Lenguaje C

Teoría y ejercicios

Evelio Granizo Montalvo

ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

LENGUAJE C

Teoría y Ejercicios

EVELIO GRANIZO MONTALVO

Lenguaje C, teoría y ejercicios
Ing. Evelio Granizo Montalvo

Primera edición electrónica. Junio de 2015
ISBN: 978-9978-301-35-7
Par revisor: Ing. Vinicio Carrera; Ing. Darwin Alulema Mgs.

Universidad de las Fuerzas Armadas - ESPE
Grab. Roque Moreira Cedeño
Rector

Publicación autorizada por:
Comisión Editorial de la Universidad de las Fuerzas Armadas - ESPE

Producción
David Andrade Aguirre
Pablo Zavala A.

Diseño Gráfico:
Ing. Evelio Granizo
Pablo M. Padilla G.

Derechos reservados. Se prohibe la reproducción de esta obra por cualquier medio impreso, reprográfico o electrónico.

El contenido, uso de fotografías, gráficos, cuadros, tablas y referencias es de **exclusiva responsabilidad** del autor.

Los derechos de esta edición electrónica son de la **Universidad de las Fuerzas Armadas - ESPE,** para consulta de profesores y estudiantes de la universidad e investigadores en: htpp//www.repositorio.espe.edu.ec.

Universidad de las Fuerzas Armadas - ESPE
Av. General Rumiñahui s/n, Sangolquí, Ecuador.
htpp//www.espe.edu.ec

Prólogo

Esta obra está diseñada con el objeto de ofrecer a los estudiantes, y a cualquier persona que tenga interés, un texto-guía de fácil comprensión para el aprendizaje eficiente del LENGUAJE C. La obra se fundamenta en la información recopilada y desarrollada por el autor durante los años que ejerció la cátedra universitaria como profesor de esté lenguaje y de temáticas afines.

Dada la gran acogida que tuvo la primera edición, surgió el compromiso de realizar una segunda edición, la cual respecto a la primera, dispuso de las siguientes mejoras:

- Teoría actualizada y ejercicios depurados, tanto resueltos como propuestos.
- Inclusión de dos apéndices para facilitar la consulta y el análisis. El primero proporciona la lista de los códigos ASCII para el PC, y el segundo, los Prototipos de las Funciones de Biblioteca del lenguaje C más utilizadas.

La tercera edición de esta obra dispone de mejoras adicionales en la exposición teórica, recogidas la mayoría de ellas a través de las observaciones y comentarios gentilmente aportados por los profesores que utilizan esta obra. Cabe señalar que el aspecto formal de esta tercera edición ha sido sometida a una revisión por pares realizada por el Dr. Vinicio Carrera y el Ing. Darwin Alulema Mgs., Docentes de la Universidad de la Fuerzas Armadas – ESPE, quienes han permitido ampliar la perspectiva de esta nueva edición.

Una vez más es grato dejar en las manos y la mente de los estudiantes y de sus profesores esta obra que ha sido mejorada gracias a sus aportes, los cuales siempre serán bienvenidos.

Evelio Granizo Montalvo

Introducción

La teoría presentada en esta obra está orientada hacia el estándar ANSI, por lo que puede utilizarse en cualquier compilador de lenguaje C, omitiéndose el tratamiento de los distintos compiladores. Excepto el capítulo undécimo "archivos" que solo se utiliza en el Sistema Operativo Windows.

El texto contiene, a más de la teoría, ejercicios resueltos para reforzar lo presentado en cada tema, y una gran cantidad de ejercicios propuestos al final de cada capítulo.

Los capítulos de esta obra están distribuidos de la siguiente manera:

El primer capítulo, INTRODUCCION AL LENGUAJE C, contiene el origen y las ventajas del lenguaje C. También, se analiza al lenguaje C como lenguaje de nivel medio, como lenguaje de alto nivel y como lenguaje estructurado. Además se establece la diferencia entre enlazar y compilar separadamente un programa. Por último, se establece la estructura de un programa en lenguaje C, lo que es una biblioteca, y cómo se realiza la compilación de un programa en lenguaje C.

El segundo capítulo, TIPOS DE DATOS, contiene las definiciones básicas y necesarias de los tipos de datos que definen un conjunto de valores que puede tener una variable, junto con un conjunto de operaciones que se pueden realizar sobre esa variable, siendo los tipos de datos más comunes los números enteros, los números reales y los caracteres. El lenguaje C tiene cinco tipos de datos básicos incorporados; aunque no se trata de un lenguaje fuertemente tipificado, porque se permiten casi todas las conversiones de tipos.

El tercer capítulo, ENTRADA/SALIDA POR CONSOLA, contiene la entrada y salida de datos que se realizan a través de funciones de biblioteca, siendo un sistema de E/S que ofrece un mecanismo flexible, a la vez que consistente, para transmitir datos entre dispositivos. Las funciones de E/S por consola son aquellas que controlan la entrada por teclado y la salida a través de pantalla, y es la E/S estándar del sistema; aunque esta E/S puede ser dirigida a otros dispositivos. En lenguaje C existe E/S por consola y por archivo, que son conceptualmente diferentes, pero técnicamente el lenguaje C hace poca distinción entre la E/S por consola y la E/S por archivo.

El cuarto capítulo, SENTENCIAS DE CONTROL, contiene los tipos de sentencias de un programa en lenguaje C, que son las piezas con las que se construye un programa. Es decir, en lenguaje C una sentencia es una instrucción para que el computador realice una tarea determinada. Además este capítulo contiene las "sentencias de control", cuyas formas básicas son decisión y repetición, que tienen la función de controlar el "flujo del programa" (flujo del programa es la secuencia que el computador sigue para ejecutarlo).

En el quinto capítulo, PUNTEROS, se define a los punteros que son variables que almacenan una dirección de memoria de otra variable, esta dirección es la posición interna de la variable en la memoria RAM del computador, es decir, son variables apuntando a otras. Para acceder a un dato se necesita a más de la dirección de memoria, el "tipo base del puntero" que define el tipo de variable a la que puede apuntar el puntero, especificándose a la vez el tamaño del dato; ya que el puntero solo apunta al primer byte del dato y técnicamente cualquier tipo de puntero puede apuntar a cualquier lugar de la memoria, pero toda la aritmética de punteros está hecha en relación a su tipo base. El tamaño de la variable puntero varía de acuerdo a la arquitectura de la memoria RAM del computador, en la actualidad se tiene de 16, 32 y 64 bits; por lo que, para facilitar el entendimiento del manejo de punteros, tanto en la teoría como en los ejemplos, se adaptó la versión simplificada de la arquitectura de 16 bits; de esta manera, todos lo ejemplos de esta obra muestran direcciones que se almacenan en dos bytes.

El sexto capítulo, FUNCIONES, contiene la definición de una función, que es un bloque de código de programa autocontenido diseñado para realizar una tarea determinada. La filosofía del diseño del lenguaje C está basada en el empleo de funciones, ya que un programa dividido en funciones es más modular y por tanto más fácil de leer, modificar o arreglar. La razón principal para usar funciones es para evitar tediosas repeticiones de programación; escribiendo una sola vez la función apropiada se la puede emplear cualquier número de veces en un determinado programa y en diferentes situaciones, únicamente llamándola por su nombre. Es decir, cuando una función es lo suficientemente general, se la puede emplear en diferentes programas, teniéndola a disposición en una librería. En general, se plantean las funciones como "cajas negras", definiéndoles mediante su ***entrada*** y su ***salida***, es decir, mediante la información que hay que suministrarlas y el producto recibido de ellas, respectivamente.

El séptimo capítulo, MODOS DE ALMACENAMIENTO, contiene los "modos de almacenamiento" de una variable, que permiten determinar el ***alcance*** de una variable (dónde son reconocidas y en qué parte del programa se las pueden usar) y el ***tiempo*** que permanece la variable en la memoria del computador. El "modo de almacenamiento" queda a su vez determinado por el ***lugar*** donde se declara la variable y por su ***palabra clave***. Entonces, una variable tiene a más de su tipo de dato, un "modo de almacenamiento".

En el octavo capítulo, ARREGLOS, se define a los arreglos que son colecciones de datos del mismo tipo que se referencian por un mismo nombre, cuyos datos, llamados "elementos", se distinguen entre sí con índices o direcciones de memoria. Los arreglos se clasifican por el número de índices de acceso a los elementos, por lo que pueden tener de una a varias dimensiones: unidimensionales, bidimensionales y multidimensionales. Además, existen arreglos de punteros y arreglos asignados dinámicamente.

El noveno capítulo, CADENAS DE CARACTERES, contiene la definición de una cadena en lenguaje C, que es un **arreglo** de tipo **char** que termina en un caracter nulo, '\0'. Además, se establece la utilización de cadenas dentro un programa y se estudian las funciones estándar de cadenas de caracteres más usadas. Por último, se realiza un estudio del uso de los argumentos de la línea de comandos, para utilizarlos dentro de un programa.

En el décimo capítulo, ESTRUCTURAS, se definen estructuras, "campos de bits", uniones y enumeraciones. Pues estos tipos de datos tienen uno de los usos más importantes en la creación de nuevos formatos de datos, porque resultan mucho más eficientes que utilizar arreglos o estructuras simples. Estos formatos son las pilas, colas, árboles, tablas y grafos, los mismos que se construyen a partir de estructuras encadenadas. Por último, se define la sentencia **typedef**, que permite asignar un nombre alternativo a un tipo de dato ya existente, con un nombre arbitrario otorgado por el usuario; es decir, realmente no se crea un nuevo tipo de dato.

El capítulo undécimo, ARCHIVOS, contiene las operaciones de E/S a archivos que tienen lugar a través de llamadas a las funciones de la biblioteca estándar. Esta biblioteca es llamada "sistema de E/S" y en lenguaje C son definidos tres tipos: el sistema de E/S definido por el estándar ANSI, el sistema de E/S tipo UNIX y el sistema de E/S de bajo nivel. En este capítulo solo se estudiará el sistema de E/S para compiladores en el Sistema Operativo Windows, que puede ser usado para leer y escribir cualquier tipo de datos, siendo estos datos transferidos en su representación binaria interna o en formato de texto normal.

El capítulo duodécimo, PREPROCESADOR, contiene las "directivas de preprocesador" que son sentencias dirigidas al compilador en el código fuente de un programa en lenguaje C, y no son realmente parte del lenguaje C, pero amplían el ámbito de entorno de programación.

Además, esta obra contiene los anexos: CÓDIGO ASCII PARA EL PC, PROTOTIPOS DE LAS FUNCIONES DE BIBLIOTECA DEL LENGUAJE C MÁS UTILIZADOS E ÍNDICE DE PROGRAMAS. También se presentan en la dirección Web ***http://eagranizo.wix.com/librospublicados***, los programas demostrativos que refuerzan lo presentado en la teoría.

Capítulo 1

INTRODUCCIÓN AL LENGUAJE C

1.1. ORIGEN

El lenguaje C fue creado por Dennis Ritchie de los Laboratorios Bell, en 1972, cuando trabajaba junto con Ken Thompson en el diseño del sistema operativo UNIX. El lenguaje C no surgió como idea espontánea de Ritchie; fue el resultado de un proceso que comenzó con los lenguajes: BCPL que fue desarrollado por Martín Ritchards y B inventado por Ken Thompson.

El lenguaje C fue implementado por primera vez por Dennis Ritchie en un computador DEC PDP-11 usando el Sistema Operativo UNIX como sistema operativo. Siendo durante muchos años el estándar del lenguaje C el proporcionado en la versión V del sistema operativo UNIX.

Luego, con la popularidad de las microcomputadoras, se crearon muchas implementaciones de lenguaje C, apareciendo algunas discrepancias porque no existía ningún estándar definido del lenguaje C. Para solucionar este problema, el Instituto de Estándares Nacional Americano (ANSI, siglas en inglés) estableció un comité en el año de 1983 para crear un estándar del lenguaje C, el cual fue terminado en 1990. Por lo que todos los principales compiladores de lenguaje C a partir de 1990 ya se han implementado en base al estándar ANSI.

El lenguaje C es el lenguaje predominante en el mundo de las computadoras de sistemas UNIX y computadoras personales, siendo usado por compañías de software, estudiantes de informática, etc.

Inicialmente el lenguaje C fue usado para la "***programación de sistemas***", que es una parte de una amplia clase de programas que forman parte del sistema operativo de la computadora o de sus utilidades de soporte. Por ejemplo, los sistemas operativos, intérpretes, compiladores, editores de texto, programas de ensamblado, gestores de base de datos, etc.

A medida que el lenguaje C creció en popularidad, se comenzó a usarlo para programar cualquier tipo de tarea, debido a sus ventajas y desempeño. (SCHILDT, 1994)

1.2. VENTAJAS

El lenguaje C es un lenguaje moderno que incorpora características como la planificación escalonada, programación estructurada y diseño modular; el resultado es un programa más fiable y comprensible. Además, respecto a los otros lenguajes de

programación, el lenguaje C tiene las siguientes ventajas: eficiencia, portabilidad, potencia y flexibilidad. A continuación se describe cada una: (SCHILDT, 1994)

1. **Eficiencia**. El lenguaje C aprovecha óptimamente las características del hardware de las microcomputadoras, por lo tanto, los programas de lenguaje C tienden a ser más compactos y se ejecutan con mayor rapidez.

2. **Portabilidad**. Los programas de lenguaje C escritos en un sistema computacional o tipo de computador pueden ejecutarse en otros sin ninguna modificación, o con modificaciones mínimas, por ejemplo en los sistemas Apple, IBM, etc. Si se necesita modificar un programa de lenguaje C, a menudo esto se reduce a cambiar unas cuantas sentencias de entrada en un archivo de cabecera (Header) en el programa principal. Por lo que los programas de lenguaje C son fáciles de modificar y de adaptar a nuevos modelos de computadoras.

3. **Potencia**. En lenguaje C están escritos casi todos los compiladores e intérpretes como: Pascal, Fortran, Logo, APL, LISP, Basic, etc. También están escritos en lenguaje C la mayoría de los sistemas operativos, bases de datos, hojas electrónicas, procesadores de texto, emuladores gráficos (producción de secuencias animadas). Además, se utiliza el lenguaje C en ingeniería para manejo de pórticos, comunicaciones, acceso a las capacidades de hardware, etc.

4. **Flexibilidad**. El lenguaje C posee control sobre aspectos del computador asociados con el lenguaje ensamblador y las ventajas del lenguaje de alto nivel. Es decir, se puede lograr mayor eficiencia del computador sin mayor complicación en el desarrollo de los programas.

1.3. EL C COMO LENGUAJE DE NIVEL MEDIO

El lenguaje C se considera un lenguaje de "nivel medio", porque combina elementos de lenguajes de "alto nivel" con el funcionalismo del lenguaje "ensamblador". Esto no significa que sea menos potente, más difícil de usar o menos evolucionado que los lenguajes de "alto nivel" como el Pascal, Modula_2, etc. Ni implica que el lenguaje C sea similar al lenguaje ensamblador que es de "bajo nivel", con sus problemas asociados; sino porque aprovecha sus características. (SCHILDT, 1994)

El lenguaje C como lenguaje de nivel medio permite la manipulación de bits, bytes, palabras de memoria y punteros, que son los elementos básicos con que funciona la computadora.

Otro aspecto importante del lenguaje C es que solo tiene 32 "palabras clave", que constituyen las órdenes que lo conforman. Estas palabras junto con la sintaxis, forman el lenguaje C. A continuación se lista las 32 "palabras clave":

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

De esta lista, 27 "palabras clave", fueron definidas en la versión original del lenguaje C estándar de Ritchie y Kemigham, y las cinco siguientes fueron añadidas por el ANSI: **const**, **enum**, **signed**, **void** y **volatile**. Posteriormente se han añadido diversas "palabras clave", dependiendo del compilador y sistema operativo. (SCHILDT, 1994)

En lenguaje C las mayúsculas y las minúsculas son diferentes, por eso todas las palabras clave de lenguaje C están en minúscula y una "palabra clave" no puede ser usada como nombre de variable o de función.

1.4. EL C COMO LENGUAJE DE ALTO NIVEL

El lenguaje C como un lenguaje de alto nivel tiene estructuras en bloques, funciones independientes, y un compacto conjunto de palabras clave. Entonces, un programa en lenguaje C puede frecuentemente alcanzar la eficiencia del código ensamblador, como su velocidad, junto con la estructura de los lenguajes de alto nivel como el ALGOL o Modula_2, la extensibilidad del lenguaje FORTH y pocas de las restricciones del Pascal o Modula_2.

El lenguaje C normalmente es compilado, y en la actualidad los compiladores de lenguaje C tienen un intérprete para ayudar en la depuración.

1.5. EL C COMO LENGUAJE ESTRUCTURADO

Técnicamente, un lenguaje estructurado por bloques permite declarar procedimientos dentro de otros procedimientos, de esta forma, se extiende los conceptos de globalidad y localidad mediante el uso de "reglas de alcance", que gobiernan la "visibilidad" de las variables o de los procedimientos. Además, la característica distintiva de un lenguaje estructurado es la capacidad de seccionar y ocultar del resto del programa toda la información y las instrucciones necesarias para llevar a cabo una determinada tarea. Esto se lo hace mediante el uso de subrutinas que emplean variables locales.

Entonces, dado que el lenguaje C no permite la creación de funciones dentro de funciones, no puede ser formalmente denominado lenguaje estructurado en bloques. Pero se lo considera un lenguaje estructurado ya que con él se programa más fácilmente

de forma organizada en bloques secuenciales, es más fácil de mantener el programa, tiene similitudes estructurales al ALGOL, Pascal y Modula_2, y es posible escribir subrutinas con el uso de variables locales de forma que lo que ocurra en su interior no provoque efectos secundarios en otras partes del programa. (SCHILDT, 1994)

Además, como la componente de estructuración principal del lenguaje C es la función, que es una subrutina independiente del lenguaje C, permite definir cada tarea de un programa y codificarla por separado, para que los programas sean modulares. (KERNIGHAN & RITCHIE, 1985)

Otra forma de estructuración en lenguaje C es mediante el uso de "***bloque de código***", que es un grupo de sentencias de un programa conectadas de forma lógica y que es tratado como una unidad o proceso. En lenguaje C se crean bloques de código colocando una serie de sentencias entre llaves, y cada sentencia puede ser una sentencia simple o un bloque de sentencias.

1.6. ESTRUCTURA DE UN PROGRAMA EN LENGUAJE C

Todos los programas en lenguaje C consisten en una o más funciones, la única función que debe estar siempre presente es la denominada **main()** (técnicamente **main()** no es parte parte integral del lenguaje C), siendo la primera función que se llama cuando comienza la ejecución del programa, y ella puede a su vez llamar a otras funciones. Sin embargo, hay funciones de tipo vacío lo que significa que ellas no retornan valores. También el programador es libre de ignorar algunos valores que una función retorne.

Para que un programa en lenguaje C esté bien escrito, la función **main()** contiene un esbozo de lo que el programa hace, es decir, está compuesto por las llamadas a otras funciones, las cuales realizan una tarea específica del programa. (KERNIGHAN & RITCHIE, 1985)

La estructura de un programa en lenguaje C es la siguiente:

<Comandos de preprocesador>
<Definición de tipos>
<Prototipos de funciones>
<Variables globales>
<Funciones>

NOTA: Lo que está demarcado con los caracteres, **<>**, puede o no ir, excepto la función principal **main()**.

Cada función tiene la siguiente estructura: (DEITEL & DEITEL, 1995)

<**Tipo**> Nombre de la función (<**Declaraciones de parámetros**>)

```
{
    <Declaraciones locales>
    <Sentencias>
}
```

Una función se identifica cuando un nombre va seguido por paréntesis, dentro de los cuales puede o no haber parámetros. Una función consta de un encabezamiento y un cuerpo:

1. **Encabezamiento**. Contiene cualquier tipo de dato que retornará la función, el nombre de la función, y entre paréntesis la declaración de los parámetros, si existieran.

2. **Cuerpo**. Está delimitado por las llaves, **{}**, que indican el bloque de código, y consiste en un grupo de sentencias, cada una de las cuales termina en un punto y coma.

1.7. BIBLIOTECA DEL LENGUAJE C

La biblioteca es un conjunto de archivos pequeños, que contienen funciones para llevar a cabo las tareas necesarias más comunes, aunque dependiendo del compilador contienen más o menos funciones, como en la versión de lenguaje C original de UNIX donde existen algunas funciones que no están definidas en el ANSI, ya que eran redundantes. Entonces, los programas en lenguaje C tienen llamadas a varias funciones contenidas en la biblioteca estándar. Por ejemplo, las operaciones de E/S, funciones matemáticas, etc.

Si se escribe una función que se va a usar repetidas veces, también puede ser ubicada en la biblioteca estándar, considerando que algunos compiladores permiten colocar esas funciones en esa biblioteca y otros obligan a crear una biblioteca adicional.

Cada programador de lenguaje C puede crear y mantener su propia biblioteca de funciones de acuerdo a su necesidad y que pueda ser usada en muchos programas diferentes. Además, el lenguaje C tiene la compilación separada, para permitir que los programas trabajen fácilmente en grandes proyectos y minimizar la duplicidad de esfuerzo. (DEITEL & DEITEL, 1995)

1.8. ENLAZADOR

El enlazador es un programa que enlaza funciones compiladas por separado para producir un solo programa, también combina las funciones de la biblioteca estándar de lenguaje C con el código que se haya escrito. La salida del enlazador es un programa ejecutable.

