ciclo, si es verdadera se ejecuta la sentencia o bloque, sino, se termina el ciclo y la sentencia de incremento se ejecuta al final de cada ciclo. Cualquiera de ellas o todas pueden ser una sentencia vacía, en este último caso se tiene un ciclo infinito equivalente a `while(1)`. Algunos ejemplos de lazos `for` son los siguientes:

a) modifica la variable de control de 1 a 100 en incrementos de 1.
```
for(i = 1; i <= 100; i++)
```
b) modifica la variable de control de 100 a 1 en decrementos de 1.
```
for(i = 100; i >= 1; i--)
```
c) modifica la variable de control de 7 a 77 en incrementos de 7.
```
for(i = 7; i <= 77; i += 7)
```
d) modifica la variable de control de 20 a 2 en decrementos de -2.
```
for(i = 20; i >= 2; i -= 2)
```

e) modifica la variable de control de 2 a 20 en incrementos de 3.

```
for(i = 2; i <= 20; i += 3)
```

f) modifica la variable de control de 99 a 0 en decrementos de -11.

```
for(i = 99; i >= 0; i -= 11)
```

Una variable puede ser declarada dentro de cualquier bloque. La variable declarada dentro de un bloque solo puede ser utilizada dentro de ese bloque. Es un hecho común que las variables de control de un ciclo `for` sean declaradas dentro del ciclo como sigue:

```
for(int i = 0; i < 100; i++){
  suma += i;
}
```

En el ejemplo anterior la variable i solo puede ser utilizada dentro del bloque, sería ilegal usarla fuera de él a menos que exista otra declaración antes del ciclo de la misma variable. La variable de control de un ciclo no tiene que ser forzosamente de tipo entero, puede utilizarse una variable de cualquier tipo. Por ejemplo:

```
for(float x = 0.0, suma = 0.0; x <= 10.0; x+=0.1){
  suma += x;
}
```

Este ejemplo suma todos los números reales desde 0 a 10 en pasos de 0.1. Note que es posible iniciar más de una variable, para esto separamos cada inicialización con el operador coma (,).

Ejemplo

El interés que otorgan los bancos por invertir un capital puede calcularse con la siguiente fórmula

$$I = C\,(1 + r)^n$$

Donde C es el capital, r es la tasa de interés y n es el número de periodos. Ejemplo:
Si C = 20,000, r = 7% y n = 3,

$$I = 20000(1 + 0.07)3 = 24{,}500.86$$

El siguiente programa utiliza un ciclo for para imprimir el monto del capital acumulado año por año de un capital al 5% de interés.

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

int main() {
```

```
   float monto;     // monto del depósito
   float principal = 1000.0; // monto principal (al inicio)
   float tasa = .05;         // tasa de interés
   int anio;
   printf("Anio   Monto del depósito\n");
   for (anio = 1; anio <= 10; anio++ ) {
      monto = principal * pow( 1.0 + tasa, anio );
      printf("%4d%21.2f\n",anio,monto);
   }
   getch();
}
```

Suma de series

Una aplicación importante de los ciclos es la suma de series. Una serie es una secuencia de números que cumplen con alguna relación. Por ejemplo:

$$\frac{1}{2},\frac{1}{4},\frac{1}{8},\ldots$$

En esta serie cada elemento es la mitad del anterior. La suma de esta serie es igual a 1 cuando tomamos un número muy grande de elementos. El siguiente programa calcula la suma de esta serie

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

main(){
  float fraccion = 1, suma = 0.0;
  int i;
  for(i=0; i<20; i++){
    fraccion /= 2.0;
    suma += fraccion;
  }
  printf("%8.6f\n",suma);
  getch();
}
```

Podemos sumar series en las que los elementos cambien de signo. Por ejemplo:

$$\frac{1}{2}-\frac{1}{3}+\frac{1}{4}-\frac{1}{5}+\ldots$$

Para cambiar de signo en cada término definimos una variable entera `signo`, la cual se inicia a 1 y en cada paso del ciclo se invierte el signo mediante la asignación `signo =`

`-signo`. Esta serie converge muy lentamente, se recomienda al lector consultar algún libro de cálculo para averiguar a que valor converge. El programa es el siguiente:

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

main(){
  float fraccion = 1, suma = 0.0;
  int i,signo = 1;
  for(i=2; i<1000; i++){
    fraccion = 1.0/i;
    suma += signo*fraccion;
    signo = -signo;
  }
  printf("\n%8.5f\n",suma);
  getch();
}
```

Problemas propuestos

5.7.1. Escriba un programa utilizando un ciclo for para sumar los múltiplos de 7 entre 0 y 1000.

5.7.2. Escriba un programa para sumar los primeros 100 números primos.

5.7.3. Escriba las siguientes instrucciones con ciclo while utilizando un ciclo for.

```
int j = 0, suma = 0;
while(j < 60){
  suma += 2*j+1;
  j += 3;
}
```

5.7.4. Escriba un programa utilizando un ciclo for que despliegue una tabla con valores de los cuadrados y cubos de los números de 1 a 20.

5.7.5. Un número es perfecto si es igual a la suma de sus divisores, por ejemplo 6 es perfecto porque 6 = 1 + 2 + 3. Escriba un programa para encontrar todos los números perfectos entre 1 y 10000 utilizando ciclos for.

5.7.6. Escriba un programa que encuentre los factores primos de un número. Ejemplo: si se introduce 30, el programa dará como salida: 2, 3, 5.

5.7.7. Escriba un programa que factorice un número en potencias de números primos. Por ejemplo: si se introduce 3000, el programa dará como salida: 2^3*3^1*5^3.

5.7.8. Escriba un programa para sumar la serie siguiente:

$$\frac{1}{3}-\frac{1}{5}+\frac{1}{7}-...$$

5.7.9. Escriba un programa para sumar la serie siguiente:

$$\frac{1}{4}+\frac{1}{7}+\frac{1}{10}+...+\frac{1}{3n+1}+...$$

5.7.10. Una terna de números *a*, *b*, *c* es pitagórica $a^2 = b^2 + c^2$. Escriba un programa que determine todas las ternas pitagóricas para números *a*, *b*, *c* ente 1 y 100.

5.8. Ciclo do-while

La tercer forma de hacer ciclos en C es mediante el ciclo **do-while**. La diferencia entre este tipo de ciclo y los dos anteriores es que se ejecuta al menos una vez, por esto es útil cuando se requiere ejecutar una o más instrucciones al menos una vez. La sintaxis del ciclo **do-while** es:

```
do
  instrucción
while(condición);
```

Se pueden usar llaves alrededor de la instrucción, aunque no son necesarias para instrucciones simples, solo si la instrucción es una instrucción compuesta, sin embargo, es conveniente utilizar las llaves siempre para no confundir el **while** con el inicio de un ciclo **while**. El ciclo **do-while** se utiliza menos que los otros dos ciclos. Sin embargo en algunas construcciones algorítmicas resulta más natural. Como ejemplo, considere el siguiente programa que imprime los primeros 10 enteros y sus cuadrados.

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

main(){
  int n = 1;
  do{
    printf("%2d\t%d\n",n,n*n);
    n++;
  }while(n<11);
  getch();
}
```

Una aplicación práctica del ciclo **do-while** es cuando se desea ejecutar un conjunto de instrucciones una o más veces. En el capítulo 4 se hizo un programa para encontrar la intersección de dos rectas dadas las pendientes y las ordenadas al origen. La siguiente

versión del programa que solicita al final una confirmación si se desea resolver otro problema.

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

int main()
{
  float pendiente1, pendiente2, ordenada1 ,ordenada2,x,y;
  int opcion;
  /*Leer datos de las dos rectas*/
  do{
    printf("teclee la pendiente y ordenada al origen 1: ");
    scanf("%f%f",&pendiente1,&ordenada1);
    printf("teclee la pendiente y ordenada al origen 2: ");
    scanf("%f%f",&pendiente2,&ordenada2);
    /*Determina intersección, si existe*/
    if(pendiente1 != pendiente2){
      x = (ordenada2 - ordenada1)/( pendiente1- pendiente2);
      y = pendiente1*x+ ordenada1;
      printf("Las rectas se interceptan en: %f, %f\n",x,y);
    }
    else
      printf("Las rectas no se interceptan...\n");
    printf("otras rectas (1-si 0-no)? ");
    scanf(" %d",&opcion);
  }while(opcion!=0);
}
```

Otra aplicación natural es el cálculo de raíces de ecuaciones. Los métodos de solución generalmente consisten de un algoritmo que se repite hasta que se cumple alguna condición como un determinado número de pasos o la satisfacción de un criterio de convergencia, o ambas.

Un método muy sencillo de solución es el del *punto fij*o. Este método consiste en despejar de la ecuación la incógnita en términos de una expresión algebraica. Por ejemplo, considere la ecuación $x^3 + 4x^2 – 3x + 7 = 0$, podemos despejar x de la siguiente forma:

$x = (–4x^2 – 7)/(x^2 – 3)$

El siguiente paso es proponer un valor para la incógnita y calcular con esta expresión la siguiente aproximación. El proceso se repite hasta hacer un cierto número de pasos o hasta que el valor de la incógnita no cambia. El siguiente programa resuelve la ecuación anterior mediante este método. Cabe aclarar que la elección de la expresión algebraica es determinante en la convergencia del método. Una mala elección puede hacer que no se llegue a la solución.

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

main(){
  float x,xvieja;
  int pasos = 0;
  x = 0.0;
  do{
    xvieja = x;
    x = (-4*x*x-7)/(x*x-3);
    pasos++;
  }while(pasos<20&&(fabs(x-xvieja)>1e-4));
  printf("x = %.5f\n  pasos = %d\n",x,pasos);
  printf("eq = %.5f\n",x*x*x+4*x*x-3*x+7);
  getch();
}
```

Todos los tipos de ciclos pueden dar lugar a ciclos infinitos, es decir, ciclos que se repiten por siempre. Por ejemplo, un lazo **for** como el siguiente es un ciclo infinito. Note que las expresiones en un ciclo **for** pueden ser nulas.

```
for(;;)
  instrucción;
```

Similarmente, un lazo **while** como el siguiente:

```
while(1)
  instrucción;
```

De forma similar, para un ciclo **do-while**:

```
do
  instrucción
while(0);
```

La sentencia **break** puede utilizarse para terminar cualquier ciclo. Por ejemplo, el siguiente programa lee una serie de números y calcula su suma. El ciclo termina cuando se escribe un número negativo. Se utiliza una sentencia **break** para terminar el lazo infinito. De forma similar puede hacerse con ciclos **while** y **do-while**.

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

main(){
  int n,suma = 0;
  for(;;){
```

```
    scanf(" %d",&n);
    if(n<0) break;
    suma +=n;
  }
  printf("la suma es: %d\n",suma);
  getch();
}
```

Problemas propuestos

5.8.1. Escriba los siguientes ciclos utilizando la construcción do-while.

a)
```
for(i = 0; i<20; i++){
      printf("sen(%f) = %f\n", i*180/M_PI/20,
sin(i*M_PI/180/20));
    }
```

b)
```
i = 3;
    while(i<=33){
      printf("5d\n",i);
      i += 2;
    }
```

5.8.2. Escriba un programa para calcular el promedio de N números reales, N deberá ser leído desde el teclado. Utilice ciclo **`do-while`**.

5.8.3. Resuelva la siguiente ecuación utilizando el método del punto fijo: $x^4 + x^3 - 3x^2 + 8 = 0$. Pruebe varias formas de despejar la incógnita hasta conseguir una solución que converja.

5.8.4. Escriba un programa que determine todos los divisores primos de un número leído desde el teclado. Ejemplo, si el número de entrada es 45, la salida será: 1, 3, 5. Utilice ciclos **`do-while`**.

5.8.5. Escriba un programa que determine la descomposición de un número leído desde el teclado productos potencias de sus divisores primos. Ejemplo, si el número de entrada es 45, la salida será: 3^2 5^1. Utilice ciclos **`do-while`**.

5.9. Validación de la entrada

Una aplicación común del ciclo **`do-while`** es en la validación de valores de entrada, es decir, cuando se requiere restringir el conjunto de valores que el usuario debe introducir para garantizar el buen funcionamiento de un programa. El algoritmo de validación puede resumirse en los siguientes pasos:

Algoritmo de validación:
1. Hacer ciclo
2. Leer datos

3. Mientras datos inválidos

El siguiente programa es un ejemplo de validación que permite al usuario introducir un valor mayor que cero, luego otro menor que cero y calcula su producto.

```
#include <stdio.h>

main(){
  int a,b;
  do{
    printf("Teclee un número positivo para A: ");
    scanf("%d",&a);
  }while(a<=0);
  do{
    printf("Teclee un número positivo para B: ");
    scanf("%d",&b);
  }while(b<=0);
  printf("a * b = %d",a*b);
  getch();
}
```

Ejemplo de validación de una fecha

Para validar una fecha debemos asegurar que los días caigan en el intervalo de 1 a 31 y los meses de 1 a 12. Una mejora consiste en verificar los días de acuerdo con el mes. Para febrero en año bisiesto el máximo de días es 29 en otro caso es 28. Supondremos la declaración de las siguientes variables de tipo entero: `a, m, d, anyoBisiesto, mes30dias, mes31dias y fechaValida`. Para verificar si el año bisiesto debemos determinar si es múltiplo de cuatro pero no múltiplo de 100, para eso usamos la siguiente expresión booleanas:

```
anyoBisiesto = (a%4==0)&&(a%100!=0);
```

Para verificar los meses de 30 y 31 días:

```
mes30dias = (m==4 || m==6 || m==9 || m==11);
mes31dias = (m==1||m==3||m==5||m==7||m==8||m==10||m==12);
```

Una expresión para validar fecha debe ser verdadera si el número de días es menor o igual a 30 y el mes es de 30 días o si el número del día es menor o igual que 31 y el mes es de 31 días o si el número del día es menor o igual a 28 y el mes es febrero o si el número del día es menor o igual a 29 y el mes es febrero y es año bisiesto. Esto puede escribirse en C como:

```
fechaValida = (d<=31 && mes31dias)||(d<=30 && mes30dias)||
(d<=28 && m==2)||(d<=29 && anyoBisiesto && m==2);
```

El siguiente programa lee una fecha validándola.

```
#include <stdio.h>

main(){
  int d,m,a;//día, mes, año
  int anyoBisiesto,mes30dias,mes31dias,fechaValida;
  do{
    printf("Teclee una fecha (dd mm aa): ");
    scanf("%d%d%d",&d,&m,&a);
     anyoBisiesto = (a%4==0)&&(a%100!=0);
     mes30dias = (m==4 || m==6 || m==9 || m==11);
     mes31dias = (m==1 || m==3 || m==5 || m==7
       || m==8 || m==10 || m==12);
    fechaValida = (d<=31 && mes31dias)|| (d<=30 &&
     mes30dias)||(d<=28 && m==2)||(d<=29 &&
     anyoBisiesto);
  }while(!fechaValida);
  printf("%d de ",d);
  switch(m){
    case 1:printf("enero");break;
    case 2:printf("febrero");break;
    case 3:printf("marzo");break;
    case 4:printf("abril");break;
    case 5:printf("mayo");break;
    case 6:printf("junio");break;
    case 7:printf("julio");break;
    case 8:printf("agosto");break;
    case 9:printf("septiembre");break;
    case 10:printf("octubre");break;
    case 11:printf("noviembre");break;
    case 12:printf("diciembre");break;
  }
  printf(" de %d\n",a);
  getch();
}
```

Revisaremos el problema del cálculo del promedio de números reales estudiado en la sección del ciclo `while`. Podemos reescribir el algoritmo como sigue:

Algoritmo
1. HACER
2. Solicitar nuevo valor para promediar
3. SI valor > -1 ENTONCES
4. Acumular suma e incrementar contador
5. MIENTRAS valor diferente de -1

6. SI contador > 0 ENTONCES
7. Calcular promedio e imprimir

Note que en este algoritmo solamente es necesario leer una vez dentro del ciclo. El programa es:

```
int main(){
    float suma = 0.0, num, promedio;
    int contador = 0;
    do{
        printf("Teclee un número (-1 = fin):");
        scanf(“%d,&num);
        if(num != -1){
          suma = suma + num;
          contador = contador + 1;
        }
    }while(num != -1);
    if(contador>0){
       promedio = suma/contador;
       printf("\nEl promedio es: &f\n",promedio);
    }
    else
       printf("\nNo se teclearon valores\n");
    getch();
}
```

Problemas propuestos

5.9.1. Escriba un programa que lea tres números enteros positivos validando de acuerdo a lo siguiente: el tercer número deberá ser mayor o igual al mayor de los otros dos y menor o igual a la suma de los dos primeros.

5.9.2. Una compañía de autos usados paga $5,000.00 a sus empleados por mes, más una comisión por cada automóvil vendido de $250.00, más 5% del valor de venta. Desarrolle un algoritmo controlado por centinela que lea el número de autos vendidos por un empleado y el valor de cada uno de ellos y determine el monto que se le deberá pagar al final del mes.

5.9.3. Escriba un programa que encuentre todos los enteros u entre 0 y m que satisfagan u mod 7 = 1, u mod 11=6 y u mod 13 = 5, para m leída desde el teclado.

Capítulo 6. Funciones

6.1. Declaración de funciones

Las funciones son los bloques de construcción más fundamentales en C. El uso de funciones permite la construcción de programas más grandes, mejor diseñados y en general más eficientes en su ejecución y depuración. Las funciones son bloques de instrucciones que efectúan un proceso específico y en algunos casos regresan valores. El propósito fundamental de las funciones es dividir los programas en módulos más manejables que permitan resolver problemas más complejos. Las funciones en C se declaran mediante el siguiente formato:

Tipo_devuelto nombre_de_funcion(lista de parámetros o argumentos){
  Sentencias (declaraciones, instrucciones, etc.)
}

Tradicionalmente en C se declaran como *prototipos* al inicio del programa. Después se declara la función `main`, y después se hace la declaración formal de las funciones. También pueden declararse las funciones al inicio del programa y después declarar la función `main` sin declarar prototipo. Los prototipos constan de solo el encabezado, es decir, el tipo devuelto, el nombre de la función y la lista de parámetros, en este caso la lista de parámetros puede constar solo de los nombres de los tipos de cada parámetro.

Los siguientes prototipos de funciones muestran ejemplos posibles:

Prototipo	Significado
`int f(int a, float x, float y);`	Función f que devuelve un valor entero y tiene como argumentos un valor entero y dos de punto flotante.
`double g(double a, int c);`	Función g que devuelve un valor de doble precisión y tiene como argumentos un valor de doble precisión y un valor entero.
`short m(int n, char c);`	Función m que devuelve un entero corto y tiene como parámetros un entero y un carácter.

El siguiente programa define una función para calcular el área de una esfera y la llama dentro de la función `main`. El usuario debe escribir el radio de la esfera.

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

/*prototipo de función, devuelve un flotante y lleva
un parámetro de tipo flotante*/
float areaEsfera(float);
```

```
main(){
  float r,area;
  printf("radio: ");
  scanf("%f",&r);
  /*llamada a la función areaEsfera*/
  area = areaEsfera(r);
  printf("El area de la esfera es: %f\n",area);
  getch();
}

/*definición de la función areaEsfera*/
float areaEsfera(float radio){
  return 4*3.14159265358979*radio*radio;
}
```

La función consta de solamente una sentencia **`return`**. La sentencia **`return`** se utiliza para que la función termine y regrese el valor calculado. En este caso el valor es $4\pi radio^2$. Al ser llamada el control del programa pasa a ejecutar las instrucciones de la función hasta encontrar una sentencia **`return`** o la última instrucción de la función. Posteriormente el control del programa pasa a la siguiente instrucción después de la llamada a la función. En el ejemplo la siguiente instrucción es la asignación a la variable `area` del valor calculado por la función.

La ejecución del programa inicia en la función `main`. En esta función se declaran dos variables de tipo `float`: `r` y `area`. El programa continuaría con la sentencia `printf` y `scanf`, que despliega la solicitud del radio y lee la variable `r`, respectivamente. La siguiente sentencia es una asignación a la variable `area`, a la que se le asigna el valor calculado con la función `areaEsfera`. En este punto se hace una *llamada a la función* `areaEsfera`. El control pasaría a las instrucciones de la función `areaEsfera`. La función lleva un parámetro llamado `radio`, en función `main` el valor de este parámetro es el de la variable r, que sería el valor tecleado por el usuario. La función multiplica 4 por 3.14159265358979 por el valor de `r` al cuadrado. El resultado se regresaría a quien hizo la llamada a la función, es decir, la instrucción de asignación, la que asignará el valor calculado a la variable `area`. Una vez que se ejecute la asignación, se ejecutará la sentencia siguiente, es decir, el `printf("El area de la esfera es: %f\n",area)`.

Se hace notar que el nombre del parámetro en la definición de la función es irrelevante, aunque es conveniente que sea significativo. A este parámetro se le llama *parámetro formal*. En la función `main` este parámetro es sustituido por el *parámetro real*, es decir, la variable `r`. El valor de `r` es sustituido en todos los lugares en donde aparece el parámetro `radio`.

El siguiente ejemplo define la función `max`. En este ejemplo no se define un prototipo de la función, simplemente se declara antes de la función `main`. La función contiene una sentencia `if-else` en cada alternativa hay una sentencia **`return`** que regresa el valor de `a` si la condición del `if` es verdadera, sino regresa el valor de `b`. La condición del `if` es verdadera solo cuando el parámetro `a` es mayor que el parámetro `b`. En la función `main` se llama a la función `max` dentro de la sentencia de salida `printf`. En esta llamada los parámetros reales de `max` son `numero1` y `numero2`, respectivamente. El valor de `numero1` se usará como `a` y el de `numero2` como `b` dentro de la función `max`.

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

//declaración de la función max, no se define prototipo
double max( double a, double b){
   if (a > b)
     return a;   // a es el mayor
   else
     return b;   // b es el mayor
}

int main()
{
   double numero1;
   double numero2;
   printf("Introduzca dos numeros: ");
   scanf(“%d%d”,&numero1,&numero2);
   printf("El mayor es: %d\n",max( numero1, numero2));
   getch();
}
```

Las funciones son bloques de construcción con los cuales se pueden construir nuevas funciones. Podemos definir la función `max3` que determine el mayor de tres números utilizando la función `max`. La definición sería como sigue:

```
double max3(double a, double b, double c){
  return max(max(a,b),c);
}
```

Las funciones pueden tener cualquier número de parámetros, incluso cero. El siguiente ejemplo es una función que calcula el área de un triángulo. La función acepta como parámetros los valores de la base y la altura del triángulo.

```
float areaTriangulo(float base, float altura){
  return base*altura;
}
```

De la misma forma que declaramos variables dentro de la función `main`, se pueden declarar variables dentro de una función. Las variables declaradas dentro de una función se llaman *variables locales*. Se les llama así porque solo pueden ser utilizadas dentro de la función en que son declaradas. Se produce un error de compilación de identificador no declarado o desconocido si se utiliza una variable fuera de la función en la que fue declarada. De acuerdo con esto, si se declaran variables con el mismo nombre en la función `main` y en alguna otra función, cada una de las variables tendrá validez solo en la función en que fue declarada, y no se producirá ningún tipo de error.

Considere el siguiente ejemplo de una función con una variable local. La función calcula el área de un triángulo pero dando como parámetros las longitudes de los lados. La función utiliza dos variables locales: `s` y `area`. Note que se utiliza la función `sqrt` de la biblioteca `math.h`, por lo tanto para compilar esta función deberá incluirse esa biblioteca al inicio del archivo.

```
float areaTriangulo2(float a, float b, float c){
  float s = (a+b+c)/2.0, area;
  area = sqrt((s-a)*(s-b)*(s-c)*s);
  return area;
}
```

La función `areaTriangulo2` puede mejorarse verificando que los valores pasados como parámetros satisfagan la restricción de que puedan formar un triángulo. Los valores que satisfacen la esta restricción es equivalente a que el radicando sea mayor que cero, entonces la función quedaría de la siguiente manera.

```
float areaTriangulo2(float a, float b, float c){
  float s = (a+b+c)/2.0, w;
  w = (s-a)*(s-b)*(s-c)*s;
  if(w>=0)
    return sqrt(w);
  else
    return 0.0;
}
```

Al hacer una llamada a una función el compilador verifica que los tipos de los argumentos coincidan con los tipos declarados en al función. Por ejemplo, si tenemos el siguiente prototipo de función

```
int f(int a, float x);
```

La siguiente llamada generará una advertencia al compilar. Es responsabilidad del programador hacer o no caso de dicha advertencia.

```
int x = f(3.5, 40);
```

Problemas propuestos

6.1.1. Explique el significado de los siguientes prototipos de funciones:

a) `int g(int a, float c);`
b) `char d(float f, int a, int b, char x);`
c) `double a(double c, double n, int f);`

6.1.2. Dados los prototipos del problema 1 escriba una llamada apropiada para cada función.

6.1.3. Escriba prototipos para cada una de las especificaciones de funciones:

a) Una función llamada `potencia` que devuelva un valor flotante y lleve como argumentos dos enteros.
b) Una función llamada `siguiente` que devuelva un valor entero y lleve como parámetros un flotante y un entero.
c) Una función llamada `reciproco` que devuelve un valor doble precisión y acepte un parámetro de tipo entero corto.
d) Una función llamada `inversa` acepta un carácter y devuelve un entero largo.

6.1.4. Escriba prototipos adecuados para las funciones main que se muestran.

a)
```
main(){
       int n, m, k;
       ...
       k = func1(m,n);
       ...
     }
```
b)
```
main(){
       float x, y;
       int a,b;
      …
       a = func2(2*x, b, y);
       ….
    }
```

c)
```
main(){
      int i, j;
      char c;
      double x;
      …
      x = func3(c, i, x+j);
      ….
     }
```

6.1.5. Escriba una función en C para calcular el volumen de una esfera. Escriba una función `main` para probar la función.

6.1.6. Escriba una función en C para determinar el máximo común divisor de dos enteros.

6.1.7. Escriba una función en C para calcular el volumen de un cilindro con radio R y altura h. Escriba una función `main` que solicite el radio y la altura del cilindro y calcule su volumen utilizando la función que definió.

6.1.8. Escriba una función que calcule la raíz cúbica de un número real. Utilice la función `pow` de la biblioteca `math.h`. Haga un programa para desplegar una tabla con las raíces cuadradas y cúbicas de los números de 1 a 20 para probar su función.

6.1.9. Escriba funciones que regresen un valor entero 0 o 1 si los argumento enteros cumplen con lo siguiente:
a) Es un número primo.
b) Es un número perfecto, es decir, la suma de sus divisores es igual a él mismo.
c) Son una terna pitagórica, es decir, $a^2+b^2=c^2$, donde *a*, *b* y *c* son los argumentos.
d) Sus dos argumentos son primos entre si.
e) Sus cuatro argumentos son diferentes.
e) Sus primeros cuatro argumentos son diferentes al quinto argumento.

6.1.10. El factorial de un número entero *n* positivo, denotado por *n*!, se define como el producto de todos los números desde 1 hasta *n*, es decir, *n*! = 1*2*3*…**n*, por definición 0! = 1. Defina una función que calcule el factorial de un número. Utilizando esta función se puede calcular los coeficientes de la expansión de un binomio a cualquier potencia entera positiva. El coeficiente *k*-ésimo de la potencia *n* es $C_k^n = \frac{n!}{k!(n-k)!}$, donde $0 \le k \le n$, utilice la función factorial para definir una función que calcule los coeficientes de la expansión binomial. Escriba una función `main` que solicite los coeficientes *a*, *b* y el exponente *n* de un binomio de la forma $(ax + by)^n$, y calcule y despliegue la expansión binomial. Ejemplos: $(x+y)^4 = x^4+4x^3y+6x^2y^2+4xy^3+y^4$, $(2x+3y)^3 = 8x^3+36x^2y+54xy^2+27y^3$.

6.2. Funciones de tipo void

Se pueden definir funciones que no devuelvan ningún valor. En algunos lenguajes como Pascal esto se conoce como un procedimiento. Se utiliza el tipo **`void`** para indicar que la función no regresa ningún valor. Una función tipo **`void`** no debe aparecer en una instrucción de asignación. El prototipo para funciones tipo **`void`** es:

```
void nombre_función(agrumentos)
```

Un ejemplo es la función `printf`, esta no regresa ningún valor. Por ejemplo, si deseamos una función que se encargue de desplegar el nombre del programador en la pantalla, podríamos usar la siguiente:

```
void despliegaNombre(){
  printf("Programa hecho por Fulanito de Tal.\n");
}
```

Note que la función no tiene argumentos, no obstante debemos poner los paréntesis vacíos. También sería válido definir la función poniendo la palabra **`void`** dentro de los paréntesis como se muestra a continuación.

```
void despliegaNombre(void){
  printf("Programa hecho por Fulanito de Tal.");
}
```

Para llamar una función de tipo **`void`**, lo hacemos como con la función `printf`, es decir, basta con poner el nombre de la función seguido de sus argumentos o al menos paréntesis vacíos si no lleva argumentos. La llamada a la función `despliegaNombre` sería

```
despliegaNombre();
```

Es necesario poner los paréntesis aunque la función no requiera argumentos para que el compilador sepa que se trata de una función.

La entrada y salida de datos se puede facilitar con el uso de funciones. Es importante diseñar los programas basándonos en funciones. Una forma de diseño de aplicaciones es el diseño *descendente*. En este enfoque primero se diseña la aplicación partiendo de los aspectos generales del problema que se va a resolver. Esto se hace definiendo la estructura básica de la función `main` basándonos en la utilización de funciones que posteriormente serán definidas con precisión. Después se procede a construir cada una de las funciones que se han definido en términos más precisos. Puede ser que estas funciones requieran de otras funciones que se especificarán posteriormente. El proceso se continúa hasta terminar la aplicación.

Como ejemplo construyamos una aplicación basada en menús para hacer conversiones de unidades físicas. Primero definiremos el esquema básico de la función main. Supondremos que el programa se ejecutará hasta que el usuario elija la opción de terminar. Una primera versión de la función main sería:

```
main(){
  do{
    desplegarMenu();
    opcion=leerOpcion();
    ejecutarOpcion(opcion);
  }while(opcion!=SALIR);
```

```
}
```

La función `desplegarMenu()` borra la pantalla y muestra las opciones de conversión del programa. Esta función es de tipo **`void`** ya que no se requiere regresar ningún valor al desplegar el menú. `leerOpcion` es una función que leerá un carácter que corresponda a alguna de las opciones de menú, ignorando los caracteres ilegales. Una vez leída la opción a ejecutar se llamará a la función `ejecutarOpcion(opcion)` para solicitar los datos necesarios y llevar a cabo la conversión. Comencemos definiendo con más precisión la función `desplegarMenu()`. Utilizamos la función `system("cls")` de la biblioteca `stdlib.h` para borrar la ventana de salida, esta función permite invocar los comandos del sistema operativo, el comando "cls" borra la pantalla.

```
void desplegarMenu(){
  system("cls");
  printf("\n\nCONVERSION DE UNIDADES\n\n"); //algunas  líneas
vacías
  printf("1. Velocidad m/s a km/h");
  printf("2. Velocidad km/h a mi/hr");
  printf("3. Temperatura ºC a ºF");
  printf("4. Temperatura ºF a ºC");
  printf("5. Salir");
  printf("\n\n\n        opcion: ");
}
```

Limitaremos las conversiones a velocidad y temperatura. Con esto en mente podemos definir la función `ejecutarOpcion(opcion)`. Esta llevará como parámetro un carácter para seleccionar la opción a ejecutar. Note que en la instrucción **`switch`** se usan caracteres como etiquetas:

```
void ejecutarOpcion(char opcion){
  float v,t;
  switch(opcion){
    case '1':printf("Velocidad en mi/hr: ");
             scanf(" %f",&v);
                      printf("La  velocidad  en  km/hr  es:
%.2f",v*1.609);
             break;
    case '2':printf("Velocidad en km/hr: ");
             scanf(" %f",&v);
                      printf("La  velocidad  en  mi/hr  es:
%.2f",v/1.609);
             break;
    case '3':printf("Temperatura en C: ");
             t = leerTemp(1);
             printf("La Temperatura en F es: %.2f",t);
```

```
            break;
    case '4':printf("Temperatura en F: ");
             t = leerTemp(2);
             printf("La Temperatura en C es: %.2f",t);
             break;
  }
}
```

En esta función hemos utilizado otro función que aún no hemos definido, `leerTemp`. Esta función tiene el propósito de validar la entrada de temperaturas. La validación consiste en garantizar que el valor tecleado sea una temperatura válida, es decir, mayor que el cero absoluto según la escala. Si el parámetro de esta función es 1, se leerán grados centígrados, sino, se leerán grados Farenheit. La función `leerTemp` es:

```
float leerTemp(int tipo){
  float x;
  do{
    scanf(" %f",&x);
    if(tipo==1&&x<-273.0)
              printf("La  temperatura  debe  ser  mayor  a
-273!\nTemperatura en C: ");
    if(tipo==1&&x<-450.0)
              printf("La  temperatura  debe  ser  mayor  a
-450!\nTemperatura en F: ");
  }while((tipo==1&&x<-273.0)||(tipo==2&&x<-450.0));
  return x;
}
```

Solo falta la función que lee las opciones de menú. En esta función se usa la función `getch()` que lee un carácter desde el teclado. Esta función no despliega el carácter que se presiona, así que hay que desplegarlo por nuestra cuenta.

```
char leerOpcion(){
  char c;
  do{
    c = getch();
    if(c>='1'&&c<='5')
      printf("%c",c);
  }while(c<'1'||c>'5');
  return c;
}
```

El programa completo agregando alguna declaración es el siguiente. Se han declarado los prototipos las funciones al principio para respetar la costumbre del lenguaje C. Se utilizo la biblioteca `iostrem` para acceder a la función `system`. Esta función permite invocar comandos del sistema, en este caso invocamos el comando "`cls`" que borra la ventana, lo cual da un aspecto más profesional.

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

void desplegarMenu(void);
char leerOpcion(void);
void ejecutarOpcion(char );
float leerTemp(int);

main(){
  char opcion;
  do{
    desplegarMenu();
    opcion=leerOpcion();
    ejecutarOpcion(opcion);
  }while(opcion!='5');
}

void desplegarMenu(){
  system("cls");
  printf("\n\nCONVERSION DE UNIDADES\n\n");
  printf("1. Velocidad mi/hr a km/hr\n\n");
  printf("2. Velocidad km/h a mi/hr\n\n");
  printf("3. Temperatura C a F\n\n");
  printf("4. Temperatura F a C\n\n");
  printf("5. Salir\n");
  printf("\n\n\n        opcion: ");
}

char leerOpcion(){
  char c;
  do{
    c = getch();
    if(c>='1'&&c<='5')
      printf("%c",c);
  }while(c<'1'||c>'5');
  return c;
}

float leerTemp(int tipo){
  float x;
  do{
    scanf(" %f",&x);
    if(tipo==1&&x<-273.0)
                printf("La  temperatura  debe  ser  mayor  a
-273!\nTemperatura en C: ");
```

```
    if(tipo==1&&x<-450.0)
                printf("La  temperatura  debe  ser  mayor  a
-450!\nTemperatura en F: ");
  }while((tipo==1&&x<-273.0)||(tipo==2&&x<-450.0));
  return x;
}

void ejecutarOpcion(char opcion){
  float v,t;
  switch(opcion){
    case '1':printf("\nVelocidad en mi/hr: ");
             scanf(" %f",&v);
                          printf("La  velocidad  en  km/hr  es:
%.2f",v*1.609);
             break;
    case '2':printf("\nVelocidad en km/hr: ");
             scanf(" %f",&v);
                        printf("\nLa  velocidad  en  mi/hr  es:
%.2f",v/1.609);
             break;
    case '3':printf("\nTemperatura en C: ");
             t = leerTemp(1);
             printf("La Temperatura en F es: %.2f",
t*9.0/5.0+32);
             break;
    case '4':printf("\nTemperatura en F: ");
             t = leerTemp(2);
             printf("La Temperatura en C es: %.2f", (t-
32)*5.0/9.0);
             break;
  }
  getch();
}
```

Problemas propuestos

6.2.1. Escriba una función que despliegue un rectángulo de asteriscos (‘*’) como se muestra. Pase como argumentos el número de columnas y renglones del rectángulo. Por ejemplo, si los parámetros son 6 y 3, se desplegará

```
******
******
******
```

6.2.2. Escriba una función que despliegue un cuadrado usando la función del problema anterior.

6.2.3. Escriba una función que imprima una fecha, ponga como parámetros el día, el mes y el año. Incluya tres formatos de salida: dd-mm-aaaa, dd/mm/aaaa, dd de mm de aaaa, en este último mm es el nombre del mes. Por ejemplo si los parámetros son 5, 7, 2011, la fecha es: 5-7-2011 o 5/7/2011 o 5 de julio de 2011. Incluya un cuarto parámetro para especificar el formato deseado.

6.2.4. Escriba una función que acepte un tiempo en segundos y lo despliegue en años, meses, días, horas, minutos y segundos. Por ejemplo, 233874 seg. = 2 días, 16 horas, 57 min., 54 seg.

6.2.5. Para calcular la trayectoria de un proyectil podemos usar las ecuaciones

$$x = (V \cos \theta)\, t$$
$$y = (V \operatorname{sen} \theta)\, t - \tfrac{1}{2}\, g t^2$$

Donde V es la velocidad inicial del proyectil (m/s), θ es el ángulo de inclinación (en radianes), t es el tiempo, g es la aceleración de la gravedad (9.8 m/s^2), y x y y son las coordenadas del proyectil en el tiempo t. Escriba una función que acepte los parámetros para V y θ y despliegue las coordenadas del proyectil en intervalos de 0.1 s hasta que el proyectil llegue a tierra.

6.2.6. Escriba una aplicación basada en menú para resolver ecuaciones de primer grado, ecuaciones de segundo grado, ecuaciones simultánea de 2x2 y ecuaciones simultáneas de 3x3. Utilice el diseño descendente.

6.3. Ejemplos de funciones matemáticas

Algunas funciones matemáticas no se encuentran en la biblioteca `math.h`. Podemos implementarlas fácilmente en C. Por ejemplo, la raíz cúbica puede extraerse mediante la función `pow`. El código en C es el siguiente:

```
double root3(double x){
  return pow(x,1.0/3.0);
}
```

Note que debemos escribir 1.0/3.0 para el exponente ya que si usamos 1/3 se hará una división entera dando como resultado de elevar a la potencia cero. Podemos definir fácilmente las funciones trigonométricas complementarias, por ejemplo la cotangente se define como:

```
double cot(double x){
  return 1.0/tan(x);
}
```

La función factorial es muy útil en muchos cálculos. La función factorial n se define como $n! = 1 \cdot 2 \cdot 3 \cdot 4 \ldots n$. Es fácil calcularla utilizando un lazo **`for`**.

```
f=1;
for(int i=2;i<=n;i++)
  f *=i;
```

El valor de la función factorial crece con mucha rapidez, por ejemplo 10! = 3628800. Por este motivo es conveniente definir el factorial de tipo doble precisión. La función sería.

```
double fact(int n){
  double f = 1.0;
  for(int i=2;i<=n;i++)
    f *=i;
  return f;
}
```

Algunas funciones requieren la evaluación de series de potencias. Por ejemplo, la función exponencial esta definida por la expansión.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Podemos calcularla mediante la siguiente función.

```
double exp(double x){
  int i;
  double suma=0.0,pot=1.0,f=1.0;
  for(i=1;i<=20;i++){
    suma += pot/f;
    pot *= x;
    f *= i;
  }
  return suma;
}
```

Esta función evaluada en 1.0 genera 2.7182818284590455. La función `exp` de la biblioteca `math.h` genera 2.7182818284590451. Es de esperarse que para valores más grandes de x los valores generados sean diferentes, habría que hacer un ciclo de más de 20 pasos para obtener la misma precisión o usar otro método para calcularla. Las funciones trigonométricas también se definen en términos de series de potencias. La serie para el seno es:

$$\text{sen } x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

La función para evaluarla es:

```
double sen(double x){
  int i,s=1;
  double suma=0.0,pot=x,f=1.0;
  for(i=1;i<=20;i+=2){
    suma += s*pot/f;
    pot *= x*x;
    f *= (i+1)*(i+2);
    s = -s;
  }
  return suma;
}
```

El valor que genera esta función para x = 89º es 0.99984769515639105, la función sin de la biblioteca math.h genera 0.99984769515639270.

Problemas propuestos

6.3.1. Escriba funciones en C para calcular seno, coseno y tangente hiperbólicas. Las funciones hiperbólicas se definen de la siguiente forma:

$$\operatorname{senh} x = \frac{e^x - e^{-x}}{2} \qquad \cosh x = \frac{e^x + e^{-x}}{2} \qquad \tanh x = \frac{\operatorname{senh} x}{\cosh x}$$

6.3.2. Defina funciones para calcular lo siguiente:

a) cosecante de x b) secante hiperbólica de x c) ln|1+*x*| d) $f(x) = \frac{1}{\sqrt{1-x^2}}$

6.3.3. Escriba una función para calcular e^{-x^2}. Puede usar la expansión de e^x sustituyendo cada *x* por $-x^2$. Escriba una función main para comparar el resultado calculado por su función y usando las funciones de la biblioteca math.h.

6.3.4. Diseñe una función con dos parámetros *x* y *n*, tal que regrese el valor de las siguientes expresiones dependiendo del valor de *x*:

$$f(x,n) = \begin{cases} x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2}, si\ x \geq 0 \\ -\frac{x^{n-1}}{n-1} + \frac{x^{n+1}}{n+1}, si\ x < 0 \end{cases}$$

6.3.5. Escriba una función para calcular ln(1+*x*) para |*x*|<1 basandose en la expansión siguiente

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$$

6.4. Números aleatorios

Los números aleatorios tienen mucha aplicación sobre todo en juegos, simulaciones y para probar algoritmos. Vamos a revisar brevemente las funciones útiles en la generación de números aleatorios.

La función básica de generación de aleatorios es la función `rand()` de la biblioteca `stdlib.h`. Esta función genera un número semialeatorio en el intervalo entre 0 y 32767. El hecho de que sea semialeatorio indica que los valores que se generan son siempre siguiendo la misma secuencia que depende de una semilla. La semilla puede cambiarse mediante la función `srand(semilla)`. Esta función establece la nueva semilla de números aleatorios.

Si se desea generar números aleatorios en un intervalo diferente se debe tomar el módulo del valor generado y quizás sumarle un valor constante. Por ejemplo si deseamos números aleatorios entre 0 y 9, inclusive, se deberá escribir

```
rand() % 10
```

Para simular el tiro de un dado debemos generar números entre 1 y 6. Esto lo hacemos con la expresión:

```
1 + rand() % 6
```

Podemos manejar estas expresiones dentro de funciones. Por ejemplo: la función que genere un tiro de un dado sería:

```
double tiroDado(){
  return 1 + rand()%6;
}
```

Con la función `rand()` podemos generar números aleatorios de doble precisión en algún intervalo definiendo una función apropiada como la siguiente:

```
double randDoble(){
  return (double)(rand()/32767.0);
}
```

Note que la división se hace entre el número flotante 32767.0 no entre el entero 32767 ya que se al dividir entre un entero obtendríamos siempre cero de resultado. La función anterior genera números aleatorios en el intervalo de 0.0 a 1.0. Para probar la función escriba la siguiente función `main` y ejecútela.

```
main(){
  for(int i=0; i<20;i++)
    printf("%f",randDoble());
  getch();
```

```
}
```

Note que siempre que se ejecuta el programa se genera la misma secuencia de números aleatorios. Para secuencias distintas debemos modificar la semilla de los números aleatorios. Una forma práctica y que es muy difícil que se repita es tomar la hora del día para establecer dicha semilla. La función `time(0)` de la biblioteca `time.h` nos da la hora actual. Si pasamos esta hora como argumento a la función `srand` estableceremos siempre una semilla diferente. En la función `main` del programa debemos poner

```
srand(time(0));
```

Una forma de obtener números aleatorios entre dos límites cualesquiera es mediante la expresión:

```
valor = limite_inferior + rand()*(limite_superior – limite_inferior);
```

Esta expresión podemos usarla en una función para obtener números aleatorios entre dos límites:

```
int aleatorio(int limite_inferior, int limite_superior){
  return limite_inferior + rand()%(limite_superior – limite_inferior);
}
```

Como un ejemplo de números aleatorios consideremos un juego en el que jugador trata de adivinar un número elegido por la computadora en seis intentos. El jugador elige un número entre 1 y un límite máximo. La computadora responde indicando si el número es mayor o menor que el número elegido por ella, o si es igual, indicando que el jugador gano. Este proceso se repite seis veces, si al final de los seis intentos el jugador no ha adivinado, el jugador pierde, sino, gana. El algoritmo es el siguiente:

Algoritmo Juego de Adivinanza.

1. La computadora elige número aleatorio.
2. Repetir 6 veces
3. leer siguiente intento
4. si los números son iguales, jugador ganó y termina
5. sino si el elegido es mayor
6. informar que el elegido es mayor
7. sino
6. informar que el elegido es menor
5. fin de ciclo
6. Computadora ganó.

Como el ciclo ha de ejecutarse al menos una ocasión, es conveniente un lazo **do-while**. El programa completo es el siguiente:

```
#include <stdio.h>

int aleatorio(int limite_inferior, int limite_superior){
     return  limite_inferior  +  rand()%(limite_superior  -
limite_inferior);
}

int elegir(int inferior, int superior){
  int elegido = aleatorio(inferior,superior);
  printf("La computadora ha elegido...\n");
  return elegido;
}

int leerIntento(int n){
  int intento;
  printf("Teclee su intento %d: ",n);
  scanf(“%d”,&intento);
  return intento;
}

main(){
  int oculto,numIntentos=0,numero;
  printf("JUEGO DE ADIVINANZA\n");
  oculto = elegir(1,100);
  do{
    numIntentos++;
    numero = leerIntento(numIntentos);
    if(oculto==numero){
      printf("Usted GANA...\n");
      getch();
      return 0;
    }else
      if(oculto>numero)
        printf("Numero mayor.\n");
      else
        printf("Numero menor.\n");
  }while(numIntentos<6);
  Printf("El número fue: %d\n",oculto);
  getch();
}
```

En el programa se utiliza la función para generar números aleatorios entre dos límites. También se definió la función `elegir` para seleccionar el número aleatorio de la computadora y la función `leerIntento` para leer cada uno de los números del jugador,

el parámetro de esta función se utiliza para indicar al jugador cual es el número de intentos que lleva.

Problemas propuestos

6.4.1. Escriba una función que genere un número real aleatorio entre dos valores reales positivos o negativos.

6.4.2. Escriba una función que genere un juego pronósticos de “melate”. Este consiste de la elección de seis números aleatorios distintos, llamados números naturales, entre 1 y 56, y un número aleatorio llamado el adicional. Deberá imprimir los números aleatorios en orden y después el adicional.

6.4.3. Escriba una función que verifique una quiniela de “melate” y despliegue el número de aciertos que obtuvo el jugador. La función llevará como parámetros la quiniela de “melate” (6 números y el adicional) y los seis números elegidos por el jugador. Imprima el lugar que ocupa entre los premios (si obtuvo premio) de acuerdo a la tabla siguiente.

Lugar	Asiertos	Premio
1er	6 números naturales	Variable
2º	5 números naturales y el adicional	Variable
3er	5 números naturales	Variable
4º	4 números naturales y el adicional	Variable
5º	4 números naturales	Variable
6º	3 números naturales y el adicional	$161.29
7º	3 números naturales	$ 43.01
8º	2 números naturales y el adicional	$ 43.01
9º	2 números naturales	$ 21.50

6.4.4. Escriba un programa que genere la quiniela del “melate” en seis variables n1, n2, n3, n4, n5 y n6 para los números naturales y una variable para el adicional. Luego simule que 10000 jugadores juegan para ganar generando las quinielas correspondientes cada una con seis valores. Determine cuantos jugadores obtienen cada uno de los lugares de la tabla del problema anterior.

6.5. Modificador `static`

El modificador **`static`** es uno de los modificadores de las diferentes clases de almacenamiento de que dispone el lenguaje C. Los otros modificadores son **`auto`**, **`register`** y **`extern`**. El modificador **`auto`** se utiliza para variable locales y es el valor por omisión, por eso se utiliza rara vez. Las variables de clase **`register`** se utiliza para sugerir al compilador que esa variable sea conservada en los registros del procesador. El modificador **`extern`** se utiliza para declarar variables o funciones externas a un archivo, también este es el valor por omisión.

Si declaramos una variable local como **`static`** su valor se conservará entre diferentes llamadas a la función. Por ejemplo considere el siguiente programa.

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

void f(){
  int x=5;
  printf("x = %d\n",x);
  x++;
}

void g(){
  static int x=5;
  printf("x = %d\n",x);
  x++;
}

main(){
  f();
  g();
  f();
  g();
  getch();
}
```

Este programa imprimirá

```
5
5
5
6
```

La variable local `x` en `f` toma el valor de 5 cada vez que se llama a la función. Por otro lado la variable local `x` en `g`, que esta declarada como **`static`**, retiene su valor entre llamadas. Las funciones también pueden ser **`static`**, en tal caso serán locales al archivo donde se declaran y no podrán usarse en otro archivo.

La declaración de variables en C tiene el alcance de un bloque. Recuerde que un bloque es un conjunto de sentencias encerrados dentro llaves. Si declaramos una variable dentro de un bloque, esta solo será conocida dentro de el. El atributo **`static`** permite que la variable retenga su valor cada que el bloque se ejecuta. El siguiente ejemplo muestra dos variables declaradas en un bloque, una llamada temporal se inicia cada que el bloque se ejecuta y la otra llamada permanente que solo se inicia la primera vez que se ejecuta el bloque, conservando su valor entre llamadas.

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

```
#include <conio.h>

main() {
  int i;
  for (i = 0; i < 3; ++i) {
    int temporal = 1;
    static int permanente = 1;
    printf("Temporal %d Permanente %d\n",temporal,
            permanente);
      ++temporal;
      ++permanente;
  }
  getch();
}
```

Numeros de Fibonacci

Una secuencia de números de mucha importancia es la secuencia de Fibonacci. Los dos primeros valores de la secuencia de Fibonacci son 1 y 1. Los demás valores se obtienen sumando los dos anteriores, la secuencia que se obtiene es: 1, 1, 2, 3, 5, 8, 13, 21, 34, … La siguiente función llamada `fibonacci` calcula dicha secuencia. Se utilizan dos variables `f1` y `f2` declaradas como estáticas para recordar su valor anterior. La función regresa un entero largo ya que crece con bastante rapidez. Note que para generar la secuencia se debe llamar a la función desde 1 hasta el valor deseado.

```
long int fibonacci(int count){
  static long int f1=1,f2=1;
  long int f;
  if(cont<3)
    f = 1;
  else
    f = f1+f2;
  f2 = f1;
  f1 = f;
  return f;
}
```

Un ejemplo de ejecución de la función `fibonacci` es el siguiente.

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

long int fibonacci(int count);

main(){
  int cont,n;
  printf("Cuantos numeros de Fibonacci desea: ");
```

```
  scanf(" %d",&n);
  for(cont=1;cont<=n;cont++)
    printf("n= %d\t Fib=%d\n",cont,fibonacci(cont));
  getch();
}

long int fibonacci(int count){
  static long int f1=1,f2=1;
  long int f;
  if(count<3)
    f = 1;
  else
    f = f1+f2;
  f2 = f1;
  f1 = f;
  return f;
}
```

Problemas propuestos

6.5.1. Describir la salida generada por los siguientes programas.

a)
```
#include <stdio.h>
  int func1(int cont);
  main(){
    int a, cont;
    for(cont=1;cont<=5;cont++){
      a=func1(cont);
      printf("%d ",a);
    }
  }
  func1(int x){
    int y=0;
    y+=x;
    return y;
  }
```

b)
```
#include <stdio.h>
  int func1(int cont);
  main(){
    int a, cont;
    for(cont=1;cont<=5;cont++){
      a=func1(cont);
      printf("%d ",a);
    }
  }
  func1(int x){
    static int y=0;
    y+=x;
```

```
  return y;
}
```

c)
```
#include <stdio.h>
int func1(int cont);
int func2(int cont);
main(){
  int a, b = 1,cont;
  for(cont=1;cont<=5;cont++){
    b= func1(a)+ func2(a);
    printf("%d ",b);
  }
}
func1(int a){
  int b;
  b = func2(a);
  return b;
}
func2(int a){
  static int b=1;
  b++;
  return b+a;
}
```

6.5.2. Escriba una versión de la función `fibonacci` que no utilice variables de clase **`static`** y calcule el n-ésimo número de Fibinacci.

6.5.3. Escriba un programa que utilice la función `fibonacci` del problema anmterior para mostrar que el cociente de dos números de Fibonacci consecutivos tiende a $(1+\sqrt{5})/2.0 = 1.61803$ conforme los números son más grandes.

6.6. Parámetros por valor y por referencia

En la función `scanf` requiere que sus argumentos variables sean precedidos por el operador dirección &. Este operador informa al compilador que se pasará como parámetro la dirección de la variable, no su valor. Si se omite este operador el programa podrá ser compilado pero no funcionará como se espera. Los parámetros precedidos de ampersand (&) se conocen como parámetros de *referencia* e indican al compilador que las variables pueden ser modificadas dentro de la función. Por ejemplo, el siguiente programa aparentemente modifica una variable pero al ejecutarlo encontrará que no es así.

```
void f(int a){
  a = 5;
}
main(){
  int b=3;
  printf("%d ",b);
```

```
  f(b);
  printf("%d ",b);
  getch();
}
```

Para definir parámetros de referencia es necesario anteponer a las variables el operador de indirección o desreferencia (*) que indica que lo que pasará como parámetro será una dirección de memoria. Si declaramos una variable `int *x`, para hacer referencia al valor de la variable debemos usar la notación `*x`, por ejemplo si deseamos asignarle un valor al lugar de memoria apuntado por `x`, escribiremos `*x = valor`. Cabe aclarar que `x` no es un entero, sino un apuntador a un entero, en un capítulo posterior trataremos ampliamente de los apuntadores. Por otro lado si tenemos la declaración `int y`, `&y` indica la dirección donde se encuentra el valor de `y`. La función corregida sería.

```
void f(int *a){
  *a = 5;
}
main(){
  int b=3;
  printf("%d ",b);
  f(&b);
  printf("%d ",b);
  getch();
}
```

Los valores impresos serán: 3 5. Como un ejemplo (poco práctico) del uso de parámetros de referencia veamos una función que suma dos tiempos. Suponemos que el tiempo lo representamos como horas, minutos y segundos, cada uno como una cantidad entera. La función `suma` acepta como parámetros los valores de dos tiempos y regresa con parámetros por referencia el resultado de la suma de estos tiempos debidamente normalizado. El programa siguiente define la función `suma` y una función llamada `escribe` para desplegar un tiempo.

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

void suma(int,int,int,int,int,int,int *,int *,int *);
void escribe(int,int,int);

main(){
  int h1=5,m1=34,s1=17,h2=16,m2=45,s2=53,h3,m3,s3;
  escribe(h1,m1,s1);
  escribe(h2,m2,s2);
  suma(h1,m1,s1,h2,m2,s2,&h3,&m3,&s3);
  escribe(h3,m3,s3);
  getch();
}
```

```
void suma(int h1, int m1, int s1, int h2, int m2, int s2,
              int *h3, int *m3, int *s3){
  *m3 = 0;
  *h3 = 0;
  *s3 = s1+s2;
  if(*s3>59){
    *s3 = *s3-60;
    *m3 = 1;
  }
  *m3 = m1+m2+*m3;
  if(*m3>59){
    *m3 = *m3-60;
    *h3 = 1;
  }
  *h3 = h1+h2+*h3;
}

void escribe(int h, int m, int s){
  printf("%dh%dm%ds\n",h,m,s);
}
```

Note que no es necesario poner los identificadores de los parámetros en los prototipos de las funciones. Las dos funciones son de tipo **`void`**, por tanto basta con escribir el nombre de la función y los argumentos para invocarlas.

No puede usarse un valor entero constante como parámetro real si es un parámetro por referencia. Por ejemplo, si definimos lo siguiente

```
void f(int *x){
  *x = 5;
}

main(){
  int a;
  f(&a);
  f(5);
}
```

Se generará un error en la línea `f(5)` indicando que no se puede convertir de **`int`** a **`int`***. También `f(a)` generará un error similar.

Nuevamente la cuadrática

Definiremos una función con parámetros por referencia para resolver la ecuación cuadrática. Recordemos que las soluciones de una cuadrática pueden ser de dos tipos, reales o complejas. Debemos incluir un argumento por referencia para informar la naturaleza de la

solución. La función requiere por tanto tres parámetros por valor para los coeficientes, dos parámetros por referencia para la raíces y un parámetro por referencia para indicar si las raíces son reales o complejas. La función es la siguiente:

```
void cuadratica(double a,double b,double c,
                double *x1,double *x2,int *caso){
  double d = b*b - 4*a*c;
  if(d>0){
    *x1 = (-b - sqrt(d))/2/a;
    *x2 = (-b + sqrt(d))/2/a;
    *caso = 1;
  }else{
    *x1 = -b/2/a;
    *x2 = (sqrt(-d))/2/a;
    *caso = 2;
  }
}
```

Problemas propuestos

6.6.1. Dada la siguiente definición de funciones diga cuales de las llamadas son válidas y cuales no.

```
void intercambia(int *a,int *b){
  int t;
  t = *a; *a = *b; *b = t ;
}
int suma(int x,int y){
  return x+y ;
}
```

a) `intercambia(suma(&x),&x) ;`
b) `intercambia(7,3) ;`
c) `suma(20);`
d) `x = suma(11) ;`
e) `z = intercambia(&x,&y);`
f) `intercambia(x,y);`
g) `intercambia(&x,&y);`

6.6.2. Escriba una función para resolver un sistema de ecuaciones simultáneas de 2x2. Ponga como argumentos por valor los coeficientes de las ecuaciones y como argumentos por referencia las incógnitas. Si el sistema no tiene solución regrese ceros en la solución.

6.6.3. Haciendo uso de la función definida en el problema anterior diseñe una aplicación utilizando el método de diseño descendente para hacer un programa interactivo para resolver ecuaciones simultáneas de 2x2.

6.6.4. Escriba una función que acepte como argumentos las ecuaciones de dos rectas (pendiente y ordenada al origen) y regrese en argumentos por referencia las coordenadas de la intersección y en otro argumento por referencia un valor 1 si hubo intersección o 0 si no la hubo.

6.6.5. Escriba una función para determinar los puntos de intersección de una recta y un círculo. Informe, mediante un argumento, cuantas intersecciones hubo, 0, 1 o 2.

6.6.6. Escriba una función para determinar los puntos de intersección de dos elipses. Informe, mediante un argumento, cuantas intersecciones hubo, 0, 1, 2, 3, 4 o 5. El valor 5 indicará que las elipses son coincidentes.

6.7. Funciones recursivas

Se dice que una función es recursiva si dentro del cuerpo de la función se hace una llamada a si misma. Por ejemplo.

```
void f(int n){
  if(n>0){
    printf("%d ",n);
    f(n-1);
  }
}
```

La función f acepta un parámetro entero, si es mayor que cero lo imprime y se llama a sí misma. Esta llamada se hace con un parámetro igual a `n-1`. Por ejemplo, si invocamos `f(5)`, se imprimirá 5 y se invocará `f(4)`, esta llamada imprimirá 4 e invocará `f(3)`, así sucesivamente hasta que se llame a `f(1)`, la cual imprimirá 1 e invocará `f(0)`. En este punto la función recibe parámetro igual a 0 y simplemente termina. Como resultado la función imprimirá la lista de todos los números desde el valor del parámetro hasta 1. Las funciones recursivas deben cumplir lo siguiente:

1. Debe existir una salida en la que no se haga la llamada recursiva
2. La llamada recursiva debe ser versión más simple que la llamada que la invocó.

En el caso de la función definida anteriormente la salida no recursiva se da cuando no se cumple la condición de la sentencia **`if`**. La llamada recursiva se ejecuta para un valor de parámetro menor que el de la llamada original, en ese sentido es una versión más simple. Podemos cambiar la llamada recursiva por `f(n+1)`, esto llevará a que la función se llame recursivamente hasta que se recorran todos los enteros positivos y se llegue a un valor negativo. Esto puede hacer que el programa agote la memoria disponible y el programa se cuelgue.

Cada llamada recursiva implica guardar la dirección de retorno y los parámetros de la llamada en la memoria. Esto implica que si se hacen suficientes llamadas recursivas, la

memoria podría agotarse. Un ejemplo típico de la recursividad es la función factorial vista anteriormente. Una versión recursiva es la siguiente.

```
double fact(int n){
  if(n==0)
    return 1;
  else
    return n*fact(n-1);
}
```

Por ejemplo se hacemos la llamada para `fact(5)` se generarán las siguientes

```
fact(5) = 5*fact(4)
     = 5*4*fact(3)
     = 5*4*3*fact(2)
     = 5*4*3*2*fact(1)
     = 5*4*3*2*1*fact(0)
     = 5*4*3*2*1*1
     = 5*4*3*2*1
     = 5*4*3*2
     = 5*4*6
     = 5*24
     = 120
```

Algunos algoritmos se plantean en matemáticas de forma recursiva. Un ejemplo de esto es la determinación del máximo común divisor (MCD). Un algoritmo recursivo es el siguiente.

1. El MCD de dos números x e y es y si y<x y x es divisible entre y.
2. El MCD de dos números x e y es igual al MCD de y y x si x<y
3. El MCD de dos números x e y es igual al MCD de y y el residuo de la división x/y.

El primer paso del algoritmo es la salida no recursiva del algoritmo. El segundo paso solo indica que el MCD de un número más pequeño que otro es igual al MCD del más grande y el pequeño, esta es una llamada recursiva que solo reacomoda los parámetros. Por último el paso 3 establece que el MCD de dos números es igual al MCD del número más pequeño y el residuo de la división del grande entre el pequeño, este residuo siempre es más pequeño que el divisor, por tanto es una versión más simple que la llamada original.

Si definimos la funcion mcd(x,y) como el MCD de x y y, El algoritmo lo podemos expresar se la siguiente manera.

1. SI (y<x)&&(x%y==0) ENTONCES
   REGRESAR y
2. SINO
3. SI x<y ENTONCES
   REGRESAR gcd(y,x)

4. SINO

REGRESAR gcd(y, x % y)

La versión en C es fácil de escribir de la siguiente manera.

```
int mcd(int x, int y){
  if((y<=x)&&(x%y == 0)
    return y;
  else if(x < y)
    return gcd(y,x);
  else
    return gcd(y,x % y);
}
```

La llamada `mcd(48,20)` generará las siguientes llamadas:

```
mcd(48,20) = mcd(20,48%20)
           = mcd(20,8)
           = mcd(8,20%8)
           = mcd(8,4)
           = 4
```

La siguiente función es una versión recursiva para calcular el n-ésimo número de Fibonacci. La diferencia con los anteriores es que se hacen dos llamadas recursivas cada vez que la función es invocada con un parámetro mayor que 1. Esto implica que el número de llamadas crezca de forma exponencial.

```
int fib(int n){
  if(n == 0 || n ==1 )
    return n;
  else
    return fib(n-1)+fib(n-2);
}
```

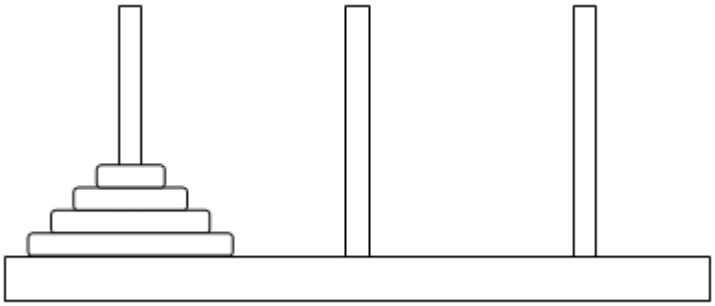
Una llamada a `fib(5)` se expande de la siguiente manera.

```
fib(5) = fib(4) + fib(3)
     = fib(3) + fib(2) + fib(2) + fib(1)
     = fib(2) + fib(1) + fib(1) + fib(0)+ fib(1) + fib(0) + 1
     = fib(1) + fib(0) + 1 + 1 + 0+ 1 + 0 + 1
     = 1 +0 + 1 + 1 + 0+ 1 + 0 + 1
     = 5
```

Como puede apreciarse se generan llamada repetidas de la función con los mismos valores de los parámetros. Esto implica un desperdicio de tiempo llevando a cabo los mismos cálculos. Esto no implica que la recursividad no sea útil en muchos casos. El procesamiento de estructuras dinámicas de datos como listas y árboles se facilita de manera notable

utilizando algoritmos recursivos sin detrimento grave en eficiencia. Algunos problemas tienen una solución muy complicada utilizando algoritmos iterativos convencionales y tienen una solución recursiva eficiente y elegante mediante un algoritmo recursivo. Algunos lenguajes basan su estructura de control en la recursión, tal es el caso del lenguaje LOGO.

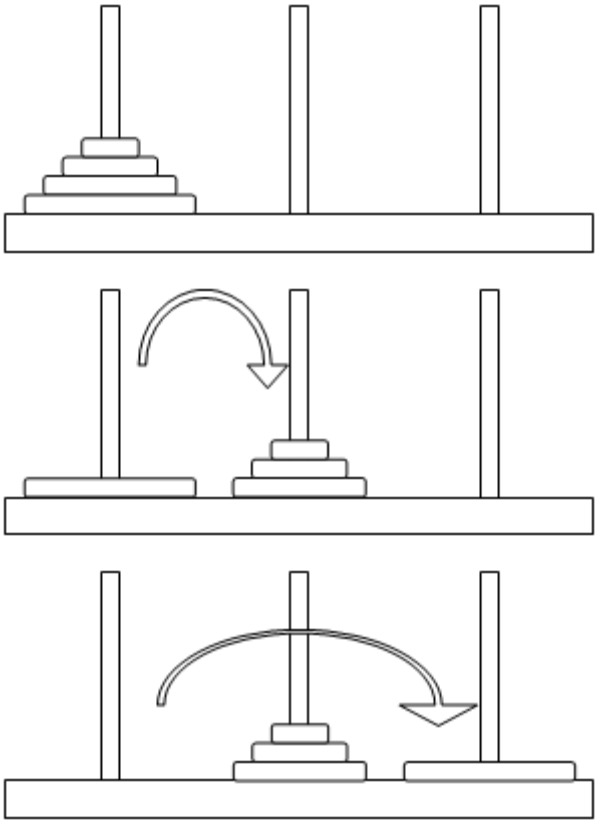
Para cerrar esta sección consideremos un problema clásico de la recursividad en la que la solución iterativa es extremadamente complicada, el problema de las torres de Hanoi. Las torres constan de tres pivotes alineados en las que se apilan una serie de discos todos de tamaños diferentes. El problema consiste en partir de la posición inicial en la que los discos perforados se encuentran en el pivote de la izquierda, como se muestra en la figura, y trasladarlos uno por uno al pivote de la derecha poniendo siempre discos más pequeños sobre los más grandes.



El pivote central puede usarse como un pivote auxiliar para hacer los movimientos. El objetivo es hacer el menor número de movimientos posible. En principio es difícil definir un algoritmo capaz de resolver este problema. Sin embargo, se puede plantear un algoritmo recursivo basándonos en que es fácil trasladar un solo disco y el problema de trasladar n se simplifica un poco si trasladamos n-1. Un primer esbozo de algoritmo es.

Algoritmo Torres. Juega el juego de Torres de Hanoi para n discos.
1. SI n=1 ENTONCES
    mover el disco al pivote derecho
2. SINO
    mover n-1 discos del pivote izquierdo al pivote central usando el derecho como auxiliar
    mover el n al pivote derecho
    mover n-1 discos del pivote central al pivote derecho usando el izquierdo como auxiliar

La siguiente figura muestra como se llevará a cabo el movimiento de los discos para una torre de 4 discos.



Note que en este algoritmo los parámetros son muy importantes. Puede notarse que existen tres parámetros: el pivote origen el pivote destino y el pivote auxiliar. Definamos la función torres de la siguiente manera.

```
void torres(int n,char desde,char hasta,char auxiliar)
```

Supondremos que los pivotes se etiquetan como ‘A’, ‘B’ y ‘C’, y los discos los numerados desde 1, para el más pequeño, hasta n el más grande. Entonces, torres(5,’A’,’C’,’B’); mueve 5 discos dl pivote ‘A’ al ‘C’ usando en pivote ‘B’ como auxiliar. Si traducimos el algoritmo Torres a C.

```
void torres(int n,char desde,char hasta,char auxiliar){
  if(n==1)
    printf(“mover disco 1 de ‘%c’ a ‘%c’”,desde,hasta);
  else{
    torres(n-1,desde,auxiliar,hasta);
    printf(“mover disco %d de %c a %c”,n,desde,hasta);
    torres(n-1,auxiliar,hasta,desde);
  }
}
```

La solución que genera esta función para n = 4 es

mover disco 1 de 'A' a 'B'
mover disco 2 de 'A' a 'C'
mover disco 1 de 'B' a 'C'
mover disco 3 de 'A' a 'B'
mover disco 1 de 'C' a 'A'
mover disco 2 de 'C' a 'B'
mover disco 1 de 'A' a 'B'
mover disco 4 de 'A' a 'C'
mover disco 1 de 'B' a 'C'
mover disco 2 de 'B' a 'A'
mover disco 1 de 'C' a 'A'
mover disco 3 de 'B' a 'C'
mover disco 1 de 'A' a 'B'
mover disco 2 de 'A' a 'C'
mover disco 1 de 'B' a 'C'

Existen muchos otros problemas que pueden resolverse utilizando la recursividad. Es importante conocer esta técnica porque algunos problemas son difíciles de plantear o resolver por métodos iterativos.

Problemas propuestos

6.7.1. ¿Cuantos movimientos de discos se hacen en el algoritmo recursivo de las torres de Hanoi para n discos?

6.7.2. Describa la salida generada por los siguientes programas.

a)
```
#include <stdio.h>
  int func1(int n );
  main(){
    int n = 10;
    printf(“%d”,func1(n));
  }
  int func1(int n){
    if(n>0) return(n + func1(n-1));
  }
```

b)
```
#include <stdio.h>
  int func1(int n );
  main(){
    int n = 10;
    printf(“%d”,func1(n));
  }
  int func1(int n){
    if(n>0) return(n + func1(n-2));
  }
```

6.7.3. La multiplicación de dos enteros puede definirse recursivamente de la siguiente manera.
1. mult(a,b) = a, si b=1
2. mult(a,b) = a+mult(a,b-1)

6.7.4. Escriba una función recursiva para calcular el producto de dos enteros.

6.7.5. El triángulo de Pascal, que se ve en la figura, se obtiene al tomar los coeficientes de la expansión de un binomio a la *n*. Estos coeficientes también pueden obtenerse a partir del número de combinaciones de *n* objetos tomados de *m* en *m*, las cuales se expresan como C_m^n.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
```

etc.
Como puede verse cada término se obtiene de la suma de los dos encima de él. Esto podemos expresarlo en forma recursiva como $C_m^n = C_{m-1}^{n-1} + C_m^{n-1}$, para 0<*m*<*n*, donde *n* es el número del renglón. Escriba una función recursiva que calcule un coeficiente del triángulo de Pascal en función de *n* y *m*. Escriba una función `main` que lea el valor de *n* e imprima el renglón del triángulo de Pascal correspondiente.

6.7.6. La función de Ackerman A está definida para todos los enteros positivos *m* y *n* como sigue:

$A(0, n) = n + 1$
$A(m, 0) = A(m - 1, 1)$
$A(m, n) = A(m - 1, A(m, n - 1))$

Escriba una función en C para calcular el valor de la función de Ackerman. Pruebe evaluando algunos valores de A(1, *n*), A(2, *n*) y A(3, *n*). Defina expresiones simples para estas funciones. La función tiene un crecimiento extremadamente grande para valores de *m* mayores que 3, por ejemplo $A(4,2) = 2^{65536} - 3$, evalúe A(4,1).