Algunos compiladores de lenguaje C tienen su propio enlazador y otros usan el enlazador estándar proporcionado por el sistema operativo.

1.9. COMPILACIÓN SEPARADA

El lenguaje C permite separar un programa en muchos archivos, los cuales son compilados por separado. Una vez que han sido compilados todos los archivos, se enlazan entre sí, junto con las rutinas de la biblioteca, para formar el código objeto completo.

La ventaja de la compilación separada es que un cambio en el código de uno de los archivos no requiere la compilación del programa entero, y así el tiempo de compilación es más corto. (DEITEL & DEITEL, 1995)

1.10. COMPILACIÓN DE UN PROGRAMA EN LENGUAJE C

La compilación de un programa en lenguaje C consiste de los siguientes tres pasos: (BECERRA, 1991)

1. **Creación del programa**. Se utiliza un editor que generalmente es incorporado por el compilador, y el archivo editado debe ser de texto ASCII estándar. El nombre de un archivo fuente escrito por el editor del lenguaje C debe tener lo siguiente: como primera parte un nombre respecto a lo que hace el programa, y como segunda parte la extensión **.c** para identificar que es un archivo de programa en lenguaje C.

2. **Compilación del programa**. Comprobará si el programa tiene algún error; si existe será indicado, caso contrario, el compilador traducirá el programa fuente a lenguaje de máquina, almacenándolo en un archivo, que es **ejecutable**, si el programa está en un solo archivo. Esto dependerá del compilador particular de lenguaje C que se use.

3. **Enlazado del programa**. Si se tiene archivos diferentes se enlazan con todas las funciones de la biblioteca que se necesiten, pero si existe un solo archivo no es necesario enlazar. Esto podrá variar entre compiladores y entornos.

A continuación, ya puede ser ejecutado el programa desde el mismo compilador o desde el sistema operativo, tecleando el nombre del nuevo archivo.

1.11. MAPA DE MEMORIA DEL LENGUAJE C

Un programa compilado en lenguaje C crea y usa 4 regiones de memoria lógicas diferentes que sirven para funciones específicas:

1. La primera región es la memoria que contiene el "***código de programa***".

2. La siguiente región es la memoria donde se guarda las "***variables globales***", cuya creación se realiza en el momento de la compilación.

3. La tercera región es el "***montón***" de la memoria libre que puede usar el programa mediante las funciones de asignación dinámica de memoria, en el momento de la ejecución del programa. Se utiliza para estructuras dinámicas como listas, árboles, etc.

4. La cuarta región es la "***pila***" que se usa en el momento de la ejecución del programa, para mantener las direcciones de vuelta de las llamadas a funciones, almacenar las variables locales, y salvar el estado de la CPU.

La disposición exacta del mapa de memoria de lenguaje C puede variar de compilador a compilador y entre distintos entornos, y la disposición física exacta para cada una de las cuatro regiones de memoria difiere entre los distintos tipos de CPU y las distintas implementaciones de lenguaje C, en la Figura 1.1 se muestra de forma conceptual cómo aparece el mapa de memoria.

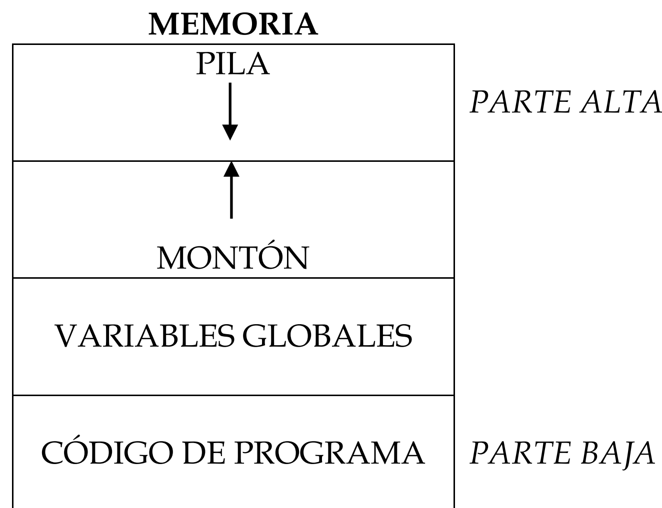


Figura 1.1. Mapa de Memoria del Lenguaje C (SCHILDT, 1994)

1.12. PROGRAMAS LEGIBLES

Con un programa legible se consigue que dicho programa sea fácil de comprender, corregir o modificar. Para lo cual se debe cumplir lo siguiente:

1. Escribir el programa en forma estructurada.

2. Escoger nombres de los identificadores de funciones, variables y constantes, apropiados de acuerdo a su función.

3. Usar una sentencia por línea.

4. Poner comentarios para resaltar una función, una declaración, una instrucción o un bloque de código, etc.

5. Emplear líneas en blanco para separar las funciones o bloques de código.

6. Identación de las instrucciones dentro de los supbloques.

Capítulo 2

TIPOS DE DATOS

2.1. INTRODUCCIÓN

Los datos se pueden representar mediante constantes o variables:

a) **Constantes**

Las constantes son datos preseleccionados en tiempo de compilación del programa, cuyos valores se mantienen sin alterar durante la ejecución del mismo. Las ventajas de las constantes son:

- En una nueva compilación se puede modificar su valor en un solo sitio, el mismo que se actualiza en donde se encuentre la constante.
- El nombre de la constante define claramente su valor. Además, el compilador es capaz de identificar el tipo de dato de la constante por el aspecto que tiene, entonces, no se necesita definir el tipo de dato.

b) **Variables**

La variable es una posición de memoria con nombre, en donde se guarda su valor, el mismo que puede ser modificado durante la ejecución del programa. En lenguaje C se debe declarar las variables previamente antes de usarlas, porque en la compilación del programa se asigna el espacio de memoria para las variables de acuerdo al tipo declarado.

2.2. TIPOS DE DATOS

El tipo de dato define un conjunto de valores que puede tener una variable, junto con un conjunto de operaciones que se pueden realizar sobre esa variable. Los tipos de datos comunes son los números enteros, números reales y caracteres.

Todos los lenguajes de alto nivel soportan el concepto de tipos de datos, aunque el lenguaje C tiene cinco tipos de datos básicos incorporados; no se trata de un lenguaje fuertemente tipificado, porque se permite casi todas las conversiones de tipos. El lenguaje C no lleva a cabo comprobaciones de errores en tiempo de ejecución, estas comprobaciones quedan enteramente a responsabilidad del programador.

En lenguaje C existen cinco tipos de datos básicos, y todos los demás tipos de datos se basan en estos tipos. Los cinco tipos de datos básicos son: (SCHILDT, 1994)

char Caracter
int Entero

float Punto flotante
double Punto flotante de doble precisión
void Sin valor

Basándose en la forma de almacenamiento en la memoria del computador, se tiene que los tipos "caracter" y "entero" están representados por tipos "enteros", y los "punto flotante" están representados en el mismo tipo.

Modificadores. A excepción del tipo **void**, los tipos de datos básicos pueden tener distintos modificadores, para rango y signo: (SCHILDT, 1994)

- **signed**
- **unsigned**
- **long**
- **short**

Estos modificadores se aplican al tipo base **int**, pero **signed** y **unsigned** también se aplican a **char**, y **long** se puede aplicar a **double**.

El uso de **signed** con enteros es redundante, su uso más importante es para modificar el tipo **char**, en implementaciones que no se tenga signo por defecto.

2.2.1. Tipo *char*

Los valores de este tipo son definidos por un conjunto de caracteres, en general por el código ASCII, el cual es utilizado por el computador para identificar a cada caracter.

2.2.2. Tipo *int*

Un entero es un número exacto, carece de parte fraccionaria, por lo que no tiene punto decimal. El tipo **int** almacena enteros de una manera directa y en su representación binaria correspondiente.

El lenguaje C presenta una gran variedad de tipos enteros: (SCHILDT, 1994)

a) ***Enteros con signo***. Son los tipos de datos: **int**, **signed int**, **short int**, **signed short int**, **long int**, y **signed long int**. Los valores permitidos para estos tipos son números enteros positivos, negativos o cero.

   El tipo **int** se refiere al tamaño de palabra estándar, en este caso en que la palabra de memoria es de 16 bits (depende del compilador y entorno), el rango de valores está desde -32768 a 32767.

En su representación binaria del tipo **int** se utiliza el bit más significativo para indicar el signo del número, por lo que el mayor entero con signo que se puede almacenar en una palabra será:

$2^{n-1} - 1$

donde:

- **n**, número de bits de la palabra de memoria.
- **n - 1**, por el signo.
- **-1**, por el cero.

b) ***Enteros sin signo***. Son los tipos de datos: **unsigned int**, **unsigned short int**, y **unsigned long int**; los cuales pueden ser únicamente positivos o cero.

El mayor entero "sin signo" será mayor que el mayor entero "con signo", porque el bit de signo de los enteros "con signo" se utiliza para la magnitud en los enteros "sin signo", por lo que el rango de un tipo **unsigned int**, en este caso en que la palabra de memoria es de 16 bits, tiene el rango de 0 a 65535.

Se usa este tipo de datos para:

1. Asegurar que el valor de una variable nunca sea negativo.
2. Tener mayor rango de números positivos.
3. Acceder direcciones de memoria.
4. Usar variables de este tipo como contadores.

2.2.3. Tipo *float*

Corresponde a los números reales que abarcan un rango mayor a los enteros, incluyendo estos números.

Para almacenar un número de tipo **float** se lo divide en dos partes:

- Parte fraccionaria o mantisa (precisión del número).
- Parte del exponente (tamaño del número).

Los números de punto flotante sirven cuando se necesitan números de mayor rango, incluyendo fracciones, y para mayor precisión cuando se tienen cálculos matemáticos.

Los números en punto flotante se representan en notación científica, por ejemplo:

Número		Notación Científica	Notación Exponencial
1000000000	=	1.0×10^{9}	1.0e9
123000	=	1.23×10^{5}	1.23e5
0.000056	=	5.6×10^{-5}	5.6e-5

El tipo **float** depende de la implementación del lenguaje C y de la arquitectura del CPU, en este caso que se utiliza 32 bits para almacenar un número en punto flotante. De ellos, se usa 1 bit para el signo del número, 8 bits para el exponente con su signo, y 23 bits para la parte fraccionaria o mantisa. Esto permite una precisión de 6 cifras decimales y un rango de ±(1.8e-38 a 3.4e+38). (SCHILDT, 1994)

2.2.4. Tipo *double*

Son los mismos números de punto flotante, pero con mayor precisión (número de dígitos que se pueden almacenar).

El tipo **double** (doble precisión) utiliza el doble número de bits que el tipo **float**, en este caso 64 bits. Se incrementan los bits de la parte fraccionaria o mantisa a 11 y la parte exponencial a 52, para incrementar la precisión (es decir reducir los errores de redondeo) y aumentar el rango de los números, respectivamente. Esto permite una precisión de 15 cifras decimales y un rango de ±(2.2e-308 a 1.8e308). (SCHILDT, 1994)

2.2.5. Tipo *void*

void indica que no tiene tipo de dato. Es decir, no existencia o no atribución de un tipo en una variable o declaración; y se utiliza para: declarar explícitamente una función que no devuelve valor alguno, establecer que la función no tiene parámetros, o crear punteros genéricos.

En la Tabla 2.1 se muestra todas las combinaciones de los tipos de datos que se ajustan al estándar ANSI, para el lenguaje C en el sistema operativo Windows de 32 bits.

Tabla 2.1. Tipos de Datos de C (SCHILDT, 1994)

TIPO		TAMAÑO EN BITS	RANGO MÍNIMO
CARACTER	char	8	-128 a 127
	unsigned char	8	0 a 255
	signed char	8	-128 a 127
ENTERO	int	16	-32768 a 32767
	unsigned int	16	0 a 65535
	signed int	16	Igual que **int**
	short int	16	Igual que **int**
	unsigned short int	16	0 a 65535
	signed short int	16	Igual que **short int**
	long int	32	-2147483648 a 2147483647
	signed long int	32	Igual a **long int**
	unsigned long int	32	0 a 4294967295
PUNTO FLOTANTE	float	32	±(1.8e-38 a 3.4e+38) 6 dígitos de precisión
	double	64	±(2.2e-308 a 1.8e+308) 15 dígitos de precisión
	long double	80	±(3.4e-4932 a 1.2e+4932) 18 dígitos de precisión
SIN TIPO	void	0	Sin valor

2.3. CONSTANTES

Las constantes en lenguaje C son valores fijos que no pueden ser alterados por el programa. Estas constantes son de cualquier tipo de datos básicos y la forma en que se representa cada constante depende de su tipo.

2.3.1. Constantes Carácter

Una constante **char** puede representar tan sólo un único caracter que normalmente corresponde al código ASCII, y que va encerrada entre apóstrofes. Por ejemplo, los caracteres: 'a' y '*'.

Un caracter que no se puede introducir desde teclado, puede ser introducido de las dos formas siguientes:

1. **Secuencias de escape**. Son caracteres especiales predefinidos, que deben usar una "barra invertida" (\) y luego su código, en lugar de sus equivalentes ASCII; asegurando así la portabilidad del código. Estos caracteres se indican en la Tabla 2.2.

Tabla 2.2. Caracteres Secuencias de Escape (SCHILDT, 1994)

CÓDIGO	SIGNIFICADO
\b	Retroceso (**b**ack space - Espacio atrás)
\f	Salto de página
\n	Salto de línea (**n**ew line - nueva línea)
\r	Retorno de carro (carriage **r**etum)
\t	Tabulación horizontal (**t**ab - tabulador)
\v	Tabulador vertical
\"	Comillas rectas
\'	Apóstrofe
\\	Barra invertida (back slatch)
\0	Nulo
\a	Alerta (sonido)

Observaciones:

- Los seis primeros caracteres son de control.

- Los tres siguientes: \", \' y \\ permiten utilizar los caracteres: ", **'** y ****; que son utilizados como delimitadores de cadenas, de caracteres e identificadores de caracteres.

- El caracter nulo '\0' corresponde al número ASCII 0, que también se puede representar por el caracter nulo '\0'.

- El caracter '\a' es un sonido de alerta.

  También estos caracteres deben encerrarse entre apóstrofes, por ejemplo: '\n', '\b', etc.

2. **Número de código ASCII**. Este número indica el caracter correspondiente, como se muestra en la Tabla 2.3; para esto dicho número de código debe escribirse en octal o en hexadecimal, luego de una barra invertida y todo encerrado entre apóstrofes, para identificarlo como un único caracter.

Tabla 2.3. Caracteres con número de código ASCII

CODIGO	SIGNIFICADO
\N	Constante octal (N es una constante octal)
\xN	Constante Hexadecimal (xN es una constante hexadecimal)

Por ejemplo: '\07' y '\0x20', es el sonido o pito, y el espacio en blanco; respectivamente.

2.3.2. Constantes Enteras

Especifican números enteros, es decir sin parte fraccionaria, por ejemplo: 10, 50 y -100.

Estas constantes pueden ser también:

Octal Una constante octal comienza por el caracter '**0**' (cero), por ejemplo: 020 (16 en decimal).

Hexadecimal Una constante hexadecimal comienza por los caracteres **0x**, por ejemplo: 0x20 (32 en decimal).

Usando un sufijo se puede especificar de forma precisa el tipo de una constante entera, como se muestra en la Tabla 2.4.

Tabla 2.4 Constantes Enteras

SUFIJO	TIPO DE DATO
U	el número será **unsigned**
L	el número será **long**

A continuación se presenta ejemplos de constantes enteras:

CONSTANTES	TIPO DE DATO
35000U	unsigned int (2 bytes)
35000L	long int (4 bytes)

2.3.3. Constantes de Punto Flotante

Requieren del punto decimal seguido de los números de la parte fraccionaria, también permite el uso de la notación científica, por ejemplo: -11.14, 1.56e-6, respectivamente.

Usando un sufijo se puede especificar de forma precisa el tipo de una constante de punto flotant, como se muestra en la Tabla 2.5.

Tabla 2.5. Constantes Punto Flotante

SUFIJO	TIPO DE DATO
F	el número será **float**
L	el número será **long doble**

A continuación se presenta ejemplos de constantes de punto flotante:

CONSTANTES	TIPO DE DATO
-234F	float
1001.2L	long doble

NOTA: Por defecto, el compilador de lenguaje C escoge el tipo de dato compatible más pequeño que pueda albergar a una constante numérica entera. Por ejemplo, 40000 corresponde al tipo **unsigned** y 100000 al tipo **long**.

La excepción es la constante en "punto flotante", que se asumen de tipo **double**.

También, un valor numérico dentro del rango del tipo caracter no puede ser almacenado como **char**, porque se mezclan los tipos.

2.3.4. Constantes Cadena

La cadena es otro tipo de constante, aunque el lenguaje C no tiene formalmente un tipo de dato cadena.

Una constante cadena es una secuencia de caracteres encerrada entre comillas. Por ejemplo:

"Esta es una cadena"

No deben confundirse las cadenas con los caracteres, por ejemplo:

"a" es una cadena.
'a' es un caracter.

2.4. IDENTIFICADORES

Son los nombres usados para referenciar las constantes simbólicas, variables, funciones, etiquetas y otros objetos definidos por el usuario.

La longitud de un identificador depende del compilador del lenguaje C, en es el sistema operativo Windows puede variar entre 1 y 32 caracteres. Sin embargo, si el identificador está envuelto en el proceso de enlazado, los seis primeros caracteres son significativos; caso contrario, si no está envuelto, los 32 primeros caracteres son los significativos.

El primer caracter siempre debe ser una letra del alfabeto inglés o el carácter de subrayado, mientras que los caracteres siguientes pueden ser letras, números o símbolos de subrayado. Por ejemplo, son identificadores: **cont**, **num_total** y **curso1**.

Un identificador puede ser más largo que el número de caracteres significativos reconocibles por el compilador, pero los caracteres que pasan este límite serán ignorados.

En lenguaje C las letras minúsculas y mayúsculas se tratan como distintas. Así **MANUAL** y **manual** son identificadores distintos.

Un identificador no puede ser igual a una palabra clave del lenguaje C, tampoco debe tener el mismo nombre que alguna función ya escrita o que se encuentre en la biblioteca del lenguaje C. (SCHILDT, 1994)

2.5. DECLARACIÓN DE VARIABLES

Todas las variables en lenguaje C deben ser declaradas antes de poder ser usadas, la forma general de la "**Sentencia de declaración**" es: (SCHILDT, 1994)

tipo lista_variables;

donde:

- **tipo**, debe ser un tipo de dato válido de lenguaje C con cualquier modificador.
- **lista_variables**, consiste en uno o más nombres de identificadores separados por comas.
- **Punto y coma (;)**, la sentencia de declaración finaliza siempre con un punto y coma.

Por ejemplo:

int perno, tuerca;
unsigned int alumnos, jugadores;
char oro, plomo, plata;
float blanco, rojo;

Las variables de un mismo tipo se pueden declarar en una sola sentencia o en varias para poner comentarios.

Existen tres sitios donde se pueden declarar variables, las mismas que toman el nombre de ese sitio; a continuación se detalla cada uno: (BECERRA, 1991)

a) Variables Locales

Son las variables que se declaran dentro de un "bloque de código", que generalmente son las funciones. Estas variables pueden ser referenciadas solo por sentencias que estén dentro del bloque en el que han sido declaradas. Por lo que las variables locales no son conocidas fuera de su propio bloque de código.

Las variables locales solo existen mientras se ejecuta el bloque de código en el que ha sido declarada. Ya que la variable local se crea en el "segmento de pila" al entrar en el bloque de código y se destruye al salir de él.

Se acostumbra a declarar las variables al principio de una función, esto se hace para que sea más fácil saber que variables se usan en dicha función. Sin embargo, se puede declarar variables locales en cualquier bloque de código.

Las ventajas de declarar una variable local en un bloque son:

- Que se dispondrá de la memoria necesaria solo cuando se necesita la variable.

- Ayuda a prevenir efectos secundarios no deseados, porque la variable no existe fuera del bloque en el que está declarada, esto impide que sea accidentalmente modificada.

Se puede inicializar una variable local con algún valor determinado. Este valor será asignado cada vez que se entre en el bloque de código en que la variable está declarada.

b) Parámetros Formales

Si la función usa argumentos, deben declararse las variables que van a aceptar los valores de esos argumentos; estas variables son los parámetros formales de la

función. Su declaración se hace tras el nombre de la función encerrado entre paréntesis.

Los parámetros formales se comportan como cualquier otra variable local de la función, por lo que también son dinámicas y se destruyen al salir de la función.

c) Variables Globales

Estas variables se conocen a lo largo de todo el programa y se usan en cualquier parte del mismo. Además, mantienen su porción de memoria durante toda la ejecución del programa, ya que se almacenan en la región de memoria de datos.

Las variables globales se crean al declararlas en cualquier parte fuera de una función, y pueden ser accedidas desde cualquier función luego de su declaración.

Sin embargo, lo mejor es declarar todas las variables globales al principio del programa.

Cuando una función tiene declarada una variable local del mismo nombre de una global, la función reconoce a la variable local y no reconoce a la variable global.

Las variables globales son muy útiles cuando se usan los mismos datos en varias funciones del programa. Pero el uso de una variable global hace que una función sea menos general, debido a que depende de algo que debe estar definido fuera de ella.

2.6. VARIABLES CADENAS DE CARACTERES

En lenguaje C no existe un tipo especial de variables cadenas. Una cadena de caracteres consiste en una serie de uno o más caracteres entre comillas. Por ejemplo:

"Es una cadena"

Las comillas rectas no forman parte de la cadena, solo sirven para delimitarla.