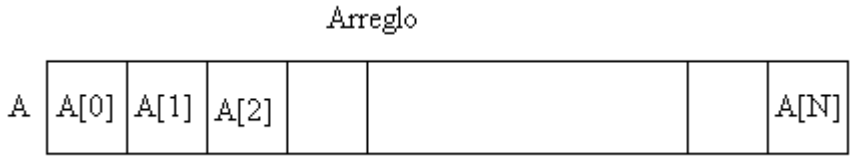
6.7.7. Desarrolle la expansión de la función de Ackerman del problema anterior para los siguientes parámetros: A(1,2), A(1,3) y A(2,1)

Capítulo 7. Arreglos

7.1. Arreglos de una dimensión

Un arreglo es una colección ordenada de elementos del mismo tipo. También se utiliza el término vector, formación u ordenación. Los arreglos permiten referirse a una colección de valores mediante un solo nombre. Para acceder a los elementos individuales de un arreglo se utiliza uno o más *índices* que deben ser un número entero o una expresión de tipo entero. Los arreglos pueden ser de cualquier tipo: **`int`**, **`float`**, **`double`**, **`char`**, o se puede tener elementos que sea estructuras más complejas. Cada elemento del arreglo almacena un valor del tipo básico.

Los arreglos de una dimensión solo utilizan un índice para localizar cada uno de sus elementos componentes. Para referirse a un elemento del arreglo se utilizan corchetes ([,]) para encerrar el índice del elemento del arreglo. En el lenguaje C los índices varían entre cero y un máximo igual al número de elementos del arreglo menos uno. Una representación gráfica es la de la figura siguiente.



El arreglo se llama A tiene N+1 elementos. Podemos direccional cada uno de sus elementos mediante el índice, así el primer elemento es A[0], el segundo A[1], el quinto elemento será A[4], y así sucesivamente. El último elemento del arreglo anterior es A[N]. La declaración de variables de tipo arreglo se hace en C siguiendo la siguiente sintaxis:

```
Tipo nombre[tamaño];
```

El tamaño es cualquier constante entera positiva mayor que cero. Los elementos Ejemplos:

`int` `a[23];` - declara un arreglo de 23 elementos de tipo entero.
`float` `m[120];` - declara un arreglo de 120 elementos de tipo flotante.
`char` `c[256];` - declara un arreglo de 256 elementos de tipo carácter.
`unsigned char` `y[1024];` - declara un arreglo de 1024 elementos de tipo carácter sin signo.

Al igual que las variables de tipo simple, se pueden iniciar los valores de un arreglo al momento de declararlo. Se le asigna valor a algunos o a todos los elementos del arreglo. Si hay algunos elementos que no hayan sido inicializados, a estos se les asignan valores nulos (0). En los siguientes ejemplos le asignamos valor a todos los elementos del arreglo `z` y se le asigna valor a los primeros tres elementos del arreglo `w` a los elemento con índices 3 a 9 se les asigna 0.0.

```
int z[10] = {2,4,6,8,10,12,14,16,18,20};
double w[10] = {12.5,-45.3,8.7};
```

Los arreglos de caracteres pueden iniciarse con cadenas de caracteres, como se muestra.

```
char s[10] = “hola”;
```
o
```
char s[10] = {‘h’,’o’,’l’,’a’};
```

El siguiente programa inicializa y despliega dos arreglos de caracteres y muestra cada uno de los elementos. En el ejemplo se despliega cada arreglo utilizando el formato `%s` que sirve para desplegar o leer cadenas de caracteres. También se despliega cada elemento como un carácter y como un número entero. Note que a los elementos no iniciados el compilador les asigna un valor nulo (0).

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

main(){
  char a[10]="hola";
  char b[10]={'h','o','l','a'};
  printf("%s\n",a);
  printf("%s\n",b);
  for(int i=0;i<10;i++)
    printf("%4d %c%4d %c\n",a[i],a[i],b[i],b[i]);
  getch();
}
```

El programa desplegará los siguientes valores:

```
hola
hola
 104 h 104 h
 111 o 111 o
 108 l 108 l
  97 a  97 a
   0     0
   0     0
   0     0
   0     0
   0     0
   0     0
```

En el caso de arreglo de caracteres solo podrán iniciarse usando la notación de cadenas si la cadena es de longitud más corta que el tamaño del arreglo, ya que C inserta automáticamente un carácter nulo (‘\0’) al final de la cadena.

Se puede omitir el tamaño de un arreglo si se inicializa en la declaración. En el caso de arreglos de caracteres se agrega al final un carácter nulo. Si no se inicializa es forzoso declarar el tamaño del arreglo. Por ejemplo.

```
int a[] = {1, 2, 3, 4};
char m[] = {'F','i','n'};
char s[] = "color";
```

Se declara un arreglo de enteros de 4 elementos con los siguientes valores.

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
```

El arreglo de caracteres `m` tendrá tres elementos con el siguiente contenido.

m[0] = 'F';
m[1] = 'i';
m[2] = 'n';

Si se despliega, por ejemplo utilizando `printf(m)` aparecerá basura, después de la cadena "Fin", que se encuentre en la memoria ya que no se agrega el terminador nulo. El arreglo `s` se inicia con seis elementos con el siguiente contenido.

s[0] = 'c';
s[1] = 'o';
s[2] = 'l';
s[3] = 'o';
s[4] = 'r';
s[5] = '\0';

En los siguientes ejemplos se inician los arreglos elemento por elemento. El siguiente ejemplo inicializa la variable `x` con números impares consecutivos:

```
int x[100], i;
for(i = 0; i<100 ; i++)
  x[i] = 2*i + 1;
```

A continuación se inicia un arreglo `x` con números aleatorios entre 25 y 74:

```
int x[100], i;
for(i = 0; i<100 ; i++)
  x[i] = rand()%50 + 25;
```

Para inicia los elementos pares x con números múltiplos de 3 y los impares con múltiplos de 4 usamos:

```
int x[100], i;
for(i = 0; i<100 ; i++)
                                                    if(i%2==0)
    x[i] = 3*i;
  else
    x[i] = 4*i
```

El siguiente código inicia x con los números de Fibonacci:

```
int x[30], i;
x[0] = 1;
x[1] = 1;
for(i = 2; i<30 ; i++)
    x[i] = x[i-1]+x[i-2]; //1,1,2,3,5,8,13,21,34,..
```

Los siguientes arreglos de caracteres se inicializan con las vocales y con las consonantes:

```
char v[]={'a','e','i','o','u'};
char c[]={'b','c','d','f','g','h','j',
'k','l','m','n','p','q','r','s','t','v','w','x','y','z'};
```

Se puede leer directamente los elementos de un arreglo. Por ejemplo, el siguiente fragmento programa lee los 10 elementos de un arreglo z, luego calcula la suma y la suma de los cuadrados de estos elementos. Observe que debemos colocar el operador & antes del elemento en la función scanf.

```
float z[10],suma=0.0,sumaCuadrado=0.0;
for(i = 0;i<10;i++)
  scanf("%f",&z[i]);
for(i = 0;i<10;i++){
  suma += z[i];
  sumaCuadrado += z[i];
}
```

Cálculo de desviación estándar

El siguiente programa que analizaremos calcula la desviación estándar de una colección de datos. La desviación estándar se calcula utilizando la siguiente expresión en la que se requiere calcular previamente el valor promedio.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}\left(x - \bar{x}\right)}{N-1}}$$

El programa está escrito utilizando funciones, cada una encargada de llevar a cabo una acción en particular. El arreglo `x` y el entero `max` se declaran como variables globales antes de la declaración de las funciones. Esto hace posible que se pueda tener acceso a estas variables dentro de cualquier función. Esta práctica no es de lo más recomendado ya que cualquier función puede modificar arbitrariamente los valores del arreglo, lo correcto sería pasar como parámetro el arreglo y su tamaño a las funciones. El paso de parámetros de arreglo a funciones lo veremos más adelante. El programa es el siguiente:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define TANANYO 50

double x[TANANYO];
int max;

double promedio();
double desviacion();
void saludo();
void LeerDatos();
void resultados();

main(){
  saludo();
  LeerDatos();
  resultados();
  getch();
}

void saludo(){
  printf("\tPROMEDIO Y DESVIACION ESTANDAR DE N VALORES\n");
}

void LeerDatos(){//lee los elementos del arreglo
  int i=0;
  double num;
  do{
    printf("Teclee un numero (-1 = terminar):");
    scanf(“%lf”,&num);
    if(num>0){
      x[i] = num; i++;
    }
```

```
  }while(num>0);
  max = i;
}

double promedio(){
  int i;
  double suma=0;
  for(i=0;i<max;i++)
    suma +=x[i];
  return suma/max;
}

double desviacion(){
  int i;
  double suma=0,prom;
  prom=promedio();
  for(i=0;i<max;i++)
    suma +=(x[i]-prom)*(x[i]-prom);
  return sqrt(suma/(max-1));
}

void resultados(){
  int i;
  printf("VALORES INTRODUCIDOS\n”);
  for(i=0;i<max;i++){
    printf(“%8.3f”,x[i]);
    if(i%8==0&&i>0)
      printf(“\n”);
  }
  printf(“\n”);
  printf("El promedio es: %8.3f\n",promedio());
  printf("La desviacion estandar es: %8.3f\n",desviacion());
}
```

La función `saludo()` informa el propósito del programa. La función `LeerDatos()` lee los datos desde el teclado almacenando cada valor en el arreglo x. El proceso de lectura termina cuando el usuario teclea un valor menor que cero. Las funciones `promedio()` y `desviación()` calculan el promedio de los valores y la desviación estándar y regresan un número de doble precisión. Por último la función `resultados()` muestra los valores introducidos en renglones, el promedio y la desviación estándar de estos valores.

Problemas propuestos

7.1.1. Describir el arreglo definido en cada una de las siguientes instrucciones:
a) `char nombre[80];`

b)
```
#define N 50
  char color[N];
```
c)
```
#define MAX 30
 float x[MAX];
```
d)
```
double v[30];
```

7.1.2. Describirla el arreglo definido en cada una de las siguientes instrucciones. Indicar qué valores son asignados a los elementos individuales del arreglo.

a)
```
float z[8] = {2., 5., 3., -4., 12., 12., 0., 8.};
```
b)
```
float x[8] = {2., 5., 3., -4.};
```
c)
```
int num[12] = {0, 2, 3, 5, 8};
```
d)
```
char valor [4] = {‘T’, 'R’, 'U', ‘E’};
```
2)
```
char valor [5] = {‘T', 'R’, ’U', 'E’};
```
f)
```
char valor [] = "TRUE";
```
g)
```
char valor [] = "FALSE";
```

7.1.3. Escribir una definición apropiada de arreglo para cada uno de los siguientes problemas.

a) Definir un arreglo de 12 elementos enteros llamada `c`. Asignar los valores 1, 4, 7, 10, ..., 34 a los elementos del arreglo.
b) Definir un arreglo de caracteres llamada `direccion`. Asignar la cadena "OESTE" a los elementos del arreglo. Terminar la cadena con el carácter nulo.
c) Definir un arreglo de cuatro caracteres llamada `letras`. Asignar los caracteres ‘N’, ‘T’, ‘S’ y ‘C’ a los caracteres del arreglo.
d) Definir un arreglo de seis elementos en coma flotante llamada `constante`. Asignar los siguientes valores a los elementos del arreglo: 3.1416, 2.71828, 1.4142, 0.69314, 0.30103, 0.31831.

7.1.4. Describir la salida producida por los siguientes programas

a)
```
#include <stdio.h>
main(){
  int a, b = 0;
  static int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  for (a = 0; a < 10; ++a)
    if (c[a] % 2 == 0) b+= a[a];
  printf("%d", b);
}
```
b)
```
#inc1ude <stdio.h>
main(){
  int a, b = O;
  static int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  for(a = 0; a < 10; ++a)
    if ((a % 2) != 0) b+= c[a];
  printf("%d", b);
}
```

c)
```
#inc1ude <stdio.h>
main(}{
  int a,b=0;
  int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  for(a = 0; a < 10; ++a)
    b+= c[a];
  printf{"%d", b);
}
```

d)
```
 #inc1ude <stdio.h>
  int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
main(){
  int a, b = O;
  for(a = 0; a < 10; ++a)
    if((c[a] % 2) != 1) b+= c[a];
  printf{“%d", b);
}
```

7.1.5. Inicie un arreglo de 20 enteros con números consecutivos de 120 en adelante.

7.1.6. Inicie un arreglo de 50 números flotantes con valores recíprocos de los números de 1 a 50.

7.1.7. Un vector en el plano puede representarse como un arreglo de dos componentes. Escriba un programa que lea dos vectores y calcule el ángulo entre ellos utilizando la definición de producto escalar.

7.1.8. Escriba un programa que inicie un arreglo con los 50 primeros números primos.

7.1.9. Considerar las siguientes monedas extranjeras y sus equivalencias en dólares americanos:

Libra esterlina: 0.65 libras por dólar USA
Dólar canadiense: 1.4 dólares por dólar USA
Yen japonés: 98 yenes por dólar USA
Peso mexicano: 3.4 pesos por dólar USA
1 Franco suizo: 1.3 francos por dólar USA

Escribir un programa interactivo, guiado por menús, que acepte dos monedas extranjeras y devuelva el valor de la segunda moneda por cada unidad de la primera moneda. (Por ejemplo, si las dos monedas son el yen japonés y el peso mejicano, el programa devolverá el número de pesos mexicanos equivalentes a un yen japonés.) Utilizar los datos dados anteriormente para realizar las conversiones. Diseñar el programa de modo que se ejecute repetidamente, hasta que se seleccione la condición de salida del menú.

7.2. Arreglos y apuntadores

Una característica muy poderosa del lenguaje C son los apuntadores. Un apuntador es una variable que almacena una dirección de memoria, en la cual se puede almacenar algún tipo de datos. Los apuntadores se utilizan para referirse a datos comunes, arreglos y sobre todo a estructuras dinámicas más sofisticadas como pilas, listas, colas, etc. Los apuntadores se utilizan para definir parámetros por referencia.

Consideremos el siguiente programa que define una variable entera x y un apuntador a un entero px. Como puede verse para definir una variable apuntador basta con anteponer a la variable el operador de indirección (*). El programa asigna valor a la variable x y luego asigna valor al apuntador, para esto utilizamos el operador dirección (&) que al aplicarlo a una variable regresa la dirección de memoria de la variable, en este caso le asignamos al apuntador px la dirección de la variable x. No puede asignarse directamente un valor a un apuntador excepto el valor nulo (NULL), la asignación debe hacerse a través del operador dirección o alguna función adecuada.

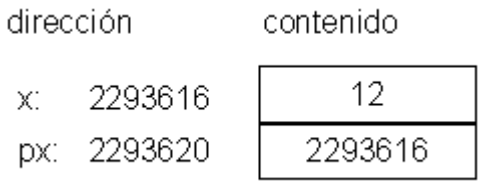
```
#include <stdio.h>
#include <conio.h>

main(){
  int *px,x;
  x = 12;
  px = &x;
  printf("x:%d   px:%d\n",x,px);
  printf("&x:%d   *px:%d\n",&x,*px);
  printf("*&x:%d   &px:%d\n",*&x,&px);
  getch();
}
```

El primer `printf` despliega el contenido de `x` y el de `px`. Se desplegará un 12 y la dirección de memoria que se le haya asignado a la variable `x`. El segundo `printf` despliega la dirección de `x` utilizando el operador & y el contenido de la dirección señalada por `px` utilizando el operador `*`. Se desplegará nuevamente la dirección de la variable `x` y el contenido de la dirección que es señalada por `px`, es decir 12. El último `printf` despliega el resultado de aplicar el operador & y `*` a la variable `x`, como estos son operadores inversos, al aplicar ambos lo que obtenemos es el valor de la variable `x`, y por último se despliega la dirección del apuntador `px`. Una posible salida es la siguiente:

```
x:12   px:2293616
&x:2293616   *px:12
*&x:12   &px:2293620
```

El mapa de memoria indica este resultado se representa en la siguiente figura. Se muestra el contenido de la memoria después de ejecutar las asignaciones. Note que en `px` se almacena la dirección de la variable `x`. La dirección de `px` está 4 bytes más allá que la dirección de `x`, esto significa que la variable `x` ocupa 4 bytes. El operador `sizeof`() regresa el tamaño en bytes de una variable o tipo. Si ejecutamos `printf(“%d”,sizeof(int));` se

imprimirá un 4. Un apuntador también tiene un tamaño de 4 byte. Estos tamaños pueden variar dependiendo del compilador y la máquina en que se ejecute.



Los arreglos en C tienen muchas similitudes con los apuntadores. Una variable de tipo arreglo es en realidad un apuntador al primer elemento del arreglo. En el siguiente ejemplo se declara un arreglo de enteros, un arreglo de **`double`**, un apuntador a enteros y un apuntador a **`double`**. El arreglo de enteros se inicializa con los números del 0 al 9 y el de **`double`** con los valores 0.9, 0.7, 0.5, 0.3 y 0.1. Se asigna al apuntador a enteros la dirección del primer elemento del arreglo de enteros y se despliega el primer elemento utilizando el subíndice 0 y el apuntador a enteros. A continuación se suma 1 al apuntador, el compilador suma en realidad una cantidad igual al tamaño de la variable apuntada por el apuntador, en este caso 4. Después se hace lo mismo con el apuntador de **`double`**, note que en este caso se incrementa el apuntador en 8 ya que el tamaño de un **`double`** es 8 bytes.

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

main(){
  int *px,x[10] = {0,1,2,3,4,5,6,7,8,9};
  double *pd,d[5] = {9.0,7.0,5.0,3.0,1.0};
  px = x;
  printf("x[0]:%d   *px:%d px:%d\n",x[0],*px,px);
  px = px+1;
  printf("*px:%d px:%d\n",*px,px);
  pd = d;
  printf("d[0]:%5.1f   *pd:%5.1f  pd:%d\n",d[0],*pd,pd);
  pd = pd+1;
  printf("*pd:%5.1f   pd:%d\n",*pd,pd);
  getch();
}
```

Note que no se requiere el uso del operador de dirección en la asignación `px = x`, ya que `x` es la dirección del elemento 0 del vector `x`, también se puede escribir esta asignación como `px = &x[0]`. Podemos sumar o restar cualquier cantidad a un apuntador y pueden usarse los operadores de incremento y decremento. La salida del programa es la siguiente:

```
x[0]:0   *px:0 px:2293552
*px:1 px:2293556
d[0]:  9.0   *pd:  9.0  pd:2293488
```

```
*pd:  7.0   pd:2293496
```

Las cadenas de caracteres pueden ser inicializados con cadenas constantes si son declarados indistintamente como arreglos o como apuntadores. El siguiente programa muestra dos cadenas, una declarada como arreglo y otra como apuntador de tipo carácter.

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

main(){
  char x[] = “Esta cadena es un arreglo.”;
  char *y = “Esta cadena es un apuntador.”;
  printf(“%s\n”,x);
  printf(“%s\n”,y);
  getch();
}
```

Lo anterior no puede hacerse en arreglos de otros tipos. Es forzoso declararlos como arreglos para inicializarlos en la declaración como se muestra.

```
int x[] = {1,2,3,4,5};
int y[8] = {3,4,5,8};
int *z = {1,2,3}; //NO es aceptable
```

Antes de iniciar los valores de un arreglo declarado como apuntador se debe crear el arreglo dinámicamente. La función `malloc` de la biblioteca `stdlib.h` permite la asignación de memoria a apuntadores. Por ejemplo, el siguiente programa declara un apuntador m a enteros. Luego crea un arreglo de enteros de 10 elementos dinámicamente y le asigna su dirección de inicio a m. Una vez creado podemos usar la notación de arreglo o de apuntador para hacer operaciones con el arreglo. Debe usarse el cambio de tipo para crear el arreglo de enteros ya que `malloc` regresa un apuntador a **`void`** (sin tipo). No asignar una dirección válida a un apuntador puede ser desastroso. Después de usar la memoria es conveniente regresarla mediante la función `free`.

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

main(){
  int *m;
  m = (int *)malloc(10*sizeof(int));
  m[4] = 5;
  printf("%d\n",*(m+4));
  getch();
  free(m);
}
```

Note que `m+4 = 4+m`, eso significa que debe ser aceptable escribir `m[4]` como `4[m]`, algunos compiladores que soportan ANSI C lo aceptan.

Problemas propuestos

7.2.1. Explicar cada una de las siguientes declaraciones:

a)
```
int *pi;
```
b)
```
float a, b;
float *pa, *pb;
```
c)
```
float x = 3.5;
float *px = &x;
```
d)
```
char cl, c2, c3;
char *pcl, *pc2, *pc3 = &c1;
```

7.2.2. Un programa en C contiene las siguientes instrucciones:
```
char  u, v = 'A';
chat *pu, *pv = &v ;
*pv = v+1;
pu = *pv+1;
pu = &u ;
```
Supongamos que cada carácter ocupa un byte de memoria. Si el valor asignado a u se almacena en la dirección F8C {hexadecimal) y el valor asignado a v se almacena en F8D, entonces:

a) ¿Qué valor es representado por `&v`?
b) ¿Qué valor es asignado a `pv`?
c) ¿Qué valor es representado por `*pv`?
d) ¿Qué valor es asignado a `u`?
e) ¿Qué valor as representado por `&u`?
j) ¿Qué valor es asignado a `pu`?
g) ¿Qué valor es representado por `*pu`?

7.2.3. Un programa tiene la siguiente declaración:
```
int a[10] = {3,5,7,9,11,13,15,17,19,21};
```
Diga el significado o el valor de las siguientes expresiones:

a) `a`
b) `a+2`
c) `*a`
d) `*(a+5)`
e) `*(a+*(a+1))`

7.2.4. Dada la siguiente declaración escriba las siguientes expresiones utilizando notación de arreglos y el operador &.
```
int a[10] = {3,5,7,9,11,13,15,17,19,21};
```
a) `a`
b) `a+2`

c) `*a`
d) `*(a+5)`
e) `*(a+*(a+1))`

7.2.5. Dada la siguiente declaración escriba las siguientes expresiones utilizando notación de apuntadores.

```
int a[10] = {3,5,7,9,11,13,15,17,19,21};
```

a) `a[3]`
b) `a[a[2]]`
c) `a[i-2*j]`
d) `&a[7]`
e) `&a[a[1]]`

7.3. Paso de arreglos como parámetros

Las funciones pueden aceptar arreglos como parámetros. El tamaño del arreglo puede o no especificarse en la declaración de la función. Los parámetros siempre pasan por referencia, es decir, no se copian los valores a la función y siempre es posible modificar los valores de los parámetros reales.

En los prototipos de las funciones pueden o no incluirse el nombre de los parámetros formales. Si no se incluyen hay que colocar unos corchetes para indicar que se trata de un arreglo.

Note que no se utiliza el operador de indirección, los paréntesis cuadrados ([]) tienen la misma función. En la mayoría de los casos hay que indicar el tamaño del arreglo como un parámetro adicional a la función. Por ejemplo, si deseamos definir una función que calcule el promedio de un arreglo, es necesario pasar como parámetro el número de elementos que se procesarán.

```
double promedio(double x[], int tamanio){
  int i;
  double suma = 0;
  for(i=0 ; i<tamanio; i++)
    suma += x[i];
  return suma/tamanio;
}

int main(){
  double a[]={3,4.5,6.5,7,5,6,8};
  double b[]={5.3,6.2,7.5,6,5,4,5,7,6,9};
  printf("Promedio de a es %6.2f\n",promedio(a,7));
  printf("Promedio de b es %6.2f\n",promedio(b,10));
  getch();
}
```

Una función para calcular la desviación estándar se puede basar en esta función que calcula el promedio.

```
double desviacion(double x[], int tamanio){
  int i;
  double suma=0,prom;
  prom=promedio(x,tamanio);
  for(i=0;i<tamanio;i++)
    suma +=(x[i]-prom)*(x[i]-prom);
  return sqrt(suma/(max-1));
}
```

La siguiente función es un ejemplo de impresión de un arreglo. El arreglo se despliega en renglones de ocho elementos.

```
void imprimeArreglo(double a[], int n){
  int i;
  for(i = 0; i< n; i++){
    printf(“%8.2f”,a[i]);
    if((i+1)%8==0)
      printf("\n");
  }
  printf("\n");
}
```

Como otro ejemplo de procesamiento de arreglos veremos una función que invierte los elementos de un arreglo. El en proceso se utiliza una variable auxiliar para almacenar temporalmente los elementos que se intercambian.

```
void invierteArreglo(double a[], int n){
  int i;
  double temp;
  for(i = 0; i< n/2; i++){
    temp = a[i];
    a[i] = a[n-i-1];
    a[n-i-1] = temp;
  }
}
```

Una operación importante es la búsqueda de datos en un arreglo. Esta se realiza comparando cada elemento con el elemento buscado. La siguiente función busca un elemento dentro de un arreglo y regresa el índice al elemento donde encontró la primera coincidencia o un -1 si la búsqueda fracasó.

```
int busca(double x, double a[], int n){
  int i;
  for(i = 0; i<n; i++)
```

```
    if(x==a[i])
      return i;
  return -1;
}
```

La búsqueda puede mejorarse si el arreglo está ordenado. La ordenación es otra operación importante en arreglos. Veremos el método más simple de ordenación. Este consiste en comparar elementos del arreglo, si el de índice menor es más grande, se invierten, al final de este proceso tendremos el elemento más pequeño al principio del arreglo. El proceso se repite hasta acomodar los restantes. La siguiente función lleva a cabo este proceso.

```
void Burbuja(double a[],int tam){
  for(int i = 0; i< tam - 1 ; i++)
    for(int j = i; j< tam;j++)
      if(a[i]>a[j]){
        double temp = a[i];
        a[i]=a[j];
        a[j]=temp;
    }
}
```

Una vez ordenado un arreglo podemos utilizar un algoritmo más eficiente para llevar a acabo búsquedas en él. El algoritmo de búsqueda binaria (o de bisección) es uno de los más comunes. Este algoritmo consiste en elegir como primer elemento para buscar al elemento de la mitad del arreglo, si es el que buscamos, el proceso termina, sino y el elemento es más grande que el que buscamos, elegimos el elemento de la mitad inferior del arreglo, sino elegimos el de la mitad superior del arreglo. El proceso continúa hasta encontrar el elemento o tener un intervalo de búsqueda nulo.

Algoritmo Búsqueda binaria. Busca en un arreglo a de datos ordenados. Se utilizan las variables bajo y alto para definir el intervalo de búsqueda. La función regresa el índice del elemento si se encuentre el valor buscado, sino regresa -1.

```
1. HACER
2.      medio = (bajo+alto)/2
3.      SI a[medio]==x ENTONCES
                regresar medio
4.      SINO
                SI(a[medio]>x)
                        alto = medio-1
                SINO
                        bajo = medio+1
5. MIENTRAS(bajo<=alto)
6. regresar -1
```

El siguiente programa define un arreglo a y busca cuatro elementos en él.

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

```
#include <conio.h>

int buscaBinaria(int a[],int x,int n);

main(){
  int b[]={1,3,4,5,8,12,15,20,25,34};
  printf("%d\n",buscaBinaria(b,1,10));
  printf("%d\n",buscaBinaria(b,25,10));
  printf("%d\n",buscaBinaria(b,34,10));
  printf("%d\n",buscaBinaria(b,26,10));
  getch();
}

int buscaBinaria(int a[],int x,int n){
  int medio,bajo=0,alto=n-1;
  do{
    medio = (bajo+alto)/2;
    if(a[medio]==x)
      return medio;
    else
      if(a[medio]>x)
        alto = medio-1;
      else
        bajo = medio+1;
  }while(bajo<=alto);
  return -1;
}
```

Se puede utilizar la notación de apuntador para el paso de arreglos como parámetros a funciones. Por ejemplo, la función Burbuja que ordena un arreglo de **`double`** puede escribirse de la siguiente manera.

```
void Burbuja(double *a,int tam){
  for(int i = 0; i< tam - 1 ; i++)
    for(int j = i; j< tam;j++)
      if(*(a+i)> *(a+j)){
        double temp = *(a+i);
        *(a+i) = *(a+j);
        *(a+j) = temp;
    }
}
```

Problemas propuestos

7.3.1. Escriba una función que acepte un arreglo de flotantes y un entero que represente el número de elementos del arreglo y despliegue un histograma del arreglo. Ajuste el tamaño de las barras para mostrar el elemento más grande con una cadena de asteriscos de longitud

30. Ejemplo, si un arreglo de 5 elementos contiene 2.0, 3.0, 5.0, 1.0 y 6.0, desplegar la siguiente figura:

```
2     **********
3     ***************
5     ************************
1     *****
6     ******************************
```

7.3.2. Escriba funciones para lo siguiente:

a) Una función que regrese el valor del elemento más grande de un arreglo. Escriba otra función para encontrar el elemento más pequeño.

b) Una función que regrese el valor del elemento más grande en valor absoluto de un arreglo. Escriba otra función para encontrar el elemento más pequeño en valor absoluto.

c) Una función que cuente los elementos mayores que cero en un arreglo.

d) Una función que calcule la suma de los cuadrados de un arreglo de elementos de doble precisión.

e) Una función que elimine los elementos repetidos de un arreglo de enteros. Deberá modificar el tamaño del arreglo dentro de la función.

f) Una función que cuente el número de elementos de un arreglo de enteros que son menores que cero, iguales a cero y mayores que cero. Puede regresar los resultados en otro arreglo de tres elementos.

g) Una función que encuentre el valor del elemento que más se repite en un arreglo.

h) Una función que encuentre el k-ésimo entero más pequeño de un arreglo de números enteros a.

7.3.3. Si un arreglo tiene n elementos, ¿cuál es el número máximo de ciclos hechos por el algoritmo de búsqueda binaria?

7.3.4. Escribir una forma recursiva del algoritmo de búsqueda binaria.

7.4. Arreglos de dos dimensiones

Los arreglos pueden tener más de una dimensión. Cada dimensión puede tener diferente número de componentes. Un arreglo de dos dimensiones puede representarse como una tabla. Los arreglos de dos dimensiones se declaran de acuerdo al esquema

```
tipo variable[renglones][columnas];
```

Por ejemplo el siguiente arreglo declara una tabla con cinco renglones y cuatro columnas. Los renglones varían de 0 a 4 y las columnas de 0 a 3.

```
int a[5][4];
```

Podemos representarlo como una tabla como se muestra.

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]

Los arreglos de dos dimensiones se inicializan de forma similar a los de una dimensión. Hay varias opciones en la notación que se muestran a continuación.

```
int a[4]
[3] = {{3,6,1},{5,7,2},{2,1,5},{8,7,6}};

int a[]
[3] = {{3,6,1},{5,7,2},{2,1,5},{8,7,6}};

int a[4]
[3] = {3,6,1,5,7,2,2,1,5,8,7,6};
```

Note que la opción

```
int a[4][] = {{3,6,1},{5,7,2},{2,1,5},{8,7,6}};
```

No es aceptada por el compilador ya que debe definirse el tamaño de todas las dimensiones excepto la primero.

La inicialización de arreglos de dos dimensiones requiere del uso de ciclos anidados. El siguiente fragmento de código inicializa un arreglo de 4 renglones y 5 columnas con valores iguales al producto de sus índices.

```
int a[4][5],i,j;
for(i=0;i<4;i++)
  for(j=0;j<5;j++)
    a[i][j] = i*j;
```

Como un ejemplo del uso de arreglos bidimensionales veamos el juego de la vida definido por el matemático John Horton Conway en 1970. Este consiste en simular una población bidimensional de células. Las células pueden estar en uno de dos estados posibles, vivas o muertas. Las células interactúan con sus ocho vecinos, dos a cada lado, uno arriba, otro

abajo y cuatro en dirección diagonal. En cualquier momento pueden ocurrir las siguientes transiciones:

1. Una célula que tenga menos de dos vecinos muere por causa de la baja población.
2. Una célula que tenga más de tres vecinos muere por causa de la alta población.
3. Una célula que tenga dos o tres vecinos sobrevive a la siguiente generación.
4. Una célula muerta que tenga exactamente tres vecinos vivos, vuelve a la vida.

Representaremos el mundo donde viven las células mediante un arreglo de dos dimensiones llamado mundo. Usaremos un arreglo de las mismas dimensiones para hacer los cálculos en cada paso sin alterar el estado inicial. Se utilizan dos vectores, dx y dy, para generar los índices de los vecinos de cada célula. Inicialmente el mundo tiene valores aleatorios de 0 y 1. A cada paso se copia el mundo en el arreglo temporal, y después se aplican las reglas de sobre vivencia al arreglo temporal elemento por elemento y se actualiza el arreglo mundo. El algoritmo es el siguiente.

Algoritmo Juego de la vida. Simula le evolución de un sistema de células. Se utilizan dos arreglos uno que represente el mundo de células y otro temporal para hacer los cálculos. Los arreglos toman dos valores 1, si la célula está viva y 0 si está muerta. Se utilizan dos arreglos dx = {1,0,1,1,1,0,-1,-1} y dy = {-1,-1,-1,0,1,1,1,0} para calcular los vecinos de cada célula, x y y son las coordenadas del vecino.

```
1. Inicial el mundo con valores aleatorios.
2. REPETIR
3.       copiar mundo al arreglo temporal
4.       PARA renglon = 0 HASTA MAXREN -1 HACER
5.                PARA columna = 1 HASTA MAXREN-1 HACER
6.                         suma = 0
7.                          PARA vecino=1 HASTA 8 HACER
8.                                   x = i+dx[vecino]
9.                                   SI x<0 ENTONCES
                                              x = MAXREN-1
10.                                  SI x = MAXREN ENTONCES
                                              x = 0
11.                                  y = j+dy[vecino]
12.                                  SI y<0 ENTONCES
                                              y = MAXCOL-1
13.                                  SI y = MAXCOL ENTONCES
                                              y = 0
14.                                  suma = suma + mundo[x][y])
15.                        FINPARA [vecino]
16. [Se aplican las reglas de sobre vivencia]
                           SI (mundo[renglon][columna]=1)ENTONCES
                                  SI(suma<2 O suma>3) ENTONCES
                                              mundo[renglon][columna] = 0
                           SINO
                                  SI suma=3 ENTONCES
                                              mundo[renglon][columna] = 1
```

17. FINPARA [columna]
18. FINPARA [renglon]
17. HASTA siempre

El programa completo en C es el siguiente. Los arreglos dx y dy se suman a los índices renglón y columna del arreglo bidimensional para obtener las coordenadas (x, y) del vecino. La coordenada x del vecino se almacena en x y la coordenada y en y. Una célula en la frontera del arreglo es vecina de las células del otro extremo del arreglo, lo mismo de aplica a las esquinas. Por ejemplo, la célula en (0,0) es vecina de las células en (0, 1), (1, 1), (1, 0), (MAXREN -1, 1), (MAXREN -1, 0), (MAXREN -1, MAXCOL -1), (0, MAXCOL -1) y (1, MAXCOL -1), donde se ha puesto primero el renglón y luego la columna. El programa se detiene en cada ciclo hasta que se presione la letra ‘q’.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#define MAXREN 10
#define MAXCOL 10

main(){
  int mundo[MAXREN][MAXCOL],temp[MAXREN][MAXCOL];
  int dx[8]={-1,0,1,1,1,0,-1,-1};
  int dy[8]={-1,-1,-1,0,1,1,1,0};
  int i,j,k,suma,x,y;
  char c;
  srand(time(0));
  for(i=0;i<MAXREN;i++)
    for(j=0;j<MAXCOL;j++)
      mundo[i][j]=rand()%2;
  do{
    printf("\n");
    for(i=0;i<MAXREN;i++){
      for(j=0;j<MAXCOL;j++){
        temp[i][j]=mundo[i][j];
        printf("%d ",mundo[i][j]);
      }
      printf("\n");
    }
    for(i=0;i<MAXREN;i++)
      for(j=0;j<MAXCOL;j++){
        suma = 0;
        for(k=0;k<8;k++){
          x = i+dx[k];
          if(x<0)x=MAXREN-1;
          if(x==MAXREN)x=0;
          y = j+dy[k];
```

```
          if(y<0)y=MAXCOL-1;
          if(y==MAXCOL)y=0;
          suma += temp[x][y];
        }
        if(mundo[i][j]){
          if(suma<2||suma>3)
            mundo[i][j]=0;
        }else
           mundo[i][j]=suma==3;
      }
    c=getch();
  }while(c!='q');
}
```

Problemas propuestos

7.4.1. Describa los arreglos declarados en cada caso.

a) `int m[5][5];`

b)
```
#define A 66
#define B 132
char memo[A][B];
```

c)
```
#define MAXX 20
#define MAXY 25
double tablero[MAXY][MAXX];
```

d)
```
#define ANCHO 8;
#define ALTO 8
unsigned char juego[ALTO][ANCHO];
```

7.4.2. Describir el arreglo definido en cada una de las siguientes instrucciones. Indicar qué valores son asignados a los elementos individuales del arreglo.

a) `int p[2][4] = {1, 3, 5, 7};`

b) `int p[2][4] = {1, 1, 3, 3, 5, 5, 7, 7};`

c) `int P[2][4] = {{1, 3, 5, 7},{2, 4, 6, 8}};`

d) `int p[2][4] ={{1, 3},{5, 7}};`

e)
```
char damas[8][8] = {{0,1,0,1,0,1,0,1}, {1,0,1,0,1,0,1,1},
{0},{0},{0},{0},{0,2,0,2,0,2,0,2}, {2,0,2,0,2,0,2,0}};
```

7.4.3. Escribir una definición apropiada de formación para cada uno de los siguientes problemas.

a) Definir una formación bidimensional de enteros, de 3x4, llamada `ventas`. Asignar los siguientes valores iniciales a los elementos de la formación

```
10 12 14 16
20 22 24 26
30 32 34 36
```

b) Definir una formación bidimensional de enteros, de 4x7, llamada compras. Asignar los siguientes valores iniciales a los elementos de la formación

```
10 12 14  0  0  0  0
0  20 22  0  0  0  0
0  30 32  0 24 12 39
0  0   0  0  0  0 12
```

7.4.4. Modifique el programa del juego de la vida para permitir que el usuario defina la configuración inicial de las células. Algunas configuraciones no cambian, por ejemplo, cuatro células formando un rombo o un cuadrado. Otras son cíclicas, tres células alineadas alternan entre una distribución horizontal y una vertical. El programa es más ilustrativo si el mundo se mantiene fijo en la ventana y se muestran los cambios a cada paso. Consulte como lograr esto con el compilador de C que utiliza e impleméntelo.

7.5. Paso de arreglos de dos dimensiones como parámetros a funciones

Para especificar arreglos de dos dimensiones como parámetros en una función, se debe especificar el tamaño de segunda dimensión, en la primera dimensión se puede poner solo un par de corchetes vacíos. Por ejemplo, la siguiente función despliega un arreglo de dos dimensiones de caracteres. Pasamos como argumentos el número de renglones y columnas del arreglo.