Una cadena en lenguaje C se almacena como un **arreglo** de tipo **char**. El ejemplo de la cadena anterior se muestra gráficamente en la Figura 2.1.

E	s		u	n	a		c	a	d	e	n	a	\0

Figura 2.1. Ejemplo de cadena

El caracter nulo '\0' indica el final de la cadena, por lo que el **arreglo** deberá siempre disponer de un elemento adicional de lo que se quiera almacenar. Una cadena es nula si el primer caracter es '\0'.

La declaración de un **arreglo** de caracteres es de la siguiente manera:

char nombre[35];

donde:

- **[]**, identifican la variable **nombre** como un arreglo.
- **35**, indica el número de elementos máximo del **arreglo**.
- **char**, indica el tipo de dato de cada elemento.

Observación: La cadena "x" no es igual que el caracter 'x', por las siguientes razones:

1. 'x' es de tipo **char**, y
   "x" es de tipo **arreglo** de **char**.
2. 'x' contiene un solo byte, y
   "x" contiene 2 bytes:

x		'x' como caracter.
x	\0	"x" como cadena.

2.7. INICIALIZACIÓN DE VARIABLES

Inicializar una variable consiste en otorgar un valor a dicha variable, en el momento que se compila el programa. Se inicializa la variable a la vez que se la declara.

La forma general de inicialización de una variable es: (SCHILDT, 1994)

tipo nombre_variable = constante;

Por ejemplo:

```
char ch = 'a' ;
int perros, caballos = 32;
float vacas = 34, cabras = 14;
```

Únicamente se inicializan las variables que tienen el valor luego del signo igual. Por ejemplo: ch = 'a', caballos = 32, vacas = 34 y cabras = 14; pero **perros** tiene un valor indeterminado si es una variable local.

2.8. CONSTANTES SIMBÓLICAS

Para definir constantes simbólicas se utiliza el preprocesador **#defìne**, el cual se añade al comienzo del programa. El uso de estas constantes hace que un programa sea más legible.

El preprocesador. Mira el programa antes de compilar el mismo (de aquí el término preprocesador), para cambiar las abreviaturas simbólicas (macros) por sus respectivas directivas, buscar los archivos necesarios y cambiar la condición de compilación.

Para indicar que es una sentencia de preprocesador ésta comienza con el caracter **#**, y se sitúan normalmente fuera de las funciones. Además, estas sentencias no terminan con punto y coma, porque cada sentencia comienza con el caracter **#** y siempre va una sentencia por línea. (SCHILDT, 1994)

Por ejemplo:

```
#define PI 3.1416
```

Al compilar el programa el valor 3.1416 será sustituido en cualquier lugar que aparezca el identificador PI. Este proceso se denomina "**sustitución en tiempo de compilación**", ya que al ejecutar el programa estarán previamente hechas todas las sustituciones.

Se acostumbra a escribir las constantes en letras mayúsculas para diferenciarlas de las variables que se escriben en minúsculas.

Es aconsejable crear constantes simbólicas, porque:

1. Su identificador da más información que un número.

2. Se puede modificar con facilidad su valor, porque esa modificación será realizada solamente en la declaración de la constante.

Por ejemplo:

```
#define PITA '\007'
#define OPCION 'S'
#define GANAR "Lo lograste"
#define NUMERO 10
```

```
#define ERROR 2e-6
```

En las declaraciones de las constantes no es necesario especificar su tipo, porque éste se establece de acuerdo al valor que toma la constante.

2.9. MODIFICADORES DE ACCESO

2.9.1. El modificador *const*

El modificador **const** ayuda a identificar las variables que nunca cambian de valor.

Se define una constante declarando la palabra **const** antes de las palabras clave para indicar el tipo: **int**, **float**, **double**, etc. Además, como una constante no puede ser cambiada, se debe inicializarla en su declaración.

Por ejemplo:

```
const int MAXIMO = 100, INTERVALO = 20;
const float PI = 3.1416;
```

En un programa se utilizan las constantes y las variables de la misma forma, la única diferencia es que los valores iniciales asignados a las constantes no pueden cambiarse.

Se pueden definir las constantes de dos formas: mediante la palabra clave **const** y mediante la directiva de compilación **#define**. En muchos programas la acción de estos dos métodos es esencialmente la misma, pero el uso de la palabra clave **const** tiene por resultado una "variable" cuyo valor no puede ser cambiado. (SCHILDT, 1994)

2.9.2. El modificador *volatile*

El modificador **volatile** ayuda a identificar a las variables que pueden cambiar inesperadamente debido a causas que escapan del control del programa, aunque podrían cambiarse por hardware.

Por ejemplo, declarar la variable **reloj** para que cambie su valor sin que el programa tenga constancia: el programa que contiene la variable **reloj** puede ser interrumpido por el reloj del sistema y a la vez la variable **reloj** cambiada. Por ejemplo, la declaración de la variable **reloj** sería:

```
volatile int reloj;
```

Se debe declarar un dato **volatile** si fuera un registro de dispositivo direccionado sobre memoria o un dato compartido por procesos separados, como es el caso de un entorno operativo multitarea. (SCHILDT, 1994)

2.10. COMENTARIOS

En lenguaje C todos los comentarios ***empiezan*** con el par de caracteres **/*** y ***terminan*** con ***/**. El compilador ignora cualquier texto entre los símbolos de comentario inicial y final. Por ejemplo:

/* Este es un ejemplo.*/

Los comentarios pueden colocarse en cualquier lugar del programa, siempre y cuando no aparezcan en la mitad de una palabra reservada o de un identificador. Aunque no interesa poner un comentario en mitad de una expresión porque oscurece el significado. Además, los comentarios no se pueden anidar, es decir un comentario no puede contener otro comentario.

Se deben incluir comentarios cuando: se necesite explicar una expresión y resaltar variables, constantes, sentencias o bloques de códigos. Todas las funciones se recomienda tener un comentario al principio, que diga qué función es, cómo se llama y qué devuelve. (BECERRA, 1991)

2.11. OPERADORES

Operador. Es el que realiza una acción sobre uno o más datos.

Operando. Es sobre lo que opera el operador: los datos.

A más del operador de asignación, el lenguaje C define cuatro clases de operadores:

- Aritméticos.
- Relacionales.
- Lógicos.
- A nivel de bits.

2.11.1. Operador de Asignación

El operador de asignación (**=**) se puede usar en cualquier expresión válida y consiste en dar valores a las variables. La forma general de una "**Sentencia de asignación**" es: (SCHILDT, 1994)

nombre_variable = expresión;

donde:

- **expresión**, puede ser una constante, una variable, o la unión de constantes y variables a través de operadores, o una llamada a función que toma un valor.

- **nombre_variable**, debe ser una variable o un puntero, que es el destino donde se asigna el valor de la **expresión**.

Por ejemplo:

```
int bmw;
bmw = 2002;
```

En la variable entera **bmw** se asigna el valor 2002.

El lenguaje C acepta las múltiples asignaciones que permite asignar a muchas variables el mismo valor con una sola sentencia, y se realiza de derecha a izquierda. Por ejemplo, las siguientes variables toman el mismo valor, que es 100:

```
jane = tarzan = chita = 100;
```

NOTA: No se puede realizar asignaciones a las variables cadenas, porque sus nombres son punteros.

2.11.2. Operadores Aritméticos

Los operadores aritméticos en lenguaje C operan de izquierda a derecha como en la mayoría de lenguajes, y pueden aplicarse a casi todos los tipos de datos predefinidos en lenguaje C.

En la Tabla 2.6 se muestra los operadores aritméticos, con la correspondiente acción.

Tabla 2.6. Operadores Aritméticos (SCHILDT, 1994)

OPERADOR	ACCIÓN
-	Resta o unario de signo
+	Suma
*	Multiplicación
/	División
%	Residuo
++	Incremento
--	Decremento

Los primeros cinco operadores son binarios y los operadores **-**, **++** y **--** son unarios. Las características generales de estos operadores se indican a continuación:

- **Los operadores**: **-**, **+** y * operan igual que en el álgebra.

- **La división** (/). Funciona de 2 maneras:

  a) La división entre números en **punto flotante** da un resultado en **punto flotante**.

  b) La división entre **enteros** produce un resultado **entero**, eliminando la parte fraccionaria.

- **El residuo** (%). Determina el residuo de la división entera, utilizándose solo para números enteros. Por ejemplo:

  13 % 5 ,da un resultado de 3
  -2 % 6 ,da un resultado de -2
  5 % -7 ,da un resultado de 5

- **Incremento** (**++**). Es un operador unario que incrementa uno al valor de su operando entero, que debe ser una variable. Es decir, actualiza el valor de la variable incrementándolo en 1. Existen 2 tipos:

  a) **Prefijo**. El **++** aparece antes del operando, con lo que el lenguaje C lleva a cabo la operación de incremento antes de utilizar el valor del operando.

  b) **Sufijo**. El **++** aparece después del operando, con lo que el lenguaje C utiliza su valor antes de realizar la operación de incremento.

  Por ejemplo, si se escribe:

  ```
  x = 10;
  y = ++x;
  ```

  se asigna a **y** el valor de 11.

  Pero, si se escribe:

  ```
  x = 10;
  y = x++;
  ```

  y toma el valor de 10.

  En ambos casos **x** queda con el valor de 11.

Ventajas:

1. Hace los programas más compactos, elegantes y fáciles de leer.

2. Genera un código compilado más rápido y eficiente, ya que su estructura se asemeja más al código de máquina real, que en cualquier otro lenguaje de alto nivel.

- **Decremento** (**--**). Este operador se comporta en forma similar al operador incremento, pero éste decrementa en 1 al valor de la variable.

NOTAS:

- Para las cuatro operaciones aritméticas, la operación ***mixta*** entre operandos de tipo entero y punto flotante da un resultado en **punto flotante**. Es decir, el entero que intervenga en esta operación se convierte en punto flotante antes de realizar la operación.

- Cuando se utiliza el operador de incremento o decremento en solitario en una sentencia, no importa la modalidad escogida. Por ejemplo, las siguientes sentencias son equivalentes:

  ++talla;
  talla++;

Abreviaturas. Son operadores de actualización de la variable, que simplifican la escritura de ciertas sentencias de asignación. Las abreviaturas son una combinación de los operadores aritméticos binarios con el operador de asignación (=).

La forma general de las abreviaturas es:

variable operador = expresión;

donde:

- **variable**, se actualiza utilizando el valor de la expresión, de acuerdo a la operación indicada por el operador binario.

- **operador**, en este caso es un operador aritmético, también puede ser un operador lógico.

Por ejemplo:

conta += 3 * y + 12;	equivale a	conta = conta + (3 * y + 12);
debe -= 6;	"	debe = debe - 6;
factor *= 40;	"	factor = factor * 40;
tiempo /= 5;	"	tiempo = tiempo/ 5;

reduce %= 20;   "   reduce = reduce % 20;

Estos operadores producen un código de máquina más eficiente y compacto.

2.11.3. Operadores Relacionales

Los operadores relacionales en lenguaje C se emplean para hacer comparaciones entre valores del mismo tipo como: enteros, punto flotante y caracteres. Estos operadores operan de izquierda a derecha.

En la Tabla 2.7 se muestra los operadores relacionales.

Tabla 2.7. Operadores Relacionales (SCHILDT, 1994)

OPERADOR	ACCIÓN
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor 0 igual que
==	Igual que
!=	Diferente que

Observaciones de los operadores relacionales**:**

- Para la comparación de caracteres se emplea el código ASCII.
- Las cadenas de caracteres no se pueden comparar (se verá más adelante).
- Para comparar números de punto flotante se usan únicamente los operadores **<** y **>**, porque dos números de punto flotante no pueden ser necesariamente iguales, cuando son resultados de operaciones aritméticas, debido a errores de redondeo o truncamiento.

En lenguaje C es **verdadero** cualquier valor distinto de 0 y **falso** el valor de 0.

Las expresiones que utilizan los operadores relacionales devuelven el valor 1 en caso de **verdadero** y 0 en caso de **falso**.

Por ejemplo:

3 > 5   el resultado es 0
2 == 9   el resultado es 0
'x' > 0   el resultado es 1   /* Compara el ordinal de x con 0 */

2.11.4. Operadores Lógicos

Los operadores lógicos combinan dos o más expresiones de relación. En lenguaje C existen tres operadores lógicos, como se muestra en la Tabla 2.8.

Tabla 2.8. Operadores Lógicos (SCHILDT, 1994)

OPERADOR	ACCIÓN
&&	AND
||	OR
!	NOT

Los operadores lógicos producen un resultado **verdadero** (1) o **falso** (0), en la Tabla 2.9 se muestran las tablas de verdad para los operadores lógicos.

Tabla 2.9. Tablas de Verdad de los Operadores Lógicos

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

donde:

- **1**, es verdadero.
- **0**, es falso.

En lenguaje C se ***garantiza*** que las expresiones lógicas se evalúen de izquierda a derecha. También ***garantiza*** que tan pronto se encuentre un elemento que invada la expresión completa, cesa la evaluación de la expresión.

Por ejemplo, al evaluar la siguiente expresión:

```
numero != 0 && 12 / numero >= 0
```

Si **numero** tiene un valor de **0**, la subexpresión izquierda al operador && es falsa y por ende la expresión total es falsa, por lo que el resto de la misma no se evalúa, evitándose una división por cero. De igual forma se cumple para el operador ||; basta que el primer operando sea verdadero para obtener una verdad.

2.11.5. Operadores a Nivel de Bits

Dado que el lenguaje C se diseñó para sustituir al lenguaje ensamblador en muchas tareas de programación, éste cubre todas las operaciones que se pueden hacer en ensamblador.

Las operaciones a nivel de bits se refieren a la comprobación, asignación o desplazamiento de los bits individuales que componen un byte o una palabra de memoria, que a su vez corresponden a los tipos estándar del lenguaje C: **char** e **int** con sus variantes.

Las operaciones a nivel de bits no se pueden usar sobre los tipos: **float**, **double**, **long double** y **void**.

En la Tabla 2.10 se presentan los operadores a nivel de bits.)

Tabla 2.10. Operadores a Nivel de Bits (SCHILDT, 1994)

OPERADOR	ACCIÓN
&	AND
|	OR
^	OR exclusiva (XOR)
~	Complemento a 1 (NOT)
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Los operadores AND, OR, y NOT a nivel de bits están gobernados por las mismas tablas de verdad que sus equivalentes lógicos, pero trabajan bit a bit.

La tabla de verdad del operador XOR, se indica en la Tabla 2.11.

Tabla 2.11. La tabla de verdad del operador XOR

p	q	p ^ q
0	0	0
0	1	1
1	0	1
1	1	0

Las operaciones sobre bits son frecuentemente usadas en aplicaciones de controladores de dispositivos, tales como programas para modems, rutinas de archivos de disco y rutinas de impresora. Porque permiten enmarcar ciertos bits, por ejemplo, el bit de paridad.

A continuación se describe cada operador:

- La operación **AND** pone los bits a cero, ya que cualquier bit que esté a cero en un operando hará que el correspondiente bit de la variable se ponga a cero. Por ejemplo, 5 & 127 sería:

```
    00000101 (5)
&   01111111 (127)
    00000101
```

- La operación **OR** pone los bits a uno, ya que cualquier bit del operando que esté a uno hace que el correspondiente bit de la variable se ponga a uno. Por ejemplo, 128 | 3 sería:

```
    10000000 (128)
|   00000011 (3)
    10000011
```

- La operación **XOR** pone los bits a uno, cuando los bits de los operandos son distintos. Por ejemplo, 127 ^ 120:

```
    01111111 (127)
^   01111000 (120)
    00000111
```

- El operador **NOT** (complemento a 1), cambia el estado de cada bit en la variable especificada, o sea, los unos los pone en cero y los ceros en uno. Por ejemplo, ~ 123:

```
~   01111011 (123)
    10000100
```

- Las operaciones de desplazamiento, **>>** y **<<**, mueven todos los bits de una variable a la derecha o a la izquierda, respectivamente.

  - La expresión de desplazamiento a la derecha es:

    variable >> número de posiciones en bits

  - La expresión de desplazamiento a la izquierda es:

    variable << número de posiciones en bits

Un desplazamiento no es una rotación, ya que los bits que salen por un extremo se pierden, es decir, no se introducen por el otro extremo, ya que a medida que se desplazan los bits hacia un extremo se van rellenando con ceros por el extremo opuesto.

Las operaciones de desplazamiento de bits pueden utilizarse para llevar a cabo operaciones muy rápidas de multiplicación y división de enteros, así:

- Un desplazamiento a la izquierda equivale a una multiplicación por 2.
- Un desplazamiento a la derecha equivale a una división por 2.

Por ejemplo:

char x;	**x luego de la operación**	**valor de x**
x = 7;	00000111	7
x = x << 4;	01110000	112, multiplica por 2^4 = 16
x = x >> 2;	00011100	28, divide por 2^2 = 4

Por último, los operadores a nivel de bits son útiles cuando se decodifica la entrada a través de dispositivos externos, como Conversores D/A. También, se usan en rutinas para descifrado, cuando se quiere que el archivo aparezca ilegible; basta con llevar a cabo en él algunas manipulaciones a nivel de bits. Uno de los métodos más sencillos es el de complementar cada byte usando el complemento a uno para invertir cada bit del byte, por ejemplo, el doble complemento en un byte produce el mismo valor:

Byte original: 01011000
Complemento a 1: 10100111
Complemento a 1: 01011000 (Dato original)

2.11.6. Otros operadores

A continuación se describen siete tipos de operadores: (SCHILDT, 1994)

a) El operador ?:

Es un operador condicional que abrevia a la sentencia de control **if-else**, y se denomina "**expresión condicional**".

Este operador ternario toma la forma general:

Expresion1 ? Expresion2 : Expresion3

- Si **Expresion1** es verdadera (distinta de cero) la expresión total toma el valor de **Expresion2**.

- Si **Expresion1** es falsa (igual a cero) la expresión total toma el valor de **Expresion3**.

Por ejemplo, calcular el valor absoluto de un número entero:

```
X = ( y < 0 ) ? -y : y;
```

- Si y < 0, entonces **x** toma el valor de **-y**.
- Si y >= 0, entonces **x** toma el valor de **y**.

Entonces, el operador **?:** se usa cuando se tiene una variable que puede tomar dos valores posibles.

NOTA: Para mejor claridad del programa en lenguaje C, la condición de las sentencias va entre paréntesis.

b) Operadores & y *

Antes de describir estos operadores hacen falta los siguientes conceptos previos:

- **Un puntero.** Es la dirección de memoria de una variable de cualquier tipo, que se fija en el momento de la compilación.

- **Variable puntero.** Es una variable declarada para almacenar un puntero a su tipo especificado.

  Las variables que vayan a contener direcciones de memoria (punteros), deben declararse como punteros colocando un ***** delante del nombre de la variable. Por ejemplo:

  ```
  int *m, cont, q;
  ```

  donde: - **m** es una variable puntero a entero, mientras **cont** y **q** son variables de tipo **int**.

- **Tipo base del puntero**. Es el tipo de dato al que apunta un puntero, en este caso **int**.

  La "***variable puntero***" es una variable que mantiene la dirección del primer byte de un dato del *tipo base*, por lo tanto, un puntero a cualquier tipo de dato se

puede guardar en 2, 4 u 8 bytes (depende del compilador del lenguaje C y la arquitectura del computador).

Los punteros en lenguaje C tienen tres funciones básicas:

1. Pueden proporcionar una rápida forma para referenciar los elementos de un arreglo.

2. Permiten a las funciones de lenguaje C modificar los parámetros de llamada.

3. Dan soporte a las listas enlazadas y a otras estructuras dinámicas de datos.

Los operadores dirección e inderección se describen a continuación:

- **Operador de Dirección &**. Es un operador unario que devuelve la "**dirección**" de memoria del operando, que puede ser una variable de cualquier tipo. Por ejemplo:

  m = & cont;

  se asigna en **m** la dirección de memoria de la variable **cont** (**m** recibe la dirección **cont**), pero no tiene relación con el valor de **cont**.

- **Operador de Indirección ***. Es el complementario de **&**, es un operador unario que devuelve el "***contenido***" del operando, que debe ser un puntero. Es decir, da el valor de la variable ubicada en la dirección que se especifica. Por ejemplo:

  m = &cont;
  q =*m:

  q recibe el valor de la dirección **m**.

  Esto es lo mismo que asignar la variable **cont** a **q**:

  q = cont;

c) Operador de compilación sizeof

El operador sizeof es un operador unario en tiempo de compilación, que devuelve la longitud en bytes de la variable, o del especificador de tipo entre paréntesis. Por ejemplo:

float f;

- **Para la variable**: sizeof f , da un valor de 4
- **Para el tipo**: sizeof (int) , da un valor de 2

Técnicamente el valor devuelto por **sizeof** es de tipo **size_t**. Donde, síze_t es un tipo entero sin signo, definido por el estándar ANSI en el archivo de cabecera "***stdio.h***". Sin embargo, en la práctica se puede usarlo como si fuera un valor sin signo, lo mismo que **unsigned**.

sizeof ayuda a generar códigos portables que dependen del tamaño de los tipos de datos incorporados del lenguaje C, ya que **sizeof** detecta la longitud real de los datos.

d) La coma como operador

El operador coma encadena varias expresiones, donde la parte izquierda del mismo se evalúa siempre como **void**; esto significa que la expresión de la parte derecha se convierte en el valor de la expresión total, garantizándose que las expresiones separadas por este operador se evalúen de izquierda a derecha. Por ejemplo:

x = (y = 3, y + 1);

primero se asigna el valor **3** a **y**, luego se suma y con el valor 1, y por último se asigna el valor **4** a **x**.