```
void despliega(char a[][10],int r,int c){
  int i,j;
  for(i=0;i<r;i++){
    for(j=0;j<c;j++)
      printf("%2c",a[i][j]);
    printf("\n");
  }
}
```

Calificaciones de un grupo de alumnos

Un profesor califica haciendo tres exámenes parciales los cuales cuentan el 80% de la calificación final y un examen final que cuenta el otro 20%. Para procesar las calificaciones definiremos un arreglo de dos dimensiones con 25 renglones y 5 columnas. Las primeras 3 columnas almacenarán las 3 calificaciones parciales, la cuarta columna almacenará la calificación del examen final y la quinta el promedio final.

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

void leerCalificaciones(float cal[][5],int nAlumnos);
void procesaCalificaciones(float cal[][5],int nAlumnos);
void imprime(float cal[][5],int nAlumnos);

main(){
```

```
  float calificaciones[25][5];
  int numAlumnos;
  printf("Cuantos alumnos?");
  scanf("%d",&numAlumnos);
  leerCalificaciones(calificaciones,numAlumnos);
  procesaCalificaciones(calificaciones,numAlumnos);
  imprime(calificaciones,numAlumnos);
  getch();
}

void leerCalificaciones(float cal[][5],int nAlumnos){
  int i,j;
  for(i = 0;i<nAlumnos;i++){
    printf("Teclee calificaciones de alumno %i\n",i);
    for(j=0;j<6;j++){
      scanf("%f",&cal[i][j]);
    }
  }
}

void procesaCalificaciones(float cal[][5],int nAlumnos){
  int i,j;
  float suma;
  for(i = 0;i<nAlumnos;i++){
    suma=0;
    for(j=0;j<5;j++)
      suma+=cal[i][j];
    cal[i][6]=suma/5*0.8+cal[i][5]*0.2;
  }
}

void imprime(float cal[][5],int nAlumnos){
  int i,j;
  printf("No.\t1\t2\t3\t4\t5\t6\tfinal\n",i);
  for(i = 0;i<nAlumnos;i++){
    printf("%d",i);
    for(j=0;j<7;j++){
      printf("\t%5.1f",cal[i][j]);
    }
    printf("\n");
  }
}
```

La función de lectura de datos permite que las calificaciones sean introducidas en un solo renglón separando cada número por espacios en blanco. Un ejemplo de entrada es el siguiente:

```
Cuantos alumnos?3
Teclee calificaciones de alumno 0
2 4 8 6
Teclee calificaciones de alumno 1
6 6 7 7
Teclee calificaciones de alumno 2
7 7 8 8
No.       1       2       3       EF      Cal final
0         2.0     4.0     8.0     6.0     5.5
1         6.0     6.0     7.0     7.0     6.7
2         7.0     7.0     8.0     8.0     7.7
```

Una matriz es un arreglo de dos dimensiones de números de punto flotante. Las matrices tienen muchas aplicaciones en el análisis de sistemas lineales, tales como redes eléctricas. Las operaciones con matrices más comunes son las operaciones aritméticas: suma, resta y multiplicación. Las matrices se caracterizan por el número de renglones y de columnas que tienen, a estos números se les llama rango y se expresa como renglones X columnas. Por ejemplo, una de 5 renglones y 8 columnas se expresa por 5x8.

Por ejemplo, la suma de matrices se define como una matriz en la que los elementos son iguales a la suma de los elementos correspondientes de las matrices originales. Las matrices deben tener el mismo número de renglones y de columnas para poder sumarse. Una función para sumar matrices es la siguiente.

```
void sumaMatriz(double a[][10],double b[][10],int ren, int
col,double c[][10]){
  for(int i=0 ;i<ren ;i++)
    for(int j=0 ;j<col ;c++)
      c[i][j] = a[i][j]+ b[i][j];
}
```

La multiplicación es una operación un tanto más complicada. Dadas dos matrices A de m x n y B de n x p, el producto de estas es una matriz C de m x p, tal que el elemento (*i*, *j*) del producto se calcula como.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + ... + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Donde se ha utilizado la notación de sumatoria. Además a_{ik} es el elemento del renglón *i* columna *k* de la matriz A, b_{kj} es el elemento del renglón *k* columna *j* de la matriz B, y c_{ij} es el elemento del renglón *i* columna *j* de la matriz C. Una función para encontrar el producto de dos matrices es la siguiente.

```
void productoMatriz(double a[][10],int ra, int ca,
           double b[][10],int rb, int cb,double c[][10]){
  double suma ;
```

```
  for(int i=0 ;i<ra ;i++)
    for(int j=0 ;j<cb ;c++){
      suma = 0.0;
      for(int k=0 ;k<ca ;c++){
        suma = a[i][k]*b[k][j];
      c[i][j] = suma;
    }
}
```

Otra forma de manejar arreglos de dos dimensiones es utilizar apuntadores. El apuntador señala al inicio de un arreglo unidimensional y se pasa como parámetro el número de renglones y columnas de la matriz. Una función para sumar matrices es la siguiente.

```
void sumaMat(int nren,int ncol,float *mat1, float *mat2,
float *mat3){
  int i, j;
  for( i=0; i<nrow; i++ )
    for( j=0; j<ncol; j++ )
      mat3[i*ncol+j] = mat1[i*ncol+j]+mat2[i*ncol+j];
/* *(mat3+i*ncol+j)] = *(mat1+i*ncol+j)+*(mat2+ i*ncol+j)*/
}
```

El siguiente programa suma y multiplica dos matrices de 4x4. Se incluye una función para imprimir una matriz.

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

void leerMat(int nren,int ncol,float *mat){
  int i, j;
  for( i=0; i<nren; i++)
    for( j=0; j<ncol; j++)
      scanf(" %f",&mat[i*ncol+j]);
}

void sumaMat(int nren,int ncol,float *mat1, float *mat2,
float *mat3){
  int i, j;
  for( i=0; i<nren; i++ )
    for( j=0; j<ncol; j++ )
      mat3[i*ncol+j] = mat1[i*ncol+j]+mat2[i*ncol+j];
/* *(mat3+i*ncol+j)] = *(mat1+i*ncol+j)+*(mat2+ i*ncol+j)*/
}

void prodMat(int nren,int ncol,float *mat1,float *mat2,float
*mat3){
```

```
  int i,j,k;
  for(i=0; i<nren; i++ )
    for(j=0; j<ncol; j++ ){
      mat3[i*ncol+j] = 0;
      for(k=0;k<ncol;k++)
         mat3[i*ncol+j] += mat1[i*ncol+k]*mat2[k*ncol+j];
    }
}

void printMat(int nren,int ncol,float *mat) {
  int i,j;
  for(i=0; i<nren; i++ ){
    for(j=0; j<ncol; j++ )
      printf("%.0f ",mat[i*ncol+j]);
    printf("\n");
  }
  printf("\n");
}

int main(void) {
  float *a,*b,*c;
  int ren,col;
  printf(“Teclee num. de renglones y columnas: “);
  scanf(“ %d %d”,&ren,&col);
  a = (float *)malloc(ren*col*sizeof(float));
  b = (float *)malloc(ren*col*sizeof(float));
  c = (float *)malloc(ren*col*sizeof(float));
  printf(“Teclee datos para la matriz A.“);
  leerMat(ren,col,a);
  printf(“Teclee datos para la matriz B.“);
  leerMat(ren,col,b);
  printMat(ren,col,(float *)a);
  printMat(ren,col,(float *)b);
  sumaMat(ren,col,(float *)a,(float *)b,(float *)c);
  printMat(ren,col,(float *)c);
  prodMat(ren,col,(float *)a,(float *)b,(float *)c);
  printMat(ren,col,(float *)c);
  getch();
}
```

Problemas propuestos

7.5.1. Para cada una de las siguientes situaciones, escribir las definiciones y declaraciones necesarias para transferir las variables y los arreglos indicados desde main hasta la función llamada muestra. En cada caso, asignar el valor devuelto por la función a la variable float x.
a) Transferir las variables de tipo float a y b y la formación unidimensional de 20 elementos enteros j star.

b) Transferir la variable entera n, la variable carácter c y la formación unidimensional de 50 elementos en doble precisión valores.
c) Transferir la formación bidimensional de caracteres de 12 x 50, llamada texto.
d) Transferir la formación unidimensional de 40 caracteres mensaje y Ia formación bidimensional de 50x100 elementos en coma flotante cuentas.

7.5.2. Describir la salida producida por los siguientes programas

e)
```
#inc1ude <stdio.h>
#define FILAS 3
#define COLUMNAS 4
int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12};
main(){
  int a, b, c = 999;
  for(a = 0; a < FILAS; ++a)
    for(b = 0; b < COLUMNAS; ++b)
      if(z[a][b]<c) c = z[a][b];
  printf('%d”, C);
}
```

f)
```
#inc1ude <stdio.h>
#define FILAS 3
#define COLUMNAS 4
int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12};
main(){
  int a, b, c;
  for(a = 0; a<FILAS; ++a){
    c = 999;
    for(b = 0; b<COLUMNAS: ++b)
      if(z[a][b]<c)c = z[a][b];
    printf(“%d ”, c);
  }
}
```

g)
```
#include <stdio.h>
#define FILAS 3
#define COLUMNAS 4

void sub1(int z[][COLUMNAS]);
main(){
  static int z[FILAS][COLUMNAS]={1,2,3,4,5,6,7,8,9,10,11,12};
  sub1(z};
}
void sub1(int x[][4]){
  int a,b,c;
  for(b = 0; b<COLUMNAS; ++b){
    c = 0;
    for(a = 0; a<FILAS; ++a)
```

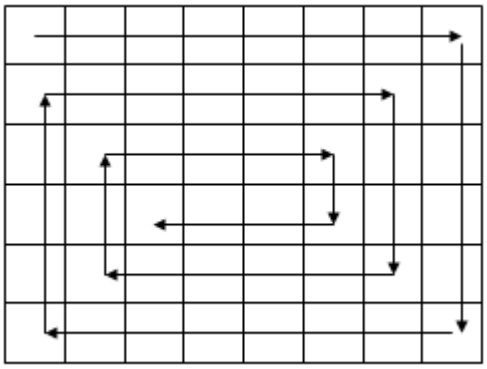
```
      if(x[a][b]>c)c = x[a][b];
    printf("%d ", C);
  }
}
```

7.5.3. Escriba una función que calcule la "traza", es decir, la suma de los elementos de la diagonal de una matriz cuadrada.

7.5.4. Escriba una función que regrese el elemento más grande de una matriz de N x M de elementos enteros.

7.5.5. Escriba una función que regrese el elemento más pequeño de una matriz de N x M de elementos enteros.

7.5.6. Escriba una función que reciba como dato un arreglo bidimensional de enteros y lo despliegue en forma espiral como la figura que se muestra.



7.5.7. Escriba una función que obtenga la traspuesta de una matriz cuadrada. La traspuesta de una matriz es otra matriz obtenida de la original al intercambiar renglones por columnas. Por ejemplo:

$$\begin{bmatrix} 1 & 2 & 7 \\ 5 & 6 & 3 \\ 4 & 7 & 8 \end{bmatrix}^T = \begin{bmatrix} 1 & 5 & 4 \\ 2 & 6 & 7 \\ 7 & 3 & 8 \end{bmatrix}$$

7.5.8. Una matriz es simétrica si es igual a su traspuesta. Escriba una función que determine si una matriz es simétrica, en cuyo caso la función deberá regresar un 1, sino, deberá regresar un 0.

7.5.9. Escriba un programa basado en menús para hacer operaciones con dos matrices. Haga funciones para capturar una matriz, restar dos matrices, desplegar una matriz.

7.6. Arreglos multidimensionales y apuntadores

Los arreglos de dimensiones pueden declararse como apuntadores. La declaración

```
Tipo arreglo[tamaño1][tamaño2];
```

Es equivalente a

```
Tipo (*arreglo)[tamaño2];
```

Esta última declaración define un puntero a arreglo de enteros de tamaño tamaño2. Es diferente a la siguiente que es un arreglo de punteros a enteros, debido a que el operador [] tiene mayor prioridad que el operador *.

```
Tipo *arreglo[tamaño 2];
```

Por ejemplo si declaramos el arreglo siguiente:

```
char a[3][5] = {{‘1’,’2’,’3’,’4’,’5’},{‘a’,’e’,’i’,’o’,’u’},
{‘A’,’E’,’I’,’O’,’U’}};
```

Estaremos definiendo el la memoria el siguiente contenido para el arreglo a.

```
12345aeiouAEIOU
```

Si declaramos un apuntador a arreglo de enteros de la siguiente manera, se tendrá un apuntador a arreglos de 5 caracteres.

```
char (*b)[5] = a;
```

Sin embargo si se declara un arreglo de apuntadores a **char**, de la siguiente manera. Lo que se tiene es un arreglo de 3 apuntadores a **char**. Necesitamos 3 apuntadores porque a es un arreglo de 3 arreglos de tipo **char**, cada uno de 5 caracteres.

```
char *c[3];
```

El siguiente programa contiene estas declaraciones. También se imprimen los tamaños de las variables y de sus contenidos.

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

main(){
  char a[3][5] = {{'1','2','3','4','5'},
{'a','e','i','o','u'},{'A','E','I','O','U'}};
  char (*b)[5] = a;
  char *c[3];
  c[0] = a[0];
```

```
  c[1] = a[1];
  c[2] = a[2];
  printf("tamaño a: %d tamaño a[0]:
%d\n",sizeof(a),sizeof(a[0]));
  printf("tamaño b: %d tamaño *b:
%d\n",sizeof(b),sizeof(*b));
  printf("tamaño c: %d tamaño *c:
%d\n",sizeof(c),sizeof(*c));
  getch();
}
```

Los valores que se imprimirán son:

```
tamaño a: 15 tamaño a[0]: 5
tamaño b: 4 tamaño *b: 5
tamaño c: 12 tamaño *c: 4
```

Note que a tiene un tamaño de 3*5=15 bytes, y un renglón de a son 5 bytes. El tamaño de es el tamaño de un apuntador, 4 bytes y lo apuntado por b es un arreglo de 5 caracteres (5 bytes). Por último, c es un arreglo de 3 apuntadores cada uno de 4 bytes o sea 12 bytes, y cada elemento del arreglo c es un apuntador de 4 bytes. El siguiente código despliega el contenido de cualquiera de las variables a, b o c utilizando notación de arreglos o de punteros, solo sustituya x por a, b o c. Note que x[i][j] = *(*(x+i)+j).

Notación de arreglos	Notación de punteros

`for(i=0; i<3;i++){`
`  for(j=0;j<5;j++)`
`  printf(“%c”,x[i][j]);`
`  printf("\n");`
`}`
	`for(int i=0; i<3;i++){`
	`  for(int j=0;j<5;j++)`
	`  printf("%c",*(*(x +i)+j));`
	`  printf("\n");`
	`}`

Las dos notaciones pueden utilizarse indistintamente pata variables que son arreglos de dos dimensiones o apuntadores a arreglos o arreglos de apuntadores o apuntadores de apuntadores. Un arreglo de dos dimensiones se puede ver como un apuntador a apuntadores. Cuando declaramos un arreglo el compilador le asigna una dirección fija, la cual se comporta como una constante y no puede ser modificada, en cambio, al declarar apuntadores estos pueden cambiar su valor durante la ejecución del programa.

Como en el caso de los arreglos de una dimensión, los arreglos de dos dimensiones pueden especificarse como apuntadores cuando se pasan como parámetros a una función. La función para sumar dos matrices la podemos especificar de la siguiente manera.

```
void sumaMatriz(double *a[10],double *b[10],int ren, int
col,double *c[10]){
  for(int i=0 ;i<ren ;i++)
    for(int j=0 ;j<col ;j++)
      *(*(c+i)+j) = *(*(b+i)+j)+ *(*(b+i)+j);
```

```
}
```

El siguiente ejemplo lee el tamaño las matrices que se desea sumar. Después lee los elementos de las matrices, calcula la suma y despliega el resultado. Note que es necesario crear las matrices dinámicamente mediante las llamadas necesarias a la función `malloc`.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAXREN
#define MAXCOL

void sumaMatriz(double *a[MAXCOL],double *b[MAXCOL],int ren,
int col,double *c[MAXCOL]){
  for(int i=0 ;i<ren ;i++)
    for(int j=0 ;j<col ;j++)
      *(*(c+i)+j) = *(*(a+i)+j)+ *(*(b+i)+j);
}

void leerMatriz(double *a[MAXCOL], int ren, int col){
  int i,j;
  for(i = 0; i<ren; i++){
    printf("\nIntroducir datos para el renglon %2d: ", i+1);
    for (j = 0; j<col; j++)
      scanf("%lf", (*(a + i) + j));
  }
}

void escribeMatriz(double *a[MAXCOL], int ren, int col){
  int i,j;
  for(i = 0; i<ren; i++){
    for (j = 0; j<col; j++)
      printf("%f ", *(*(a + i) + j));
    printf("\n");
  }
}

main(){
  double *a[10],*b[10],*c[10];
  int i,ren,col;
  printf("teclee numero de renglones: ");
  scanf(" %d",&ren);
  printf("teclee numero de columnas : ");
  scanf(" %d",&col);
  /*Debemos crear las matrices*/
  for(i=0; i<ren; ++i){
    a[i] = (double *) malloc (col * sizeof(double));
```

```
    b[i] = (double *) malloc (col * sizeof(double));
    c[i] = (double *) malloc (col * sizeof(double));
  }
  leerMatriz(a,ren,col);
  leerMatriz(b,ren,col);
  sumaMatriz(a,b,ren,col,c);
  escribeMatriz(c,ren,col);
  getch();
}
```

Problemas propuestos

7.6.1. Un programa en C contiene la siguiente declaración:

```
static int x[8] = {10, 20, 30, 40, 50, 60, 70, 80};
```

a) ¿Cuál es el significado de x?
b) ¿Cuál es el significado de (x+2)?
c) ¿Cuál es el valor de *x?
d) ¿Cuál es el valor de (*x+2)?
e) ¿Cuál es el valor de *(x+2)?

7.6.2. Un programa en C contiene la siguiente declaración:

```
static int a[2][3] = {{1.1, 1.2, 1.3},{2.1, 2.2, 2.3}};
```

a) ¿Cuál es el significado de `a`?
b) ¿Cuál es el significado de `(a+1)`?
c) ¿Cuál es el significado de `*(a+1)?`
d) ¿Cuál es el significado de `(*(a+1)+1)`?
e) ¿Cuál es el significado de `(*(a)+1)`?
f) ¿Cuál es el valor de `*(*(a+1)+1)`?
g) ¿Cuál es el valor de `*(*(a)+1)`?
h) ¿Cuál es el valor de `*(*(a+1))`?
i) ¿Cuál es el valor de `*(*(a)+1)+1`?

7.6.3. Escriba el programa de calificaciones de alumnos visto en la sección 7.5 utilizando la notación de punteros.

7.6.4. Escriba una función para calcula el determinante de una matriz cuadrada definido recursivamente de la siguiente manera:
1. Si a es una matriz de 1x1, *det*(*a*) = *a*[0][0].
2. Si a es de orden mayor que 1, calcule el determinante de la siguiente manera:

   Elija un renglón o columna. Para cada elemento *a*[i][j] de ese renglón o columna forme el producto:

   $(-1)^{i+j} * a[i][j] * det(menor(a[i][j]))$

   El determinante de a es la suma de todos esos productos.

   El menor de *a*[*i*][*j*] = *menor*(*a*[*i*][*j*]) es la matriz que se obtiene al eliminar de la matriz *a* el renglón *i* y la columna *j*.

Escriba la función creando dinámicamente el menor de la matriz que pasa como parámetro. Utilice la notación de apuntadores en su función.

7.7. Arreglos de más de dos dimensiones

Los arreglos de más de dos dimensiones son utilizados con menos frecuencia. La declaración es similar a de los arreglos de dos dimensiones. El número de elementos de arreglos multidimensionales crece rápidamente con el número de dimensiones. Por ejemplo, considere el arreglo siguiente.

```
int a[4][3][5];
```

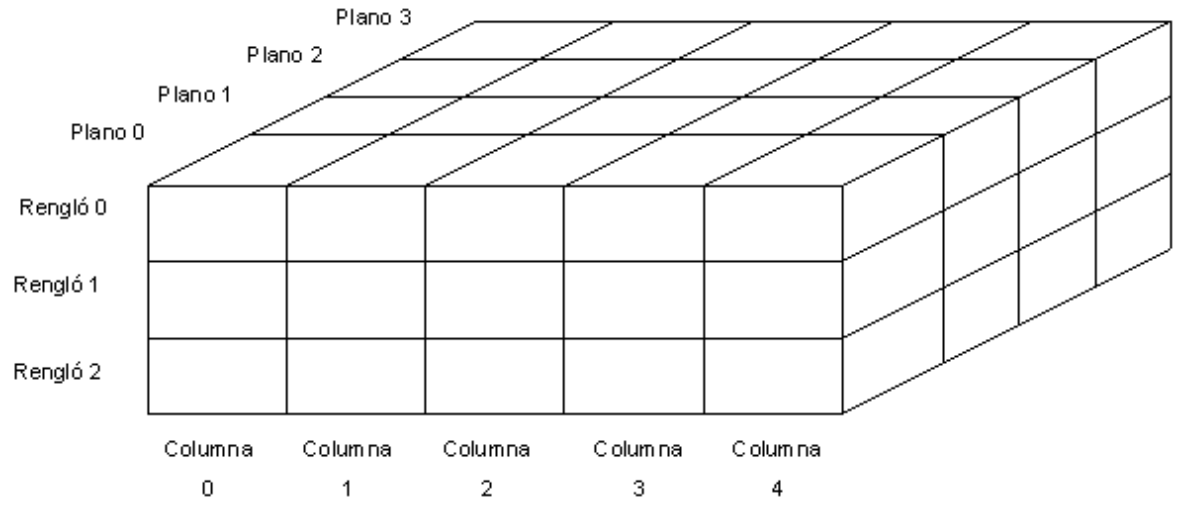
El arreglo a tendrá un total de 4 x 3 x 5 = 60 elementos de tipo entero. Una representación gráfica de tal estructura es la que se muestra en la siguiente figura. Los elementos se especifican mediante tres subíndices, el primero indica el plano, el segundo el renglón y el tercero la columna. Un ejemplo práctico puede ser una distribución de temperaturas en el espacio, otro puede ser los datos de una tomografía, que generalmente son valores de la densidad del tejido humano. Arreglos de más dimensiones son difíciles de visualizar gráficamente, sin embargo C permite cualquier número de dimensiones, limitado solo por la memoria disponible.

Para referirse a un elemento del arreglo podemos utilizar tres subíndices o la notación de apuntadores. Por ejemplo, ele elemento del plano 2, renglón 1 y columna 3 es

```
a[2][1][3]
```

o

```
*(*(*(a + 2) + 1) +3)
```



Como ejemplo consideremos el control de vehículos de una ciudad que se lleva a cabo considerando la siguiente información: fabricante (código entero de 0 a 10), modelo del vehículo (de 1990 a 2010) y la condición en que se encuentra (entero entre 0 a 3 para mal estado, regular, bueno y excelente). Esta información puede ser representada en un arreglo de tres dimensiones, la primera dimensión para la condición, la segunda para el modelo y la tercera para el fabricante. Podemos asignar índice 0 para el modelo a los vehículos de 1990, el 1 para 1991 y así sucesivamente hasta 20 para 2010. La declaración del arreglo de carros será

```
#define NUM_ANYOS 20
#define NUM_MARCAS 10
#define NUM_COND 4
int carros[NUM_MARCAS][NUM_ANYOS][NUM_COND];
```

Un elemento de este arreglo como carros[3][12][3] representa el número de vehículos del fabricante 3, modelo 2002 que se encuentran en buenas condiciones. La siguiente función reporta el número de vehículos que hay de un cierto año y su condición promedio.

```
void reporte(int car[][NUM_ANYOS][NUM_COND],int anyo){
  int cuenta=0,i,j,num;
  float cond_prom = 0.0;
  for(i=0;i<NUM_MARCAS;i++){
    for(j=0;j<NUM_COND;j++){
      num = car[i][anyo][j];
      if(num!=0){
        cond_prom += num*(j+1);
        cuenta += num;
      }
    }
  }
  printf("anyo: %d\ncantidad: %d\npromedio de condicion:
%.2f",2010-NUM_ANYOS+anyo,cuenta,cond_prom/cuenta);
}
```

Problemas propuestos

7.7.1. Indicar el significado de cada una de las declaraciones siguientes y especificar los valor que tomen los elementos de los arreglos cuando estos de declaren. Indique también el número de elementos de cada arreglo.

a) `char cc[5][4][3];`

b) `unsigned int num[2][2][3] = {{{0,5,6},{3,2,1}},{{0,0,1},{4,5,5}}};`

c) `double cuentas[50][20][80];`

e) `int q[2][3][4] = {{{1,2,3},{4,5},{6,1,8,9},{{10, 11},{},{12, 13,14}}};`

7.7.2. Dado el arreglo carros del ejemplo de vehículos de una ciudad diseñe algoritmos para calcular los siguientes datos estadísticos:
a) Número de automóviles fabricados antes de 2000 considerados como buenos o excelentes.
b) La marca de automóvil más popular.
c) Determinar el fabricante cuyos automóviles aparecen en las mejores condiciones promedio.

7.7.3. Un tomógrafo puede digitalizar un objeto de 20x20x40 cm en pasos de 0.5 cm. Declare un arreglo que permita almacenar los datos de una tomografía considerando que los valores digitalizados son números reales. A partir de los datos de un escáner como el del problema anterior se desea procesarlos y detectar las coordenadas en las que los valores caen entre 0.5 y 0.6.

Capítulo 8. Cadenas de caracteres

8.1. El tipo `char`

Las variables de tipo **`char`** ocupan un byte en la memoria. Se utilizan para representar los caracteres generados desde el teclado y también los caracteres imprimibles en la pantalla. Son compatibles con las variables de tipo entero. Los valores posibles de una variable tipo `char` van de -128 a 127. Generalmente se les asigna valor utilizando caracteres encerrados entre comillas sencillas, como se muestra a continuación.

```
char a, b, c;
a = ‘x’;/* le asigna el carácter x, o a = 120 o a = 0x78 */
b = ‘a’;/* le asigna carácter a, o a: b = 97 o a = 0x61 */
c = a – b;/* c = 120-97 = 23 */
```

Las variables de tipo `char` pueden utilizarse en cualquier expresión de tipo entero. Por ejemplo.

```
char x, y;
int a, b;
x = 45;
y = ‘a’;
a = 2*x;
b = 3*x – y;
```

Los caracteres imprimibles en la pantalla están basados en la tabla de caracteres ASCII (American Standard Code for Information Interchange). Esta tabla define el significado de los caracteres del 0 al 127. Una extensión de la tabla define los 256 caracteres agregando letras en otros idiomas además del inglés y caracteres gráficos.

La siguiente tabla muestra los valores asignados. Un Carácter tiene un código hexadecimal determinado por el número del renglón seguido del número de la columna, así por ejemplo, la letra ‘a’ tiene código 0x61 = 95, el carácter ‘0’ tiene código 0x30 = 48, el carácter ‘}’ tiene código 0x7D = 125 y así sucesivamente. Los caracteres del 0 al 31 (hexadecimal 0x00 al 0x1F) son caracteres de control y normalmente no son imprimibles. Los caracteres del 32 al 127 son los caracteres normales que comprenden letras del alfabeto inglés mayúsculas y minúsculas, números y signos de puntuación. Del 128 en adelante son letras con diversos acentos, líneas y signos especiales.

	0123456789ABCDEF
0	☺☻♥♦♣♠•◘○◙♂♀♪♫☼
1	►◄↕‼¶§▬↨↑↓→←∟↔▲▼
2	!"#$%&'()*+,-./
3	0123456789:;<=>?
4	@ABCDEFGHIJKLMNO

5	PQRSTUVWXYZ[\]^_
6	`abcdefghijklmno
7	pqrstuvwxyz{|}~⌂
8	ÇüéâäàåçêëèïîìİÅ
9	ÉæÆôöòûùÿÖÜø£Ø×ƒ
A	áíóúñÑªº¿®¬½¼¡«»
B	▒▓│┤ÁÂÀ©╣║╗╝¢¥┐
C	└┴┬├─┼ãÃ╚╔╩╦╠═╬¤
D	ðÐÊËÈıÍÎÏ┘┌█▄¦Ì▀
E	ÓßÔÒõÕµþÚÛÙýÝ¯´
F	±‗¾¶§÷¸°¨·¹³²■

La impresión de caracteres mediante `printf` se hace con el formato %c. El siguiente programa imprime la tabla ASCII. Note que algunos caracteres no se imprimen, como el 0x07 el cual es el retroceso, 0x09 que es la campana, etc.

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

main(){
  int i,j;
  char c=0;
  for(i=0;i<16;i++){
    for(j=0;j<16;j++)
      printf("%c",c++);
    printf("\n");
  }
  getch();
}
```

La tabla que despliega es la siguiente:

```
☺☻♥♦♣
♫☼
►◄↕‼¶§▬↨↑↓→←∟↔▲▼
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~⌂
ÇüéâäàåçêëèïîìİÅ
ÉæÆôöòûùÿÖÜø£Ø×ƒ
áíóúñÑªº¿®¬½¼¡«»
▒▓│┤ÁÂÀ©╣║╗╝¢¥┐
└┴┬├─┼ãÃ╚╔╩╦╠═╬¤
```

ðÐÊËÈıÍÎÏ┘┌▄▄¦Ì▀
ÓßÔÒõÕµþÞÚÛÙýÝ¯´
±‗¾¶§÷¸°¨.¹³²■

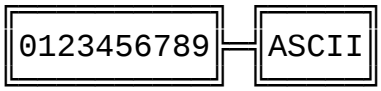
Se pueden utilizar secuencia de escape para definir valores ASCII, para eso se pueden utilizar números en octal o hexadecimal. Para usar números en octal simplemente se especifican tres dígitos octales después de la diagonal invertida, por ejemplo \044 (36 en decimal) es el carácter '$'. Para hexadecimal se antepone 'x' a los dos dígitos hexadecimales, por ejemplo \x24 representa también '$'. El siguiente programa dibuja un pequeño cuadrado de líneas sencillas.

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

main(){
  printf("\xda\xc4\xbf\n");
  printf("\xc0\xc4\xd9\n");
  getch();
}
```

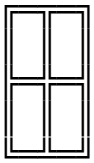
Problemas propuestos

8.1.1. Determine el código ASCII en decimal, octal y hexadecimal de los siguientes caracteres: '3', 'g', 'G', '#', '\', '~', 'î', '▀' y '¾'.

8.1.2. Escriba un programa que despliegue la siguiente figura utilizando caracteres del código ASCII.



8.1.3. Escriba una función para desplegar un tablero de n x n, donde n = 1, 2, …, 8, utilizando caracteres gráficos. La figura siguiente muestra el tablero de 2x2.



8.2. Operaciones básicas con caracteres

La lectura de caracteres puede hacerse de varias maneras. Las más comunes son las funciones `scanf`, `getch` y `getchar`. La diferencia entre estas funciones se muestra en la siguiente tabla.

función	biblioteca	Comentarios
scanf	stdio.h	Debe usarse el formato %c. Requiere que el usuario presione <ENTER>, hace eco en la pantalla.
getch	conio.h	No espera a que se presione <ENTER>, no hace eco en la pantalla.
getchar	stdio.h	Requiere que el usuario presione <ENTER>, no hace eco en la pantalla.
getche	conio.h	No espera a que se presione <ENTER>, hace eco en la pantalla.

Para desplegar un carácter se emplea `printf` o `putchar`. El siguiente ejemplo muestra algunas lecturas y escrituras de caracteres.