Los paréntesis son necesarios debido a que el operador coma tiene menor precedencia que el operador de asignación.

Esencialmente, la coma produce una secuencia de operaciones cuando se utiliza en la parte derecha de una sentencia de asignación; el valor asignado a la variable correspondiente es el valor de la última expresión separada por comas.

e) Los Operadores punto y flecha

Los operadores punto (**.**) y flecha (**->**) referencian elementos individuales de las "estructuras" y de las "uniones".

Las "**estructuras**" y las "**uniones**" son tipos de datos compuestos que almacenan distintos tipos de datos, que pueden referenciarse bajo un solo nombre.

Los operadores punto y flecha se describen a continuación:

- **El Operador punto**. Se usa cuando se trabaja **directamente** con la "estructura" o la "union".

- **El operador flecha**. Se aplica cuando se usa un **puntero** a una "estructura" o una "union".

f) Los paréntesis y los corchetes como operadores

Los operadores paréntesis y corchetes se describen a continuación:

- **Los paréntesis ()**. Son operadores que aumentan la *precedencia* de las operaciones que contienen.

- **Los corchetes []**. Llevan a cabo la indexación de los elementos de un arreglo, mediante un índice.

2.11.7. Precedencia de operadores

La precedencia de operadores es el orden en el cual se evalúan los operadores. En la Tabla 2.12 se lista las precedencias desde la más alta a la más baja, entre todos los operadores. Tómese en cuenta que los operadores en la misma línea de la tabla tienen el mismo nivel de precedencia.

Tabla 2.12. Precedencia de los operadores (SCHILDT, 1994)

DESCRIPCIÓN	OPERADORES
---------	()[] -> .
unarios	! ~ ++ -- - (tipo) * & sizeof
multiplicativos	* / %
aditivos	+ -
desplazamiento	<< >>
relación	< <= > >=
relación	== !=
AND sobre bits	&
XOR sobre bits	^
OR sobre bits	|
AND lógico	&&
OR lógico	||
condicional	?:
asignación y abreviaturas	= += -= *= /= %=
coma	,

Observaciones:

- Los operadores: unarios, condicional, de asignación y abreviaturas, se evalúan de derecha a izquierda.

- El resto de los operadores que tienen el mismo nivel de precedencia son evaluados de izquierda a derecha.

Por supuesto, se puede utilizar paréntesis para alterar el orden de evaluación, ya que los paréntesis fuerzan a que una operación, o un conjunto de operaciones, tengan un nivel de precedencia mayor.

Cuando se utiliza operadores lógicos y de relación su resultado es siempre 1 o 0. Además, no se necesitan paréntesis porque los operadores lógicos tienen menor precedencia que los operadores de relación.

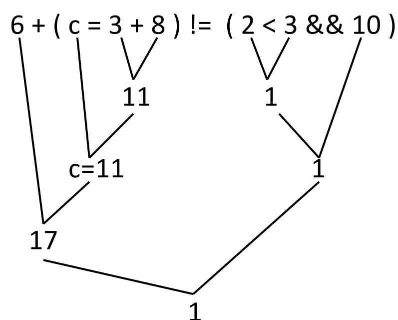
2.12. EXPRESIONES. EVALUACIÓN DE EXPRESIONES

Una expresión está constituida por la combinación de operandos y operadores de acuerdo a la sintaxis correspondiente, y además una expresión siempre tiene un valor.

La expresión más simple es un operando aislado, donde los operandos pueden ser constantes o variables o combinaciones de ambas. A partir del operando se pueden ir construyendo expresiones de mayor o menor complejidad. Por ejemplo, las siguientes expresiones son válidas:

a * (6 + c / d) / 20
9 > 3
x = ++q % 3

La evaluación de las expresiones se realiza de acuerdo a la precedencia preestablecida. Por ejemplo:



NOTA: El estándar ANSI estipula que las subexpresiones (entre paréntesis) de una expresión, pueden ser evaluadas en cualquier orden.

Por ejemplo:

x = (a + b) * (c + d);

En esta expresión podrá evaluarse primero (a + b) o (c + d), indistintamente.

Problemas. Existen problemas cuando la expresión tiene operaciones de incremento. Por ejemplo:

a=3;
x = (a++ - 8) * (a - 6);

- Al evaluarse primero el paréntesis (a++ - 8):

```
x = (a++ - 8) * (a - 6);
      \   /  a=4  \ /
       -5          -2
         \        /
            10
```

- Al evaluarse primero el paréntesis (a - 6):

```
x = (a++ - 8) * (a - 6);
      \   /  a=3  \ /
       -5          -3
         \        /
            15
```

Obteniéndose diferentes valores, por ello se recomienda no usar el operador incremento en expresiones de este tipo.

2.13. CONVERSIONES DE TIPOS

Cuando en una expresión se mezclan constantes y variables de distintos tipos, todas ellas se convierten a un tipo único.

Existen dos formas de conversión de tipos: Automática y Moldes (Casting), las cuales se describen a continuación:

2.13.1. Automática

Se realiza la conversión automáticamente de todos los operandos al mismo tipo.

Las reglas básicas son las siguientes: (SCHILDT, 1994)

a) La categoría de los tipos está dado por su tamaño de bytes; el de mayor valor tiene mayor categoría. A continuación se muestran las categorías de mayor a menor:

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

b) El resultado de una "sentencia de asignación" se convierte al tipo de la variable a la que se asigna el valor de la expresión.

El proceso puede ser una "***promoción de tipo***" o una "***pérdida de rango***", según sea la categoría de la variable a asignar.

Por ejemplo, con las siguientes declaraciones:

```
int i;
char ch;
float f;
```

Al realizar las asignaciones:

```
ch =i;
f = ch;
```

- **i** se convierte de entero a caracter (pérdida de rango). Lo que algunos compiladores obligan a utilizar el casting correspondiente.
- **ch** se convierte de caracter a punto flotante (promoción de tipo).

En la "pérdida de rango" los bits más significativos se pierden, de acuerdo a la Tabla 2.13.

Tabla 2.13 Pérdida de Rango (SCHILDT, 1994)

TIPO DESTINO	**TIPO EXPRESIÓN**	**POSIBLE PERDIDA DE INFORMACIÓN**
signed char	char	si valor > 127, destino negativo
char	short int	8 bits más significativos
char	int	8 bits más significativos

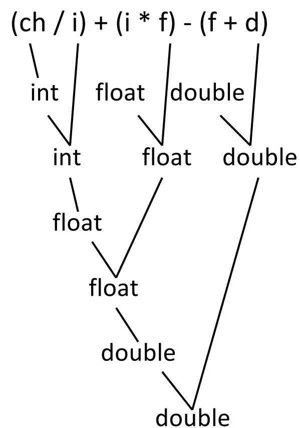
char	long int	24 bits más significativos
int	long int	16 bits más significativos
int	float	parte fraccional
float	double	precisión, resultado redondeado
double	long double	precisión, resultado redondeado

c) En cualquier operación que aparezcan dos tipos diferentes de datos, se convierte el operando de "menor categoría" al de "mayor", y el resultado de la operación será también del mismo tipo que los dos operandos. Este proceso se denomina "***promoción de tipo***".

Por ejemplo, si se tiene la siguiente declaración:

```
char ch;
int i;
float f;
double d;
```

El tipo de la expresión será:



2.13.2. Moldes (Casting)

Usualmente lo mejor es mantenerse apartado de las conversiones de tipo, en especial de las pérdidas de rango.

Es posible convertir una expresión a un tipo determinado, especificando el tipo concreto de conversión al que se desee; pero se debe tener en cuenta que el dato que se convierte al nuevo tipo, debe estar dentro del rango de este tipo.

La forma general de un **molde** es: (SCHILDT, 1994)

(**tipo**) expresión

donde:

- **tipo**, es uno de los tipos estándares del lenguaje C.
- **expresión**, es una expresión de cualquier tipo.

Por ejemplo, si se quiere asegurar que la expresión: x / y, se evalúe como de tipo **float**, se escribe:

(float) i / j

donde: **i** y **j** son enteros.

A menudo los moldes son considerados como operadores unarios.

PROBLEMAS PROPUESTOS

1) De los siguientes identificadores determinar ¿Cuáles son válidos y explicar por qué?.

a) registro1	d) return	g) nombre dirección
b) 1registro	e) $impuesto	h) nombre_dirección
c) archivo_3	f) nombre	i) 123-45-6789

2) Si se tiene un programa enlazado, determinar ¿Cuáles de los siguientes pares de nombres de identificadores se consideran idénticos y cuáles diferentes?.

a) nombre, nombres	e) lista1, lista2
b) direccion, direcciones	f) respuesta, RESPUESTA
c) identificador1_, identificador2	g) num1, num_1
d) numero, Numero	h) promedio, promedios

3) Determinar ¿Cuáles de los siguientes valores numéricos son constantes válidas?. Si una constante es válida, especificar si es entera o punto flotante. Especificar también la base en que está escrita cada constante entera válida.

a) 0.5	e) 12345678	i) 018CDF
b) 27,822	f) 12345678L	j) OXBCFDAL
c) 9.3e12	g) 08	k) 0515
d) 0x87e3ha	h) 9.3e-12	l) 0.8e8

4) Determinar ¿Cuáles de los siguientes caracteres son constantes válidas?.

a) 'a'	d) '\n'	g) '\\'
b) 'xyz'	e) '$'	h) '\n'
c) '\a'	f) '\0'	i) '\052'

5) Determinar ¿Cuáles de las siguientes cadenas de caracteres son constantes válidas?.

a) '8:15 P.M.'
b) "Rojo, Blanco y Verde"
c) "Nombre:
d) "Capítulo 3 (cont\'d) "
e) "1.3e-12"
f) "28018 QUITO"
g) "El Profesor dijo, "Por favor, no se duerman en clases"
h) "Este es un \"ejemplo\""
i) "\\¿Qué es esto?\\"

6) Escribir las declaraciones apropiadas para cada grupo de variables y cadenas.

a) Variables enteras: p, q
Variables de punto flotante: x, y, z
Variables de caracter: a, b, c

b) Variables de punto flotante: raiz1, raiz2
Variable entera larga: cont
Variable entera corta: indicador

c) Variable entera: indice
Variable entera sin signo: cliente_num
Variable de doble precisión: bruto, impuesto, neto

d) Variable de caracter: actual, ultimo
Variable entera sin signo: contador
Variable de punto flotante: error

e) Variable de caracter: primero, ultimo
Cadenas de caracteres de 80 elementos: mensaje, curso
Cadena de caracteres de 30 elementos: nombre

7) Escribir declaraciones apropiadas y asignar los valores iniciales dados para cada grupo de variables y cadenas.

a) Variables de punto flotante: a = -8.2, b = 0.005
Variables enteras: x = 129, y = 87, z = -22
Variables de caracter: c1 = 'w', c2 = 'g'

b) Variables de doble precisión: d1 = 2.88e-8, d2 = -8.4e5
Variables enteras: u = 711 (octal), v = fff (hexadecimal)
Variables enteras: i = 1000 (entera larga), j = 1 (entera)

c) Variable de entero largo: grande = 123456789
Variable de doble precisión: c = 0.33333333
Variable de caracter: eol = caracter de nueva línea

d) Cadena de caracteres: error = "ERROR, no hay memoria"
Cadena de caracteres: aviso = "El archivo está lleno\n"
Cadena de caracteres: mensaje = "\n\"FIN\" de entrada\n"

8) Escribir declaraciones apropiadas y asignar los valores iniciales dados para cada grupo de variables.

a) Variables de caracter: c1 = 'A' , c2 = 'B' , pito = caracter alarma

b) Variables enteras: u = 711 (octal), v = abcd (hexadecimal), w = 123, x = 45, y = -390, grande = 123456789 (largo).
c) Variables de punto flotante: a = -1.2, b = 0.005, c = 3.1416
d) Variables de doble precisión: d1 = 1.22e-4, d2 = -2.3e6, c = 0.66666

9) Escribir una definición apropiada para cada una de las siguientes constantes simbólicas:

	Constante	Texto
a)	FACTOR	-18
b)	ERROR	0.0001
c)	BEGIN	{
d)	END	}
e)	NOMBRE	"Anita"
f)	EOLN	'\n'
g)	COSTO	"$19.95"
h)	PI	3.1416
i)	FIN	"Fin del programa\n"
j)	MENSAJE	"No se puede abrir el archivo\n"

10) Indicar ¿Cuáles de los siguientes literales son verdaderos y cuáles son falsos?. Explique sus respuestas.

a) Los siguientes identificadores de variables son todos nombres válidos: _sobre_bar_, m928134, t5, j7, aqui_sales, informe_total, a, b, c, z, z2.
b) Los siguientes identificadores de variables son todos nombres inválidos: 3g, 87, 67h2, h22, 2h.
c) Los operadores de lenguaje C se evalúan de izquierda a derecha.
d) Una expresión aritmética válida en lenguaje C que no contenga paréntesis se evalúa de izquierda a derecha.

11) Identificar los errores en cada uno de los siguientes enunciados, pudiera existir más de un error por cada enunciado. Luego corregir esos errores:

a) */ Programa para determinar el largo de 3 enteros /*
b) Primer_numero + Segundo_numero = Suma_de_numeros
c) largo == (numero => largo) ? numero :;
d) igual = (x = y) ? x : y;
e) largo == (numero => largo) ? numero 2:;
f) igual = (x = y) ? x : y;

12) Escribir solo un enunciado de lenguaje C que cumpla con cada uno de los siguientes literales:

a) Asignar el producto de las variables **b** y **c** a la variable **a**.
b) Declarar que un programa realiza un reporte alfabético de una nómina de estudiantes, es decir, utilizar un comentario para documentar el programa.

c) Expresar la siguiente ecuación matemática en forma computacional:

$$y = \frac{x^3 - \frac{3}{5}x^2 + \frac{4}{3}x + 1}{x^3\left(\frac{4}{5} + 7\right)x^2 + \frac{1}{2}x}$$

d) La variable x tiene el valor de 20, ésta debe ser actualizada multiplicándola por el valor de y; a su vez y se obtiene al asignarle el valor inicial de 10 y luego incrementándolo en 50.

13) Explicar el propósito de cada una de las siguientes expresiones:

a) a - b && a + b
b) (a % 5) == 0
c) a >= t
d) --a
e) d = a * (b + c)
f) a * (b + c)
g) a < (b / c)
h) (a = 6) * b

14) ¿Cuáles de las siguientes expresiones del lenguaje C, correspondientes a la ecuación $y = ax^3 + 7$, son correctas?.

a) y = a *(x * x * x + 7)
b) y = a * x * x *(x + 7)
c) y = (a * x) * x * (x + 7)
d) y = (a * x) * x * x + 7
e) y = a * (x * x * x) + 7
f) y = a * x * (x * x + 7)
g) y = a * x * x * x + 7

15) Declarar el orden de cálculo de los operadores en cada una de las siguientes expresiones, y mostrar el valor de **x** después de que se ejecute cada una de ellas.

a) x = 7 + 3 * 6 / 2 - 1;
b) x = 2 % 2 + 2 * 2 – 2 / 2;
c) x = (3 * 9 * (3 + (9 * 3 / (3))));

16) Considerando las siguientes declaraciones:

```
int a = 8;
int b = 3;
int c = -5;
```

Determinar el valor de cada una de las expresiones aritméticas, indicando el orden de cálculo:

a) a + b + c	d) a % b	g) a * b / c
b) 2 * b + 3 * (a - c)	e) a + c / b	h) a * (b / c)
c) a / b	f) a % (b * c)	i) (a % c) * b

17) Dadas las siguientes declaraciones:

```
float x = 1.0;
float y = 3.0;
float z = -2.0;
```

Determinar el valor de cada una de las expresiones aritméticas, indicando el orden de cálculo:

a) x + y + z	d) x % y	g) 2 *x / 3 * y
b) 2 * y + 3 * (x - z)	e) x / (y + z)	h) 2 * x / (3 * y)
c) x / y	f) (x / y) + z	i) x + y / z

18) Considerar las declaraciones:

```
char c1 = 'A' ;
char c2 = '5' ;
char c3 = '?' ;
```

Determinar el valor numérico de las siguientes expresiones, basándose en el conjunto de caracteres ASCII.

a) c1	d) c3 + '#'	g) 3 * c2
b) c1 - c2 + c3	e) c1 % c3	h) '3' *c2
c) (c1 /c2) * c3	f) c2 – 2	i) c2 - '2'

19) Un programa en lenguaje C contiene las siguientes declaraciones:

```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

Determinar el tipo de datos de cada una de las expresiones mostradas a continuación:

a) i + c	d) ix + j	g) ix + c
b) x + c	e) i + x	h) s + c

c) (int) dx + ix   f) dx + x   i) s + j

20) Un programa en lenguaje C contiene las siguientes declaraciones y asignaciones iniciales:

```
int i = 8, j = 5;
float x = 0.005, y = -0.01;
char c = 'c' ,d = 'd';
```

Determinar el valor de cada una de las expresiones mostradas a continuación, utilizando para cada una de ellas los valores asignados inicialmente a las variables.

a) (3 * i - 2 * j) % (2 * d - c) *j != 6
b) 2 * ((i / 5) + (4 * (j - 3))) /c == 9
c) (i - 3*j)%(c+2*d)+5*(i + j)>'c'
d) -(i + j)/(x / y) % (i + j - 2) - (2 * x + y) == 0
e) ++j / 2 * x + (y == 0)
f) i++ % 2 * x + y == 0
g) --j - !(i <= j)
h) ++x * !(c == 9)
i) y-- + !(x > O)
j) i <= j | | (i > 0) && (j < 5)
k) c > d | | (i > 0) | | (j < 5)
l) x >= 0 && (x > y) && (i > 0) | | (j < 5)
m) x < y | | (x > y) && (i > 0) && (j < 5)
n) 2 * x >= 5 * j && i > j % 3 < i / j

21) Un programa en lenguaje C contiene las siguientes declaraciones y asignaciones iniciales:

```
int = 8, j = 5, k;
float x = 0.0005, y = -0.01, z;
char a, b, c ='c' ,d ='d';
```

Determinar el valor de cada una de las expresiones de asignación, utilizando para cada expresión el valor inicial asignado a las variables.

a) k = (i + j)
b) z = (x + y)
c) i = j = 1.1
d) k = (x + y)
e) k = c
f) z = i / j
g) a = b = d
h) z = k = x
i) k = z = x
j) i += 2
k) y -= x
l) x += 2
m) i /= j
n) i %= j
o) i += (j - 2)
p) k = (j == 5) ? i : j
q) k = (j > 5) ? i : j
r) z = (y >= 0) ? x : 0
s) z = (y >= 0) ? x : 0
t) a = (c < d) ? c : d
u) i -= (j > 0)? j : 0

22) Un programa en lenguaje C contiene las siguientes declaraciones y asignaciones iniciales:

```
int i = 8, j = 5;
double x = 0.005, y = -0.1;
char c= 'c' , d = 'd';
```

Determinar el valor de cada una de las funciones estándar, además, identificar el propósito de cada función y la librería estándar a la cual corresponde.

a) abs (i - 2 * j)
b) fabs (x + y)
c) isprint (c)
d) isdigit (c)
e) toupper (d)
f) cos (x + y)
g) islower (c)
h) ceil (x)
i) isupper (j) q)
j) exp (x)
k) log (x)
l) sqrt (x * x + y * y)
m) isalnum (10 * j)
n) isalpha (10 * j)
o) isascii (10 * j)
p) floor (x + y)
q) toascii (10 * j)
r) fmod (x, y)
s) tolower (65)
t) pow (x - y, 3.0)
u) sin (x - y)
v) strlen ("hola\0")
w) strchr ("Ho\0", 'e')
x) tan (x)

23) Considerando las siguientes declaraciones, evaluar las expresiones mostradas a continuación, indicando el orden de cálculo.