```
#include <stdio.h>
#include <stdio.h>
#include <conio.h>
main(){
  char p1,p2,p3=65;
  printf("\ningrese un caracter: ");
  p1 = getchar();
  putchar(p1);
  printf("\n");
  printf("\nel caracter p3 es:");
  putchar(p3);
  printf("\n");
  printf("\ningrese otro caracter: ");
  scanf("%c",&p2);
  printf("\n%c",p2);
  printf("\ningrese otro caracter: ");
  p2 = getch();
  printf("\ncaracter leido: %c",p2);
  getch();
}
```

Note que no funciona como se espera, ya que no espera para la lectura del con `scanf`, esto se debe a que el <ENTER> (o cualquier otro carácter después del primero) se asigna a p2. Para evitar este problema es necesario borrar el buffer de entrada antes de cada lectura. El buffer lo borramos utilizando la función `fflush(sdtin)` de la biblioteca `stdio.h`. La versión corregida es la siguiente.

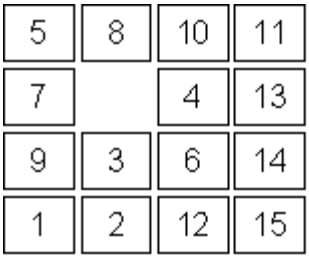
```
#include <stdio.h>
#include <conio.h>

main(){
  char p1,p2,p3=65;
  printf("\ningrese un caracter: ");
  p1 = getchar();
  putchar(p1);
```

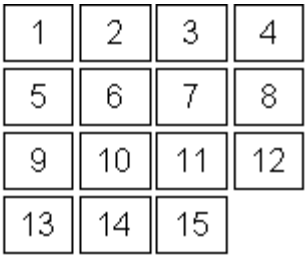
```
  printf("\n");
  fflush(stdin);
  printf("\nel caracteer p3 es:");
  putchar(p3);
  printf("\n");
  printf("\ningrese otro caracter: ");
  scanf("%c",&p2);
  printf("\n%c",p2);
  printf("\ningrese otro caracter: ");
  fflush(stdin);
  p2 = getch();
  fflush(stdin);
  printf("\ncaracter leido: %c %d",p2,p2);
  getch();
}
```

La lectura de teclas especiales requiere de un procesamiento diferente. Cada que se presiona un tecla especial se generan dos caracteres, las teclas de función generan 0 seguido de 0x3B, para F1 hasta 0x44 para F10, en hexadecimal. Las teclas de movimiento del cursor generan 0xE0 seguido de 0x48 (`H') arriba, 0x4B ('K') izquierda, 0x4D ('M') derecha, 0x50 ('P') abajo, 0x49 ('I') Regresar Página, 0x51 ('Q') Avanzar Página, 0x47 ('G') inicio, 0x4F ('O') fin, 0x52 ('R') insertar y 0x53 ('S') suprimir. Con base en esto podemos leer las teclas de flecha mediante el siguiente código

```
key = getch();
if(key == '\xe0'){
  key = getch();
  switch(key){
    case 'H' : cursor hacia arriba
    case 'K' : cursor hacia la izquierda
    case 'M' : cursor hacia la derecha
    case 'P' : cursor hacia abajo
  }
}
```

Como un ejemplo de lectura de las flechas veremos un ejemplo del rompecabezas 15. Este rompecabezas consta de un tablero de 4x4 en el que se encuentran 15 piezas numeradas del 1 al 15. El objetivo es partir de una posición desordenada llegar a una posición ordenada donde el 1 se encuentre en la posición superior izquierda, a la derecha el 2, y así sucesivamente, como se muestra en la figura.





El espacio en blanco se utiliza para desplazar las piezas alrededor del tablero. El programa dibujará el tablero anterior generando al inicio un tablero al azar y permitirá al usuario manipular las piezas desplazando el espacio vacío según convenga.

Representaremos el tablero con un arreglo de 4x4 de enteros con valore iniciales de 1 a 15 y con un 0 en el último elemento. Para dibujar el tablero y las piezas debemos posicionar el cursor dentro de la ventana utilizando una llamada al sistema de ventanas de `windows`. Para hacer llamadas al sistema de ventanas incluimos la biblioteca `windows.h` y usamos la función `SetConsoleCursorPosition` que posiciona el cursor. Usando esta función se define una función llamada `gotoxy` que posiciona el cursor.

```
void gotoxy(int x, int y) {
  COORD c;
  c.X = x - 1;
  c.Y = y - 1;
   SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE),c)
;
}
```

Para dibujar el tablero definimos los caracteres necesarios para dibujar líneas en la ventana de texto. Después de dibujar el marco se dibuja cada una de las piezas. La función para mover una pieza acepta un parámetro de tipo **`char`** que especifica la dirección de acuerdo con el código de la flecha que se presiono. Esta función retiene en dos variables **`static`** la posición anterior del espacio en blanco. La función inicia llama a mover con cien valores aleatorios de los códigos de las flechas para iniciar el tablero. Por último, la función jugar tiene un ciclo **`while`** que se ejecuta hasta presionar la tecla ESC, la función dibuja el tablero y lee el teclado para realiza el movimiento correspondiente. Falta una función que detecte cuando el tablero ha sido resuelto. El programa es el siguiente:

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>

char tablero[4][4]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0};

void gotoxy(int x, int y) {
```

```
  COORD c;
  c.X = x - 1;
  c.Y = y - 1;
   SetConsoleCursorPosition (GetStdHandle(STD_OUTPUT_HANDLE),
c);
}

void dibujaTablero(){
  char si='\xda', lh='\xc4', sd='\xbf', lv='\xb3', ii='\xc0',
id='\xd9';
  int i,j;
  /* Dibuja el marco del tablero */
  gotoxy(1,1);
  printf("%c",si);
  for(i=0;i<16;i++)
    printf("%c",lh);
  printf("%c\n",sd);
  for(j=0;j<12;j++){
    gotoxy(1,j+2);
    printf("%c",lv);
    gotoxy(18,j+2);
    printf("%c",lv);
  }
  gotoxy(1,14);
  printf("%c",ii);
  for(i=0;i<16;i++)
    printf("%c",lh);
  printf("%c\n",id);
  /* dibuja las piezas */
  for(i=0;i<4;i++){
    for(j=0;j<4;j++)
      if(tablero[j][i]){
        gotoxy(2+i*4,2+j*3);
        printf("%c%c%c%c",si,lh,lh,sd);
        gotoxy(2+i*4,2+j*3+1);
        printf("%c%2d%c",lv,tablero[j][i],lv);
        gotoxy(2+i*4,2+j*3+2);
        printf("%c%c%c%c",ii,lh,lh,id);
      }else{
        gotoxy(2+i*4,2+j*3);
        printf("    ");
        gotoxy(2+i*4,2+j*3+1);
        printf("    ");
        gotoxy(2+i*4,2+j*3+2);
        printf("    ");
      }
  }
```

```
}

void mover(char c){
  static int x=3,y=3;
  int x1,y1,t;
  x1 = x;
  y1 = y;
  switch(c){
    case 'P':if(y>0)y--;break;
    case 'M':if(x>0)x--;break;
    case 'K':if(x<3)x++;break;
    case 'H':if(y<3)y++;break;
  }
  t = tablero[y1][x1];
  tablero[y1][x1] = tablero[y][x];
  tablero[y][x] = t;
}

void inicia(){
  int i;
  char s[]="PMKH";
  srand(time(0));
  for(i=0;i<100;i++)
    mover(s[rand()%4]);
}

void jugar(){
  char key=0;
  while(key!=27){
    dibujaTablero();
    key = getch();
    if(key == '\xe0'){
      key = getch();
      mover(key);
    }
  }
}

main(){
  inicia();
  jugar();
}
```

Problemas propuestos

8.2.1. Escriba una función que regrese un 1 si se obtiene la solución del juego del rompecabezas 15. Modifique la función `main` para utilizar esta función e informar al usuario sobre la solución.

8.2.2. Modifique el programa del rompecabezas 15 para incluir un contador de movimientos. Despliegue el contador cada que se hace un movimiento.

8.2.3. Escriba un programa interactivo para calcular el *índice de masa corporal* de una persona. Este índice se calcula como el peso en kilogramos dividido entre el cuadrado de la estatura en metros. Utilice las teclas de flechas derecha e izquierda para incrementar y decrementar en una unidad el peso, las flechas arriba y abajo para incrementar y decrementar en una unidad la estatura, la tecla “Inicio” para ir a los valores inferiores de peso y estatura (30 kg y 1.0 m), la tecla “Fin” para ir a los valores máximos (200kg y 2.0 m), la tecla “Re Pag” y “Av Peg” para incrementar y decrementar peso y estatura en 10 unidades, respectivamente. Utilice la siguiente tabla para desplegar el letrero adecuado.

Etiqueta	intervalo
Infrapeso	<18,50
Delgadez severa	<16,00
Delgadez moderada	16,00 - 16,99
Delgadez aceptable	17,00 - 18,49
Normal	18.5 - 24,99
Sobrepeso	≥25,00
Preobeso	25,00 - 29,99
Obeso	≥30,00
Obeso tipo I	30,00 - 34,99
Obeso tipo II	35,00 - 39,99
Obeso tipo III	≥40,00

8.3. Biblioteca `ctype.h`

La biblioteca `ctype.h` suministra un conjunto de funciones que son muy útiles en la manipulación de caracteres. Las funciones se muestran en la tabla.

Función	Significado
`isalnum(c)`	Regresa no nulo si el argumento c es una letra o un dígito decimal, sino regresa 0.
`isalpha(c)`	Regresa no nulo si el argumento c es una letra, sino regresa 0.
`isascii(c)`	Regresa no nulo si el argumento c es ASCII (0x00 a 0xff)
`iscntrl(c)`	Regresa no nulo si el argumento c es carácter de control (0x00 a 0x1f)
`isdigit(c)`	Regresa no nulo si el argumento c es un dígito, sino regresa 0.
`isgraph(c)`	Regresa no nulo si el argumento c es imprimible excepto ‘ ‘.
`isprint(c)`	Regresa no nulo si el argumento c es imprimible.
`ispunct(c)`	Regresa no nulo si el argumento c es signo de puntuación.
`isspace(c)`	Regresa no nulo si el argumento c es espacio, tab, retorno de línea, cambio

	de línea, tab vertical, salto de página (0x09 a 0x0D, 0x20).
`isxdigit(c)`	Regresa un 1 si el argumento c es un dígito hexadecimal.
`islower(c)`	Regresa un 1 si el argumento c es una letra minúscula, sino regresa 0.
`isupper(c)`	Regresa un 1 si el argumento c es una letra mayúscula, sino regresa 0.
`tolower(c)`	Regresa el carácter c convertido a minúscula.
`toupper(c)`	Regresa el carácter c convertido a mayúscula.

Las funciones `tolower` y `toupper` convierten letras del alfabeto inglés a minúsculas y a mayúsculas, respectivamente. Para incluir las vocales acentuadas habrá que escribir una función que incluya la conversión de estas letras. De la tabla ASCII podemos obtener los códigos de las letras minúsculas acentuadas, estos son: xa0, xa1, xa2, xa3, xa4, xa5, los cuales corresponden a á, í, ó, ú, ñ y Ñ, respectivamente. El código para é es x82. Las letras mayúsculas acentuadas son: Á (xb5), É (x90), Í (xd6), Ó (xe0) y Ú (xe9). La siguiente función convierte letras acentuadas y ñ de minúsculas a mayúsculas y deja inalterado cualquier otro carácter.

```
char toupperAcentos(char c){
  if(c=='\xa0') return '\xb5';
  if(c=='\x82') return '\x90';
  if(c=='\xa1') return '\xd6';
  if(c=='\xa2') return '\xe0';
  if(c=='\xa3') return '\xe9';
  if(c=='\xa4') return '\x82';
  return c;
}
```

Para convertir cualquier letra en español a mayúscula definimos la siguiente función:

```
char toupper2(char c){
  return toupperAcentos(toupper(c));
}
```

De forma similar se pueden convertir de mayúsculas a minúsculas. El siguiente programa lee caracteres desde el teclado y los despliega en mayúscula utilizando la función `toupper2` hasta que el usuario teclee la tecla ESC.

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

char toupperAcentos(char c);
char toupper2(char c);

main(){
  char c;
  do{
```

```
    c = getch();
    c = toupper2(c);
    putchar(c);
  }while(c!=27);
}


char toupperAcentos(char c){
  if(c=='\xa0') return '\xb5';
  if(c=='\x82') return '\x90';
  if(c=='\xa1') return '\xd6';
  if(c=='\xa2') return '\xe0';
  if(c=='\xa3') return '\xe9';
  if(c=='\xa4') return '\x82';
  return c;
}

char toupper2(char c){
  return toupperAcentos(toupper(c));
}
```

Problemas propuestos

8.3.1. Escriba un programa que lea un carácter y determine el código del carácter, alfabético, dígito, de puntuación, especial o no imprimible.

8.3.2. Escriba un programa que determine cuantos caracteres generan 1 para cada una de las funciones de la biblioteca `ctype.h`.

8.3.3. Escriba una función, con un parámetro de tipo **`char`**, que regrese 1 si el parámetro no es letra o es signo de puntuación y regrese 0 en otro caso.

8.3.4. Escriba una función, con un parámetro de tipo **`char`**, que regrese 1 si el parámetro no es dígito hexadecimal o no es carácter de control y regrese 0 en otro caso.

8.3.5. Escriba una función en C que convierta caracteres acentuados y la Ñ de mayúsculas a minúsculas.

8.4. Cadenas de caracteres

Una variable de tipo cadena de caracteres es un arreglo unidimensional de caracteres. Las cadenas pueden tener longitud máxima fija o variable, dependiendo de la forma de declara la variable. Las cadenas pueden ser declaradas de dos formas:

```
char a[20] = “Hola mundo”;
```

y

```
char *b = “Hola mundo”;
```

La primera define una cadena de longitud fija. La variable a es un apuntador a un espacio de 20 caracteres. La dirección de la variable a no cambia durante toda la ejecución del programa. La segunda declaración define un apuntador a caracteres. El compilador le asigna la dirección de la cadena constante “Hola mundo”. Se le puede asignar en cualquier momento el valor de otra cadena utilizando el operador de asignación. Por otro lado la cadena señalada por a puede ser modificada en cualquier momento, la cadena señalada por b no puede ser modificada. La siguiente instrucción generaría un error de ejecución.

```
b[0] = ‘h’;
```

Si utilizamos el operador **sizeof** en ambas cadenas, se reportará 20 para la primera y 4 para la segunda. El siguiente programa ilustra algunas declaraciones e inicializaciones de cadenas.

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

main(){
  // declara una cadena de longitud 20 como máximo.
  char a[20];
  // declara una cadena de longitud 10 como máximo.
  // el contenido es la cadena “hola”, x[4] = ‘\0’.
  char x[10] = "hola";
  // declara un apuntador a cadena
  char *m = "que tal";
  // declara una cadena de 11 caracteres
  char s[] = {'B','i','e','n','v','e','n','i','d','o','\0'};
  // declara una cadena con longitud 20
  char nom[] = "desoxirribonucleico";
  x[0] = 'H';
  printf("a = %d x = %d m = %d s = %d nom = %d\n",
      sizeof(a),sizeof(x),sizeof(m),sizeof(s),sizeof(nom));
  puts(x);
  puts(m);
  puts(s);
  puts(nom);
  printf("x es \"%s\"\n",x);
  printf("m es \"%s\"\n",m);
  printf("s es \"%s\"\n",s);
  printf("nom es \"%s\"\n",nom);
  printf("x[0] es %c\n",x[0]);
  printf("m[1] es %c\n",m[1]);
```

```
  printf("s[2] es %c\n",s[2]);
  printf("nom[3] es %c\n",nom[3]);
  getch();
}
```

Se ha utilizado la función `puts` (put string, poner cadena) para la salida de cada cadena. La función `puts` agrega un cambio de línea al final de cada cadena. La salida es la siguiente:

```
a = 20 x = 10 m = 4 s = 11 nom = 20
Hola
que tal
Bienvenido
desoxirribonucleico
x es "Hola"
m es "que tal"
s es "Bienvenido"
nom es "desoxirribonucleico"
x[0] es H
m[1] es u
s[2] es e
nom[3] es o
```

Las cadenas pueden leerse con la función `gets` (get string, obtener cadena). Esta función lee todos los caracteres hasta presionar la tecla <ENTER>. Se puede leer una cadena con la función `scanf` utilizando el formato %s. De esta manera se leerán todos los caracteres hasta el primer espacio (o tabulador, o fin de línea). Puede indicarse que caracteres se desea leer dentro del formato, por ejemplo, "%[a-z]" leerá una cadena con solo caracteres alfabéticos, la lectura de la cadena termina al leer cualquier otro carácter. El siguiente ejemplo ilustra estas funciones de lectura.

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

main(){
  char cad[10];
  gets(cad);
  puts(cad);
  scanf("%s",cad);
  puts(cad);
  fflush(stdin);
  scanf("%[a-z]s",cad);
  puts(cad);
  getch();
}
```

Un ejemplo de entrada es el siguiente. Las cadenas en cursiva son las que se teclearon. Note que el segundo `scanf` lee solo los primeros caracteres alfabéticos.

```
hola, que tal?
hola, que tal?
como estan?
como
abcd1234 fin
abad
```

Note que la primera cadenas del ejemplo rebasa el tamaño de la variable a, esto puede provocar que el programa no funcione correctamente. Debemos asegurar que las cadenas asignadas quepan en las variables destino. Una forma adecuada es usar `fgets` en lugar de `gets`. La función `fgets` acepta tres parámetros, el primero es el arreglo de caracteres, el segundo el tamaño y el tercero es el archivo de entrada, en nuestro caso el archivo es el teclado especificado por `stdin`. También se puede establecer un límite a la longitud de la cadena en el formato %s. El programa anterior modificado es el siguiente.

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

main(){
  char cad[10];
  fgets(cad,10,stdin);
  puts(cad);
  fflush(stdin);
  scanf("%10s",cad);
  puts(cad);
  fflush(stdin);
  scanf("%10[a-z]",cad);
  puts(cad);
  getch();
}
```

Un ejemplo de entrada es:

```
hola que tal?
hola que
desoxirriboncleico
desoxirrib
numero1
numero
```

Para leer una cadena declarado como puntero hay que reservar memoria antes de la lectura. El siguiente código hará que el programa falle.

```
char *s;
gets(s);
```

Antes de leer reservamos memoria para la cadena con la función `malloc`. Un código más correcto es el siguiente. No olvidar liberar el espacio utilizado por `malloc`.

```
char *s;
s = (char *)malloc(sizeof(char)*20);
gets(s);
..
free(s);
```

Se puede leer cadenas con cualquier carácter usando `scanf` con el formato "`%[^\n]`". El tilde se utiliza para negar los caracteres que serán aceptados, en este caso solo se rechaza el carácter de nueva línea. Por ejemplo, el siguiente fragmento lee un nombre y la edad separados por coma.

```
char nombre[50];
int edad;
printf("Escriba nombre completo y edad separados por
coma:\n" );
scanf("%[^,],%d",nombre,&edad );
printf("\nTu nombre es %s y tu edad es %d\n", nombre, edad);
```

El siguiente programa incluye una función que cuanta el número de letras y números de una cadena de caracteres.

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

void cuenta(char *s,int *letras,int *digitos){
  *letras = 0;
  *digitos = 0;
  char *p = s;
  while(*p!='\0'){
    if(isdigit(*p)) (*digitos)++;
    if(isalpha(*p)) (*letras)++;
    p++;
  }
}

main(){
  int letras,digitos,edad;
  char cad[80],nombre[50];
  printf("Escriba una frase:\n");
  scanf("%79[^\n]s",cad);
  cuenta(cad,&letras,&digitos);
  printf("letras: %d\n",letras);
  printf("digitos: %d\n",digitos);
```

```
  getch();
}
```

De los ejemplos anteriores se puede observar que es más conveniente leer cadenas usando `gets` o `fgets`. La función `scanf` no fue diseñada para estas situaciones y puede dar resultados inesperados.

Problemas propuestos

8.4.1. ¿Qué diferencia hay entre ‘x’ y “x”?

8.4.2. Escriba sentencias que lea una frase y cambie las letras mayúsculas en minúsculas y las minúsculas en mayúsculas.

8.4.3. Escriba sentencias que lea una frase y cambie las letras acentuadas por letras no acentuadas.

8.4.4. Escriba sentencias que lea una frase y cambie loas dígitos hexadecimales por el carácter ‘X’.

8.5. Bibliotecas `stdlib.h` y `string.h`.

La biblioteca `stdlib.h` incluye algunas funciones de conversión relacionadas con cadenas de caracteres. La tabla muestra estas funciones y su significado.

Función	Significado
`atoi(cad)`	Convierte cadena a **`int`**
`atof(cad)`	Convierte cadena a **`float`**
`atol(cad)`	Convierte cadena a **`long int`**
`strtod(cad,&cad2)`	Convierte cadena a **`double`**
`strtol(cad,&cad2,base)`	Convierte cadena a **`long`**
`strtoul(cad)`	Convierte cadena a **`long`** sin signo

Un ejemplo sencillo del uso de atoi se da en el siguiente programa. El programa lee una cadena seguida de un carácter y seguido de otra cadena. Las cadenas deben estar formadas por solo dígitos decimales, y el carácter debe ser algún símbolo de operación, sino es así, el programa no desplegará nada.

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

main(){
  char op1[10],op2[10],opnd;
  int o1,o2,res;
```

```
  printf("Tecle expresion: ");
  scanf("%[0-9] %c %[0-9]",op1,&opnd,op2);
  o1 = atoi(op1);
  o2 = atoi(op2);
  switch(opnd){
    case '+':printf("%s%c%s = %d",op1,opnd,op2,o1+o2);break;
    case '-':printf("%s%c%s = %d",op1,opnd,op2,o1-o2);break;
    case '*':printf("%s%c%s = %d",op1,opnd,op2,o1*o2);break;
    case '/':printf("%s%c%s = %d",op1,opnd,op2,o1/o2);break;
  }
  getch();
}
```

La función strtol permite convertir de cualquier base entre 2 y 36 a base 10, si el valor de la base es 0, se convertirá con base en las dos primeros caracteres de la cadena de entrada, si el primero es 1-9, se convierte a base 10, si el primero es 0 y el segundo 0-7, se convertirá de octal y si el primero es 0 y el segundo x o X, se convierte de hexadecimal. El primer argumento es la cadena que se desea convertir, el segundo argumento es la dirección de una cadena de caracteres donde se guarda la subcadena que no fue convertida y el tercer argumento es la base. La función regresa un valor de tipo long. La función strtoul es similar a esta función. El siguiente ejemplo ilustra esta función.

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

int main()
{
  char numPtr[] = "12abg78", *finalPtr;
  int base;
  long n;
  for( base=2; base<=17; base++ ){
    n = strtol(numPtr, &finalPtr, base);
      printf( "Cadena \"%s\" numero en base %d: %u, final:
%s\n",
            numPtr, base,n,finalPtr);
  }
  getch();
}
```

La función `strtod` convierte una cadena a **`double`**. Acepta como argumentos la cadena que se va a convertir y una cadena por referencia que almacena la subcadena con los caracteres que no fueron convertidos. La función regresa un valor de doble precisión. El siguiente programa es una versión del programa de calculadora visto antes e ilustra el uso de esta función.

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

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

main(){
  char op1[10],op2[10],opnd,*final;
  double o1,o2,res;
  printf("Tecle expresion: ");
  scanf("%[0-9+eE.-] %c %[0-9+eE.-]",op1,&opnd,op2);
  o1 = strtod(op1,&final);
  o2 = strtod(op2,&final);
  switch(opnd){
    case '+':printf("%s%c%s = %d",op1,opnd,op2,o1+o2);break;
    case '-':printf("%s%c%s = %d",op1,opnd,op2,o1-o2);break;
    case '*':printf("%s%c%s = %d",op1,opnd,op2,o1*o2);break;
    case '/':printf("%s%c%s = %d",op1,opnd,op2,o1/o2);break;
  }
  getch();
}
```

Note que hemos agregado los caracteres ‘+’, ‘e’, ‘.’ Y ‘-‘ al formato de entrada las cadenas correspondientes a los números para aceptar números con signo en notación científica. Unos ejemplos de ejecución son los siguientes:

```
Tecle expresion: 17.5e3*5.3e-2
17.5e3*5.3e-2 = 927.500000

Tecle expresion: 3E5/2e-1
3E5/2e-1 = 1500000.000000

Tecle expresion: 1.6e-19*-5.2e20
1.6e-19*-5.2e20 = -83.200000
```

La biblioteca `string.h` contiene las funciones para la manipulación de cadenas de caracteres. La siguiente tabla es un resumen de las más importantes.

Función	Descripción
`strcpy(cad1,cad0)`	Copia la cadena cad0 en la cadena cad1
`strncpy(cad1,cad0,n)`	Copia n caracteres de la cadena cad0 en la cadena cad1
`strcat(cad1,cad0)`	Añade una copia de cad0 al final de la cadena cad1.
`strncat(cad1,cad0,n)`	Añade n caracteres de cad0 al final de la cadena cad1
`strstr(cad1,cad0)`	Localiza la primera ocurrencia de la cadena cad0 en la cadena cad1, regresa el apuntador a esa cadena.
`strlen(cad0)`	Regresa la longitud de la cadena cad0
`strcmp(cad0,cad1)`	Compara cad0 con cad1, regresa un entero mayor, igual o menor que cero según si la cadena cad0 es mayor, igual o

	menor que cad2.
`strchr(cad,c)`	Localiza la primera ocurrencia del carácter c dentro de la cadena cad y regresa el apuntador a la cadena restante.
`strtok(cad1,cad0)`	Rompe en trozos una cadena cad1, en donde cada trozo está delimitado por algún carácter de cad0. Destruye cad1 en la primera llamada, posteriores llamadas con argumento NULL recorren los demás trozos.
`memset(cad,c,n)`	Copia el carácter c en los primeros n bytes de la cadena cad.

Por ejemplo, la función `strlen` cuenta los caracteres de la cadena. Recordemos que una cadena está delimitada por el carácter nulo (‘\0’). Sabiendo esto, es fácil implementar esta función de la siguiente manera.

```
int strlen2(char *cad){
  int i = 0;
  while(cad[i]!='\0') i++;
  return i;
}
```

De manera similar se pueden implementar las otras funciones. Como segundo ejemplo implementaremos strcpy. Esta copia una cadena en otra y regresa el apuntador a la segunda.

```
char *strcpy2(char *cad1, char *cad0){
  int i = 0;
  while(cad0[i] != '\0'){
    cad1[i] = cad0[i];
    i++;
  }
  cad1[i] = '\0';
  return cad1;
}
```

Como ejemplo de aplicación de las funciones de la biblioteca `string.h` hagamos un programa que lea una oración y la separe en las palabras componentes. Definimos una función llamada separa, encargada de llevar a cabo la separación. Esta función utiliza la función `strtok` que separa la frase en trozos, la primera llamada lleva como primer parámetro el argumento frase y las llamadas posteriores llevan NULL. Los delimitadores son los caracteres de la cadena `" \n\t,.?!"`. Al terminar de analizar toda la frase la función `strtok` regresa el valor NULL.

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

void separa(char *frase){
```

```
  char *s;
  char delimitadores[] = " \n\t,.?!";
  s = strtok(frase,delimitadores);    //Primera llamada
  printf("%s\n", s);
  while(s!= NULL){
    s = strtok( NULL,delimitadores);
    if(s!=NULL)
      printf("%s\n", s);
  }
}

main(){
  char cad[80];
  printf("Escriba una frase:\n");
  scanf("%[^\n]",cad);
  separa(cad);
  getch();
}
```

La función `strstr` busca una cadena dentro de otra y regresa el apuntador al carácter donde encontró la ocurrencia. Si no encuentra la cadena buscada, regresa NULL. Como ejemplo, el siguiente programa cuenta el número de veces que una cadena está dentro de otra. Las dos cadenas se leen desde el teclado.

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

int cuenta(char *s1,char *s2){
  int total = 0;
  char *ptr=s1;
  do{
    ptr = strstr(ptr,s2);
    if(ptr!=NULL){
      total++;
      ptr++;
    }
  }while(ptr!=NULL);
  return total;
}

main(){
  char cadena[80],palabra[20];
  printf("Introduzca cadena: ");
  scanf("%[^\n]",cadena);
  printf("Introduzca palabra: ");
  scanf(" %[^\n]",palabra);
```

```
  printf("Total de ocurrencias: %d",cuenta(cadena,palabra));
  getch();
}
```

Note que en la función `cuenta` se incrementa el apuntador `ptr` si se localiza una ocurrencia, de no hacerlo así, se entraría en un lazo infinito volviendo a encontrar la misma ocurrencia. Con esta función `cuenta` podemos fácilmente contar, por ejemplo las vocales de una cadena con el siguiente código.

```
vocales = cuenta(cadena,"a")+ cuenta(cadena,"e")+
cuenta(cadena,"i")+ cuenta(cadena,"o")+ cuenta(cadena,"u");
```

Para buscar caracteres está la función `strchr`. Esta lleva como argumentos la cadena donde se buscar y el carácter que se buscará, la función regresa el apuntador al la primera ocurrencia del carácter o NULL si no se encontró ninguna ocurrencia. Una función para contar caracteres es la siguiente:

```
int cuentaChr (char *s,char c){
  int total = 0;
  char *ptr=s;
  do{
    ptr = strchr(ptr,c);
    if(ptr!=NULL){
      total++;
      ptr++;
    }
  }while(ptr!=NULL);
  return total;
}
```

Por ejemplo, para contar todas las letras minúsculas de una cadena usamos el siguiente código:

```
suma =0;
for(c = 'a';c<='z';c++)
  suma += cuentaChr(cadena,c);
```

Para ejemplificar la función strcat desarrollaremos un algoritmo para convertir números arábigos a números romanos. Vamos a limitar la conversión a números de 1 a 1000. Recuerde que en la notación romana se utilizan las letras I, V, X, L, C, D y M. Podemos distinguir dos tipos de símbolos, los que se usan para representar 1, 10, 100 y 1000 y el otro grupo para representar 5, 50 y 500. Usaremos dos cadenas, uno para los símbolos para representar 1, 10, 100 y 100 y otra para 5, 50 y 500 las cuales serán u = "IXCM" y = "VLD ". La función para convertir acepta una cadena de hasta cuatro dígitos decimales y regresa la cadena que representa el número romano. Todos los dígitos se convierten a romano concatenando símbolos de la misma posición de las cadenas u y q, excepto el dígito 9, que se convierte a romano con dos símbolos de u. El algoritmo es el siguiente.

Algoritmo A Romano. Convierte un número en arábigo a romano. Acepta una cadena representando el número en arábigo y devuelve una cadena en romano.

1. len = Longitud(cadena)
2. romano = '\0'
2. PARA i = len-1 HASTA 0 HACER
3. uni = u[len-i-1]
4. quin = q[len-i-1]
5. deci = u[len-i]
6. ch = cadena[i]
7. SELECCIONA OPCION ch
   - '1' : concatena uni a romano
   - '2' : concatena uni+uni a romano
   - '3' : concatena uni+uni+uni a romano
   - '4' : concatena uni+quin a romano
   - '5' : concatena quin a romano
   - '6' : concatena quin+uni a romano
   - '7' : concatena quin+unin+uni a romano
   - '8' : concatena quin+unin+unin+uni a romano
   - '9' : concatena uni+deci a romano
8. FINSELECCION
9. FINPARA
10. regresar romano.

La función strcat concatena dos cadenas pero solo por la derecha, eso implica que se construirá una cadena romana inversa, por esa razón la invertimos al final del proceso mediante la función strrev. La función en C es la siguiente:

```
char *aRomano(char *s){
  char u[] = "IXCM",q[] = "VLD ";
  int i,len = strlen(s);
  char *r;
  char uni[2]=" ",deci[2]=" ",quin[2]=" ",ch;
  r = (char *)malloc(20*sizeof(char));
  r[0] = '\0';
  for(i=len-1;i>=0;i--){
    ch = s[i];
    uni[0] = u[len-i-1];
    quin[0] = q[len-i-1];
    deci[0] = u[len-i];
    switch(ch){
      case '1':strcat(r,uni);break;
      case '2':strcat(r,uni);strcat(r,uni);break;
                                                    case
'3':strcat(r,uni);strcat(r,uni);strcat(r,uni);break;
      case '4':strcat(r,quin);strcat(r,uni);break;
      case '5':strcat(r,quin);break;
```

```
      case '6':strcat(r,uni);strcat(r,quin);break;
                                                    case
'7':strcat(r,uni);strcat(r,uni);strcat(r,quin);break;
      case '8':strcat(r,uni);strcat(r,uni);
strcat(r,uni);strcat(r,quin);break;
      case '9':strcat(r,deci);strcat(r,uni);break;
    }
  }
  r = strrev(r);
  return r;
}
```

La siguiente función main imprime los romanos de 30 a 60. Hace uso de la función `sprintf` que convierte un número a y una cadena formateada.

```
main(){
  int i;
  char s[6];
  for(i=30;i<=60;i++){
    sprintf(s,"%d",i);
    printf("%s = %s\n",s,aRomano(s));
  }
  getch();
}
```

Como último ejemplo de esta sección se mostrará un ejemplo para sumar números muy grandes que no caben en el tipo long. Para esto utilizaremos cadenas para representar a los números. Lo primero que hay que considerar es que al sumar dos números, el resultado puede tener una longitud más grande que el mayor de los dos números. Además, para facilitar la operación es conveniente invertir las cadenas que representan los números. La función es la siguiente:

```
char *sumaL(char *num1, char *num2){
  char *rnum1, *rnum2, *result;
  int i, mayor;
  //invertir los números
  rnum1 = strrev(num1);
  rnum2 = strrev(num2);
  if(strlen(num1)>strlen(num2))
    mayor = strlen(num1);
  else
    mayor = strlen(num2);
  //reserva memmora para el resultado
  result = (char *)malloc((mayor+2)*sizeof(char));
  memset(result,'\0',mayor+2);
  for(i=0;i<mayor;i++){
    if(i<strlen(num1))
```

```
      result[i] += rnum1[i]-'0';
    if(i<strlen(num2))
      result[i] += rnum2[i]-'0';
    if(result[i]>9){//suma mayor que 9?
      result[i] -= 10;
      result[i+1] = 1;//acarreo
    }
    result[i] += '0';//convierte a ASCII
  }
  if(result[mayor]==1)//hubo acarreo en el último dígito
    result[i]='1';
  return strrev(result);
}
```

Problemas propuestos

8.5.1. Modifique el programa de calculadora de doble precisión para incluir la operación de potenciación. Utilice el carácter '^' para esto.

8.5.2. Agregue la operación de módulo flotante. Este módulo lo calcula la función `fmod` de la biblioteca `math.h`. El módulo flotante es igual al cociente menos el producto de la parte entera del cociente por el divisor.

8.5.3. Implemente la función strncpy.

8.5.4. Escriba una función que acepte dos cadenas de caracteres S1 y S2 y elimine todas las ocurrencias de los caracteres de S1 en S2. por ejemplo:

```
S1=",.?¨"
S2="hola, ¨que tal?, espero que bien."
salida "hola que tal espero que bien"
```

8.5.5. Dada una cadena caracteres, escriba una función para eliminar los caracteres blancos repetidos, ejemplo:

entrada: " hola que tal "
salida : "hola que tal"

8.5.6. Escriba una función que acepte una cadena S y dos subcadenas S1 y S2, y reemplace todas las ocurrencias de la subcadena S1 en S por la subcadena S2.

8.5.7. Escribir una función que tome como parámetro una cadena de caracteres y la transforme de manera que las palabras aparezcan en orden inverso. Ejemplo: "Esto es un ejemplo" => "ejemplo un es Esto".

8.5.8. Escribir una función que acepte una cadena de caracteres e invierta cada una de las palabras de la cadena, suponga que el espacio es el separador. Ejemplo: "SAN LUIS POTOSI" = > "NAS SIUL ISOTOP"

8.5.9. Escriba una función que determine si una cadena es un PALINDROMO, o sea si se lee igual de izquierda a derecha y al revés. Note que debe ignorar signos de puntuación y espacios.

8.5.10. Escriba una función para convertir un número romano a su correspondiente número arábigo. Considere solo números de 1 a 1000.

8.5.11. Escriba una función para imprimir una cantidad con letras, es decir, para 1234 deber desplegar “mil doscientos treinta y cuatro”.

8.5.12. Escriba una función para restar dos números enteros largos representados como cadenas de caracteres. Tome como ejemplo la función sumaL vista en el ejemplo.

8.5.13. Desarrolle un algoritmo para multiplicar dos enteros largos y escriba la función correspondiente. Use la función para encontrar 100! con todos sus dígitos.

8.5.14. Desarrolle un algoritmo para dividir dos enteros largos y escriba la función correspondiente.

8.6. Arreglos de cadenas.

Los arreglos de cadenas son muy útiles para procesar lista de nombres, direcciones, etc. En esta sección revisaremos algunos métodos para la manipulación de arreglos de cadenas. Como vimos anteriormente la declaración más frecuente de arreglos de cadenas es mediante la sentencia:

```
char arregloCadenas[RENGLONES][TAMANIO];
```

Esta declaración define un arreglo de `RENGLONES` elementos, donde cada uno es una cadenas de `TAMANIO` caracteres. La cadena i-ésima sería `arregloCadenas[i]`, por ejemplo, el siguiente código copia la cadena `s` en el elemento 5 del arreglo:

```
char s[TAMANIO]=”Universidad”;
strcpy(arregloCadenas[5],s);
```

La siguiente función lee desde el teclado una serie de cadenas y las almacena en un arreglo que pasa como parámetro. Primero se solicita el número de elementos que van a leerse luego se leen las cadenas. El argumento n pasa por referencia para reportar el número de cadenas que se leyeron. Note que usamos `fflush` para evitar que el <ENTER> se pase a la lectura de la primera cadena.

```
void leerArreglo(char arreglo[][TAMANIO],int *n){
  int i;
  printf("Cuantas cadenas se leeran? ");
  scanf("%d",n);
```

```
  fflush(stdin);
  for(i=0;i<*n;i++){
    printf("Escriba cadena %d: ",i);
    gets(arreglo[i]);
  }
}
```

La siguiente función imprime la lista de cadenas de un arreglo.

```
void imprimeArreglo(char arreglo[][TAMANIO],int n){
  int i;
  for(i=0;i<n;i++)
    printf("%s\n",arreglo[i]);
}
```

Una operación importante es el ordenamiento de una lista de cadenas. Anteriormente utilizamos el algoritmo de la burbuja para ordenar un arreglo de números. La versión de este algoritmo para un arreglo de cadenas es el siguiente. Para comparar las cadenas usamos la función strcmp y para hacer el intercambio de dos elementos se una la función strcpy.

```
void Burbuja(char a[][TAMANIO],int tam){
  char temp[TAMANIO];
  for(int i = 0; i< tam - 1 ; i++)
    for(int j = i; j< tam;j++)
      if(strcmp(a[i],a[j])>0){
        strcpy(temp,a[j]);
        strcpy(a[j],a[i]);
        strcpy(a[i],temp);
      }
}
```

En algunos casos se utilizan arreglos paralelos, esto es, arreglos de tipos diferentes que mantienen información relacionada. Por ejemplo, en un arreglo de cadenas se puede tener los nombres de los alumnos de un grupo y en una arreglo bidimensional de flotantes las calificaciones de los diferentes exámenes parciales que realizaron. El siguiente programa ordena una lista de nombres de alumnos con base en los promedios obtenidos por cada uno. La función `promedia` tiene como argumentos el arreglo de nombres, el arreglo de calificaciones, el número de alumnos y el número de exámenes. La función calcula los promedios y los almacena en un arreglo de flotantes `prom`. Luego ordena los arreglos de nombres, de promedios y de calificaciones y finalmente los despliega.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define TAMANIO 50
#define MAX 10
```

```
#define ALUMNOS 10

void  promedia(char  nombres[][TAMANIO],float  cal[][MAX],int
numAlumnos,int numExamenes){
  int i,j,k;
  float prom[ALUMNOS],t;
  char temp[TAMANIO];
  /*Calcula el promedio de cada alumno*/
  for(i=0;i<numAlumnos;i++){
    prom[i] = 0;
    for(j=0;j<numExamenes;j++)
      prom[i] += cal[i][j];
    prom[i] /= numExamenes;
  }
  /* Ordena la lista con base en el promedio */
  for(i = 0; i< numAlumnos-1 ; i++)
    for(j = i; j< numAlumnos;j++)
      if(prom[i]<prom[j]){
      /* Intercambia elementos de los tres arreglos */
        t = prom[i];
        prom[i] = prom[j];
        prom[j] = t;
        strcpy(temp,nombres[j]);
        strcpy(nombres[j],nombres[i]);
        strcpy(nombres[i],temp);
        for(k=0;k<numExamenes;k++){
          t = cal[i][k];
          cal[i][k] = cal[j][k];
          cal[j][k] = t;
        }
      }
  /* despliega los datos ordenados*/
  for(i=0;i<numAlumnos;i++){
    printf("%20s ",nombres[i]);
    for(j=0;j<numExamenes;j++)
      printf("%4.1f ",cal[i][j]);
    printf(" = %4.1f \n",prom[i]);
  }
}

main(){
  char alumnos[ALUMNOS][TAMANIO] =
       {"Juan Perez","Olivia Suarez","Marco Vinicio",
       "Lourdes Guerrero","Abdul Lopez"};
  float calif[ALUMNOS][MAX] =
       {{6.7, 7.5, 4.5, 2, 5, 7, 5, 6, 4, 9},
       {4, 6, 7.5, 6.3, 5.8, 8, 7.8, 8.5, 5, 8},
```

```
        {3.4, 5, 5, 3.2, 7.5, 2, 1, 3, 3, 5},
        {5, 5.6, 6.7, 5.4, 7, 8, 9, 3.4, 5, 6},
        {2, 4, 7, 9, 6, 0, 6, 5.4, 5, 4}};
  printf("DATOS ORIGINALES\n");
  for(int i=0;i<5;i++){
    printf("%20s ",alumnos[i]);
    for(int j=0;j<10;j++)
      printf("%4.1f ",calif[i][j]);
    printf("\n");
  }
  printf("DATOS ORDENADOS\n");
  promedia(alumnos,calif,5,10);
  getch();
}
```

La función `main` declara el arreglo de nombres y el arreglo bidimensional de calificaciones. Luego despliega los datos iniciales y llama a la función promedia.

Problemas propuestos

8.6.1. Dada la siguiente lista de países y sus capitales, escriba un programa interactivo que lea el nombre de un país y despliegue su capital y viceversa. El programa se deberá ejecutar hasta que el usuario escriba la palabra `fin`.

México	Cd. De México
Francia	París
Chile	Santiago
Panamá	Panamá
Cuba	La Habana
Alemania	Berlín
Rusia	Mascú
China	Pekín
Estados Unidos	Washington
Inglaterra	Londres
Israel	Jerusalén
Italia	Roma

8.6.2. Escriba un programa para convertir una oración de español a pig latín. La conversión se lleva a cabo trasponiendo la primera letra de cada palabra al final de la misma y añadiéndole luego la terminación “a”. Por ejemplo: perro -> erropa. El programa deberá leer varias líneas de texto hasta que se introduzca la palabra fin y desplegará el texto convertido a pig latín.

8.7. Apuntadores y cadenas

Como se vio en la sección 7.2, existe una relación muy estrecha entre arreglos y apuntadores. Como las cadenas son arreglos, podemos manipularlas utilizando apuntadores.

Por ejemplo, la función para determinar la longitud de una cadena podemos escribirla utilizando subíndices de la siguiente forma.

```
int strlen(char []s){
  int i;
  for(i=0; s[i] != '\0';i++);
  return i;
}
```

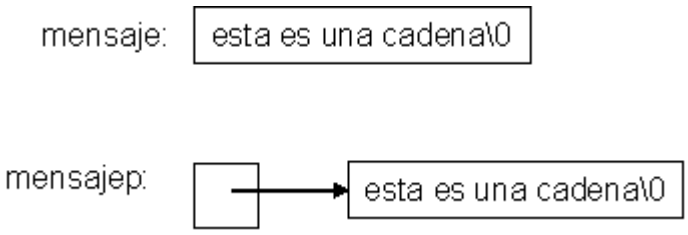
Una forma alternativa sería utilizar apuntadores como sigue. Note que el apuntador es incrementado mediante s++, esto no modifica el parámetro real que haya pasado a la función, solo modifica la privada del apuntador.

```
int strlen(char *s){
  int i;
  for(i=0; *s != '\0';s++)
    i++;
  return i;
}
```

Las siguientes declaraciones no son completamente equivalentes. La primera declara un arreglo de caracteres, los caracteres de este arreglo pueden ser modificados en cualquier momento, pero la variable mensaje siempre señalará a la misma localidad de memoria. La segunda declara un apuntador a carácter, el resultado de modificar el contenido de la cadena esta indefinido, sin embargo la variable mensajep puede modificarse para señalar a otra cadena.

```
char mensaje[] = "esta es una cadena";
char *mensajep = "esta es una cadena";
```

En la memoria se tendría la siguiente representación.



Consideremos ahora la función strcpy. Esta función copia el contenido de una cadena s1 en otra s2. Si simplemente escribiéramos s2 = s1, el resultado sería copiar el apuntador, no los caracteres. Para copiar debemos utilizar un ciclo como el siguiente.

```
void strcpy(char *s2, char *s1){
  int i;
  i = 0;
```

```
  while((s2[i] = s1[i])!= '\0')
    i++;
}
```

Note que hemos hecho la asignación dentro de la secuencia de control del ciclo. Esto es posible porque el operador asignación regresa el valor asignado, si este valor es nulo, el ciclo termina. Lo mismo hecho con apuntadores es como sigue:

```
void strcpy(char *s2, char *s1){
  while((*s2 = *s1)!= '\0'){
    s++;
    t++;
  }
}
```

Un programador experto de C escribirá el código anterior como

```
void strcpy(char *s2, char *s1){
  while((*s2++ = *s1++)!= '\0');
}
```

El valor de *s2++ es el carácter al que apunta s2 antes de incrementarse, cuando se llega al carácter nulo este es copiado antes de salir del ciclo. Como el lazo se ejecuta mientras la expresión de control no sea nula, se puede escribir la función de la siguiente forma, aun más compacta. Aunque esta forma es muy críptica, es muy conveniente entenderle código ya que aparece con frecuencia en programas en C.

```
void strcpy(char *s2, char *s1){
  while(*s2++ = *s1++);
}
```

Como otro ejemplo del uso de apuntadores, considere la función strcmp que compara dos cadenas, si son iguales regresa 0, si la primera es más grande (lexicográfica mente), regresa un valor positivo y sino regresa un valor negativo. La implementación con subíndices podría ser la siguiente. Note que podemos restar libremente los caracteres para determinar el mayor.

```
int strcmp(char *s1, char *s2){
  int i;
  for(i=0; s1[i] == s2[i];i++)
    if(s[i]=='\0')
      return 0;
  return s1[i]-s2[i];
}
```

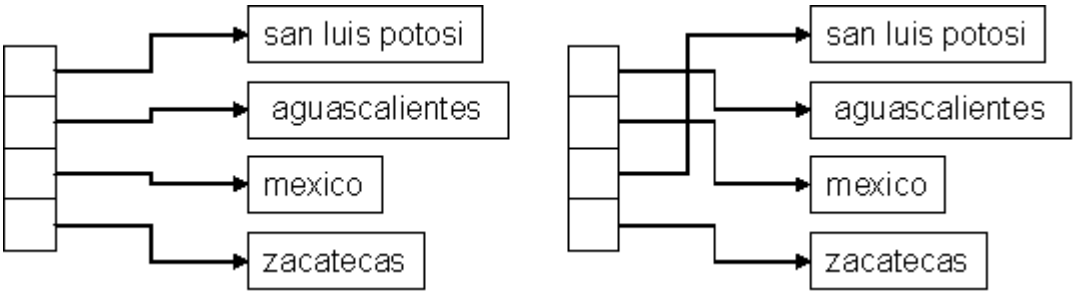
La versión con apuntadores es la siguiente:

```
int strcmp(char *s1, char *s2){
  int i;
  for( ; *s1 == *s2;s1++,s2++)
    if(*s1=='\0')
      return 0;
  return *s1-*s2;
}
```

Como último ejemplo de apuntadores consideraremos nuevamente el problema de ordenar un arreglo de cadenas. En la sección anterior se discutió una función para ordenar un arreglo de cadenas, esta función podemos modificarla para ordenar un arreglo de punteros a cadenas. De esta manera la ordenación se llevará acabo en el arreglo de apuntadores, no en las cadenas, ahorrando una considerable cantidad de tiempo. La función modificada es:

```
void Burbuja(char *a[],int tam){
  char *temp;
  for(int i = 0; i< tam - 1 ; i++)
    for(int j = i; j< tam;j++)
      if(strcmp(a[i],a[j])>0){
        temp = a[j];
        a[j] = a[i];
        a[i] = temp;
      }
}
```

Note que solo se reordena el arreglo de apuntadores, no se copian las cadenas cuando se intercambian. Para arreglos grandes esta función será considerablemente más rápida. Como ejemplo de una función main considere la siguiente. Hemos utilizado una versión modificada de la función imprimeArreglo para desplegar el arreglo desordenado y ordenado. El efecto de la función puede visualizarse en la siguiente figura.




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

void Burbuja(char *a[],int tam);
void imprimeArreglo(char *a[],int tam);
```

```
main(){
  char *s[] = {"san luis potosi","aguascalientes",
"mexico","zacatecas"};
  imprimeArreglo(s,4);
  Burbuja(s,4);
  imprimeArreglo(s,4);
  getch();
}

void imprimeArreglo(char *a[],int tam){
  for(int i = 0; i< tam ; i++)
    printf("%s\n",a[i]);
  printf("\n");
}

void Burbuja(char *a[],int tam){
  char *temp;
  for(int i = 0; i< tam - 1 ; i++)
    for(int j = i; j< tam;j++)
      if(strcmp(a[i],a[j])>0){
        temp = a[j];
        a[j] = a[i];
        a[i] = temp;
      }
}
```

Problemas propuestos

8.7.1. Escriba la siguiente función utilizando apuntadores.

```
void f(char a[],int n){
  char temp[];
  int i=0;
  while(a[i]!='\0'){
    if(isalpha(a[i]))
      a[i] = 'x';
  }
}
```

8.7.2. Escriba una versión lo más eficiente posible de la función del problema 1 usando apuntadores.

8.7.3. Escriba dos versiones para strcat una utilizando subíndices y otra con apuntadores.

8.7.4. Escriba dos versiones para strncpy que copia los primeros n caracteres de sus argumentos, una utilizando subíndices y otra con apuntadores.

Capítulo 9. Estructuras y uniones

9.1. Definición de estructuras

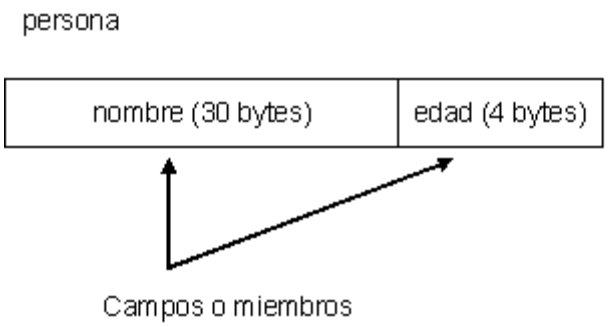
Una estructura es un agrupamiento de variables, que pueden ser de tipo diferente, bajo un solo nombre. En algunos lenguajes se les llama registros. A diferencia de los arreglos en una estructura podemos tener elementos de diferente tipo. Las estructuras permiten agrupar datos en una sola unidad para hacerlos más manejables. Las estructuras se pueden copiar, asignar, pasar a funciones y ser regresadas por funciones. De esta última manera las funciones pueden regresar algo más complejo que un valor escalar simple.

La declaración de estructuras se hace mediante la palabra reservada **`struct`**. La sintaxis es la siguiente. Es importante poner el punto y coma al final de la declaración.

```
struct nombre{
  campos;
};
```

En donde cada campo o miembro es la declaración de una o más variables. Esta declaración define el nombre como una estructura que puede utilizarse posteriormente para declarar variables de este tipo. La siguiente declaración define una estructura que consta de una cadena llamada nombre de 30 caracteres y un campo numérico entero llamado edad. La figura muestra la una representación gráfica de la estructura persona.

```
struct persona{
  char nombre[30];
  int edad;
};
```



Una vez declarada la estructura se pueden definir variables de tipo estructura. En ANSI C es necesario anteponer la palabra reservada **`struct`** cada que se declaran variables de tipo estructura, pero algunos compiladores como dev-C no requieren esto. Por ejemplo, la siguiente declaración define p1 y p2 como estructuras persona, cada una con un campo para nombre y otro para edad.

```
persona p1, p2;

o

struct persona p1, p2; /* en ANSI C*/
```

También se pueden declarar variables de tipo estructura al definir la estructura. Es decir, es válido hacer lo siguiente:

```
struct persona{
  char nombre[30];
  int edad;
} p1, p2;
```

Esta sentencia reserva espacio en memoria para almacenar de forma contigua las estructuras `p1` y `p2`. Para asignar valores a las estructuras se utiliza el operador ".", este se coloca entre el nombre de la estructura y cada uno de los campos. Por ejemplo, podemos leer con `gets` los campos nombre de las estructuras `p1` y `p2`.

```
gets(p1.nombre);
p1.edad = 25 ;
gets(p2.nombre);
p2.edad = 33 ;
```

El siguiente programa muestra como definir dos estructuras, asignar valor a los campos y mostrarlos en pantalla.

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

struct persona{
  char nombre[30];
  int edad;
};

main(){
  persona per1,per2;
  strcpy(per1.nombre,"pedro");
  strcpy(per2.nombre,"maria");
  per1.edad = 23;
  per2.edad = 17;
  printf("nombre: %s\nedad: %d\n",per1.nombre,per1.edad);
  printf("nombre: %s\nedad: %d\n",per2.nombre,per2.edad);
  getch();
}
```

Se puede inicializar una estructura al declararla con una lista de inicializadores. Las constantes deben coincidir el tipo de los campos.

```
persona p1 = {"Juan Perez",34}, p2;
```

Una estructura puede directamente asignarse a otra. Lo que no puede hacerse es comparar por igualdad o diferencia dos estructuras, es forzoso comparar campo por campo. La siguiente asignación es válida y copia todos los campos de `p1` en `p2`.

```
p2 = p1;
```

Pero, lo siguiente no es válido.

```
if(p1==p2)
 …
```

Debe compararse campo por campo

```
if((strcmp(p1.nombre,p2.nombre)==0)&&p1.edad==p2.edad)
 …
```

Una estructura puede contener cualquier tipo de variable, incluso otras estructuras, a estas estructuras se les llama *estructuras anidadas*. Como ejemplo considere la estructura para representar puntos en el plano.

```
struct punto{
  float x,y;
};
```

Con esta estructura podemos crear otra estructura para representar una recta en el plano. Una recta queda definida por los puntos extremos, como se muestra a continuación.

```
struct recta{
  punto p1, p2,
};
```

El siguiente programa declara una variable de tipo punto y dos de tipo recta. El punto `p` se inicializa con los valores 7 para `x` y 3 para `y`. La recta `r1` se inicializa dándole valores a las coordenadas de los puntos extremos. Luego se asigna al primer punto de la recta `r2` los valores del punto `p` y al segundo punto los valores del primer punto de `r1`.

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

struct punto{
  float x,y;
```

```
};

struct recta{
  punto p1, p2;
};

main(){
  punto p = {7,3};
  recta r1 = {{2,3},{5,6}};
  recta r2;
  r2.p1 = p;
  r2.p2 = r1.p1;
  printf("recta de (%f,%f) a (%f,%f)\n",r1.p1.x, r1.p1.y,
r1.p2.x, r1.p2.y);
  printf("recta de (%f,%f) a (%f,%f)\n",r2.p1.x, r2.p1.y,
r2.p2.x, r2.p2.y);
  getch();
}
```

Otros ejemplos de estructuras son los siguientes.

Elemento químico (se inician los datos para el oro). Reservamos tres caracteres para el símbolo ya que hay que almacenar el carácter nulo.

```
struct elemento{
  int nAtomico;
  float masaAtomica;
  char simbolo[3];
  char nombre[20];
} e = {79, 196.987, “Au”, “Oro”};
```

Datos bibliográficos.

```
struct libro{
  char titulo[30];
  char autor[40];
  char editorial[10];
  int edicion;
  int anyo;
} x = {“El Lenguaje de Programación C”,
       “Brian w. Kernighan, Dennis M. Ritchie”,
       “Prentice Hall Hispanoamericana”, 2, 1988};
```

Empleado

```
struct empleado{
  char nombre[15];
```

```
  char apellido[20];
  int edad;
  float sueldo;
  int anyoIngreso;
} x = {“Pedro”,“Moreno”, 35, 2567.5, 1998};
```

Como otro ejemplo considere una persona que puede requerir los siguientes datos para su identificación: nombre, apellido, dirección, teléfono, fecha de nacimiento, sexo y CURP. A su vez la dirección puede tener diferentes campos separados como: calle, número, colonia, código postal, ciudad, estado y país. También la fecha de nacimiento puede constar de: día, mes y año. Las siguientes declaraciones definen los tipos necesarios.

```
struct fecha{
  int dia, mes, anyo;
};

struct direccion{
  char calle[30];
  int numero;
  char colonia[20];
  int CP;
  char ciudad[20],estado[15],pais[15];
};

struct persona{
  char nombre[15],apellido[20];
  direccion dir;
  char telefono[15];
  fecha fechaNacimiento;
  char sexo; /* ‘h’ – hombre, ‘m’ - mujer */
  char CURP[18];
};
```

Para asignarle valor a la dirección de una persona tendremos que escribir el siguiente código:

```
persona p;
strcpy(p.dir.calle,”olmo”);
p.dir.numero = 145;
strcpy(p.dir.colonia,”Las Flores”);
p.dir.CP = 78234;
strcpy(p.dir.ciudad,”Valles”);
strcpy(p.dir.estado,”San Luis Potosí”);
strcpy(p.dir.colonia,”México”);
```

La siguiente sentencia **`if`** verifica que una persona viva en Valles y sea de sexo masculino.

```
if(strcmp(p.dir.ciudad,"Valles")==0&&p.sexo=='h')
```

Problemas propuestos

9.1.1 Defina una estructura para guardar información sobre clientes de un banco. Incluya miembros para el nombre, dirección, teléfono, número de cuenta, tipo de cuenta y saldo de la cuenta.

9.1.2. Declare variables del tipo de la estructura del problema anterior. Inicialice una estructura con los siguientes datos: nombre = "Roberto Lima", dirección = "Olmo #245", teléfono = "34-32-65-67", número de cuenta = 3457, tipo de cuenta = 2, saldo = 45928.60.

9.1.3. Defina una estructura para almacenar la siguiente información de un planeta: masa, radio, distancia al sol, gravedad superficial, duración del año en días y periodo de rotación en horas. Escriba un programa que inicialice los datos del planeta tierra y los despliegue.

9.1.4. Usando la estructura del problema anterior escriba un programa que defina una variable de tipo planeta y lea la masa del planeta y su radio. Calcule la densidad del planeta y la gravedad en la superficie y los despliegue. La densidad es masa entre volumen y la gravedad se calcula con $g = GM/R^2$, donde $G = 6.67 \times 10^{-11}$ $m^3kg^{-1}s^{-2}$, M es la masa y R el radio.

9.1.5. Un número real puede representarse como una parte entera y un multiplicador de por una potencia de 10, por ejemplo: 234.56 x 10-7 = 23456 x 10-9, la parte entera es 23456 y el exponente de la potencia de 10 es -9. Declare una estructura para representar números reales de esta forma. Escriba código para evaluar la multiplicación de dos números reales.

9.2. Estructuras y apuntadores

En muchos casos es conveniente utilizar apuntadores a estructuras. Para usar una estructura mediante un apuntador se debe primero crear la estructura con `malloc`. Considere el siguiente ejemplo.

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

struct punto{
  float x,y;
};

main(){
  punto *p;
  p = (punto *)malloc(sizeof(punto));
  (*p).x = 5;
  (*p).y = 8;
```

```
  printf("punto (%.0f,%.0f)",(*p).x,(*p).y);
  getch();
}
```

Note que usamos `(*p).x` para acceder al campo `x` ya que el operador punto tiene precedencia respecto al operador de indirección, la notación `*p.x` significa "el contenido de lo direccionado por el campo `x`" lo cual no es válido ya que `x` es un flotante. Para abreviar se puede usar el operador "`->`" para acceder a los campos. Las referencias a los campos se cambiarían por las siguientes:

```
  p->x = 5;
  p->y = 8;
  printf("punto (%.0f,%.0f)",p->x,p->y);
```

Una estructura puede tener referencias a si misma. Veamos una modificación a la estructura punto para poder representar un polígono. Un polígono es una lista de puntos. Podemos formar una lista agregando un campo apuntador a la estructura punto de la siguiente manera:

```
struct punto{
  float x,y;
  punto *siguiente;
};
```

Un polígono se representará como un apuntador a esta estructura. A su vez el primer elemento de la estructura señalara al siguiente punto del polígono, y así sucesivamente. El siguiente programa lee n puntos y los inserta en un polígono. Note que necesitamos dos apuntadores p y q, para poder hacer la inserción de cada elemento. Al final se despliegan los puntos del polígono, note que se despliegan en orden inverso por la forma en que se insertaron. La estructura creada se llama genéricamente "lista" y a cada elemento se le llama "nodo".

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

struct punto{
  float x,y;
  punto *siguiente;
};

main(){
  punto *p,*q;
  int n,i;
  printf("Numero de puntos? ");
  scanf("%d",&n);
  q = NULL;
```

```
  for(i=0;i<n;i++){
    p = (punto *)malloc(sizeof(punto));
    printf("x y: ");
    scanf("%f %f",&p->x,&p->y);
/*se hace señalar al anterior inicio de la lista */
    p->siguiente = q;
/*se actualiza el nuevo inicio de la lista*/
    q = p;
  }
  p = q;
  while(p){
    printf("punto (%.0f,%.0f)\n",p->x,p->y);
    p = p->siguiente;
  }
  getch();
}
```

Problemas propuestos

Dada la siguiente declaración de estructura y variables diga cuales de las siguientes sentencias son válidas:

```
struct prueba{
  int n, m;
  char c;
  double p;
  char *s;
};
prueba t, *r;
```

a) `t->n = 5;`
b) `r->s = “hola”;`
c) `r->s[2] = ‘c’;`
d) `scanf(“%f”,&t.p);`
e) `scanf(“%d”,&t->n);`
f) `*r.p = t.p;`
g) `(*r).m = 800;`
h) `t.c = r->s[r->n];`

Agregue código al programa del polígono para que calcule el área del polígono. Considere los casos en que no hay puntos suficientes para crear un área cerrada, en tales casos el área deberá ser cero.

9.3. Paso de estructuras a funciones.

Las funciones en C pueden recibir estructuras como argumentos y también pueden regresar estructuras. Por ejemplo, la siguiente función despliega una variable del tipo punto visto anteriormente.

```
struct punto{
  float x,y;
};

void despliegaPunto(punto p){
  printf(“(%.2f, %.2f)\n”,p.x,p.y);
}
```

Las estructuras pueden pasar por valor, como en el caso anterior, o por referencia, en cuyo caso se pasa un apuntador a la estructura. Por ejemplo, la siguiente función lee las coordenadas de un punto. Note que las coordenadas deben ir separadas por una coma, de acuerdo con el formato de `scanf`.

```
void leePunto(punto *p){
  printf("Teclee x,y: ");
  scanf("%f,%f",&p->x,&p->y);
}
```

Otra forma de leer es regresando el valor leído a través de la función. Un ejemplo es el siguiente.

```
punto leePunto2(){
  punto p;
  printf("Teclee x,y: ");
  scanf("%f,%f",&p.x,&p.y);
  return p;
}
```

El siguiente programa es un ejemplo de una estructura para representar una dirección. Se definió una función para leer una dirección y otra para desplegar. Las lecturas de cadenas se hacen con `gets` para admitir cadenas con espacios.

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

struct dir{
  char calle[20];
  int numero;
  char colonia[30];
  int cp;
};

dir dameDir(){
  dir d;
  printf("Calle? ");
```

```
  gets(d.calle);
  printf("Numero? ");
  scanf("%d",&d.numero);
  fflush(stdin);
  printf("Colonia? ");
  gets(d.colonia);
  printf("Codigo postal? ");
  scanf("%d",&d.cp);
  return d;
}

void despliegaDir(dir d){
  printf("%s #%d\n",d.calle,d.numero);
  printf("colonia %s\n",d.colonia);
  printf("CP %d\n",d.cp);
}

main(){
  dir b;
  b = dameDir();
  despliegaDir(b);
  getch();
}
```

Si las estructuras son grandes es más conveniente pasar a las funciones apuntadores. Las estructuras son muy útiles en la representación de otros tipos numéricos como los números racionales y complejos. Un número racional es un número que puede expresarse como el cociente de dos enteros. Una forma de representar números racionales es con una estructura formada por dos campos enteros, en un campo se almacena el numerador y otro para el denominador.

```
struct racional{
  int num,den;
};
```

La primera operación con racionales es la de comparación. No basta con comparar el numerador y el denominador de un racional para determinar su igualdad, ya que por ejemplo, 2/4 es igual a 1/2, pero $2 \neq 1$ y $4 \neq 2$. Para poder comparar dos racionales debemos primero representarlos de forma normalizada, esta normalización consiste en dividir ambos, numerador y denominador entre el máximo común divisor de ambos. El algoritmo de Euclides permite llevar a cabo esta operación de forma eficiente.

Algoritmo de Euclides. Permite reducir un número racional.

1. Sea a el mayor y b el menor entre numerador y denominador.
2. MIENTRAS (b > 0) HACER
3. Divídase a entre b y sea r el residuo.

4. Hágase a = b y b = r
5. FINMIENTRAS
6. Divídase numerador y denominador entre a.

La función en C que normaliza dos racionales es reduce.

```
void reduce(racional *r){
  int a,b,q;
  if(r->num > r->den){
    a = r->num;
    b = r->den;
  }else{
    a = r->num;
    b = r->den;
  }
  while(b){
    q = a % b;
    a = b;
    b = q;
  }
  r->num = r->num/a;
  r->den = r->den/a;
}
```

Una vez reducidos la comparación de dos racionales es:

```
int igual(racional a, racional b){
  reduce(&a);
  reduce(&b);
  return (a.num==b.num && a.den==b.den);
}
```

La suma de racionales se calcula con la fórmula: a/b+c/d = (ad+bc)/(bd). La siguiente función suma dos racionales a y b y regresa el resultado.

```
racional suma(racional a, racional b){
  racional c;
  c.num = (a.num*b.den+b.num*a.den);
  c.den = a.den*b.den;
  reduce(&c);
  return c;
}
```

También puede definirse la suma con una función que acepte dos racionales y regrese por referencia un racional de la siguiente manera:

```
void suma(racional a, racional b, racional *c){
```

```
  c->num = (a.num*b.den+b.num*a.den);
  c->den = a.den*b.den;
  reduce(c);
}
```

Las demás operaciones de racionales son igualmente sencillas:

```
racional resta(racional a, racional b){
  racional c;
  c.num = (a.num*b.den-b.num*a.den);
  c.den = a.den*b.den;
  reduce(&c);
  return c;
}

racional multiplica(racional a, racional b){
  racional c;
  c.num = a.num*b.num;
  c.den = a.den*b.den;
  reduce(&c);
  return c;
}

racional divide(racional a, racional b){
  racional c;
  c.num = a.num*b.den;
  c.den = a.den*b.num;
  reduce(&c);
  return c;
}
```

Podemos fácilmente definir una función para leer un racional y otra para desplegar un racional. Note que la lectura requiere que el racional sea escrito con la diagonal, ¿son necesarios los espacios entre '/'?.

```
void escribir(racional a){
  printf("%d/%d\n",a.num,a.den);
}

void leer(racional *a){
  scanf("%d / %d",&a->num,&a->den);
}
```

La siguiente función `main` declara dos números racionales, los lee y realiza las cuatro operaciones básicas con ellos.

```
main(){
```

```
  racional a,b,c;
  printf("Teclee un racional: ");
  leer(&a);
  printf("Teclee otro racional: ");
  leer(&b);
  escribir(a);
  escribir(b);
  c=suma(a,b);
  escribir(c);
  c=resta(a,b);
  escribir(c);
  c=multiplica(a,b);
  escribir(c);
  c=divide(a,b);
  escribir(c);
  getch();
}
```

Problemas propuestos

9.3.1. Escriba una función que acepte dos números racionales con la representación vista en el texto y regrese un 1 si los números son iguales y 0 si son diferentes.

9.3.2. Escriba una función de conversión entre flotantes y racionales y otra para conversión de racionales y flotantes.

9.3.3. Modifique la función que despliega un número racional para que despliegue 0 cuando el denominador se 0 y el denominador sea diferente de 0. También deberá desplegar solo el numerador si el denominador es 1. Garantice que el signo menos para racionales negativos se despliegue en el numerador.

9.3.4. Escriba un programa basado en menús para realizar operaciones con racionales.

9.3.5. Un número complejo se puede representar como una estructura formada por dos reales, uno para la parte real y otro para la parte imaginaria. Defina una estructura para representar números complejos. Implemente funciones para: sumar, restar, multiplicar, dividir números complejos y extraer el módulo y el argumento de un complejo.

9.4. Arreglos de estructuras.

Es común que se utilicen arreglos de estructuras. Si deseamos guardar los datos de un grupo estudiantes en estructuras, basta con declarar un arreglo del tipo básico. Suponga que se requiere saber: nombre, apellido, clave, carrera, promedio y fecha de ingreso. Definiremos una estructura para la fecha y otra para estudiante.

```
struct fecha{
```

```
  int dia, mes, anyo;
};

struct estudiante{
  char nombre[20];
  char apellido[15];
  char clave[10];
  int carrera;
  float promedio;
  fecha fechaIngreso;
};
```

Suponemos que el miembro carrera es un índice a una tabla con los nombres de las carreras. La declaración siguiente define un arreglo de 20 elementos del tipo estudiante.

```
estudiante est[50];
```

Para actualizar el promedio de un estudiante escribiremos lo siguiente:

```
est[i].promedio = valor;
```

La siguiente función ordena el grupo de alumnos con base en el promedio obtenido. Note que se pueden intercambiar las estructuras mediante una simple asignación.

```
void ordena(estudiante e[], int n){
  int i, j;
  estudiante temp;
  for(i = 0;i<n-1;i++)
    for(j = i;j<n;j++)
      if(estudiante[i].promedio<estudiante[j].promedio){
        temp = estudiante[i];
        estudiante[i] = estudiante[j];
        estudiante[j] = temp;
      }
}
```

El siguiente programa completo es un ejemplo de inicialización de un arreglo de estudiantes, ordenación en base a su promedio y despliegue. Se ha definido una función para desplegar el arreglo de estructura.

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

struct fecha{
  int dia, mes, anyo;
};
```

```
struct estudiante{
  char nombre[20];
  char apellido[15];
  char clave[10];
  int carrera;
  float promedio;
  fecha fechaIngreso;
};

void ordena(estudiante e[], int n);
void despliegaGrupo(estudiante e[],int n);

main(){
  estudiante est[10] = {
    {"juan","perez","001109211",3,3.4,{3,5,2005}},
    {"pedro","lopez","002367879",3,6.3,{5,1,2006}},
    {"luis","mata","003456789",3,7.5,{2,5,2003}},
    {"ana","medina","001567436",3,2.8,{2,5,2003}},
    {"oscar","dias","001234522”,3,9.3,{3,5,2005}},
    {"juana","baez","001345678",3,5.6,{5,1,2006}},
    {"maria","lima","002340015",3,7.1,{5,1,2006}},
    {"hilda","mora","001455411",3,5.2,{4,1,2007}},
    {"luisa","limon","002344431",3,8.7,{4,1,2007}},
    {"luis","mendez","004567225",3,3.8,{2,5,2003}}};
  printf("Listado sin ordenar\n");
  despliegaGrupo(est,10);
  ordena(est,10);
  printf("\n\nListado ordenado\n");
  despliegaGrupo(est,10);
  getch();
}

void ordena(estudiante e[], int n){
  int i, j;
  estudiante temp;
  for(i = 0;i<n-1;i++)
    for(j = i;j<n;j++)
      if(e[i].promedio<e[j].promedio){
        temp = e[i];
        e[i] = e[j];
        e[j] = temp;
      }
}

void despliegaGrupo(estudiante e[],int n){
  int i;
  for(i = 0;i<n;i++)
```

```
                                printf("%2d:       %s       %s\t
%6.1f\n",i,e[i].nombre,e[i].apellido,e[i].promedio);
}
```

Revisaremos un ejemplo de estructura para mantener la información de los pacientes de un hospital. Supondremos que la información necesaria consiste de:

Nombre y apellidos (cadenas de caracteres)
Edad (entero)
Sexo (carácter)
Condición (entero)
Domicilio(estructura)
  calle(cadena de caracteres)
  número (entero)
  Colonia (cadena de caracteres)
  Código postal (cadena de caracteres)
  Ciudad (cadena de caracteres)
Teléfono (cadena de caracteres)

Donde condición es un entero entre 1 y 5, 1 mínimo de gravedad, 5 máximo de gravedad. La estructura en C es la siguiente:

```
struct direccion{
  char calle[30],colonia[20],ciudad[30];
  int numero,cp;
};

struct paciente{
  char nombre[30];
  int edad;
  char sexo;
  direccion dir;
  int concicion;
  char telefono[20];
};
```

Suponga que hay un total de `NUMPACIENTES` en el hospital. El siguiente fragmento de código despliega nombre y teléfono de los pacientes con la máxima gravedad.

```
paciente pac[NUMPACIENTES];
...
printf(“Pacientes con máxima gravedad:\n”);
for(i = 0;i < NUMPACIENTES;i++)
  if(pac[i].condicion==5)
    printf(“Nombre: %s, telefono: %s\n”,
pac[i].nombre,pac[i].telefono);
```

El siguiente fragmento calcula el porcentaje de pacientes hombres y mujeres. Note que `sumaH` y `sumaF` son variables de tipo **`float`** (o **`double`**).

```
sumaH = 0.0;sumaF = 0.0;
for(i = 0;i<NUMPACIENTES;i++)
  if(pac[i].sexo==‘H’)
    sumaH++;
  else
    sumaF++;
printf(“% de Hombres= %.2f\n”,sumaH/ NUMPACIENTES *100.0);
printf(“% de Mujeres= %.2f\n”,sumaF/ NUMPACIENTES *100.0);
```

Otros ejemplos son:

Número de pacientes en cada condición, se utiliza el arreglo `c` de cinco elementos para llevar la cuenta del número de pacientes en cada condición.

```
int c[5] = {0};
for(i = 0;i< NUMPACIENTES;i++)
  c[pac[i].condicion-1]++;
for(i = 0;i<5;i++)
  printf(“Pacientes en condicion %d: %d\n”,i+1,c[i]);
```

Nombre calle y número de pacientes masculinos en condición de máxima gravedad.

```
for(i = 0;i< NUMPACIENTES;i++)
  if(pac[i].condicion==5&&pac[i].sexo=‘H’)
    printf(“Nombre: %s, direccion: %s #%d\n”,
    pac[i].nombre, pac[i].dir.calle, pac[i].dir.numero);
```

Problemas propuestos

9.4.1. Considere las estructuras para pacientes en un hospital. Escriba sentencias para lo siguiente:
a) Desplegar los pacientes de mínima gravedad entre dos edades leídas desde el teclado.
b) Desplegar los pacientes menores de 18 años de sexo femenino cuyo domicilio este en la colonia “Jardín”.
c) Encontrar el paciente más joven del hospital.
d) Encontrar el paciente más viejo del hospital con la máxima gravedad.