```
int i = 8, j = 5;
float x = 8, y = 4;
int *p;
float *q;

p = &i;
q = &x;
```

Utilizar los valores asignados inicialmente a las variables para cada expresión.

a) !(i -= (i> 0) ? j : 0) && (*p / j) I I (i = j = 1.1) && (i += 20, *p / j)
b) (*q > y) && (*q > 0 | | y < 5) * (--j % 2)

24) Si no existen errores en las siguientes expresiones, evaluarlas indicando el orden de cálculo. Considerar la declaración y asignación:

```
int i = 100, *ip;
ip = &i;
```

Utilizar los valores asignados inicialmente a las variables para cada expresión.

a) (i = *ip / sizeof (int), i += (i > 50)? 100 : 200, ++i * 10 - i + 1)
b) i++ % 2 && (10 > 5 && !(110 < i) || 3 <= 4) || !(i > 100) || 0

25) Un programa contiene las declaraciones y asignaciones iniciales:

```
int i = 4, j = 3;
float x = 0.1, y = -0.2;
char c = 'A', d = 'D';
```

Determinar el valor de cada una de las siguientes expresiones, desarrollando el proceso de evaluación. Utilizar para cada expresión los valores asignados inicialmente a las variables.

a) (3 * i - 2 * j) % (2 * d - c) + j != 6 - c > d + (i > 0) || (j < 5)
b) 2 * ((i / 5) + (4 *(j - 3))) - c == 9 + x < y + (x > y) && (i > 0)
c) (i -3 * j) % (c + 2 * d) * 5 *(i + j) > 'C'
d) -(i + j) / (x / y) % (i + j - 2) - (2 * x + y) == 0
e) ++j - 2 * x + (y == 0) + i++ % 2 * x + y == 0 + !(x > 0)

Capítulo 3

ENTRADA / SALIDA POR CONSOLA

3.1. INTRODUCCIÓN

La entrada y salida de datos se realiza a través de funciones de biblioteca, siendo un sistema de E/S el que ofrece un mecanismo flexible, a la vez que consistente, para transmitir datos entre dispositivos.

En lenguaje C existe E/S por consola y por archivo, ya que son mundos conceptualmente diferentes, pero técnicamente el lenguaje C hace poca distinción entre la E/S por consola y la E/S por archivo.

Las funciones de E/S por consola son aquellas que controlan la entrada por teclado y la salida a través de pantalla, que es la E/S estándar del sistema; aunque la E/S estándar puede ser dirigida a otros dispositivos. En general, cualquier función para ser usada en el programa, debe ser declarada como tal mediante el "prototipo de función".

"**El prototipo de función**" consiste en la declaración de la función, que permite que el lenguaje C lleve a cabo una comprobación de tipos y número de argumentos de la función, así como el tipo de dato que retorna la función. Para lo cual se declara el prototipo de la función al principio del programa.

Los prototipos de las funciones para E/S por consola que se estudiarán a continuación, se encuentran en el archivo de cabecera "***stdio.h***" (**st**andar **i**nput **o**utput).

Para incluir en el programa un archivo de cabecera, se utiliza el preprocesador de inclusión, **#include**. Por ejemplo, si se quiere incluir el archivo "***stdio.h***" en un programa así: (SCHILDT, 1994)

```
#include "stdio.h"        /* directorio  actual */
```

o

```
#include <stdio.h>        /* directorio predefinido */
```

3.2. ENTRADA/SALIDA POR CONSOLA CON FORMATO

Las funciones **printf()** y **scanf()** permiten comunicarse con el mundo exterior y realizar la E/S con formato, esto es, pueden escribir o leer datos de cualquier tipo, incluyendo caracteres, cadenas y números, en varias formas que pueden ser controladas.

3.2.1. Salida: *printf()*

3.2.1.1. Salida con Formato

Se llama salida con formato porque se especifica el tipo de dato de las constantes, variables o expresiones a imprimirse, y en qué forma deben mostrarse. Entonces, la impresión de una variable depende del tipo de dato del que se trate para mostrar. (SCHILDT, 1994)

El prototipo de la función **printf()** es:

int printf (char *cadena_control, lista_argumentos);

donde:

- **cadena_control**, está formada por dos tipos de elementos:

  1. El primer elemento, es el caracter que se mostrará en pantalla, tal como está escrito.

  2. El segundo elemento, contiene "**especificadores de formato**", que definen la forma en que se muestra la **lista_argumentos**, por ejemplo, los tipos: **int**, **char**, etc.

  Los especificadores de formato empiezan con el caracter '%' y va seguido por el "**código de formato**".

- **lista_argumentos**, son los valores de las distintas constantes, variables o expresiones a imprimirse. Pueden ser omitidas.

  Debe haber exactamente el mismo número de argumentos en la **lista_argumentos** que "**especificadores de formato**", y ambos deben coincidir en su orden de aparición de izquierda a derecha.

  Si un "**especificador de formato**" no tiene argumento, el resultado, en el mejor de los casos, se obtendrá datos sin sentido en la respectiva salida.

La función **printf()** retorna el número de caracteres escritos en la salida, o bien un valor negativo si se produce un error.

Los "**especificadores de formato**" para la función **printf()** en el lenguaje C ANSI, se muestra en la Tabla 3.1.

Tabla 3.1 Especificadores de formato para la función printf() (SCHILDT, 1994)

CÓDIGO	FORMATO
%c	Caracter
%d	Entero decimal con signo
%i	Entero decimal con signo
%e	Punto flotante en notación científica (doble longitud, imprime **e** minúscula)
%E	Punto flotante en notación científica (doble longitud, imprime **E** mayúscula)
%f	Punto flotante en notación decimal
%g	Usa **%e** o **%f**, el más corto (imprime **e**)
%G	Usa **%E** o **%f**, el más corto (imprime **E**)
%o	Entero octal sin signo
%s	Cadena de caracteres
%u	Entero decimal sin signo
%x	Entero hexadecimal sin signo (imprime letras minúsculas)
%X	Entero hexadecimal sin signo (imprime letras mayúsculas)
%p	Mostrar un puntero
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos hasta que aparezca **%n**.
%%	Imprimir el caracter **%**

Los formatos de impresión son los siguientes:

- **Impresión de un caracter**. Se usa el especificador **%c**.
- **Impresión de una cadena**. Se utiliza especificador **%s**.
- **Impresión de números**. Se realiza de la siguiente forma:

  - Para enteros decimales con signo, se utiliza los especificadores **%d** o **%i**.
  - Para enteros sin signo, se usa **%u**.
  - Para números de punto flotante en notación decimal se usa **%f**.
  - Para números de punto flotante en notación científica se utiliza **%e** y **%E**, estos números tienen la siguiente forma:

    x.ddddddE±yy

    ➘ Si se quiere imprimir la letra '**E**' en mayúscula, utilizar el formato **%E**.

➘ Si se quiere imprimir la letra '**e**' en minúscula, utilizar el formato **%e**.

Para los números de punto flotante de menor número de caracteres de **%f** o **%e**, se utiliza **%g** o **%G**. Si se usa **%G** imprime la letra '**E**' mayúscula; se imprimirá '**e**' minúscula si se utiliza **%g**.

Para enteros sin signo en formato octal o hexadecimal se utiliza **%o** y **%x**, respectivamente.

Para representar las letras '**A**' a '**F**' en sistema hexadecimal, se utiliza **%X** para mayúsculas y **%x** para minúsculas.

- **Impresión de una dirección**. Se utiliza el especificador **%p**. Por ejemplo, si **ejem** es una variable de tipo **int**, y se desea imprimir su dirección se procede así:

```
printf ("%p", &ejem);
```

- **Especificador %n**. Este especificador es diferente al resto, no imprime nada, ya que el parámetro que corresponde a este especificador debe ser un puntero a una variable, cuya dirección, después de la ejecución de **printf()**, tendrá el número de caracteres que se han impreso hasta el punto en el cual se encontró **%n**. Por ejemplo, como se muestra a continuación:

```
int cuenta;
printf ("El %nejemplo es demostrativo.\n", &cuenta);
printf("%d", cuenta);
```

Imprimirá:

El ejemplo es demostrativo.
3

%n se utiliza fundamentalmente para permitir al programa asignar formatos directamente.

Ejemplos:

➘ Las siguientes sentencias:

```
printf ("Los valores a y b son: %d y %f\n", 2, 3.1);
printf ("Los valores son: a = %d y b = %f", 2, 3.1);
```

Imprime en pantalla:

Los valores de a y b son: 2 y 3.100000
Los valores son: a = 2 y b = 3.100000

- La impresión de una frase no necesita especificadores de formato:

    printf ("Ingrese un dato");

- Por el contrario, si se desea imprimir tan sólo datos, la frase no es necesaria:

    printf ("%c %d", '$' , 23);

- Para imprimir el caracter '%' se utiliza %%, por ejemplo, imprimir **20 %:**

    printf ("%d %%", 20);

- Si se desea conocer el número de caracteres escritos en pantalla, se procede así:

    numero = printf ("Número de caracteres escritos");

    Por lo tanto, la variable **numero** tomará el valor 29.

- Para realizar una conversión de tipo de dato, se procede así:

    printf ("%c", 65);

    Esta sentencia imprime el caracter '**A**'.

3.2.1.2. Modificadores de Formato

Los modificadores se agregan a los especificadores para modificar su salida, con el objeto de alterar su significado, así: se puede especificar la longitud mínima de un campo, el número de decimales y la justificación por la izquierda.

Los modificadores de formato se colocan entre el símbolo ' %' y "**código de formato**".

1. **Especificador de longitud mínima de campo**. Hace la reservación de un espacio en la salida para la impresión de un dato.

    - Si el dato es menor a la longitud mínima de campo, se rellena la salida con espacios por omisión.

    - En el caso de que la cantidad a imprimir no quepa en el espacio asignado, se usará automáticamente un campo mayor.

Si se quiere rellenar con ceros, se coloca un cero antes del especificador de longitud de campo.

Por ejemplo:

printf ("%9d", 345);
Imprimirá:

						3	*4*	*5*

printf ("%09d", 345);
Imprimirá:

0	*0*	*0*	*0*	*0*	*0*	*3*	*4*	*5*

El modificador de longitud mínima de campo se utiliza generalmente, para crear tablas en las cuales las columnas aparezcan alineadas.

2. **El especificador de precisión**. Consiste en un punto seguido de un entero, de acuerdo al tipo dato al que se aplica. Los tipos de datos pueden ser:

a) **Punto Flotante**, es el que determina el número de posiciones decimales a imprimir.

Por ejemplo, "%10.4f", imprime un número de al menos 10 caracteres con cuatro posiciones decimales.

b) **Cadena de caracteres**, es el número máximo de caracteres que se han de imprimir.

Por ejemplo, "%8.12s", imprime una cadena de al menos 8 caracteres y no más de 12 caracteres de longitud. El especificador de precisión tiene prioridad sobre la longitud de campo.

c) **Entero**, es el que determina el número de dígitos que aparecerán por cada número a imprimir. Para alcanzar el número requerido de dígitos se añaden ceros por delante.

Por ejemplo, realizar las siguientes impresiones de los tipos de datos: punto flotante, entero y cadena.

```
/* PROG0301.C */

#incIude <stdio.h›
```

```
void main ()
{
   printf ("%.4f\n", 123.1234567);
   printf ("%3.8d\n", 1000);
   printt ("%10.15s\n", "Esto es un texto sencillo");
}
```

El programa imprimirá:

123.1235
00001000
Esto es un text

3. **Ajuste a la salida**. Por omisión, todas las salidas están ajustadas a la derecha de la longitud de campo si esta longitud es mayor que el dato a imprimir, y si es menor, no toma en cuenta la longitud de campo. Por ejemplo:

   printf ("%10.2f", 341.566);

   Imprimirá:

				3	4	1	.	5	7

   Para ajustar a la izquierda del campo, se coloca el signo menos (-) después del caracter '%'. Por ejemplo:

   printf ("%-10.2f", 341.566);

   Imprimirá:

3	4	1	.	5	7				

4. **Modificadores l y h**. Los modificadores de formato **l** y **h** se aplican a especificadores de formato **d**, **i**, **o**, **u** y **x**. Por ejemplo:

   %ld ,imprime un entero largo.
   %hd ,imprime un entero corto.
   %hud ,imprime un entero corto sin signo.

   El modificador l puede preceder a los especificadores de punto flotante **e**, **f** y **g**, los mismos que indican que se imprime un dato **double**.

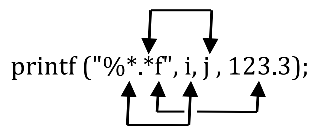
5. **Modificador #**. Si el modificador **#** precede a los especificadores de formato **g**, **f** o **e**, asegura que aparecerá un punto decimal, incluso si no hay dígitos decimales.

   Si el modificador **#** precede a los especificadores de formato **x**, se imprimirá un número hexadecimal con el prefijo **0x**.

6. **Modificador ***. Los especificadores de "longitud mínima de campo" y "precisión" pueden ser pasados como argumentos a **printf()**, en lugar de hacerlo mediante constantes. Es decir, se pasa los especificadores de formato como variables.

   Por ejemplo:

```
int i = 10, j = 4;
```



   Con esta función se tendrá una longitud mínima de campo de 10, una precisión de 4, y el dato que se imprime es:

		1	*2*	*3*	*.*	*3*	*0*	*0*	*0*

NOTA: Si se utiliza más de un modificador en el mismo sitio, el orden en que deberán ir será el siguiente: ajuste de salida, especificador de longitud mínima de campo, especificador de precisión, y modificadores de formato **l** y **h**.

3.2.2. Entrada: *scanf()*

3.2.2.1. Entrada con Formato

scanf() es la rutina de entrada por consola de propósito general, pues lee todos los tipos de datos que suministra el compilador, convirtiendo automáticamente al formato interno apropiado. Y luego de la entrada salta a la siguiente línea.

El prototipo de la función **scanf()** es: (SCHILDT, 1994)

int scanf (char* cadena_control, lista_argumentos);

donde:

- **cadena_control**, determina cómo se leen los valores de los datos en las variables a las que se hace referencia en la **lista_argumentos**.

  La **cadena_controI** consta de tres clases de caracteres:
  a) Especificadores de formato.
  b) Caracteres de espacio en blanco.
  c) Caracteres distintos de espacios en blanco.

  (Los caracteres de la cadena de control se explicaran más adelante)
- **lista_argumentos**, son punteros a variables, para fijar los mismos se debe colocar el nombre de la variable precedido por el operador de dirección **&**. Excepto, si se desea leer una variable de tipo cadena, porque su nombre de variable es un arreglo, el cual ya es un puntero.

La función **scanf()** retorna el número de datos leídos de las variables a las que se han asignado correctamente un valor. Si se produce un error, **scanf()** devuelve EOF (**E**nd **O**f **F**ile), que tiene el valor de **-1**, y se encuentra definido en el archivo de cabecera "***stdio.h***".

A continuación se muestra los caracteres de la "***cadena_control***"***:***

a) Especificadores de formato

Los especificadores de formato de entrada van precedidos por el caracter '%', e indican a **scanf()** que tipo de dato se va a leer a continuación. Los especificadores de formato se asocian de izquierda a derecha con los argumentos de la **lista_argumentos**.

Los especificadores de formato para la función **scanf()** se muestran en la Tabla 3.2.

Tabla 3.2. Especificadores de formato para la función scanf() (SCHILDT, 1994)

CÓDIGO	FORMATO
%c	Leer un caracter
%d	Leer un número entero decimal
%i	Leer un número entero decimal
%e	Leer un número de tipo punto flotante
%f	Leer un número de tipo punto flotante
%g	Leer un número de tipo punto flotante
%s	Leer una cadena
%o	Leer un número octal
%x	Leer un número hexadecimal
%p	Leer un puntero
%n	Recibe un valor entero igual al número de caracteres leídos

%u	Leer un número entero sin signo
%[]	Juego de inspección, define un conjunto de caracteres que pueden leerse utilizando **scanf()**

Los formatos de lectura son los siguientes:

- **Lectura de números**. Cuando la función **scanf()** lee números, termina la lectura en el momento que encuentra el primer caracter no numérico.

  - Para la lectura de números enteros decimales, se utiliza **%d** y **%i**.
  - Para leer un número en punto flotante, se utiliza **%e**, **%f** o **%g** sin importar la representación en notación decimal o científica.
  - Para leer un entero octal, se usa **%o**.
  - Para leer un entero hexadecimal, se usa **%x**, donde los valores de 'A' a 'F' pueden escribirse tanto en mayúsculas como en minúsculas.
  - Lectura de enteros sin signo, se utiliza **%u**.

- **Lectura de caracteres individuales**. Se usa **%c**, y también se leen como cualquier otro caracter el espacio en blanco, la tabulación y el salto de línea, que se utilizan como separadores de campo en otros tipos de datos.

- **Lectura de cadenas**. Se usa **%s**, que hace a **scanf()** leer caracteres hasta que se encuentre con un tipo de caracter blanco como un espacio, un salto de carro o una tabulación.

  Esto significa que no se puede utilizar **scanf()** para leer una cadena de caracteres o texto, porque con el primer espacio en blanco termina el proceso de lectura.

  Además, los caracteres leídos de una cadena son asignados al argumento de **scanf()**, que es un arreglo de caracteres, añadiéndose un caracter nulo al final de la cadena. Por ejemplo, para leer una cadena de caracteres se haría:

```
char cad[30];
scanf ("%s", cad);
```

- **Lectura de una dirección**. Se utiliza **%p**, que hace a **scanf()** leer una dirección en el formato propio de la CPU.

  Por ejemplo, leer un puntero para un caracter y luego mostrar su contenido.

```
/* PROG0302.C */
```

```
#include <stdio.h>

void main ()
{
   char *punt;
   printf ("Introduzca una dirección: ");
   scanf ("%p", &punt);
   printf ("En la posición %p está %c\n", punt, *punt);
}
```

Una salida del programa podría ser:

Introduzca una dirección: EF00 <ENTER>
En la posición EF00 está r

- **El especificador %n**. Indica a **scanf()** que asigne a la variable apuntada por el correspondiente argumento, el número de caracteres leídos desde la secuencia de entrada hasta el punto que aparezca **%n**.

- **Juego de inspección**. Un juego de inspección define un conjunto de caracteres que pueden leerse utilizando **scanf()**, estos caracteres leídos son asignados al correspondiente argumento que es un arreglo de caracteres, hasta que se encuentre con un caracter que no esté en el juego de inspección.

  Después de la ejecución de **scanf()**, el arreglo contendrá una cadena compuesta por los caracteres leídos y el caracter nulo.

  Un juego de inspección se determina poniendo una cadena con los caracteres que se van a leer entre corchetes, los corchetes deben ir precedidos de un signo de tanto por ciento.

  Por ejemplo, el siguiente juego de inspección indica a **scanf()** que lea solo los caracteres **x**, **y**, y **z**:

```
char cad[80];
scanf ("%[xyz]", cad);
```

  Las observaciones que se debe tener en cuenta en el formato "juego de inspección", son las siguientes:

- Si el primer caracter del conjunto es '**^**', se puede especificar un juego de caracteres que no está definido por el juego de inspección.

- Se puede especificar un rango de caracteres utilizando un guión. Por ejemplo, los caracteres desde **A** a la **Z** se especifican así: "%[A-Z] ".

- Se puede leer cualquier cadena de caracteres mediante el juego de inspección. Por ejemplo, si se desea leer un conjunto de caracteres hasta digitar ENTER, se procede así:

```
char cad [80];
scanf ("%[^\n]", cad);
```

Observaciones generales:

- Para diferenciar dos o más datos en el ingreso, éstos deben separarse por blancos o tabulados. Esto es válido en todo especificador de formato, excepto en el especificador **%c** porque lee todos los caracteres. Por ejemplo, si se quiere leer dos variables con los valores 2 para **numero** y 3.1415 para **promedio**, las sentencias serían:

```
int numero;
float promedio;
scanf ("%d%f", &numero, &promedio);
```

Y la entrada del teclado sería:

2 3.1415 <ENTER>

- Si se desea conocer el número de datos leídos, se asigna la función **scanf()** a una variable entera:

```
numero = scanf ("%d%c%f", &entero, &caracter, &real);
```

En la variable **numero** se almacena el valor que corresponde al número de datos válidos leídos.

b) Caracteres de espacio en blanco

Un caracter en blanco en la cadena de control, hace que **scanf()** salte uno o más caracteres en blanco de la secuencia de entrada.

En esencia, un caracter en blanco en la cadena de control hace que **scanf()** lea, pero no almacene, cualquier número de espacios en blanco hasta que encuentre el primer caracter que no sea de ese tipo. Por ejemplo, en las siguientes sentencias:

```
int i;
char ch;
```

```
scanf ("%d %c", &i, &ch);
```

no toma en cuenta el espacio en blanco entre los dos datos a leer.

Un caracter en blanco puede ser: un espacio, una tabulación o un caracter de salto de línea.

c) Caracteres distintos de espacios en blanco

Un caracter distinto de espacio en blanco en la cadena de control hace que **scanf()** lea y descarte los caracteres que coincidan con él en la secuencia de entrada.

Por ejemplo, la sentencia:

```
scanf ("%d*%d", &i, &j);
```

hace que **scanf()** lea un entero, lea y descarte un asterisco y posteriormente lea otro entero.

Pero si no se encuentra el caracter especificado, **scanf()** termina su ejecución.

3.2.2.2. Paso de Direcciones a *scanf()*

Todas las variables que se utilizan para recibir valores a través de **scanf()** se deben pasar por sus direcciones, esto significa que todos los argumentos de **scanf()** deben ser punteros a las variables que reciben los valores.

Por ejemplo:

```
scanf ("%d", &cuenta);
```

NOTA: Una cadena se asigna a un arreglo de caracteres y su nombre es la dirección del primer elemento del arreglo, es decir, el nombre del arreglo es un puntero.

Por lo tanto, para leer una cadena se haría:

```
char cad[30];
scanf ("%s", cad);
```

3.2.2.3. Modificadores de Formato

a) **Longitud de campo**. Este modificador determina una longitud máxima de campo, mediante un entero situado entre el % y el "código de formato", lo que limita el

número de caracteres leídos para ese campo. Si se encuentra un carácter blanco, la lectura del campo puede terminar antes de que se alcance su longitud máxima.

Por ejemplo, para leer hasta 20 caracteres en una **cadena** puede procederse así:

scanf ("%20s", cad);

Si la secuencia de entrada está formada por más de veinte caracteres, una llamada posterior a la función **scanf()** comenzará donde la anterior acabó.

Nótese que se obtendría lo mismo en la lectura de números. Por ejemplo, en la siguiente lectura:

scanf ("%2d %d", &i, &j);

al leer:

123456 <ENTER>

Almacenaría: **i=12** y **j=3456**.

b) **Lectura en enteros**. Para este propósito, los modificadores '**l**' y '**h**' se pueden usar junto con los formatos **d**, **i**, **o** y **x**. Por ejemplo: "**%ld**".

- Para leer un entero largo se pone '**l**' delante del "código de formato".

- Para leer un entero corto se pone '**h**' delante del "código de formato".

c) **Lectura en punto flotante**. Para este propósito, los modificadores '**l**' y '**L**' se pueden usar junto con los formatos **f**, **e** y **g**.

- Si se pone '**l**' delante de alguno de estos formatos, **scanf()** asigna los datos a una variable de tipo double.

- Si se usa '**L**' a los mismos formatos, se indica a **scanf()** que la variable que recibe el dato leído es de tipo **long double**.

3.2.2.4. Eliminación de Entradas

Se puede indicar a **scanf()** que lea un "**campo**" sin asignarlo a una variable. Esto se hace precediendo el código de formato de ese campo con el caracter '*****'. Por ejemplo, si en la siguiente sentencia:

scanf ("%d%*c%d", &x, &y);

se ingresa:

10, 10 <ENTER>

se lee la coma pero sin asignarla a ninguna variable. Es decir, omite la coma por tener el especificador de formato ***c**. Igual sucede para los otros especificadores.

La supresión de asignación es especialmente útil cuando sólo se necesita procesar una parte de lo que se introduce.

Por último, la lectura adecuada de un dato se recomienda realizar con su respectivo mensaje. Por ejemplo:

```
printf ("Ingrese un número entero: ");
scanf ("%d", &num);
```

3.3. ENTRADA/SALIDA DE CARACTERES POR CONSOLA

Esta entrada/salida permite solamente la lectura y escritura de caracteres por consola.

3.3.1. Entrada

Existe tres funciones para leer un caracter por consola: (SCHILDT, 1994)

a) getchar()

La función **getchar()** lee un caracter del teclado y espera un retorno de carro. Es decir, se guarda en el "buffer de entrada" (memoria temporal) el caracter hasta que se pulse la tecla ENTER.

El prototipo de la función **getchar()** es:

```
int getchar (void);
```

La función **getchar()** retorna un valor entero, siendo el byte de menor orden el que contiene el caracter introducido. Pero si se ha alcanzado el final del archivo, **getchar()** devuelve **EOF**.

La macro **EOF**, que tiene el valor -1, está definida en el archivo "***stdio.h***" y significa "**fin de archivo**".

b) getche()

La función **getche()** lee un caracter con *eco* sin esperar un retorno de carro. El eco de la tecla pulsada aparece automáticamente en la pantalla y la función **getch()** retorna el caracter pulsado.

El prototipo de **getche()** se encuentra en el archivo "***conio.h***" y es similar a la función **getchar()**.

c) getch()

La función **getch()** lee un caracter sin eco sin esperar un retorno de carro. Y no se muestra en la pantalla el eco del caracter introducido, además la función **getch()** retorna el caracter pulsado.

El prototipo de **getch()** se encuentra en el archivo "***conio.h***" y es similar a la función **getchar()**.

NOTA: Las funciones **getche()** y **getch()**, y el archivo "***conio.h***" no son parte del sistema de E/S del ANSI, solo existen en compiladores para el sistema opertivo Windows, en donde son muy utilizadas.

3.3.2. Salida

putchar()

La función **putchar()** escribe su argumento de tipo caracter en la pantalla en la posición del cursor.

El prototipo de la función **putchar()** es:

```
int putchar (int c);
```

La función **putchar()** tiene un argumento de tipo **char**, aunque se declara usando un parámetro entero, siendo el byte de menor orden el que se muestra en pantalla.

La función **putchar()** retorna el caracter escrito, o EOF si se ha producido un error. Por ejemplo:

```
char ch = '*';
putchar ('\007');      ,Sonido o pito.
putchar ('A');         ,Imprime el caracter 'A'.
putchar (ch);          ,Imprime el caracter '*'.
putchar (getchar ());  ,Imprime el caracter leído.
```

3.4. ENTRADA/SALIDA DE CADENAS DE CARACTERES POR CONSOLA

Esta entrada/salida permite solamente la lectura y escritura de cadenas de caracteres por consola.

3.4.1. Entrada: *gets()*

La función **gets()** lee una cadena de caracteres introducida por teclado, hasta que se pulse un retorno de carro o una marca **EOF** (ctrl Z), y la almacena en la dirección apuntada por su argumento de tipo puntero a caracter. El caracter de salto de línea no forma parte de la cadena, éste se transforma en un caracter nulo ('\0') para indicar el final de la cadena. Luego de realizar la entrada de la cadena, el cursor se posiciona en el inicio de la siguiente línea.

El prototipo de la función **gets()** es:

```
char *gets (char *cadena);
```

donde:

- **cadena**, es un arreglo de caracteres.

La función **gets()** retorna un puntero a la cadena, si se ejecuta correctamente, y un puntero nulo en caso de error.

No hay límite al número de caracteres que leerá **gets()**, por lo tanto, debe asegurarse de que el arreglo apuntado por cadena no supere sus límites.

Por ejemplo:

Realizar el ingreso de una cadena que acepte cualquier nombre (incluyendo espacios en blanco) de hasta 80 caracteres de largo:

```
/* PROG0303.C*/

/* Leer un nombre con la función gets (). */

#include <stdio.h>

void main ()
{
   char nombre[81];
   char *ptr;
```

```
    printf ("¿Hola, cómo te llamas?\n");
    ptr = gets (nombre);
    printf ("¿%s?. Ah! %s!\n", nombre, ptr);
}
```

Una posible salida sería:

¿Hola, cómo te llamas?
Pedro Pablo ***<ENTER>***
¿Pedro Pablo?. Ah! Pedro Pablo!

3.4.2. Salida: *puts()*

La función **puts()** escribe en la pantalla la cadena apuntada por su argumento. El caracter nulo se transforma en un caracter de salto de línea, posicionándose el cursor al principio de la siguiente línea.

El prototipo de la función **puts()** es:

int puts (char *cadena);

donde: - **cadena**, es un puntero a **char**.

La función **puts()** retorna un valor de salto de línea, en caso de error obtiene **EOF**.

La función **puts()** requiere mucho menos espacio y se ejecuta más rápidamente que **printf()**, porque **puts()** solo puede imprimir una cadena de caracteres y no imprime números ni realiza conversiones de formato.

Por ejemplo:

Imprimir cadenas de caracteres mediante la función **puts()**:

```
/* PROG0304.C */

# include <stdio.h>

# define DEF "Nací de una constante."

void main ()
{
    puts ("Soy un argumento de puts().");
    puts (DEF);
}
```

La salida sería:

Soy un argumento de puts().
Nací de una constante.

NOTA: Cuando se lee números y luego cadenas o caracteres, se presenta un error de entrada, pues la cadena o caracter no son leídos, porque al ingresar un dato numérico, el caracter nueva línea se queda en el buffer de entrada, que será leído en la siguiente lectura de una variable cadena o caracter.

PROBLEMAS PROPUESTOS

1) Identificar los errores en cada uno de los siguientes literales, pudiera existir más de un error por cada enunciado. Luego corregir esos errores:

   a) scanf ("d", valor);
   b) Scanf ("%d", un_entero);
   c) scanf ("%s", &nombre); /* nombre es una cadena de caracteres. */
   d) printf ("El producto de %d y %d es: %d" \n, x, y);
   e) printf ("Residuo de %d dividido por %d es\n", x, y, x % y);
   f) printf ("%d es igual a %d\n", x, y);
   g) printf ("La suma es %d\n, " x + y);
   h) printf ("El valor ingresado es: %d\n, &valor);

2) Indicar ¿Cuáles de los siguientes literales son verdaderos y falsos?. Explicar sus respuestas:

   a) La función **printf ("a = 5");** es un ejemplo típico de una expresión de asignación.
   b) La función **printf()** puede escribir constantes, variables y expresiones.
   c) La funcion **scanf ("Ingrese un número: %d", &numero);** puede ingresar un dato con un mensaje.
   d) La función **scanf ("%s", &nombre);** ingresa una cadena declarada como **char nombre[31];**.

3) Encontrar los errores en cada uno de los siguientes segmentos de programa. Explicar cómo pueden ser corregidos cada uno de ellos:

   a) printf ("%s\n", ' Feliz cumpleaños');
   b) printf ("%c\n", ' Hola');
   c) printf ("%c\n", ' Esta es una cadena');
   d) El siguiente enunciado deberá imprimir "Juan Pérez":
      printf (""%s "", "Juan Pérez");
   e) char dia[] = "Sabado";
      printf ("%s\n", dia[3]);
   f) printf (%f, 123.456);
   g) El siguiente enunciado deberá imprimir los caracteres ' o' y ' k' :
      printf("%s%s\n", ' o' , ' k');
   h) char s[10];
      scanf ("%c", s[7]);
   i) int ent;
      scanf ("%d", ent);

4) Interpretar el significado de la "cadena de control" de cada función **scanf()**:

```
scanf ("%12ld %5hd %15lf %15le", &a, &b, &c, &d);
scanf ("%101x %6ho %5hu %14lu",&a, &b, &c, &d);
scanf ("%12d %hd %15f %15e", &a, &b, &c, &d);
scanf ("%8d %*d %12lf %12lt", &a, &b, &c, &d);
```

5) Un programa en lenguaje C contiene la siguiente sentencia de declaración:

```
int i, j, k;
```

Escribir de forma adecuada una función **scanf()** que permita introducir los valores numéricos de **i**, **j** y **k**, asumiendo que:

a) Los valores de **i**, **j** y **k** son enteros decimales.
b) Los valores de **i**, **j** y **k** son enteros decimales y cada uno no excede los seis caracteres.
c) El valor de **i** es un entero decimal, **j** un entero octal y **k** un entero hexadecimal.
d) El valor de **i** es un entero decimal, **j** un entero octal y **k** un entero hexadecimal, y cada cantidad no excede de los ocho caracteres.
e) Los valores de **i** y **j** son enteros hexadecimales, y **k** un entero octal.
f) Los valores de **i** y **j** son enteros hexadecimales, **k** un entero octal, y cada cantidad tiene siete o menos caracteres.

6) Un programa en lenguaje C contiene las sentencias de declaración:

```
int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función **scanf()** que realice cada uno de los siguientes literales, suponiendo que todos los enteros se leerán como cantidades decimales:

a) Introducir los valores de **i**, **ix**, **j**, **x**, **u**, **dx**, **c**, **s**.
b) Introducir los valores de **i**, **j**, **x** y **dx**, suponiendo que cada cantidad entera no excede los cuatro caracteres, la cantidad en punto flotante no excede los ocho caracteres, y la cantidad en doble precisión no excede los 15 caracteres.

c) Introducir los valores de **i**, **ix**, **j**, **x** y **u**, suponiendo que cada cantidad entera no excede los cinco caracteres, el entero largo no excede más de doce caracteres y la cantidad en punto flotante no excede los diez caracteres.
d) Introducir los valores de **i**, **u** y **c**, suponiendo que cada cantidad entera no excede más de los nueve caracteres.
e) Introducir los valores de **c**, **x**, **dx** y **s**, suponiendo que la cantidad en punto flotante no excede los nueve caracteres, la cantidad en doble precisión no excede los 16 caracteres y el entero cono no excede los cuatro caracteres.

7) Un programa en lenguaje C contiene la siguiente sentencia de declaración:

```
char texto[80];
```

Escribir una función **scanf()** que lea una cadena de caracteres y se le asigne al arreglo **texto**:

a) Suponiendo que la cadena de caracteres no contiene ningún caracter de espaciado.
b) Suponiendo que la cadena de caracteres solo contiene letras minúsculas, espacios en blanco y un caracter de nueva línea.
c) Suponiendo que la cadena de caracteres solo contiene letras mayúsculas, dígitos, signo de dólar y espacios en blanco.
d) Suponiendo que la cadena de caracteres contiene cualquier caracter salvo el asterisco para indicar el final de la cadena.

8) Un programa en lenguaje C contiene las sentencias de declaración:

```
char a, b, c;
int u, v;
float x, y;
```

Suponiendo que se desea introducir el caracter '$' y se le asigna a **a**, que a **b** se le asigne el carcter '*', que a **c** se le asigne '@', que a **u** se le asigna el valor 12, que a **v** se le asigna el valor -8, que a **x** se le asigna el valor 0.011 y que a **y** se le asigna el valor -2.2×10^8.

Mostrar cómo se deben introducir los datos para cada una de las siguientes funciones **scanf()**, en caso de ser posible:

a) scanf ("%c%c%c", &a, &b, &c);
b) scanf ("%c %c %c", &a, &b, &c);
c) scanf ("%s%s%s", &a, &b, &c);
d) scanf ("%s %s %s", &a, &b, &c);
e) scanf ("%1s %1s %1s", &a, &b, &c);

f) scanf ("%d %d %f %f", &u, &v, &x, &y);
g) scanf ("%d %d %e, %e", &u, &v, &x, &y);
h) scanf ("%2d %2d %6f %6e", &u, &v, &x, &y);
i) scanf ("%3d %3d %8f %8e", &u, &v, &x, &y);

9) Un programa en lenguaje C contiene las sentencias de declaración:

```
int i, j, k;
int a = 0177, b = 055, c = 0xa8, d = 0xlff;
```

Escribir una función **printf()** para cada uno de los siguientes grupos de variables o expresiones, suponiendo que todas las variables representan enteros decimales:

a) i, j, k
b) (i + j), (i - k)
c) sqrt (i + j), fabs (i - k)
d) i, j y k, con una longitud de campo mínima de tres caracteres por cantidad.
e) (i + j) y (i - k), con una longitud de campo mínima de cinco caracteres por cantidad.
f) sqrt (i + j) y fabs (i - k), con una longitud de campo mínima de 9 para la primera cantidad y 7 para la segunda.
g) a, b, c, b
h) (a + b), (a - b)

10) Suponiendo que **i**, **j** y **k** son variables enteras, donde i representa una cantidad octal, **j** una cantidad decimal y **k** una cantidad hexadecimal. Escribir una función **printf()** adecuada para cada una de los siguientes enunciados:

a) Escribir los valores de **i**, **j** y k con una longitud de campo mínima de ocho caracteres por cada valor.
b) Repetir el literal a) con cada dato ajustado a la izquierda de su campo.
c) Repetir el literal a) con cada dato precedido de ceros o 0x.

11) Un programa en lenguaje C contiene la sentencia de declaración:

```
float x, y, z;
```

Escribir una función **printf()** para cada uno de los siguientes grupos de variables o expresiones, utilizando el especificador de formato f para cada cantidad de punto flotante:

a) x, y, z
b) (X + Y), (X - Z)
c) sqrt (x + y), fabs (x - z)

d) **x**, **y** y **z**, con una longitud de campo mínima de seis caracteres por cantidad.
e) (x + y) y (x - z), con una longitud de campo mínima de ocho caracteres por cantidad.
f) sqrt (x + y) y fabs (x - z), con una longitud de campo mínima de 12 para la primera cantidad y 9 para la segunda.
g) **x**, **y** y **z**, con una longitud de campo mínima de ocho caracteres por cantidad, con cuatro cifras decimales.
h) (x + y) y (x - z), con una longitud de campo mínima de ocho caracteres por cantidad, con tres cifras decimales.
i) sqrt (x + y) y fabs (x - z), con una longitud de campo mínima de 12 para la primera cantidad y 10 para la segunda. Presentar un máximo de cuatro cifras decimales en cada cantidad.
j) Repetir los literales anteriores utilizando el especificador de formato **e**.

12) Un programa en lenguaje C contiene las sentencias de declaración:

```
int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función **printf()** para cada uno de los siguientes enunciados. Supóngase que todos los enteros se desean presentar como cantidades decimales:

a) Escribir los valores de **i**, **ix**, **j**, **x**, **u**, **dx**, **c**, **s**.
b) Escribir los valores de **i**, **j**, **x** y **dx**, suponiendo que cada cantidad entera tiene una longitud de campo mínima de cuatro caracteres y cada cantidad de punto flotante se presenta en notación exponencial con un total de al menos 14 caracteres y no más de 8 cifras decimales.
c) Repetir el literal a) visualizando cada cantidad en una cadena.
d) Escribir los valores de **i**, **ix**, **j**, **x** y **u**, suponiendo que cada cantidad entera tiene una longitud de campo mínima de cinco caracteres y el entero largo tiene una longitud de campo mínima de 12 caracteres y la cantidad de punto flotante tiene al menos 10 caracteres con un máximo de 5 cifras decimales, no incluir el exponente.
e) Repetir el literal a) visualizando las tres primeras cantidades en una línea, seguidas de una línea en blanco y las otras cantidades en la línea siguiente.
f) Escribir los valores de **i**, **u** y **c** con una longitud de campo mínima de 6 caracteres para cantidad entera, y separar con tres espacios en blanco cada cantidad.

g) Escribir los valores de **j**, **u** y **x**, visualizando las cantidades enteras con una longitud de campo mínima de cinco caracteres, y la cantidad en punto flotante utilizando el especificador de formato **f**, con una longitud de campo mínima de 11 caracteres y máximo de 4 cifras decimales.
h) Repetir el literal g) con cada dato ajustado a la izquierda dentro de su campo.
i) Repetir el literal g) anteponiendo el signo (+ o -) de cada dato.
j) Repetir el literal g) rellenando el campo de cada cantidad entera con ceros.
k) Repetir el literal g) mostrando el punto decimal a la derecha del valor de **x**.

13) Un programa en lenguaje C contiene las sentencias de declaración:

```
int a = 12345, b = 0xabcd9, c = 077777;
int i = 12345, j = -13579, k = -2468;
long ix = 123456789;
short sx = -3333;
unsigned ux = 666;
```

Mostrar la salida resultante de cada una de las siguientes funciones **printf()**:

a) printf ("%d %d %d %ld %d %u", i, j, k, ix, sx, ux);
b) printf ("%3d %3d %3d\n\n %3ld %3d %3u", i, j, k, ix, sx, ux);
c) printf ("%8d %8d %8d\n\n %15ld %8d %8u", i, j, k, ix, sx, ux);
d) printf ("%8d %8d %8d\n\n %15ld %8d %8u", i, j, k, ix, sx, ux);
e) printf ("%08d%08d%08d\n\n %015ld%08d%08u", i, j, k, ix, sx, ux);
f) printf ("%d %x %o", a, b, c);
g) printf ("%3d %3x %3o", a, b, c);
h) printf ("%8d %8x %8o", a, b, c);
i) printf ("%-8d %-8x %-8o", a, b, c);
j) printf (" %8d %8x %8o", a, b, c);
k) printf (" %08d %#8x %#8o", a, b, c);

14) Un programa en lenguaje C contiene la declaración:

```
float a = 2.5, b = 0.0005, c = 3000;
```

Mostrar la salida resultante de cada una de las siguientes funciones:

a) printf ("%f %f %f", a, b, c);
b) printf ("%8f %8f %8f", a, b, c);
c) printf ("%8.4f %8.4f %8.4f", a, b, c);
d) printf ("%e %e %e", a, b, c);
e) printf ("%3e %3e %3c", a, b, c);
f) printf ("%8.2e %8.2e %8.2e", a, b, c);

g) printf ("%-8f %-8f %-8f", a, b, c);
h) printf ("%08f %08f %08f", a, b, c);
i) printf ("%#8f %#8f %#8f", a, b, c);
j) printf ("%g %g %g", a, b, c);
k) printf ("%#g %#g %#g", a, b, c);

15) Un programa en lenguaje C contiene la sentencia de declaración:

char a, b, c;

Realizar los siguientes literales:

a) Ingresar los valores de **a**, **b** y **c** mediante funciones **getchar()**.
b) Visualzar los valores ingresados de **putchar()**.
c) **a**, **b** y **c** mediante funciones
d) Repetir los literales a) y b) utilizando solo una función **scanf()** y **printf()**, respectivamente.

16) Un programa en lenguaje C contiene la declaración:

char a = 'A', b = 'B', c = 'C';

Mostrar la salida resultante de cada una de las siguientes funciones:

a) printf ("%c %c %c", a, b, c);
b) printf ("%c%c%c", a, b, c);
c) printf ("%3c %3c %3c", a, b, c);
d) printf ("%3c%3c%3c", a, b, c);
e) printf ("a = %c b = %c c = %c", a, b, c)

17) Mostrar lo que imprime cada una de las siguientes funciones. Suponiendo que x=2 y y=3:

a) printf ("%d", x);
b) printf ("%d", x + x);
c) printf ("x = ");
d) printf ("x = %d", x);
e) printf ("%d = %d", x + y, y + x);
f) z = x + y;
   printf ("%d\n", &z);
g) /* printf ("x + y = %d", x + y); */
h) printf ("\n");
i) ("*\n**\n***\n****\n*****\n");
j) puts ("\nEste es el fin");

k) putchar ('\n');

18) Mostrar lo que imprime cada una de las siguientes funciones. Si un literal es incorrecto, indicar por qué?.

a) printf ("%-10d\n", 10000);
b) printf ("%c\n", "Esta es una cadena");
c) printf ("%*.*li\n", 8, 3, 1024.987654);
d) printf ("%#o\n%#x\n%#e\n", 17, 17, 1008.83689);
e) printf ("%1d\n%+1d\n", 1000000, 10000O0);
f) printf ("%10.2E\n", 444.93738);
g) printf ("%10.2g\n", 444.93738);
h) printf ("%d\n", 10.987);

19) Un programa en lenguaje C contiene la declaración:

char texto[80];

Suponiendo que se le asignado a **texto** la siguiente cadena de caracteres:

Programar en lenguaje C es bonito.

Mostrar la salida resultante de cada una de las siguientes funciones:

a) printf ("%s\n", texto);
b) printf ("%18s\n", texto);
c) printf ("%.18s\n", texto);
d) printf ("%18.7s\n", texto);
e) printf ("%-18.7s\n", texto);

20) Un programa en lenguaje C contiene la sentencia de declaración:

char texto[80]

Introducir una cadena de caracteres de longitud no especificada mediante la función **gets()**. Luego escribir una función **puts()** Para visualizar el contenido de texto de los siguientes literales:

a) Toda la cadena en una línea.
b) Solo los ocho primeros caracteres.
c) Los ocho primeros caracteres precedidos de cinco espacios en blanco.
d) Los ocho primeros caracteres seguidos de cinco espacios en blanco.
e) Repetir todo lo anterior usando las funciones **scanf()** y **printf()** respectivamente.

21) Escribir solo una función que cumpla con cada uno de los siguientes literales:

a) Imprimir el mensaje "Ingrese 2 números".
b) Leer dos valores desde el teclado y colocar estos valores en las variables enteras **a**, **b** y **c**.
c) Imprimir el valor 3.1416 en una amplitud de campo de 10, ajustado a la izquierda, con dos decimales y rellenado con ceros.
d) Leer la siguiente cadena "Este es un ejemplo".

22) Escribir una función **printf()** o **scanf()** para cada uno de los siguientes literales:

a) Imprimir el entero sin signo 40000 justificado a la izquierda, en un campo de 15 dígitos y con 8 dígitos.
b) Leer un valor hexadecimal en la variable **hex**.
c) Imprimir 200 con y sin signo.
d) Imprimir 100 en forma hexadecimal precedido por **0x**.
e) Leer caracteres en la cadena **s**, hasta que se encuentre la letra 'p'.
f) Imprimir 1.234 en un campo de 9 dígitos, con ceros a la izquierda.
g) Leer la hora de la forma **hh:mm:ss** para almacenar las partes de la hora en las variables enteras **hora**, **minuto** y **segundo**. Omitir los dobles puntos (:) del flujo de entrada. Utilizando la "eliminación de entradas".
h) Leer la hora de la forma **hh:mm:ss** para almacenar las partes de la hora en las variables enteras **hora**, **minuto** y **segundo**. Omitir los dobles puntos (:) en el flujo de entrada. No utilizar la "eliminación de entradas".
i) Leer una cadena de la forma "caracteres" de la entrada estándar. Almacenar la cadena en la cadena de caracteres **s**. Eliminar las comillas del flujo de entrada.

23) En el siguiente programa escribir la salida en forma exacta:

```
#include "stdio.h"

#define TEXTO "Programar en lenguaje C"

void main ()
{
   char C1= 'A', C2= 'B';
   float f1 = l234.567;
   int i = 12345, j = 0xabcd, k = 0777;

   printf ("%s\n", "Bienvenido al programa");
   puts (TEXTO);
   printf ("%s", "es muy bonito");
   putchar ('\n');
```

```
    printf ("%-19.7s\n", "Universidad de las Fuerzas Armadas-ESPE");
    printf ("%-3c%5c\n", C1, C2);
    putchar (65);
    printf ("\n%08ld %08d %08u\n", i, 'B', 'A');
    printf ("%8d %8x %80\n", i, j, k);
    printf ("%#12f %012.2g %12.2e\n", f1, f1, f1);
}
```

24) Escribir exactamente lo que imprime el siguiente programa:

```
#include "stdio.h"

void main ()
{
    int i;
    char ch;
    float f;

    ch = 'A';
    i = (ch + 5) * 4 + 8;
    f = i * 2;
    printf ("%d %c %#f\n", i, ch, f);
    ch = (char) (i / 5);
    i = ch * 2 + 2;
    f = (float) i;
    printf ("%d %c %.2e\n", i, ch, f);
}
```

25) Escribir exactamente lo que imprime el siguiente programa:

```
#include "stdio.h"

#define PINTA "Emocionante emoción"

void main ()
{
    int cuenta;

    printf ("/%2.12ld/\n", 1234567);
    printf ("/%10.2e/\n", 1234.567);
    printf ("/%-22.7s/\n", PINTA);
    printf ("/%#x/\n", 336);
    printf ("/%c/\n", 70);
    printf ("\"Los valores: %-10d %%\", %n están dados\n", 60, &cuenta);
```

```
    printf ("%-2d\n", cuenta);
}
```

26) En algunos lenguajes de programación, las cadenas pueden ser introducidas entre apóstrofes o comillas. Escribir un programa que pueda leer cadenas de las tres siguientes formas: **Susana**, "**Susana**", y **'Susana'.** ¿Son los apóstrofes y comillas ignoradas o leídas por el lenguaje C, como parte de la cadena?.

27) Escribir un programa para imprimir el valor entero 12345 y el valor punto flotante 1.2345 en varios tamaños de longitud de campo, incluyendo campos que contiene menos dígitos que los valores mismos. Además, el programa debe imprimir el valor 100.453627 redondeado al dígito décima, centésima, milésima y decenas de millar más cercano.

28) Escribir un programa que ingrese una cadena desde el teclado y determine la longitud de la misma. Imprimir la cadena utilizando como ancho de campo dos veces su longitud para ajustarla a la izquierda.

29) Escribir un programa que introduzca dos números, para imprimir la suma, el producto, la diferencia, el cociente y el residuo de los dos números.

30) Escribir un programa que imprima en una misma línea los números desde el 1 al 4, utilizando los siguientes métodos:

    a) Utilizando una función **printf()** sin especificadores de formato.
    b) Utilizando una función **printf()** con cuatro especificadores de formato.
    c) Utilizando cuatro funciones **printf()**.

31) Escribir un programa que ingrese dos números, a continuación imprimir los números ingresados y el número mayor seguido por las palabras "es mayor"; si los números son iguales, que imprima el mensaje "Estos números son iguales". Utilizar el operador condicional.

32) Escribir un programa que lea el radio de un círculo y que imprima su diámetro, circunferencia y área. Utilizar el valor constante 3.14159 para **PI** como constante predefinida. Efectuar cada uno de estos cálculos dentro de la función o funciones **printf()**, utilizando el especificador de formato **%lf**.

33) Escribir un programa que lea 5 enteros y a continuación determine e imprima cuál es el entero mayor y menor del grupo. Utilizar el operador condicional.

34) Escribir un programa que lea un entero, determine e imprima si es par. Un número par es un múltiplo de 2 y cualquier múltiplo de 2 tiene un residuo de cero al ser dividido entre dos.

35) Escribir un programa que lea dos enteros, que determine e imprima si el primero es un múltiplo del segundo.

36) Escribir un programa que imprima los equivalentes enteros (ordinal de código ASCII) de las letras mayúsculas, las letras minúsculas, los dígitos y los símbolos especiales. Como mínimo, determinar los códigos ASCII de los siguientes caracteres: 'A', 'B', 'C', 'a', 'b', 'c', '0', '1', '2', '4', '5', '*', '+', '/' y ' ' (espacio en blanco).

37) Escribir un programa que se ingrese desde el teclado tres enteros diferentes, y a continuación imprima la suma, el promedio, el producto, el más pequeño y el más grande de estos números. Utilizar el operador condicional. El diálogo en pantalla deberá aparecer como sigue:

Ingrese tres enteros diferentes: 13 27 14 ***<ENTER>***
La suma es : 54
El promedio es : 18
El producto es : 4914
El pequeño es : 13
El grande es : 27

38) Escribir un programa que ingrese un número de cinco dígitos, para separarlo en sus dígitos individuales e imprimir los dígitos separados unos de otros mediante tres espacios. Por ejemplo, si se ingresa 42878 el programa debería imprimir:

4   2   8   7   8

39) Escribir un programa que calcule los cuadrados y los cubos de los números del 1 al 10, utilizando tabuladores para imprimir la siguiente tabla de valores:

número	cuadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	81
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

40) Escribir un programa que imprima con asteriscos un recuadro, un óvalo, una flecha y un diamante como sigue:

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

41) Desplegar un patrón cuadriculado utilizando ocho funciones **printf()**, y a continuación desplegar el mismo patrón con el mínimo posible de funciones **printf()**, como se muestra a continuación:

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

42) imprimir sus iniciales en letras de bloque hacia abajo de la página. Construir cada letra de bloque utilizando las letras como se representa a continuación:

```
EEEEEEEEE
E   E   E
E   E   E
E   E   E
E   E   E

PPPPPPPPP
    P   P
    P   P
    P   P
     P P

DDDDDDDDD
D       D
D       D
 D     D
   DDDDD
```

43) Realizar un programa que determine la cantidad de dinero futuro **F** (incluyendo la cantidad inicial) que se acumulará en una cuenta de un banco después de **n** años, si se conocen la cantidad depositada inicialmente **P** y el tanto por ciento anual de interés compuesto aplicado **i** (se expresa en tanto por uno). La fórmula para calcular es:

$F = P (1 + i)^n$

44) Realizar un programa que calcule el área y volumen de una esfera y cilindro, utilizando las siguientes fórmulas:

Esfera	Cilindro
$A = 4\pi r^2$	$A = 2\pi rh + 2\pi r^2$
$V = 4/3\ \pi r^3$	$V = \pi r^2 h$

45) Realizar un programa que calcule la masa del aire de un neumático de automóvil utilizando la fórmula:

$PV = 0.37m(T + 460)$

donde:

- **P**, presión en libras por pulgada cuadrada (psi).
- **V**, volumen en pies cúbicos.
- **m**, masa de aire en libras.
- **T**, temperatura en grados Fahrenheit.

46) Realizar un programa que determine la cantidad de dinero en dólares que existe en una alcancía que contiene: medios dólares, cuartos, dimes, níqueles y peniques.

47) Se lee el siguiente número de punto flotante en doble precisión 0.1234546789, imprimir una salida como la siguiente:

```
    .1
    .12
    .123
```

4 espacios

Utilizar una sola función para la impresión, en la que se use el paso de parámetros de longitud de campo y precisión. Además, realizar ajuste a la derecha. (Nota: Utilizar alguna sentencia de control para la repetición).

Capítulo 4

SENTENCIAS DE CONTROL

4.1. INTRODUCCIÓN

Un programa es simplemente un conjunto de sentencias con cierta sintaxis, por ello, las sentencias son las piezas con las que se construye un programa.

En lenguaje C una sentencia es una instrucción para el computador y siempre termina en punto y coma. Por ejemplo:

dedos = 20

es una expresión, que a su vez, puede formar parte de una expresión más compleja, mientras que:

dedos = 20;

es una sentencia.

Una vez ejecutada una sentencia, el computador se dirigirá a la siguiente, a realizar lo que corresponda.

4.2. TIPOS DE SENTENCIAS

1. **La sentencia de declaración**. Sirve para establecer los nombres y el tipo de variables, y hace que el computador reserve posiciones de memoria para cada una de ellas.

2. **La sentencia de asignación**. Sirve para asignar valores a las variables. Está formada por un nombre de variable, seguido del operador de asignación **=**, de una expresión y de un punto y coma.

3. **La sentencia de función**. Es una llamada a función, que se encarga de realizar una tarea de acuerdo a como fue diseñada la función.

4. **Las sentencias de control**. Son aquellas que deciden qué sentencias deben ejecutarse o cuáles sentencias deben repetirse.

5. **La sentencia nula**. No hace nada.

Sentencia compuesta, se denomina así a un conjunto de dos o más sentencias (de las anteriores) agrupadas y encerradas entre llaves, que realizan una tarea determinada. Recibe también el nombre de bloque, módulo, proceso, o bloque de código.

Un programa puede contener una sentencia simple que puede ser intercambiada por una sentencia compuesta o viceversa. Además, es aconsejable utilizar la tabulación (indentación) como una herramienta que ayude a destacar la estructura del programa.

También, todos los programas tienen las sentencias ordenadas de arriba abajo. Es decir, una programación de arriba hacia abajo es aquella que, siempre y cuando un bloque haya sido ejecutado completamente, continúa con el siguiente.

Cabe destacar que aprender a programar no es simplemente conocer las reglas del lenguaje, sino también utilizar la programación modular, que permite subdividir el trabajo en tareas manejables y menos complejas.

4.3. SENTENCIAS DE CONTROL

Un lenguaje debe disponer de dos formas básicas para controlar el "flujo del programa" (flujo del programa es la secuencia que el computador sigue para ejecutarlo), que son:

1) **Decisión**. Decide ejecutar sentencias entre acciones alternativas.

2) **Repetición**. Repite una secuencia de sentencias hasta que se cumpla una determinada condición.

4.4. SENTENCIAS DE DECISIÓN

4.4.1. Decisión Binaria

La decisión binaria decide la ejecución de las sentencias entre dos alternativas.

4.4.1.1. Sentencia *if*

La sentencia **if** permite escoger entre realizar una acción o no. Es la forma más sencilla de una sentencia **if** que permite elegir entre ejecutar una sentencia o saltarla.

La estructura más simple de la sentencia **if** es:

```
if (expresión)
    sentencia;
```

Si **expresión** es verdadera se ejecuta la **sentencia** que va a continuación, si es falsa, la **sentencia** se ignora.

Generalmente la **expresión** es de relación, es decir, una expresión que compara el tamaño de dos cantidades. Pero puede ser también cualquier **expresión**: si ésta tiene un valor 0 se toma como "*falsa*"; si tiene un valor distinto de 0 es "*verdadera*".

La sentencia puede ser una sentencia simple o una sentencia compuesta delimitada por llaves. Así:

```
if (expresión) {
    /* Sentencias. */
}
```

Por ejemplo:

```
if (num › 0)
    printf("EI número %d es positivo.\n", num); /* Sentencia simple. */

if (num == 0) {
    printt ("EI número es igual a cero.\n");
    num++;
}                                               /* Sentencia compuesta. */
```

4.4.1.2. Sentencia *if-else*

La sentencia **if-else** puede escoger entre dos acciones diferentes. Es decir, permite la elección entre dos sentencias; elige la una y la otra no la toma en cuenta, siendo excluyente en esta elección.

La estructura general de la sentencia **if-else** es:

```
if (expresión)
    sentencia1;
else
    sentencia2;
```

Si la **expresión** es verdadera, se ejecuta la primera sentencia (**sentencia1**); si es falsa se ejecuta la sentencia que está colocada a continuación de **else** (**sentencia2**).

Por ejemplo, calcular el valor absoluto de un número:

```
if (num <= 0)
    y = -num;
else
    y = num;
```

Esta sentencia también puede realizarse con el operador condicional:

```
y = (num <= 0) ? -num : num;
```

Las sentencias que son involucradas por **if-else** pueden ser simples o compuestas. Las sentencias compuestas tendrán esta estructura:

```
if (expresión) {
    /* Sentencias. */
}
else {
    /* Sentencias. */
}
```

4.4.1.3. Sentencia *else-if*

La sentencia **else-if** es la misma sentencia **if-else**, pero se utiliza para elegir una única acción entre más de dos posibles alternativas. Esto constituye una "***elección múltiple***".

Es decir, ***else-if*** consiste de una sentencia **if-else**, en la cual la parte de la sentencia **else** es, a su vez, otra sentencia **if-else**, la misma que a su vez puede contener tantos **else-if** como se deseen. En esta forma se dice que la segunda sentencia **if-else** está "***anidada***" en la primera.

Por lo tanto, se deben tener como mínimo 2 condiciones (expresiones), y la anidación debe ser realizada solo por la sentencia **else** para que sea una elección excluyente.

La estructura general de la sentencia **else-if** es:

```
if (expresión1)
    sentencia1;
else if (expresión2)
    sentencia2;
else
    sentencia3;
```

Si **expresión1** es verdadera, se ejecuta la **sentencia1.** Si **expresión1** es falsa, pero **expresión2** es verdadera, se ejecuta la **sentencia2**. Si ambas expresiones son falsas, se ejecuta la **sentencia3**.

Por ejemplo:

```
if (tanteo < 1000)
```

```
    bono = 0;
else if (tanteo < 5000)
    bono = 1;
else if (tanteo < 10000)
    bono = 2;
else
    bono = 3;
```

Toda la estructura **else-if** se contabiliza como una sola sentencia, por lo que no hay necesidad de encerrar entre llaves el **else-if** anidado. Como se puede ver el ejemplo anterior al expresarlo en forma anidada:

```
if (tanteo < 1000)
    bono = 0;
else
    if (tanteo < 5000)
        bono = 1;
    else
        if (tanteo < 10000)
            bono = 2;
        else
            bono = 3;
```

De forma similar a **if-else**, las sentencias pueden ser simples o compuestas.

Cada *else* con su *if*

También se puede anidar por la condición de verdad de la sentencia **if-else**, es decir, cuando se encuentran varios **if** y **else** anidados, existe una correspondencia de los **if** con los **else**: el **else** va con el **if** más próximo anterior, a menos que haya llaves que indiquen lo contrario.

Por ejemplo, considérese el siguiente fragmento de programa:

```
if (numero > 6)
    if (numero < 12)
        printf ("Correcto!\n");
    else /* Este else pertenece al segundo if. */
        printf ("lncorrecto!l\n");
```

el **else** va con el segundo **if**.

La ejecución de este fragmento de programa realizará las siguientes impresiones:

número	*IMPRESIÓN*
5	*ninguna*
10	*Correcto!*
15	*Incorrecto!*

Si realmente se desea que **else** vaya con el primer **if**, se debería reescribir el fragmento de programa con un bloque de código, de la siguiente manera:

```
if (numero > 6) {
    if (numero < 12)
        printf ("Correcto!\n");
}
else /* Este else pertenece al primer if. */
    printf ("Incorrecto!\n");
```

Ahora se tendrá las siguientes impresiones:

número	*IMPRESIÓN*
5	*Incorrecto!*
10	*Correcto!*
15	*ninguna*

Ejercicio

Supóngase que las tarifas por consumo eléctrico son:

CONSUMO (KWH)	VALOR (UDS por KWH)
Los primeros 200	0.5418
Hasta los 500	0.7047
Hasta los 750	0.9164
Sobre los 750	1.1254

Diseñar un programa que permite calcular su costo de la luz (recibo de luz) de un medidor.

```
/* PROG0401.C */

/* Calcula el recibo de luz. */

#include "stdio.h"

#define TARIFA1 5.418     /* Tarifa de los primeros 200 Kw. */
```

```
#define TARIFA2 7.047      /* Tarifa hasta los 500Kw. */
#define TARIFA3 9.164      /* Tarifa hasta los 750 Kw. */
#define TARIFA4 11.254    /* Tarifa por encima de los 750 Kw. */
#define LIMITE1 200.0     /* Primer bloque de tarifa. */
#define LIMITE2 500.0     /* Segundo bloque de tarifa. */
#define LIMITE3 750.0     /* Tercer bloque tarifa.  */

void main ()
{
  float KWH;       /* Kilowatios por hora gastados. */
  float recibo;    /* Precio. */

  printf ("Ingrese el gasto en KWH: ");
  scanf ("%f", &KWH);

  if (KWH < LIMITE1)                /* KWH menor a 200. */
    recibo = TARIFA1 * KWH;
  else if (KWH < LIMITE2)        /* KWH entre 200 y 500. */
    recibo = TARIFA2 * KWH;
  else if (KWH < LIMITE3)         /* KWH entre 500 y 750. */
    recibo = TARIFA3 * KWH;
  else                                       /* KWH por encima de 750. */
    recibo = TARIFA4 * KWH;

  printf ("La cuenta total por %.1f KWH es %.0f dólares.\n", KWH, recibo);
}
```

Una salida del programa sería:

Ingrese el gasto en KWH: 225.4 <ENTER>
La cuenta total por 225.4 KWH es 158.8 dólares.

4.4.2. Decisión Múltiple: Sentencia *switch*

Se utiliza la sentencia **switch** cuando se necesita elegir una o varias acciones entre algunas alternativas por lo tanto, esta sentencia no es excluyente.

La sentencia **switch** funciona de la siguiente manera: el control del programa se transfiere a la sentencia cuya etiqueta tenga el mismo valor que la **expresión** evaluada luego de **switch**. El programa se continúa ejecutando sentencia a sentencia hasta que se ejecute la sentencia **break** o se termine de ejecutar la sentencia **switch**. Si no se encuentra ninguna etiqueta con el valor de la **expresión**, el control del programa se transfiere a la sentencia etiquetada **default**, si ésta existe. En caso contrario, el control pasa a la sentencia localizada inmediatamente después de la sentencia **switch**.

La estructura general de la sentencia **switch** es:

```
switch (expresión) {
    case etiqueta1 :      sentencias;
                          break; /* Es opcional. */
    case etiqueta2 :      sentencias;
                          break; /* Es opcional. */

    /* Otras alternativas. */

    default : sentencias;
                          break; /* Es opcional. */
}
```

En conclusión, al evaluarse la **expresión**, el valor que posea se rastrea en la lista de etiquetas **case**, hasta que encuentre una que corresponda a dicho valor; entonces se transfiere el control del programa a dicha etiqueta, para ejecutar las sentencias hasta que se encuentre una sentencia **break** o termine de ejecutarse la sentencia **switch**. Si no existe dicho valor en la lista, se transfiere el control del programa a las sentencias de **default**. El programa continuará ejecutando la siguiente sentencia de **switch**.

"*La sentencia **break***". Esta es una "*sentencia de salto*" que hace que el control del programa se salga de la sentencia **switch** y se dirija a la sentencia situada inmediatamente después de la misma.

"*La **expresión***". La expresión va encerrada entre paréntesis y debe tener un valor entero o un valor de tipo **char**.

"*La **etiqueta***". La etiqueta debe ser de valor constante o ser una expresión formada únicamente por constantes de tipo entero o de tipo **char**.

Tanto el tipo de dato de la **expresión** como de la **etiqueta** deben tener el mismo tipo de dato.

Ejercicio

Realizar un programa para crear un menú que tenga las siguientes opciones:

1 Sumar dos números.
2 Restar dos números.
3 Multiplicar dos números.
4 Dividir dos números.

Se debe controlar en la división no exista división por cero, y si existe división por cero se debe indicar con un mensaje.