9.4.2. Escribir un programa que gestione una agenda de direcciones. Los datos de la agenda se almacenan en memoria en un arreglo de estructuras, cada una de las cuales tiene los siguientes campos:
Nombre
Dirección
Teléfono

Teléfono celular
Dirección de correo electrónico
El programa de poder añadir una nueva entrada a la agenda, borrar una entrada, buscar por nombre y eliminar una entrada determinada por el nombre.

9.4.3. Escriba un programa basado en menús para gestionar los pacientes de un hospital utilizando la estructura revisada en el texto. Incluya opciones para agregar un paciente, borrar un paciente, editar datos de un paciente, listar pacientes dependiendo de la condición, el sexo y la edad.

9.5. Uniones

Una unión es una estructura en la que se comparte una región de memoria para almacenar datos de tipos distintos. El tamaño de la unión es igual al del tipo de datos más grande. La declaración de uniones es muy similar a la declaración de estructuras. Por ejemplo.

```
union prueba{
  int a;
  double b;
  char c;
};

main(){
  prueba x;
  x.a = 5;
  printf("a= %d, b= %f, c= %c\n",x.a,x.b,x.c);
  x.b = 5.0;
  printf("a= %d, b= %f, c= %c\n",x.a,x.b,x.c);
  x.c = '5';
  printf("a= %d, b= %f, c= %c\n",x.a,x.b,x.c);
  getch();
}
```

La declaración de la unión prueba define una estructura que puede contener un entero, un flotante o un carácter. El espacio reservado para la unión es de 8 bytes ya que el tipo **`double`** ocupa 8 bytes, el **`int`** 4 bytes y el **`char`** 1 byte. La variable `x` declarada en `main` puede contener un valor de tipo **`int`**, **`double`** o **`char`**, es responsabilidad del programador que sea consistente con el valor que desea almacenar. El programa anterior despliega lo siguiente:

```
a= 5, b= 0.000000, c= ♣
a= 0, b= 5.000000, c=
a= 53, b= 5.000000, c= 5
```

Un ejemplo más práctico es el siguiente. Podemos almacenar la información de una persona

Dependiendo del sexo, si es hombre, deseamos guardad su estatura y peso y si es mujer sus medidas. Lo anterior podemos representarlo mediante la siguiente estructura:

```
struct fecha{
  int dia,mes,anyo;
};

struct persona{
  char nombre[20],apellido[20];
  fecha nacimiento;
  char sexo;
  union{
    struct {
      float peso,estatura;
    }varon;
    struct {
      int medidas[3];
    }hembra;
  };
};
```

El miembro `sexo` de tipo **`char`** nos permite saber que se almacena en los campos de la unión, a este miembro se le llama *discriminador*. La unión en si está formada por dos estructuras que se distinguen con un nombre: `varon` y `hembra`. La primera estructura tiene dos miembros flotantes: peso y estatura y la segunda tiene un arreglo de tres enteros para las medidas. Podemos desplegar una variable de tipo persona mediante la siguiente función `escribePersona`.

```
void escribePersona(persona p){
  printf("nombre: %s %s\n",p.nombre,p.apellido);
  printf("fecha de nacimiento: %d/%d/%d\n",
     p.nacimiento.dia,p.nacimiento.mes,p.nacimiento.anyo);
  if(p.sexo=='H'){
    printf("sexo: masculino\n");
    printf("peso: %.1f, estatura: %.1f\n",
             p.varon.peso,p.varon.estatura);
  }
  else{
    printf("sexo: femenino\n");
    printf("medidas: %d, %d, %d\n",p.hembra.medidas[0],
        p.hembra.medidas[1],p.hembra.medidas[2]);
  }
}
```

La siguiente función `main` inicializa dos estructuras del tipo persona, una con sexo masculino y otra con sexo femenino. Solo puede iniciarse el primer miembro de una unión,

de tal manera que no es posible poner valores para los tres miembros `medidas`. El despliegue de esta función se muestra a continuación.

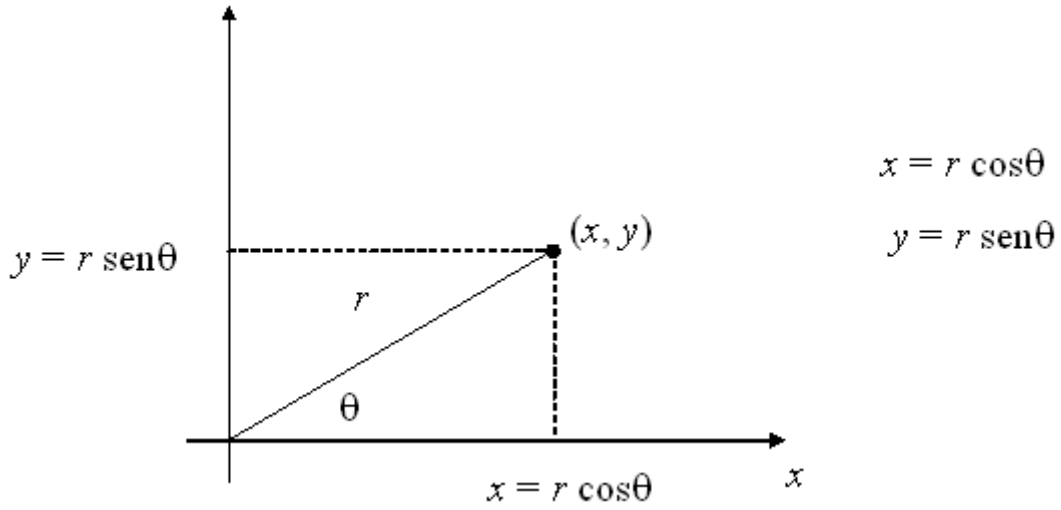
```
main(){
  persona a = {"Juan","Perez",{3,4,1980},'H',80,1.83},
          b = {"Luisa","Lane",{16,7,1990},'M',90,60};
  escribePersona(a);
  escribePersona(b);
  b.hembra.medidas[0]=90;
  b.hembra.medidas[1]=60;
  b.hembra.medidas[2]=90;
  escribePersona(b);
  getch();
}
```

Despliegue:

```
nombre: Juan Perez
fecha de nacimiento: 3/4/1980
sexo: masculino
peso: 80.0, estatura: 1.8
nombre: Luisa Lane
fecha de nacimiento: 16/7/1990
sexo: femenino
medidas: 1119092736, 1114636288, 0
nombre: Luisa Lane
fecha de nacimiento: 16/7/1990
sexo: femenino
medidas: 90, 60, 90
```

Coordenadas rectangulares y polares

Un punto en el plano se puede representar en coordenadas rectangulares o polares. La figura muestra la correspondencia entre unas y otras.



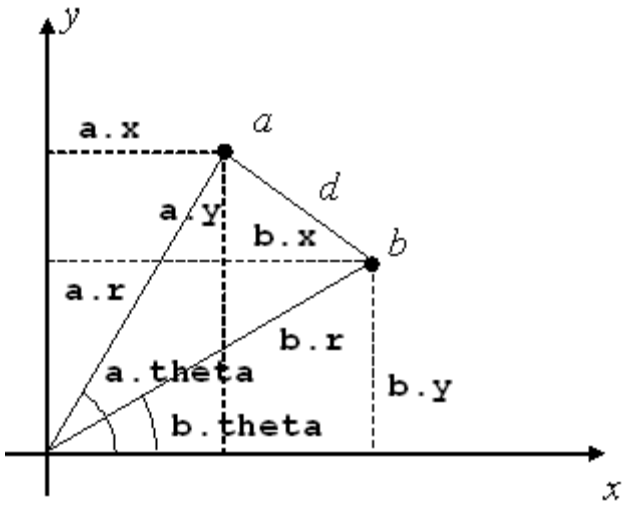
En C podemos representar un punto como una estructura formada por un miembro para reconocer el tipo de coordenada y una unión para almacenar los valores, de la siguiente manera:

```
struct coordenada{
  int tipo;
  union {
    struct{double x,y;}rect;
    struct{double r,theta;}pol;
  };
};
```

Para acceder a los miembros de un punto a utilizamos la siguiente notación.

```
coordenada a;

a.rect.x  -  componente x de a
a.rect.y  -  componente y de a
a.pol.r  -  distancia al origen de a
a.pol.theta  -  ángulo con eje x de a
```

La distancia entre dos puntos representados con la estructura anterior se puede calcular mediante el teorema de Pitágoras o haciendo uso de la ley de los cosenos. La figura muestra como hacerlo:



Con a y b en coordenadas rectangulares la distancia es: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Con a en rectangulares y b en polares la distancia es:

$d = \sqrt{(r_2 \cos\theta_2 - x_1)^2 + (r_2 \mathrm{sen}\,\theta_2 - y_1)^2}$

Con a en polares y b en rectangulares la distancia es:

$d = \sqrt{(x_2 - r_1 \cos\theta_1)^2 + (y_2 - r_1 \mathrm{sen}\,\theta_1)^2}$

Con a y b en coordenadas polares la distancia es: $d = \sqrt{r_1^2 + r_2^2 - r_1 r_2 \cos(\theta_2 - \theta_1)}$

Donde hemos puesto x_1 = `a.rect.x`, x_2 = `b.rect.x`, y_1 = `a.rect.y`, y_2 = `b.rect.y`, r_1 = `a.pol.r`, r_2 = `b.pol.r`, θ_1 = `a.pol.theta` y θ_2 = `b.pol.theta`. La siguiente función calcula la distancia entre dos puntos tomando en cuenta esta representación.

```
#define CARTESIANA 0
#define POLAR 1

double distancia(coordenada a,coordenada b){
  double d;
  switch(a.tipo){
    case CARTESIANA:switch(b.tipo){
      case CARTESIANA:
       d = sqrt((a.rect.x-b.rect.x)*(a.rect.x-b.rect.x)+
          (a.rect.y-b.rect.y)*(a.rect.y-b.rect.y));
           break;
      case POLAR:
       d = sqrt((a.rect.x-b.pol.r*cos(b.pol.theta))*
          (a.rect.x-b.pol.r*cos(b.pol.theta))+
          (a.rect.y-b.pol.r*sin(b.pol.theta))*
          (a.rect.y-b.pol.r*sin(b.pol.theta)));
          break;
      }break;
```

```
    case POLAR:switch(b.tipo){
     case CARTESIANA:
       d = sqrt((a.pol.r*sin(a.pol.theta)-b.rect.x)*
           (a.pol.r*sin(a.pol.theta)-b.rect.x)+
           (a.pol.r*cos(a.pol.theta)-b.rect.y)*
           (a.pol.r*cos(a.pol.theta)-b.rect.y));
           break;
     case POLAR:d = sqrt(a.pol.r*a.pol.r+b.pol.r*b.pol.r-
             2*a.pol.r*b.pol.r*cos(a.pol.theta-b.pol.theta));
          break;
      }
  }
  return d;
}
```

Enumeraciones

Una enumeración es un conjunto de nombres simbólicos con valores numéricos enteros. Las enumeraciones tienen por objetivo hacer más clara la escritura y comprensión de los programas. Para definir una enumeración se utiliza la palabra reservada **`enum`**. C asigna enteros consecutivos (comenzando por 0) a los valores de las etiquetas a menos que se asignen valores explícitamente. La sintaxis es la siguiente:

```
enum nombre {etiquetas};
```

El siguiente ejemplo define una enumeración para los días de la semana. C asignará 0 al identificador `lunes`, 1 a `martes`, etc.

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

enum diaSemana {lunes, martes, miercoles, jueves, viernes,
                sabado, domingo};
char dias[7][10] ={"lunes","martes","miercoles",
                   "jueves","viernes","sabado","domingo"};

main(){
  diaSemana dia;
  do{
    printf("Teclee el numero del dia (0 a 6): ");
    scanf("%d",&dia);
  }while(dia<lunes || dia>domingo);
  printf("El dia es: %s",dias[dia]);
  getch();
}
```

Las enumeraciones pueden usarse para iniciar constantes de una forma similar a la directiva define.

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

/* enum para definir constantes de tipo entero */
enum {SEC = 1, MIN = 60, HORA = 60*60, DIA = 24*60*60};

main(){
  int segundos;
  printf("Escriba numero de segundos: ");
  scanf("%d",&segundos);
  printf("%d segundos = %d minutos = %d horas"
         " = %d dias\n", segundos, segundos/MIN,
segundos/HORA, segundos/DIA);
  getch();
}
```

Otros ejemplos son los siguientes:

```
enum escapes{BELL='\a', BACKSPACE='\b', HTAB='\t',
       RETURN = '\r', NEWLINE   = '\n', VTAB = '\v' };


enum boolean { FALSE = 0, TRUE };


enum days {Jan=31, Feb=28, Mar=31,
            Apr=30, May=31, Jun=30,
            Jul=31, Aug=31, Sep=30,
            Oct=31, Nov=30, Dec=31};
```

Problemas propuestos

9.5.1. Defina una enumeración para los meses del año.

9.5.2. Utilice una enumeración para asignar valores a los colores negro, rojo, verde, amarillo, azul, magenta, cian y blanco.

9.5.3. Defina una enumeración para representar los posibles estados de un semáforo.

9.5.4. ¿Cuál es la salida del siguiente programa?

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

```
union U{
  int n;
  char s[5];
};
main(){
  U a = {5};
  printf("a.s = %s\n",a.s);
  strcpy(a.s,"hola");
  printf("a.n = %d\n",a.n);
  printf("a.n = %x\n",a.n);
  printf("a.s = %s\n",a.s);
  getch();
}
```

9.5.5. Dada la siguiente declaración de unión, ¿cuáles de las siguientes inicializaciones son válidas?

```
union u1{
  int x;
  char c;
  float r;
};
```

a) `u1 a = {34};`
b) `u1 b = {'f'};`
c) `u1 c = {4.6};`
d) `u1 d = {r = 5.6};`

9.5.6. Se desea manejar la información de libros, revistas y películas. De los libros se quiere mantener la siguiente información: código, autor, título, editorial, año; de las revistas la siguiente información: código, nombre, mes, año; de las películas la siguiente información: código, título, director, productora, año. Declare estructuras para cada elemento de información. Luego declare una unión para representar los tres tipos de elementos. Declare un arreglo llamado tabla que contenga 100 elementos del tipo unión.

9.5.7. Escriba una función para desplegar la información de un elemento de la tabla del problema anterior, ya sea libro, revista o película.

Capítulo 10. Archivos

10.1. Archivos y flujos.

Los archivos son bloques de información que tienen una duración mayor a la de los datos en memoria principal. Tradicionalmente los archivos se guardan en medios magnéticos, pero anteriormente residían en otros medios como cinta de papel perforado o tarjetas perforadas. En la actualidad se utilizan medios ópticos o electrónicos como memorias USB. El sistema operativo se encarga de manejar los diversos tipos de medios para hacerlos compatibles y transparentes al usuario.

El manejo de archivos no está ínter construido en el lenguaje C. Se provee en la biblioteca `stdio.h` con funciones que inician con "f" para la manipulación de archivos. Las funciones trabajan igual que las de entrada y salida estándar. En C los archivos se representan como secuencias de caracteres sin formato. Es fácil moverse hacia adelante para leer o escribir caracteres, pero es casi imposible moverse hacia atrás.

Antes de leer o escribir se debe asociar un flujo (`stream`) a los archivos. El flujo es una apuntador a una estructura que mantiene toda la información necesaria para leer y escribir en un archivo. Todo programa en C abre por lo menos tres flujos: `stdin`, `stdout`, `stderr`. El archivo `stdin` es la entrada estándar y esta relacionada con el teclado, `stdout` es la salida estándar y esta relacionada con la pantalla y `stderr`: es salida estándar de errores y esta relacionada con la pantalla.

Un flujo crea una región de memoria (buffer) entre el programa y el archivo en disco que tiene como propósito reducir los accesos a disco. Los caracteres se leen en bloques en el buffer, o se escriben en bloques en el disco.

El archivo `stderr` se utiliza para desplegar los errores que se produzcan en la ejecución de algún programa. Puede redireccionarse la salida del archivo `stderr`, esto es, se puede enviar a otro archivo que no sea la pantalla. Similarmente se puede redireccionar `stdout` para que la salida normal de un programa se dirija a otro archivo. Esta es la manera más simple de crear un archivo desde un programa en C. El siguiente programa envía un letrero a `stdout` y otro a `stderr`, como ambos están asignados inicialmente a la pantalla estos dos letreros aparecerán en la pantalla.

```
#include <stdio.h>
main(){
  printf("Escritura a stdout\n");
  fprintf(stderr,"Escritura a stderr\n");
  return 0;
}
```

Una forma de redireccionar la salida es usando el carácter > en la línea de comandos de MSDOS o UNIX. La sintaxis es:

```
c:>programa > salida
```

Donde "`c:>`" es el indicador de comandos, "`programa`" es el nombre del programa ejecutable y salida es el nombre del archivo en donde se almacenaran las líneas de salida del programa. Los siguientes comandos en MSDOS crearan el archivo "`archivo.txt`". Los caracteres en itálica son tecleados por el usuario.

```
c:> prog
Escritura a stdout
Escritura a stderr
c:> prog > archivo.txt
Escritura a stderr
c:> type archivo.txt
Escritura a stdout
```

La entrada estándar es un archivo de caracteres basado en líneas. Esto quiere decir que los caracteres tecleados son retenidos (buffered) hasta presionar ENTER. El siguiente ejemplo lee la entrada estándar y la muestra carácter por carácter hasta presionar ctrl-z (en MSDOS). Note que cada línea se desplegará solo al presionar la tecla ENTER.

```
#include <stdio.h>
main(){
  int ch;
  while((ch=getchar())!= EOF)
    printf("lee '%c'\n",ch);
  printf("EOF\n");
  return 0;
}
```

La entrada estándar también puede redirecionarse con el carácter <. El siguiente ejemplo lee datos desde el teclado. El programa calcula la suma y el producto de dos matrices y fue revisado en la sección 7.5. Los datos para matrices grandes es mejor almacenarlos en un archivo para evitar errores de tecleo. Suponga que deseamos sumar y multiplicar las siguientes matrices:

$$A=\begin{bmatrix}2 & 5 & 3\\ 6 & 0 & 8\\ 1 & 4 & 6\end{bmatrix} \; y \; B=\begin{bmatrix}5 & 3 & 0\\ 7 & -5 & 8\\ 7 & -4 & -9\end{bmatrix}$$

Podemos escribir en un archivo de texto (digamos "`dat.txt`") lo siguiente y introducirlo mediante el redireccionamiento al programa:

```
3
3
2 5 3
```

```
6 0 8
1 4 6

5 3 0
7 -5 8
7 -4 -9
```

El resultado es el siguiente. El resultado muestra la salida del programa que contiene las líneas de salida solicitando los datos de entrada, pero estos son tomados desde el archivo dat.txt.

```
C:>mat < dat.txt
Teclee num. de renglones y columnas: Teclee datos para la
matriz A.Teclee datos
para la matriz B.2 5 3
6 0 8
1 4 6

5 3 0
7 -5 8
7 -4 -9

7 8 3
13 -5 16
8 0 -3

66 -31 13
86 -14 -72
75 -41 -22
```

Se puede redireccionar la entrada y la salida el mismo tiempo. En el ejemplo anterior podemos guardar los resultados en un archivo "`sal.txt`" escribiendo:

```
c:>mat < dat.txt > sal.txt
```

Problemas propuestos

10.1.1. Un programa llamado "mcd" lee parejas de números separados por un espacio hasta que se teclee una pareja de ceros y genera como salida el máximo común divisor de estos números. Escriba una sentencia que permita introducir datos desde un archivo al programa mcd y mandar la salida a otro archivo.

10.1.2. El siguiente programa resuelve ecuaciones cuadráticas hasta que el usuario escriba un 0 para el coeficiente cuadrático. Escriba un archivo texto con los datos necesarios para resolver las siguientes ecuaciones: $3x^2 + 5x + 4 = 0$, $8x^2 + 7x + 1 = 0$, $x^2 - 9x + 7 = 0$, $9x^2 + 2x – 5 = 0$, $-x^2 + x + 1 = 0$, $x^2 - x + 8 = 0$, $x^2 + 12x + 40 = 0$, $37x^2 + 25x + 412 = 0$, $97x^2 - 3x$

+ 4 = 0, $-0.23x^2 + 0.65x + 4.7 = 0$. Escriba un comando en MSDOS para resolver las ecuaciones con este programa y guardar las soluciones en otro archivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
main(){
  float a,b,c,d,x1,x2,re,im;
  do{
    scanf(" %f,%f,%f",&a,&b,&c);
    if(a==0)
      exit(0);
    d = b*b-4*a*c;
    printf("a = %.4f b = %.4f c = %.4f\n",a,b,c);
    if(d>=0){
      x1 = (-b-sqrt(d))/2/a;
      x2 = (-b+sqrt(d))/2/a;
      printf("x1 = %.4f x2 = %.4f\n",x1,x2);
    }else{
      im = sqrt(-d)/2/a;
      re = -b/2/a;
      printf("re = %.4f im = %.4f\n",re,im);
    }
  }while(1);
}
```

10.2. Operaciones básicas en archivos.

Las operaciones básicas con archivos pueden resumirse de la siguiente manera: abrir el archivo, leer y/o escribir en el archivo, cerrar el archivo.

Antes de poder utilizar un archivo es necesario informarle al sistema operativo que se va a hacer uso de el. El sistema operativo establece una región de memoria en donde se almacenan los datos que serán leídos o escritos en el archivo. Esto se hace en la operación de apertura del archivo. La función `fopen` es la encargada de abrir un archivo. Es necesario primero crear una variable para que señale a la región de memoria utilizada como buffer. El tipo de la variable es un apuntador a `FILE`. O sea:

```
FILE *f;
```

Una vez definida la variable se abre el archivo con la función `fopen`, si la operación tiene éxito la función `fopen` regresa un apuntador al buffer del archivo, sino regresa un apuntador `NULL`. La función `fopen` lleva como parámetros una cadena de caracteres especificando la ruta al archivo y una cadena que especifica si se leerá o escribirá, o ambas, en el archivo.

Una vez abierto, puede utilizarse el archivo mediante las funciones `fgetc`, `fgets` y `fscanf` para leer o `fputc`, `fputs` y `fspinrf` para escribir en el archivo.

Después de leer o escribir en un archivo es conveniente cerrarlo. Esta operación tiene como propósito vaciar el buffer del archivo y liberar la memoria utilizada para el buffer. La función que cierra un archivo es `fclose`. Es aconsejable cerrar un archivo abierto pero si no se cierra el compilador cerrara el archivo al terminar la ejecución normal del programa.

El siguiente programa lee un archivo carácter por carácter y lo despliega en pantalla. El nombre del archivo se le solicita al usuario. Note que el archivo se abre con la opción "r" que indica que se abrirá para solo lectura.

```
#include <stdio.h>
#include <conio.h>
int main(){
  char nombre[80];
  FILE *entrada;
  int ch;
  printf("Archivo: ");
  scanf("%79s",nombre);
  if((entrada=fopen(nombre,"r"))==NULL){
    fprintf(stderr,"fallo la lectura de '%s' ",nombre);
    perror("debido a");
    getch();
    return 1;
  }
  while((ch=fgetc(entrada))!= EOF)
    putchar(ch);
  fclose(entrada);
  getch();
  return 0;
}
```

El programa anterior verifica que `fopen` regrese un valor igual a `NULL`, si esto ocurre se llama a la función `perror` que imprime el letrero que se pasa como parámetro más el mensaje del error responsable de la ofensa. Note que el programa en este caso regresa 1 al sistema operativo para indicar que hubo error. Si no hay error se lee el archivo carácter por carácter con la función `fgetc` y se despliega cada carácter con `putchar`. Siempre es importante verificar si se presenta algún error en la creación, lectura o escritura de archivos para evitar sorpresas.

El siguiente ejemplo crea un archivo y escribe todas las líneas que escriba el usuario hasta que escriba una línea vacía. En este caso el archivo se crea con la opción "w" que habilita el archivo solo para escritura, si el archivo existe, se borra su contenido y se crea un archivo vacío. Las cadenas se leen mediante `gets` y se escriben en el archivo con `fputs`. Muestre

el archivo `test2.txt` y verifique que `fputs` no escribe el carácter nulo de cada cadena ni agrega `'\n'` al final de las mismas.

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

main(){
  char cadena[30];
  FILE *archivo;
  archivo = fopen("test2.txt","w");
  if(archivo!=NULL){
    do{
      printf("Escriba una cadena: ");
      gets(cadena);
      if(cadena[0]!='\0')
        fputs(cadena,archivo);
    }while(cadena[0]!='\0');
    fclose(archivo);
  }
  else
    printf("No se puede crear el archivo...");
  getch();
}
```

El siguiente programa lee cadenas de 30 caracteres desde el archivo creado por el programa anterior. La lectura se hace con la función `fgets` que lee una cadena del tamaño especificado en el segundo parámetro. El fin de un archivo se puede verificar utilizando la función `feof` que regresa 1 si se ha llegado al final del archivo o 0 en cualquier otro caso. Recuerde que `puts` si agrega un '\n' al final de cada cadena.

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

main(){
  char cadena[30];
  FILE *archivo;
  archivo = fopen("test2.txt","r");
  if(archivo!=NULL){
    while(!feof(archivo)){
      fgets(cadena,30,archivo);
      puts(cadena);
    }
    fclose(archivo);
  }
  else
    printf("No se puede leer el archivo...");
  getch();
```

```
}
```

La siguiente tabla resume las diferentes formas de acceder a un archivo mediante la función `fopen`.

Tipo de archivo	Significado
“r”	Abre un archivo existente solo para lectura
“w”	Abre un archivo existente solo para escritura. Si el archivo existe, lo borra y crea uno nuevo vacío.
“a”	Abre un archivo para agregar datos al final. Si no existe crea un archivo nuevo.
“r+”	Abre un archivo existente tanto para lectura como escritura.
“w+”	Abre un archivo existente tanto para lectura como escritura. Si el archivo existe, crea uno nuevo vacío.
“a+”	Abre un archivo para leer y agregar datos al final. Si no existe crea un archivo nuevo.
“t”	Archivo de tipo texto. Solo un conjunto limitado de caracteres ASCII. Esta es la opción por omisión.
“b”	Archivo de tipo binario. Se permite cualquier carácter.

Como ejemplo veamos un programa que convierte texto en español a “pig latín”. Una palabra en español se convierte a pig latín cambiando la letra inicial al final y agregando una “a” al final. Por ejemplo: casa -> asca, lapiz -> apizla, etc. El programa leerá el nombre de un archivo de texto, leerá el archivo y escribirá en un segundo archivo las líneas de texto con las palabras convertidas a pig latín.

El primer paso en la conversión es la eliminación de todos los signos de puntuación de cada una de las líneas de texto. La función encargada de este proceso es limpia. Esta función no regresa ningún valor y lleva como argumentos la línea de entrada y regresa como segundo parámetro la línea sin signos de puntuación. El proceso es muy simple, cada carácter que sea alfabético o el carácter espacio es copiado a la cadena de salida, los demás caracteres son ignorados.

```
void limpia(char texto[], char cad[]){
  int i,j=0;
  for(i=0;i<strlen(texto);i++)
    if(isalpha(texto[i])||texto[i]==' ')
      cad[j++] = texto[i];
  cad[j] = '\0';
}
```

El proceso de conversión lo realiza la función `textApiglatin`.

```
void textApiglatin(char text[], char pig[]){
  int p1 = 0,p2,i;
  char pal[20] = "";
```

```
  strcat(text," ");
  pig[0] = '\0';
  while(p1<strlen(text)){
    i = p1;
    while(text[i]!=' ')i++;
    p2 = i;
    //copia palabra desde 2a letra
    for(i=p1;i<p2;i++)
      pal[i-p1] = text[i+1];
    //copia inicial al final
    pal[i-p1-1] = text[p1];
    //agrega una 'a'
    pal[i-p1] = 'a';
    pal[i-p1+1] = '\0';
    strcat(pig,pal);
    strcat(pig," ");
    p1 = p2+1;
  }
}
```

Por último, la función procesa leer el nombre del archivo a procesar y si el archivo existe, crea un archivo llamado “pig.txt” en el cual se escribirá la salida. La función lee línea por línea y las somete al proceso de limpieza y conversión escribiendo el resultado en el archivo de salida. La función main del programa solo llama a esta función.

```
void procesa(){
  char nom[30],r[80],s[80],t[80];
  FILE *f,*g;
  printf("Archivo de entrada: ");
  scanf("%s",nom);
  f = fopen(nom,"r");
  if(f==NULL){
    printf("Archivo %s no encontrado...",nom);
    getch();
    exit(1);
  }
  g = fopen("pig.txt","w");
  while(!feof(f)){
    fgets(r,80,f);
    limpia(r,s);
    textApiglatin(s,t);
    fprintf(g,"%s\n",t);
  }
  fclose(f);
  fclose(g);
}
```

Problemas propuestos

10.2.1. Escriba una sentencia para abrir un archivo llamado “datos.txt” para lectura y escritura sin crear un archivo vacío.

Explique el significado de las siguientes sentencias:

a) `f = fopen(“datos.txt”,”w+”);`
b) `f = fopen(“datos.txt”,”a+”);`
c) `f = fopen(“datos.txt”,”r”);`
d) `f = fopen(“datos.txt”,”w”);`

10.2.2. Escriba un programa que copie un archive de texto cambiando todas las letras minúsculas en mayúsculas.

10.2.3. Escriba un programa que lea un archivo texto y cuente el número de cada una de las letras del alfabeto que contiene.

10.2.4. Escriba un programa que escriba en un archivo de nombre “datos.txt” su nombre, dirección, teléfono y la carrera que está cursando. Cada elemento escríbalo en una línea distinta.

10.2.5. Escriba un programa que lea un archivo de un programa en C y genere otro archivo del mismo programa pero con todos los comentarios eliminados.

10.2.6. Escriba un programa que lea un archivo de un programa en C y cuente el número de líneas de comentarios que contiene y el número total de líneas del programa.

10.3. Uso de archivos secuenciales.

Existen dos tipos de archivos de datos, archivos secuenciales y archivos de acceso aleatorio. Los archivos secuenciales se subdividen en archivos de texto, que están formados por secuencias de caracteres, las cuales pueden ser interpretadas como cadenas de caracteres o como números. En los archivos de texto solo se permite un conjunto restringido de caracteres ASCII. La segunda categoría de archivos secuenciales son llamados archivos sin formato que se organizan como secuencias de bloques de información. Estos bloques representan estructuras o arreglos.

Por último están los archivos binarios que contienen todos los valores binarios posibles. Estos archivos son útiles para guardar información numérica, imágenes, sonidos, etc. Los archivos aleatorios son aquellos en los que se puede acceder a cualquier parte de ellos para lectura o escritura. Pueden existir combinaciones de los diferentes tipos de archivo, esto es, puede haber archivos de texto donde se almacene información binaria, etc.

El siguiente programa escribe una tabla de números de 1 a 20 y sus cuadrados. El archivo creado es de tipo texto y puede ser editado con cualquier editor de texto, tal como el editor

de programas de Dev-C. Se utiliza la función `fprintf` para escribir datos numéricos con formato.

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

main(){
  int i;
  FILE *archivo;
  archivo = fopen("test4.txt","w");
  if(archivo!=NULL){
    for(i = 0; i< 10; i++)
      fprintf(archivo,"%d %d\n",i,i*i);
    fclose(archivo);
  }
  else
    printf("No se puede crear el archivo...");
  getch();
}
```

El siguiente programa lee el archivo creado por el programa anterior utilizando la función `fscanf`. Debe tomarse en cuenta que los datos están separados por espacios para leerlos correctamente.

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

main(){
  int i,j;
  FILE *archivo;
  archivo = fopen("test4.txt","r");
  if(archivo!=NULL){
    for(i = 0; i< 10; i++){
      fscanf(archivo,"%d %d\n",&i,&j);
      printf("%d %d\n",i,j);
    }
    fclose(archivo);
  }
  else
    printf("No se puede crear el archivo...");
  getch();
}
```

El siguiente ejemplo abre un archivo y escribe una línea y cierra el archivo. A continuación agrega una línea al final del archivo abriéndolo con la opción "`a`" y se cierra el archivo. Finalmente se abre con la opción "`a+`" que permite leer el archivo y agregar al final.

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

main(){
  FILE *archivo;
  archivo = fopen("testApend.txt","w");
  fprintf(archivo,"Esta es una prueba de append\n");
  fclose(archivo);
  archivo = fopen("testApend.txt","a");
  fprintf(archivo,"Esta linea se agrego despues.\n");
  fclose(archivo);
  archivo = fopen("testApend.txt","a+");
  while(!feof(archivo))
    putchar(fgetc(archivo));
  getch();
  fprintf(archivo,"Esta linea se agrego al final.\n");
  fclose(archivo);
}
```

Veamos un ejemplo más práctico. Se desea un programa para registrar las temperaturas máximas y mínimas durante un cierto número de días. Lo conveniente es utilizar un archivo para ir almacenando los datos. Supondremos que antes de ejecutar por primera vez el programa el archivo de datos no existe, esto implica que debemos crearlo para agregar datos posteriormente. Escribiremos una función que se encargue de verificar si existe el archivo y si no existe lo creará. La función es la siguiente.

```
void inicia(){
  FILE *archivo;
  archivo = fopen("temps.txt","r");
  if(archivo==NULL){
    archivo = fopen("temps.txt","w");
  }
  fclose(archivo);
}
```

Esta función será llamada al inicio del programa. El programa en si consta de un menú de opciones: capturar temperaturas, ver temperaturas, datos estadísticos y terminar. La función `menu` es la encargada de leer la opción de menú y llama a las funciones restantes para cada opción. Esta función se ejecutará hasta que el usuario seleccione la opción de salir.

```
void menu(){
  char ch;
  do{
    despliegaMenu();
    do{
      ch = getch();
    }while(ch<'1'||ch>'4');
```

```
    printf("%c",ch);
    switch(ch){
      case '1':leerTemperaturas();break;
      case '2':verTemperaturas();break;
      case '3':estadisticas();break;
    }
  }while(ch!='4');
}
```

La función `despliegaMenu` simplemente borra la pantalla y despliega las opciones del menú.

```
void despliegaMenu(){
  clear();
  printf("1. Capturar temperaturas\n\n");
  printf("2. Ver temperaturas\n\n");
  printf("3. Estadisticas\n\n");
  printf("4. Terminar\n\n");
  printf("Opcion: ");
}
```

La función `leerTemperaturas` es la encargada de capturar la fecha y las temperaturas máxima y mínima de esa fecha. El archivo se abre con la opción “a” para agregar los datos al final del mismo.

```
void leerTemperaturas(){
  int dia,mes,anyo;
  float tMax,tMin;
  FILE *archivo;
  clear();
  printf("Introduzca fecha (dd mm aa): ");
  scanf("%d %d %d",&dia,&mes,&anyo);
  printf("Introduzca temperatura maxima: ");
  scanf("%f",&tMax);
  printf("Introduzca temperatura minima: ");
  scanf("%f",&tMin);
  archivo = fopen("temps.txt","a+");
  fprintf(archivo,"%d %d %d %.2f %.2f\n", dia, mes, anyo,
tMax, tMin);
  fclose(archivo);
}
-
```

La función `verTemperaturas` muestra todas las temperaturas capturadas en grupos de 20. Esta función abre el archivo con la opción “r”.

```
void verTemperaturas(){
  int dia,mes,anyo,lineas=0;
```

```
  float tMax,tMin;
  FILE *archivo;
  archivo = fopen("temps.txt","r");
  clear();
    fscanf(archivo,"%d  %d  %d  %f  %f\n",  &dia,  &mes,  &anyo,
&tMax, &tMin);
  while(!feof(archivo)){
    if(tMax<60)
        printf("Fecha %2d/%2d/%4d: Temp. max %.2f, Temp. min
%.2f\n",dia,mes,anyo,tMax,tMin);
    if(lineas==20){
      lineas = 0;
      printf("Presione una tecla...");
      getch();
      clear();
    }
    else
      lineas++;
     fscanf(archivo,"%d %d %d %f %f\n", &dia, &mes, &anyo,
&tMax, &tMin);
  };
  printf("Presione una tecla...");
  getch();
  fclose(archivo);
}
```

Por último la función `estadisticas` abre el archivo también en la opción “r” y lee todos los datos sumando todas las temperaturas capturadas y contando todas las líneas del archivo para calcular el promedio. También se registra cual es la temperatura máxima y mínima absolutas de todos los datos leídos.

```
void estadisticas(){
  int dia,mes,anyo,lineas=0;
  float tMax,tMin,maxima=-100,minima=100,promedio = 0;
  FILE *archivo;
  archivo = fopen("temps.txt","r");
  clear();
    fscanf(archivo,"%d  %d  %d  %f  %f\n",  &dia,  &mes,  &anyo,
&tMax, &tMin);
  while(!feof(archivo)){
    if(tMax<60){
      if(tMax>maxima) maxima = tMax;
      if(tMin<minima) minima = tMin;
      promedio += tMax+tMin;
    }
    lineas++;
```

```
      fscanf(archivo,"%d %d %d %f %f\n", &dia, &mes, &anyo,
&tMax, &tMin);
  };
  if(lineas)
    printf("Temp. max: %.2f\nTemp. min: %.2f\n"
      "Promedio: %.2f\n\n\n",maxima,minima,promedio/lineas);
  printf("Presione una tecla...");
  getch();
  fclose(archivo);
}
```

La función `main` solo llama a la función que inicializa el archivo y al menú.

```
main(){
  inicia();
  menu();
}
```

Veamos un ejemplo de archivos secuenciales y estructuras. Las carreras de fórmula otorgan puntos a los pilotos de acuerdo con el lugar que ocupas en la llegada. La tabla de puntos es la siguiente:

Lugar	puntos
1º	25
2º	18
3º	15
4º	12
5º	10
6º	8
7º	6
8º	4
9º	2
10º	1

Los datos de los pilotos se almacenan en un archivo de estructuras. Cada estructura consta del nombre del piloto, la escudería, la nacionalidad y el número de puntos. La estructura en C es la siguente:

```
struct piloto{
  char nombre[20];
  char nacionalidad[4];
  int puntos;
};
```

Desarrollemos un programa para actualizar los datos después de cada carrera. Para esto mantendremos un archivo con los datos de los pilotos. Inicialmente crearemos el archivo

con un editor de texto. La información de cada piloto estará contenida en cuatro líneas de texto. El archivo se leerá hasta llegar al final de archivo.

El programa realizará los siguientes pasos:

1. Leer archivo de datos de todos los pilotos y almacenarlo en un arreglo.
2. Leer los primeros diez lugares y actualizar el arreglo de pilotos.
3. Ordenar el arreglo de pilotos.
4. Exhibir la nueva lista de pilotos ordenada por puntaje.
5. Escribir el arreglo en el archivo con los puntos acualizados.

La función `leerDatos` lee los datos de los pilotos y los almacena en un arreglo que pasa como argumento. La función tiene un parámetro por referencia que retiene el número de pilotos. El programa termina si no se encuentra el archivo de datos. Observe que la lectura del último dato (el número de puntos) incluye la lectura de un ENTER para asegurar que se llegue al final del archivo cuando se lean los datos del último piloto.

```
void leerDatos(piloto lideres[],int *n){
  int i = 0;
  FILE *f;
  f = fopen("formula1.txt","r");
  if(f==NULL){
    printf("No existe el archivo 'formula1.txt'");
    getch();
    exit(1);
  }
  while(!feof(f)){
    fscanf(f," %[^\n]",lideres[i].nombre);
    fscanf(f," %[^\n]",lideres[i].nacionalidad);
    fscanf(f," %d\n",&lideres[i].puntos);
    i++;
  }
  *n = i;
  fclose(f);
}
```

La función `listaDatos` despliega una lista de los pilotos ordenada por puntos.

```
void listaDatos(piloto lideres[],int n){
  int i;
  printf("FORMULA 1\n");
  printf(" # Nombre                    pts"
  "     # Nombre                   pts\n");
  for(i=0;i<n;i++){
    printf("%2i %-20s %3s %3d      ",i+1,lideres[i].nombre,
      lideres[i].nacionalidad,lideres[i].puntos);
    if(i%2==1)
```

```
      printf("\n");
  }
}
```

La lectura de las posiciones de una carrera la efectúa la función `leerPrimeros`. La función lee el número de cada piloto (de acuerdo a la lista ordenada) en el orden en que quedó en la carrera. No se hace ningún manejo de errores, tal como detectar números repetidos o número fuera de rango. Los números de los pilotos se guardan en el arreglo lugar. Posteriormente se actualizan los puntos de cada piloto.

```
void leerPrimeros(piloto lideres[]){
  int i,lugar[10];
  int puntos[10] = {25,18,15,12,10,8,6,4,2,1};
   printf("Escriba los numeros de los 10 pilotos en el orden
en que\n");
  printf("llegaron en la ultima carrera: ");
  for(i=0;i<10;i++)
    scanf(" %d ",&lugar[i]);
  for(i=0;i<10;i++)
    lideres[lugar[i]-1].puntos += puntos[i];
}
```

Posteriormente la función `ordenar` ordena la lista de pilotos con base en el número de puntos.

```
void ordenar(piloto lideres[],int n){
  int i,j;
  piloto temp;
  for(i=0;i<n-1;i++)
    for(j=i;j<n;j++)
      if(lideres[i].puntos<lideres[j].puntos){
        temp = lideres[i];
        lideres[i] = lideres[j];
        lideres[j] = temp;
      }
}
```

Por último la función `escribirDatos` escribe el archivo borrando la versión anterior.

```
void escribirDatos(piloto lideres[],int n){
  int i;
  FILE *f;
  f = fopen("formula1.txt","w");
  for(i=0;i<n;i++){
    fprintf(f,"%s\n",lideres[i].nombre);
    fprintf(f,"%s\n",lideres[i].nacionalidad);
    fprintf(f,"%d\n",lideres[i].puntos);
```

```
  }
  fclose(f);
}
```

La función `main` llama a las funciones anteriores en el orden adecuado, de acuerdo con el algoritmo descrito anteriormente.

```
main(){
  piloto lideres[24];
  int max;
  leerDatos(lideres,&max);
  listaDatos(lideres,max);
  leerPrimeros(lideres);
  system("cls");
  ordenar(lideres,max);
  listaDatos(lideres,max);
  lescribirDatos(lideres,max);
  getch();
}
```

Problemas propuestos

10.3.1. Escriba un programa que lea el nombre de un archivo y lo cree para escritura, luego solicite al usuario el número de valores que desea guardar en el archivo, después solicite al usuario los valores y los almacene en el archivo. Luego cierre el archivo y ábralo para lectura, lea todos los valores y calcule la suma total y despliegue la suma.

10.3.2. Modifique la función `leerPrimeros` del programa de fórmula 1 para que verifique si se introduce algún número de piloto repetido o fuera de rango.

10.3.3. Desarrolle un programa para llevar los resultados de la liga mexicana de fútbol. La liga consta de 18 equipos organizados en dos grupos y juegan entre si una vez por temporada.

10.3.4. Un banco emplea un archivo para almacenar los detalles de las cuentas corrientes de sus clientes. Cada registro contiene el número de cuenta, el nombre, la dirección, el saldo actual y el límite de saldo negativo (cero si no se permiten saldos negativos). Desarrolle un programa que escudriñe este archivo y cree uno nuevo con los registros de los clientes que tengan un saldo negativo. Así mismo, se deben exhibir en la pantalla loc detalles de cualquier cliente que rebase en más del 10% su límite acordado de saldo negativo, o que presente saldo negativo si no está permitido tenerlo en su caso particular.

10.3.5. Dos compañías de tarjetas de crédito se fusionaron recientemente y necesitan combinar sus registros computarizados de clientes. Para hacerlo, cada compañía a preparado un archivo con el número de cuenta, nombre, dirección y saldo actual de cada uno de sus clientes, almacenado en orden por número de cuenta. Escriba un programa para

fusionar estos dos archivos y producir un archivo único nuevo que contenga todos los datos ordenados según el mismo criterio.

10.3.6. La liga escocesa de fútbol de primera división consta de 12 equipos que juegan entre sí cuatro veces cada temporada. Desarrolle un programa que pueda aceptar los resultados (es decir, equipos y marcadores) de una serie semanal de seis encuentros y que actualice la tabla de la liga que se almacena en un archivo de datos entre cada ejecución del programa.

10.4. Archivos binarios

Los archivos binarios pueden contener bytes con cualquier valor, de 0 a 255. Los archivos ejecutables son un ejemplo común de archivos binarios. Otros ejemplos son archivos que contienen información de imágenes, tales como los archivos GIF, TGA, JPG, BMP, etc. Estos archivos pueden ser leídos si se conoce el formato gráfico. Los archivos binarios pueden contener datos numéricos. Los archivos binarios pueden desplegarse en pantalla o en un editor de texto, sin embargo en estos casos habrá algunos caracteres que no son imprimibles y el resultado puede ser inesperado.

El siguiente programa permite desplegar cualquier archivo binario o de texto. Note que la función `main` lleva dos argumentos, esto permite se introduzcan argumentos al ejecutar el programa. El primer argumento de la función `main` es `argc`, es el número de cadenas de caracteres que contiene la línea de comandos. Las cadenas se almacenan en el segundo argumento `argv` que es un arreglo de cadenas. Las cadenas se numeran de 0 en adelante, siendo el argumento 0 el nombre del programa. Si el número de argumentos es menor que dos, el programa termina informando la forma de utilizarlo.

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

void muestra(char *nom);

main(char argc,char **argv){
  if(argc<2){
    printf("uso: ver archivo\n");
    getch();
    exit(1);
  }else{
    muestra(argv[1]);
  }
}

void muestra(char *nom){
  FILE *f;
```

```
  unsigned char ch;
  char a[16] = {0};
  int i=0,j=0,k;
  f = fopen(nom,"r");
  if(f==NULL){
    printf("NO existe el archivo: %s",nom);
    getch();
    exit(1);
  }else{
    ch = fgetc(f);
    while(!feof(f)){
      printf("%02X ",ch);
      a[i] = ch;
      i++;
      if(i==16){
        j++;
        for(k=0;k<16;k++)
          if(isprint(a[k]))
            printf("%c",a[k]);
          else
            printf(".");
        i = 0;
        printf("\n");
      }
      if(j==24){
        printf("Presione cualquier tecla...");
        getch();
        printf("\n");
        j = 0;
      }
      ch = fgetc(f);
    }
  }
  j = i;
  while(j<16){
    printf("   ");
    j++;
  }
  for(k=0;k<i;k++)
    if(isprint(a[k]))
      printf("%c",a[k]);
    else
      printf(".");
  printf("\nPresione cualquier tecla...");
  getch();
}
```

La función `muestra` es la encargada de leer y desplegar el archivo. Los archivos son desplegados en renglones de 16 bytes en formato hexadecimal y en su equivalente ASCII. La sentencia `printf("%02X ",ch)` despliega el formato hexadecimal, el 0 se utiliza para rellenar el campo de 0's en lugar de usar espacios. Se utiliza la función `isprint` para desplegar solo los caracteres imprimibles. Con este programa se puede explorar cualquier tipo de archivo.

Como ejemplo de archivo binario considere el siguiente programa que escribe los cuadrados de los números de 1 a 20. La escritura se lleva a cabo mediante fwrite que escribe datos sin formato. Los parámetros de fwrite son la dirección de la variable que se escribe, el tamaño de cada dato, el número de datos y el apuntador al archivo.

```
#include <stdio.h>

main(){
  int i,t[20];
  FILE *f;
  f = fopen("test.dat","w");
  for( i=1; i<=20; i++ )
    t[i-1] = i*i;
  fwrite(t, sizeof(int), 20, f);
  fclose(f);
}
```

Los datos escritos por el programa anterior pueden ser visualizados con el programa de despliegue de archivos visto al inicio de esta sección. Se deberá ver algo similar a lo siguiente:

```
01 00 00 00 04 00 00 00 09 00 00 00 10 00 00 00 ................
19 00 00 00 24 00 00 00 31 00 00 00 40 00 00 00 ....$...1...@...
51 00 00 00 64 00 00 00 79 00 00 00 90 00 00 00 Q...d...y...É...
A9 00 00 00 C4 00 00 00 E1 00 00 00 00 01 00 00 ....—...........
21 01 00 00 44 01 00 00 69 01 00 00 90 01 00 00 !...D...i...É...

Presione cualquier tecla...
```

Note que los enteros se almacenan en orden inverso, primero los bits menos significativos luego los más significativos, además cada entero ocupa 4 bytes. El siguiente programa lee los datos en binario y los despliega en pantalla.

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

main(){
  int i,t[20];
  FILE *f;
  f = fopen("test.dat","r");
```

```
  fread(t, sizeof(int), 20, f);
  for( i=0; i<20; i++ )
    printf("%d\n",t[i]);
  fclose(f);
  getch();
}
```

Problemas propuestos

10.4.1. Modifique el programa de despliegue de archivos para poder navegar en el archivo. Use las teclas de flecha arriba y abajo para avanzar y retroceder 16 bytes, las teclas “RePag” y “AvPag” para avanzar 20 renglones, la tecla “Inicio” para ir al inicio del archivo, la tecla “Fin” para ir al final y la tecla “Esc” para terminar.

Utilice archivos binarios para los siguientes programas.

10.4.2. Un banco emplea un archivo para almacenar los detalles de las cuentas corrientes de sus clientes. Cada registro contiene el número de cuenta, el nombre, la dirección, el saldo actual y el límite de saldo negativo (cero si no se permiten saldos negativos). Desarrolle un programa que escudriñe este archivo y cree uno nuevo con los registros de los clientes que tengan un saldo negativo. Así mismo, se deben exhibir en la pantalla loc detalles de cualquier cliente que rebase en más del 10% su límite acordado de saldo negativo, o que presente saldo negativo si no está permitido tenerlo en su caso particular.

10.4.3. Dos compañías de tarjetas de crédito se fusionaron recientemente y necesitan combinar sus registros computarizados de clientes. Para hacerlo, cada compañía a preparado un archivo con el número de cuenta, nombre, dirección y saldo actual de cada uno de sus clientes, almacenado en orden por número de cuenta. Escriba un programa para fusionar estos dos archivos y producir un archivo único nuevo que contenga todos los datos ordenados según el mismo criterio.

10.4.4. La liga escocesa de fútbol de primera división consta de 12 equipos que juegan entre sí cuatro veces cada temporada. Desarrolle un programa que pueda aceptar los resultados (es decir, equipos y marcadores) de una serie semanal de seis encuentros y que actualice la tabla de la liga que se almacena en un archivo de datos entre cada ejecución del programa.

10.5. Uso de archivos aleatorios.

En los archivos aleatorios se puede acceder cualquier parte del archivo directamente. A diferencia de los archivos de texto los archivos aleatorios deben estar formados por bloques del mismo tamaño, a cada bloque se le llama registro. Se utiliza las funciones `fwrite` y `fread` para escribir y leer. Otras funciones para manejo de archivos aleatorios son `fseek`, y `ftell`. Más adelante veremos como utilizar estas funciones. La siguiente tabla resume el propósito de estas funciones.

Función	Propósito
`fwrite(void *,size_t tam,size_t num,FILE *f)`	Escribe un número de bloques `num`, de tamaño `tam`, en un archivo especificado por `f`. El primer argumento señala a la variable donde se inician los bloques.
`fread(void *,size_t tam,size_t num,FILE *f)`	Lee un número de bloques `num`, de tamaño `tam`, de un archivo especificado por `f`. El primer argumento señala a la variable donde se almacenarán los bloques.
`fseek(FILE *f,long int desp,int org)`	Posiciona el apuntador de archive del archive f en la dirección `org+desp`.
`ftell(FILE *f)`	Regresa la posición del apuntador del archivo `f`.

Considere la siguiente estructura para datos de una pequeña biblioteca:

```
struct libro{
  char titulo[30];
  char autor[30];
  char editorial[15];
  int edicion;
  int anyo;
};
```

Revisaremos un ejemplo simple que permita hacer las operaciones básicas sobre un archivo de datos de una pequeña biblioteca. El programa estará basado en menú con las opciones de agregar un libro, borrar un libro, editar datos de un libro, listar los libros existentes y salir de la aplicación. Este ejemplo no pretende mostrar el diseño de una base de datos, sino solo mostrar como utilizar las operaciones básicas con archivos aleatorios.

La primera función que veremos es la que crea el archivo de datos, si este no existe. La función lleva como argumentos un apuntador a apuntador de archivo. La función modifica el apuntador de archivo, por lo tanto debe pasar por referencia. El otro argumento es un apuntador a entero que, el cual almacenará el número de registros en el archivo.

```
void iniciar(FILE **f,int *regs){
  *f = fopen("biblio.dat","r+b");
  if(*f==NULL){
    printf("Biblioteca vacia.");
    *regs = 0;
    *f = fopen("biblio.dat","wb");
    getch();
    fclose(*f);
    *f = fopen("biblio.dat","r+b");
  }else
```

```
    *regs = filesize(*f)/sizeof(libro);
}
```

La primera instrucción de la función abre el archivo. El archivo se abre para lectura y escritura en modo binario. Si el archivo no existe, se informa al usuario que no hay datos y se crea un archivo vacío, posteriormente el archivo se cierra y se vuelve a abrir en modo de lectura y escritura. Si el archivo existe, se determina el número de registros, para esto se calcula el tamaño del archivo en bytes llamando a la función `filesize` y dividiendo el número de bytes entre el tamaño de un registro. La función `filesize` se muestra a continuación.

```
long int filesize(FILE *archivo){
  long int size_of_file;
  fseek(archivo,0L,SEEK_END);
  size_of_file = ftell(archivo);
  fseek(archivo,0L,SEEK_SET);
  return size_of_file;
}
```

Para determinar el tamaño de un archivo lo que hacemos es posicionar el apuntador de archivo al final y llamar a la función `ftell` que nos indica la posición del apuntador en bytes.

La función `menu` despliega el menú del programa y no requiere más comentarios. Lleva como argumento el número de registros para desplegarlo.

```
void menu(int regs){
  system("cls");
  printf("\n\n\t1 Agregar libro");
  printf("\n\n\t2 Borrar libro");
  printf("\n\n\t3 Editar libro");
  printf("\n\n\t4 Listar libros");
  printf("\n\n\t5 Salir");
  printf("\n\n\tLibros %d",regs);
  printf("\n\n\tOpcion: ");
}
```

Para agregar y editar un registro usamos una función que permite escribir los datos de un libro. La función captura lleva un parámetro por referencia y lee los datos de un libro.

```
void capturar(libro *lib){
  printf("Titulo: ");
  scanf(" %[^\n]",lib->titulo);
  printf("Autor: ");
  scanf(" %[^\n]",lib->autor);
  printf("Editorial: ");
  scanf(" %[^\n]",lib->editorial);
```

```
  printf("Edicion: ");
  scanf("%d",&lib->edicion);
  printf("Anyo: ");
  scanf("%d",&lib->anyo);
}
```

La función `agregar` es la encargada de agregar un libro. La función borra la pantalla y llama a la función `captura`. Una vez capturados los datos, posiciona el apuntador de archivo al final y escribe el registro. Note se ha escrito `(*libros)++` para incrementar la variable libros y no `*libros++`, ya que lo último incrementa el apuntador no el contenido de la variable.

```
void agregar(FILE *f,int *libros){
  libro lib;
  system("cls");
  capturar(&lib);
  despliega(lib);
  fseek(f,0L,SEEK_END);
  fwrite(&lib,sizeof(libro),1,f);
  (*libros)++;
}
```

Para borrar un registro lo que hacemos es crear un nuevo archivo con otro nombre con todos los registros excepto el que se desea borrar. Luego se borra el archivo original (con `remove`) y se renombra el archivo nuevo (con `rename`) con el nombre del archivo borrado. Antes de borrar un archivo es necesario cerrarlo, al igual que antes de cambiar el nombre. La función `borra` hace todas estas acciones.

```
void borrar(FILE *f,int *libros){
  int n,i;
  libro lib;
  FILE *g;
  system("cls");
  printf("Borrar registro? ");
  scanf("%d",&n);
  if(n>=0&&n<=*libros){
    fseek(f,n*sizeof(libro),SEEK_SET);
    fread(&lib,sizeof(libro),1,f);
    printf("Borrando\n%2d: ",n);
    despliega(lib);
    g = fopen("biblio.new","w");
    fseek(f,0,SEEK_SET);
    for(i=0;i<*libros;i++){
      fread(&lib,sizeof(libro),1,f);
      if(i!=n)
        fwrite(&lib,sizeof(libro),1,g);
    }
```

```
    fclose(g);
    fclose(f);
    remove("biblio.dat");
    rename("biblio.new","biblio.dat");
    f = fopen("biblio.dat","r+");
    (*libros)--;
  }
}
```

Para editar un registro volvemos a leer todos los miembros del mismo y lo guardamos en la misma posición dentro del archivo. Esto hace la función `editar`.

```
void editar(FILE *f,int libros){
  int n;
  libro lib;
  system("cls");
  printf("Editar registro? ");
  scanf("%d",&n);
  if(n>=0&&n<=libros){
    fseek(f,n*sizeof(libro),SEEK_SET);
    fread(&lib,sizeof(libro),1,f);
    printf("Editando\n%2d: ",n);
    despliega(lib);
    capturar(&lib);
    fseek(f,n*sizeof(libro),SEEK_SET);
    fwrite(&lib,sizeof(libro),1,f);
  }
}
```

Por último el despliegue de todos los registros del archivo lo hace la función `listar`. Esta función despliega grupos de 20 registros hasta el final del archivo.

```
void listar(FILE *f,int libros){
  int n = 0;
  libro lib;
  system("cls");
  do{
    fseek(f,n*sizeof(libro),SEEK_SET);
    fread(&lib,sizeof(libro),1,f);
    printf("%2d: ",n);
    despliega(lib);
    n++;
    if(n%20==0)
      getch();
  }while(n<libros);
  getch();
}
```

La función `main` inicia el archivo de datos y cuenta los registros. Luego entra en un bucle del que se sale al elegir la opción 5 del menú. Los números del 1 al 4 permiten llamar a las demás funciones del programa.

```
main(){
  FILE *f;
  char ch;
  int libros;
  iniciar(&f,&libros);
  do{
    menu(libros);
    ch = getch();
    printf("%c\n",ch);
    switch(ch){
      case '1':agregar(f,&libros);break;
      case '2':borrar(f,&libros);break;
      case '3':editar(f,libros);break;
      case '4':listar(f,libros);break;
    }
  }while(ch!='5');
}
```

Problemas propuestos

10.5.1. Escriba un programa para gestionar una pequeña agenda de direcciones. Utilice archivos de acceso aleatorio. Incluya opciones para agregar registro, borrar registro, modificar registro, buscar por nombre, buscar por teléfono, etc.

10.5.2. Defina una estructura para representar los datos de un alumno en un curso. Deberá tener campos para: nombre del alumno, cinco calificaciones de exámenes, 5 calificaciones de tareas y calificación final. Escriba funciones para a) leer todos los datos de un grupo de alumnos desde un archivo y almacenarlos en un arreglo, b) calcular la calificación final tomando un peso de 60% para exámenes y 40% para tareas, y c) generar un archivo de salida con una lista de los nombres de los alumnos y su calificación final ordenado por calificación final.

Bibliografía

- Brian w. Kernighan, Dennis M. Ritchie. *El Lenguaje de Programación C*. Prentice Hall Hispanoamericana. 1988. 2ª edición.

- Deitel y Deitel. *Como programar en C y C++*, Prentice Hall. Segunda edición.2003.

- Jean-Paul Tremblay, Richard B. Bunt. *Introducción a la Ciencia de las Computadoras, enfoque algorítmico*. McGraw hill. 1981.

- Aaron M. Tenenbaum, Yedidyah Langsam, Moshe A. Augenstein. *Estructuras de datos en C*. Prentice may. 1993.

- Niklaus Wirth. *Algoritmos+Estructuras de datos=Programas*. Ediciones del Castillo. 1980.

- Byron S. Gottfried. *Programación en C*, segunda edición. McGraw Hill.1997.

- Osvaldo Cairó. *Fundamentos de Programación, piensa en C*. Prentice hall 2006.

- Daniel D. McCracken. *Programación Fortran IV*. Limusa. 1972.

- Richard W. Foley. *Turbo Pascal, Introducción a la programación*. Adison Wesley Iberoamericana. 1993.

- Luis Joyanes Aguilar, Andrés Castillo Sanz, Lucas Sánchez García, Ignacio Zahonero Martínez. *Programación en C, libro de problemas*. McGraw Hill. 2002.

- James L. Antonakos, Kenneth C. Mansfield Jr. *Programación Estructurada en C*. Prentice Hall. 2004.

- Félix García Carballeira, Jesús Carretero Pérez, Javier Hernández Munoz, Alejandro Calderón Mateos. *El lenguaje de Programación C Diseño e implementación de programas*. Prentice Hall. 2002.

Apéndices

Biblioteca <math.h>

Funciones matemáticas

`double acos(double x)`	Calcula el arco coseno de x.
`double asin(double x)`	Calcula el arco seno de x.
`double atan(double x)`	Devuelve el arco tangente en radianes.
`double atan2(double y, double x)`	Calcula el arco tangente de las dos variables x e y. Es similar a calcular el arco tangente de y/x, excepto en que los signos de ambos argumentos son usados para determinar el cuadrante del resultado.
`double ceil(double x)`	Redondea x hacia arriba al entero más cercano.
`double cos(double x)`	devuelve el coseno de x, donde x está dado en radianes.
`double cosh(double x)`	Devuelve el coseno hiperbólico de x.
`double exp(double x)`	Devuelve el valor de e (la base de los logaritmos naturales) elevado a la potencia x.
`double fabs(double x)`	Devuelve el valor absoluto del número en punto flotante x.
`double floor(double x)`	Redondea x hacia abajo al entero más cercano.
`double fmod(double x, double y)`	Calcula el resto de la división de x entre y. El valor devuelto es x - n * y, donde n es el cociente de x / y.
`double frexp(double x, int *exp)`	Se emplea para dividir el número x en una fracción normalizada y un exponente que se guarda en exp o $x = res \times 2^{exp}$.
`long int labs(long int j)`	Calcula el valor absoluto de un entero largo.
`double ldexp(double x, int exp)`	Devuelve el resultado de multiplicar el número x por 2 elevado a exp (inversa de frexp).
`double log(double x);`	Devuelve el logaritmo neperiano de x.
`double log10(double x)`	Devuelve el logaritmo decimal de x.
`double modf(double x, double *iptr)`	Divide el argumento x en una parte entera y una parte fraccional. La parte entera se guarda en iptr.
`double pow(double x, double y)`	Devuelve el valor de x elevado a y.
`double sin(double x)`	Devuelve el seno de x.
`double sinh(double x)`	Regresa el seno hiperbólico de x.
`double sqrt(double x)`	Devuelve la raíz cuadrada no negativa de x.
`double tan(double x)`	Devuelve la tangente de x.
`double tanh(double x)`	Devuelve la tangente hiperbólica de x.

Constantes matemáticas

M_E	La base de los logaritmos naturales e.
M_LOG2E	El logaritmo de e de base 2.
M_LOG10E	El logaritmo de e de base 10.
M_LN2	El logaritmo natural de 2.
M_LN10	El logaritmo natural de 10.
M_PI	π
M_PI_2	π/2
M_PI_4	π/4
M_1_PI	1/π
M_2_PI	2/π
M_2_SQRTPI	2/√π
M_SQRT2	√2
M_SQRT1_2	1/√2

Macros:

HUGE_VAL es una constante de doble precisión que representa a infinito.

Tabla de precedencia y asociatividad de operadores

Operadores	**Asociatividad**
`( ) [ ] ->`	Izquierda a derecha
`! - ++ -- + - * & (tipo) sizeof`	Derecha a izquierda
`* / %`	Izquierda a derecha
`+ -`	Izquierda a derecha
`<< >>`	Izquierda a derecha
`< <= = >=`	Izquierda a derecha
`== !=`	Izquierda a derecha
`&`	Izquierda a derecha
`^`	Izquierda a derecha
`|`	Izquierda a derecha
`&&`	Izquierda a derecha
`||`	Izquierda a derecha
`?:`	Derecha a izquierda
`= += -= *= /= %= &= ^= |= <<= >>=`	Derecha a izquierda
`,`	Izquierda a derecha

Caracteres de canversión de printf

Carácter de conversión	Significado
c	el dato se muestra cama un caracter
d	el dato se muestra cama un entera decimal
e	el dato se muestra cama un valar en cama flatante can expanente
f	el dato se muestra cama un valar en cama flatante sin exponente
g	el dato se muestra cama un valar en cama flatante usanda la canversión de tipo e o f, dependienda del valar; no se muestran ceras na significativas ni el punta decimal si no es signilicativa
i	el dato se muestra como un entero decimal can signa
o	el dato se muestra como un entero actal, sin el cero inicial
s	el dato se muestra como una cadena de caracteres
u	el dato se muestra como un entero decimal sin signa
x	el dato se muestra como un entero hexadecimal, sin el 0x del principio

Notar que algunas de estas caracteres se interpretan de modo diferente que con la función scanf .

Un prefijo puede preceder a ciertos caracteres de conversión.

Prefijo	Significado
h	dato corto (entero corto 0 entero corto sin signo)
l	dato largo (entero largo, entero largo sin signo o real en doble precisión)
L	dato largo (real en doble preeisión largo)

Ejemplo:

```
int a=100;
short b=50;
long c=10000000;
unsigned d=1000;
double x=1e3;
char cad[80] = "abcdefghijk";
printf("%5d %3hd %12ld %12lu %15.1e\n", a, b, c, d, x);
printf("%40s\n", cad);
Salida:

  100  50     10000000         1000        1.0e+003
                              abcdefghijk
```

Indicadores

Indicador	Significado
-	El dato es justificado a Ia izquierda dentro del campo (los espacios en blanco necesarios para rellenar la longitud de campo mínima se añadirán demtrás del dato en vez de delante).
+	Un signo (+ o -) precederá a cada valor numério con signo. Sin este indicador, solo los valores negativos estarán precedidos por un signo.
0	Produce la aparición de ceros iniciales en vez de espacios en blanco. Se aplica solo a datos que estan justilicados a la derecha en campos cuya longitud mínima es mayor que el dato. (Nota: algunos compiladores consideran el indicador cero como una parte de la especificación de la longitud de campo en vez de como un indicador rcal. Esto asegura que el 0 se procese el último si estan presentes múltiples indicadores.)
‘ ‘	Un espacio en blanco precedera a cada valor positivo. Este indicador es anulado (espacio en blanco) por cl indicadpr + si ambos estan presentes.
#(con conversiones de tipo o- y x-)	Hace que los datos octales y hexadecimales sean precedidos por 0 y 0x, respectivamente
#(con conversiones de tipo e-, f- y g-)	Hace que esté presente el punto decimal en todos los números en coma flotante, incluso si el número no tiene decimales. También trunca los ceros no significativos en una conversión del tipo g-.

Ejemplo:

```
int a;
short b;
1ong c;
unsigned d;
double x;
printf("%+5d %+5hd %+l2ld %-l2lu %#15.7le\n", a, b, c, d, x);
```

Salida:

```
   +2    +0     +2293672 2088763392    5.2124176e+291
```

Caracteres de conversion de `scanf`

Carácter de conversión	Significado
`c`	el dato es un caracter
`d`	el dato es un entero decimal
`e`	el dato es un valor en coma flotante
`f`	el dato es un valor en coma flotante
`g`	el clato es un valor en coma flotante _
`h`	el dato es un entero corto
`i`	el dato es un entero decimal, hexadecimal u octal
`o`	el dato es un emero octal
`s`	el dato es una cadena de caracteres seguido por un espacio en blanco (cl caracter nulo \0 se añade automaticamente al final)
`u`	el dato es un emero decimal sin si gno
`x`	el dato es un entero hexadecimal
`[ . . . ]`	el dato es una cadena de caracteres que puede contener espacios en blanco

Un prejijo puede preceder a ciertos caracteres de conversion.

Prefijo	Significado
`h`	dato corto (entero corto o entero sin signo corto)
`l`	dato Iargo {entero largo, entero largo sin signo o real en doble precisión)
`L`	dato largo (real en doble precisión largo)

Ejemplo:

```
int a;
short b;
long c;
unsigned d;
double X;
char cad[80];

scanf("%5d %3hd %l2ld %l2lu %15lf", &a, &b, &c:, &d, &x);
scanf ( "% [^\n] ", cad);
```

Biblioteca `stdlib.h`

Funciones

`abort()`	Ocaciona la terminación abnormal del programa
`abs(n)`	Valor absoluto de un entero
`atexit(func)`	Registra función para ser llamada en la terminación normal de un programa
`atof(cad)`	Convierte una cadena a un double
`atoi(cad)`	Convierte una cadena a un entero
`atol(cad)`	Convierte una cadena a un entero largo
`bsearch()`	Implementa la búsqueda binaria
`calloc(n,t)`	Reserva espacio en memoria para n objetos de tamaño t
`div(num,den)`	Calcula cociente y residuo
`exit(n)`	Termina la ejecusión de un programa
`free(ptr)`	Libera la memoria usada por ptr
`getenv(cad)`	Obtiene cadena de entorno
`labs(l)`	Valor absoluto de un entero largo
`ldiv(l)`	Calcula cociente y residuo de un entero largo
`malloc(tam)`	Reserva memoria y regresa apuntador
`mblen(cad,t)`	Calcula el numero de byte en caracter multibyte
`mbstowcs(ptr,cad,n)`	Convierte secuencia de caracteres multibyte
`mbtowc()`	Número de bytes en carácter multibyte
`qsort()`	Implementa quick sort
`rand()`	Genera un número aleatorio
`realloc(ptr,t)`	Cambia el tamaño de espaci oreservado por ptr
`srand(n)`	Inicia semilla de números aleatorios
`strtod(cad,cad[])`	Convierte una cadena a double
`strtol(cad, cad[],b)`	Convierte una cadena a long int en la base b
`strtoul(cad, cad[],b)`	Convierte una cadena a unsigned long int en la base b
`system(cad)`	Ejecuta comando del sistema opertativo
`wctomb(cad,wc)`	Determina el número de bytes necesitado para representar un carácter multibyte

Macros

EXIT_FAILURE	-1 Valor de retorno de exit si hubo falla
EXIT_SUCCESS	0 Valor de retorno de exit si no hubo falla
RAND_MAX	0x7FFF