```
/* PROG0402.C */

#include "stdio.h"

void main ()
{
  char ch;
  float num1, num2;

  puts ("1 Sumar dos números.");
  puts ("2 Restar dos números.");
  puts ("3 Multiplicar dos números.");
  puts ("4 Dividir dos números.");
  printf ("\nDigite una opción: ");
  ch = getchar ();
  printf ("\n");

  if (ch > '0' && ch <= '4') {
    printf ("Ingrese el primer número : ");
    scanf ("%f", &num1);
    printf ("Ingrese el segundo número: ");
    scanf ("%f", &num2);
  }

  switch (ch) {
    case '1' : printf ("%.1f + %.1f = %.1f\n", num1, num2, num1 + num2);
                    break;
    case '2' : printf ("%.1f - %.1f = %.1f\n", num1, num2, num1 - num2);
                    break;
    case '3' : printf ("%.1f * %.1f = %.1f\n", num1, num2, num1 * num2);

    case '4' : if (num2)
                    printf ("%.1f / %.1f = %.1f\n", num1, num2, num1 / num2);
                 else
                    printf ("División por cero\n");
                 break;
    default  :
                 printf ("ERROR, opción errada\n");
  }
}
```

Una salida del programa sería:

1 Sumar dos números.
2 Restar dos números.
3 Multiplicar dos números.
4 Dividir dos números.

Digite una opción: 2 <ENTER>

Ingrese el primer número: 6.2 <ENTER>
Ingrese el segundo número: 4.5 <ENTER>
6.2 - 4.5 = 1.7

Cuando se desea que varias etiquetas ejecuten las mismas sentencias, solamente la última etiqueta contendrá esas sentencias; las anteriores quedarán sin sentencias. Por ejemplo, determinar que una variable caracter sea una vocal:

```
switch (ch) {
    case 'a':
    case 'e'        :
    case 'i':
    case 'o'        :
    case 'u'        : printf ("EI caracter %c es una vocaI:\n", ch);
              break;
    default :  printf ("El caracter %c no es una vocaI:\n", ch);
}
```

Cuando no usar una sentencia switch:

1) No se puede emplear la sentencia **switch** cuando la elección esté basada en una comparación de variables o expresiones de tipo **float**.

2) Tampoco se puede usar la sentencia **switch** si la variable a controlar esta comprendida en un cierto rango, porque se debe indicar todos los valores del rango.

Es decir, en estos dos casos solo puede emplearse una sentencia **else-if**.

4.5. SENTENCIAS DE REPETICIÓN

4.5.1. Sentencia *while*

La sentencia **while** crea un bucle o lazo que se repite hasta que la expresión de control llamada "*test*" se vuelva falsa.

La estructura general de la sentencia **while** es:

```
while (expresión)
    sentencia;
```

Si la **expresión** es verdadera (distinto de cero) la **sentencia** se ejecuta, y la **expresión** se evalúa de nuevo para comprobar si su valor de verdad permanece inalterable. Este ciclo de "test y ejecución" se repite hasta que la **expresión** se vuelva falsa (igual a cero). Cada vez que se ejecuta el lazo (cada ciclo realizado) se denomina una "***iteración***".

La **expresión** puede ser cualquier expresión válida, pero se utiliza por lo general una expresión de relación.

La **sentencia** puede estar constituida por una sentencia simple que acabe en un punto y coma, o bien por una sentencia compuesta encerrada entre llaves. Por ejemplo:

```
while (expresión) {
    /* Sentencias. */
}
```

Cuando se construye un lazo de este tipo se debe incluir una variación del valor de la **expresión** del **test**, de manera que el lazo acabe cuando dicha expresión sea falsa, caso contrario el lazo no finalizará nunca.

El lazo **while** es "***condiciona***l", porque la ejecución de las sentencias que contiene dicho lazo, depende de la condición que se describa en la **expresión**.

Además, la sentencia **while** es un lazo con "***condición de entrada***", porque la condición del **test** debe cumplirse antes de acceder al cuerpo del lazo; caso contrario, si la condición es falsa, no se ejecuta el lazo.

El lazo **while** se emplea de dos formas:

1) **En forma indefinida**. Cuando la repetición del lazo depende de una condición indeterminada para su finalización, es decir, no se sabe de antemano cuántas veces se va a ejecutar el lazo antes de que la expresión se vuelva falsa.

Ejercicio

Realizar un programa que ingrese un texto desde teclado, para que sea mostrado en pantalla, hasta que se digite fin de archivo (Ctrl Z).

```
/* PROG0403.C */
```

```
#include "stdio.h"

void main ()
{
  char ch;

  printf ("Ingrese el texto. Termina con <Ctrl Z>.\n");

  while ((ch = getchar ()) != EOF)
    putchar (ch);
}
```

Una salida del programa sería:

```
Ingrese el texto. Termina con <Ctrl Z>.
Este es un ejemplo demostrativo, <ENTER>
Este es un ejemplo demostrativo,
para indicar lo que muestra el <ENTER>
para indicar lo que muestra el
ejercicio. <ENTER>
ejercicio.
^Z <ENTER>
```

2) **En forma definida**. Cuando se conoce el número de repeticiones del lazo de antemano. Entonces, si el lazo se va a repetir un número fijo de veces, lleva implícito tres "acciones" que son:

- "*Inicializar un contador*". La inicialización de la variable de control del lazo (contador) se debe realizar fuera del lazo.

- "*Test de comparación*". La condición del lazo **while** (expresión) efectúa la comparación con su límite de finalización.

- "*Incremento*". El operador incremento se encarga de cambiar el valor de la variable de control del lazo, cada vez que éste se ejecuta.

Ejercicio

Realizar un programa que imprima números, desde el 1 hasta el 40.

```
/* PROG0404.C */

#include "stdio.h"
```

```
void main ()
{
  int cont;

  cont = 1;              /* Inicialización.   */
  while (cont <= 40) {     /* Test.            */
    printf ("%5d", cont);   /* Acción o proceso. */
    cont++;                /* Incremento.      */
  }
  printf ("\n");
}
```

La salida de este programa será:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

Se pueden combinar el "test de comparación" e "incremento" en una sola expresión, empleando las siguientes sentencias:

```
cont = 0;                    /* Inicialización. */
while (++cont <= 40)     /* Test de incremento. */
    printf ("%5d", cont);/* Proceso */
```

Ejercicio

Leer una serie de caracteres numéricos por línea, luego transformar cada línea a números enteros largos sin signo. El programa debe finalizar cuando se haya leído el fin de archivo.

```
/* PROG0405.C */

#include "stdio.h"

void main ()
{
  char ch;
  unsigned long valor = 0;

  printf ("Ingrese caracteres numéricos por línea. Termina con <Ctrl Z>.\n");

  while ((ch = getchar ()) != EOF)
    if (ch != '\n')
```

```
      valor = 10 * valor + ch - '0';
    else {
      printf ("%lu\n", valor);
      valor = 0;
    }
}
```

Una salida del programa sería:

Ingrese caracteres numéricos por línea, <Termina con Ctrl Z>.
12345 <ENTER>
12345
27895 <ENTER>
27895
1298 <ENTER>
1298
^Z <ENTER>

4.5.2. Sentencia *for*

Este lazo agrupa en un solo lugar las tres "***acciones***": inicializar un contador, compararlo con un límite e incrementarlo cada vez que se ejecute el lazo. Es decir, en un lazo **for** se puede sustituir las tres acciones en una sola sentencia.

La estructura general de la sentencia **for** es:

```
for (expresión1; expresión2; expresión3)
     sentencia;
```

Los paréntesis contienen tres expresiones separadas por puntos y comas, para controlar el proceso de lazo:

1) **expresión1** es normalmente la "***inicialización del contador***" y se ejecuta una sola vez al comenzar el lazo **for**.

2) **expresión2** es el "***test de comparación***", se evalúa antes de cada ejecución potencial del lazo y cuando la **expresión2** es falsa (o cero) el lazo finaliza.

3) **expresión3** es el "***incremento***" o "***decremento***" del contador (actualización), se evalúa al final de cada ejecución del lazo, e inmediatamente después se efectúa de nuevo el "test de comparación".

La sentencia **for** es un lazo con "***condición de entrada***", porque la comparación del **test** se realiza antes de ser ejecutada. Por lo tanto, es posible que el lazo no se ejecute ni una sola vez.

Este lazo puede estar formado por una sentencia simple o por una compuesta. Por ejemplo:

```
for (expresión1; expresión2; expresión3) {
    /* Sentencias. */
}
```

Ejercicio

Realizar un programa que imprima números, desde el 1 hasta el 40.

```
/* PROG0406.C */

#include "stdio.h"

void main ()
{
  int cont;

  for (cont = 1; cont <= 40; cont++)
    printf ("%5d", cont);
  printf ("\n");
}
```

La salida de este programa será:

```
    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20
   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40
```

Otra aplicación común del lazo **for** es hacer un contador de tiempo, con el fin de adaptar la velocidad del computador a nivel humano. Por ejemplo, si se desea retardar la velocidad del computador en un factor igual a 10000, se procede así:

```
for (n = 1; n <= 10000; n++)
    ;
```

Este lazo hace que el computador cuente hasta 10000. El punto y coma solitario de la segunda línea es una "*sentencia nula*", por lo que el lazo no realiza ninguna tarea.

Flexibilidad de la sentencia *for*

Este lazo tiene una gran libertad en la selección de las expresiones que lo controlan, esto lo hace mucho más útil que cualquier otro lazo. Esto implica que existen muchas otras posibilidades para el empleo de este lazo, que son las siguientes:

1. Contar en forma descendente. Por ejemplo:

```
for (cont = 10; cont > 0; cont--)
    printf ("%5d", com):
```

Imprime:

10 9 8 7 6 5 4 3 2 1

2. Incrementar o decrementar en cualquier cantidad entera. Ejemplo:

```
for (cont = 0; cont <= 10; cont += 2)
    printf ("%5d", cont);
```

En cada ciclo se incrementa **cont** en el valor 2, imprimiendo los números pares:

0 2 4 6 8 10

3. Contar caracteres. Por ejemplo:

```
for (ch = 'a'; ch <= 'z'; ch++)
    printf ("%5c", ch);
```

El lazo cuenta los números enteros de los caracteres, imprimiendo las letras del alfabeto inglés:

a b c d e f g h i j k l m n o p p r s t u v w x
y z

4. Actualizar la expresión de incremento en progresión geométrica; es decir, en vez de sumarle una cantidad fija cada vez, se la multiplica. Por ejemplo:

```
for (deuda = 100.0; deuda < 130.0; deuda *= 1.1)
    printf ("Su deuda asciende a %.2f\n", deuda);
```

La variable punto flotante **deuda** se multiplica por 1.1 en cada ciclo, incrementándose en un 10 % cada vez. Imprimiendo:

Su deuda asciende a 100.00
Su deuda asciende a 110.00
Su deuda asciende a 121.00

5. Utilizar cualquier expresión legal que se desee como "tercera expresión", ésta se evaluará siempre tras cada iteración. Ejemplo:

```
y = 55;
for (x = 1; y <= 100; y += 5 * x++ +10)
    printf ("%5d%5d\n", x, y);
```

Imprimirá para cada valor de **x** el valor de **y**, de acuerdo a la expresión algebraica 5 * x +10:

1 55
2 70
3 90

Además, cada una de las tres expresiones del lazo **for** podrían emplear diferentes variables; pero esto no es adecuado, porque no se debe mezclar procesos de cambio de índice con cálculos algebraicos.

6. Dejar una o más expresiones en blanco, para lo cual siempre se debe mantener el punto y coma. Por ejemplo:

```
ans = 2;
for (n = 3; ans <= 100;) {
    printf ("%5d", ans);
    ans *= n;
}
```

El valor de **n** será constante e igual a 3 y la variable **ans** comenzará con un valor 2 que se irá imprimiendo e incrementando, así:

2 6 18 54

Por otra parte, si se tiene la segunda expresión del lazo en blanco, hay que asegurarse de tener una sentencia que finalice el lazo. Por ejemplo:

```
for (;;)
    printf ("Quiero ser alguien en la vida!\n");
```

La segunda expresión (*test vacío*) es verdadera, por lo tanto el lazo se vuelve infinito. Se puede salir de él con la sentencia **break**.

7. Comprobar una condición en vez del número de iteraciones, obteniéndose un "lazo condicional". Por ejemplo:

```
for (; num != 0;) {
    printf ("Ingrese un número. Con <0> termina: ");
    scanf ("%d", &num);
}
```

Se ingresa números enteros desde teclado. Si se ingresa un valor igual a cero, que es un valor condicional, finalizará el lazo. Por ejemplo:

Ingrese un número. Con <0> termina: 5 <ENTER>
Ingrese un número. Con <0> termina: 0 <ENTER>

8. La primera expresión puede ser cualquier expresión. Es decir, no es necesario que la expresión inicialice una variable. Por ejemplo:

```
num = 1;
sum = 0;
for (printf ("Ingrese números. Con <0> termina.\n"); num != 0; sum += num)
    scanf ("%d", &num);
printf ("La suma de los datos ingresados es: %d", sum);
```

La primera sentencia del lazo **for** es la función **printf()** que imprimirá el mensaje una sola vez en el momento de ingresar al lazo, y continuará aceptando números hasta que se introduzca un 0. Por ejemplo:

Ingrese números. Con <0> termina.
4 <ENTER>
3 <ENTER>
5 <ENTER>
0 <ENTER>
La suma de los datos ingresados es: 12

9. Alterar los parámetros de las expresiones del lazo dentro del mismo. Por ejemplo:

```
delta = 1;
for (n = 1; n < 100; n += deIta) {
    if (n == 20)
        delta = 10;
    printf ("%5d", n);
}
```

En este lazo, en las primeras 20 iteraciones, se incrementa en 1 la variable **n**; a partir del valor de **n=20** se incrementa en 10, porque **delta**, que es la variable de incremento, es modificada en ese valor. Imprimiéndose:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
30 40 50 60 70 80 90

Entonces, se puede tomar la decisión de cambiar el valor de cualquier parámetro del lazo en la ejecución del mismo, dependiendo si es demasiado pequeño o demasiado grande el parámetro de incremento. Esto puede realizar el usuario dentro del lazo, mediante una sentencia **if** en el interior de éste.

10. Incluir más de una "inicialización", "comparación" o "actualización" dentro de las especificaciones del lazo, lo que se obtiene mediante el "*operador coma*".

    Por ejemplo:

```
for (minutos = 5, costo = 20; minutos <= 15; minutos +=5, costo += 12)
    printf ("%5d%5d\n", minutos, costo);
```

Si la tarifa de una llamada telefónica es de 0,50 dólares para los primeros 5 minutos, y 12 más para cada 5 minutos adicionales. Entonces, en la primera expresión se inicializan **minutos** y **costo**, y en la tercera expresión se incrementará el valor 5 en **minutos** y el 12 en **costo** en cada iteración. Por lo que se imprimirá:

```
 5   20
10   32
15   44
```

Es decir, se puede usar en las expresiones del lazo más de una subexpresión, separadas por el operador coma.

4.5.3. Sentencia *do-while*

La sentencia **do-while** crea un lazo que se repite hasta que la expresión de control **test** sea falsa. La diferencia con la sentencia **while** es que la condición del **test** debe estar al final del cuerpo del lazo.

La estructura general de la sentencia **do-while** es:

```
do
    sentencia;
while (expresión);
```

La sentencia del lazo puede ser compuesta, como se muestra a continuación:

```
do {
    /* Sentencias. */
} while (expresión);
```

La sentencia **do-while** es un lazo con "***condición de salida***", donde la condición del test se comprueba al final de cada iteración. Por lo que el lazo **do-while** se ejecuta siempre una vez como mínimo, lo que es necesario en algunas aplicaciones.

Por ejemplo, para validar que los datos sean leídos en un rango predeterminado:

```
do {
    printf ("Ingrese un número entero positivo menor a 100: ");
    scanf ("%d", &num);
} while (num <= 0 || num > =100);
```

Ejercicio

Leer una serie de números para determinar si los números son primos o no, hasta digitar un número 0. Un número primo es aquel que puede dividirse únicamente por sí mismo y por la unidad. Por ejemplo: 1, 2, 3, 5, 7, 11, …

```
/* PROG0407.C */

#include "stdio.h"

void main ()
{
 long num = 1, res, con;

 while (num != 0) {
  do {
   printf ("Ingrese un número entero. Termina con <0>: ");
   scanf ("%ld", &num);
  } while (num < 0);

  res = 1;
  con = 1;
  while (++con <= num / 2 && res)
   if (num % con == 0)        /* Equivale a: res = num % con; */
    res = 0;
  if (num)
   if (res)
```

```
      printf ("%ld es primo.\n", num);
    else
      printf ("%ld no es primo.\n", num);
  }
}
```

Una salida del programa podría ser:

Ingrese un número entero. Termina con <0>: 17 <ENTER>
17 es primo.
Ingrese un número entero. Termina con <0>: 15 <ENTER>
15 no es primo.
Ingrese un número entero. Termina con <0>: 0 <ENTER>

NOTA: Las tres sentencias de repetición: **for**, **while** y **do-while**; pueden ser intercambiables, es decir, pueden generar el mismo lazo de repetición.

4.5.4. Lazos Anidados

Se llama lazo anidado aquel que está completamente incluido en otro lazo.

Por ejemplo:

```
for (i = 1; i <= 3; i++) {
    printf ("\ni = %-5d\n", i);
    for (j = 1; j <= 5; j++)
        printf ("%-5d", j);
}
```

Imprimirá:

i =1
1 2 3 4 5
i =2
1 2 3 4 5
i =3
1 2 3 4 5

Ejercicio

Determinar el máximo común divisor (**mcd**) de **n** números, en base al siguiente algoritmo: calcular el **mcd** de los dos primeros números, luego calcular el **mcd** de éste resultado con el tercer número, hasta terminar con los **n** números o el **mcd** sea igual a 1.

```
/* PROG0408.C */

/* Máximo común divisor de n números. */

#include "stdio.h"

void main ()
{
  int mcd, num, n, residuo, conta = 1;

  do {
    printf ("Ingrese la cantidad números de la serie: ");
    scanf ("%d", &n);
  } while (n <= 1);

  do {
    printf ("Ingrese un número positivo de la serie  : ");
    scanf ("%d", &num);
  } while (num <= 0);

  mcd = num;

  while (mcd != 1 && ++conta <= n) {

    do {
      printf ("Ingrese otro número positivo de la serie: ");
      scanf ("%d", &num);
    } while (num <= 0);

    residuo = 1;

    while (residuo) {
      residuo = mcd % num;
      mcd = num;
      num = residuo;
    }
  }

  if (mcd != 1)
    printf ("mcd = %d\n", mcd);
  else
    printf ("no hay mcd\n");
}
```

Una salida del programa podría ser:

Ingrese la cantidad números de la serie: 2 <ENTER>
Ingrese un número positivo de la serie: 12 <ENTER>
Ingrese otro número positivo de la serie: 30 <ENTER>
mcd = 6

4.6. SENTENCIAS DE SALTO INCONDICIONAL

Las sentencias tratadas anteriormente son denominadas de "salto condicional" porque cambian la secuencia de ejecución del programa cuando se da una cierta condición. Mientras que las sentencias de "salto incondicional" no necesitan condición para cambiar la secuencia del programa. El lenguaje C tiene cuatro sentencias que llevan a cabo un salto incondicional: **return**, **break**, **continue** y **goto**.

- Las sentencias **return** y **goto** se pueden usar en cualquier parte del programa.

- Las sentencias **break** y **continue** se deben usar dentro de una sentencia de lazo, aunque se puede usar **break** en una sentencia **switch**.

4.6.1. La Sentencia *return*

Es una sentencia de salto porque hace que la ejecución del programa vuelva de una función al punto de su llamada, luego de haber sido ejecutada. Además, la sentencia **return** retorna un valor por la función, teniendo dos alternativas:

a) Si en la función hay algún valor asociado con **return**, éste es el valor de vuelta por la función.

b) Si en la función no se especifica un valor de vuelta, se asume que devuelve un valor sin sentido, produciéndose un error.

La forma general de la sentencia **return** cuando la función retorna un valor es:

return expresión;

Si la función es declarada como **void** no toma ningún valor, pero puede contener sentencias **return** que no especifiquen un valor y será de la siguiente forma:

return;

Se puede usar tantas sentencias **return** como se quiera en una función, sin embargo, la función termina tan pronto como encuentre el primer **return**. También terminará

cuando su código finalice con la llave (}), que hace que el control del programa vuelva de la función al punto de llamada; esto equivale a un **return** sin valor específico.

4.6.2. La Sentencia *break*

La sentencia **break** se puede utilizar con cualquiera de los tres tipos de lazos y con la sentencia **switch**. Esta sentencia produce un salto en el control del programa, de manera que se evite o se libere de los lazos o de la sentencia **switch** en donde se encontraba, reanudándose la ejecución con la siguiente sentencia que sigue a los lazos o a **switch**.

Por ejemplo, en la siguiente sentencia **for** se termina el lazo infinito cuando se haya leído un valor igual a cero:

```
for (;;) {
    pritf ("Ingrese un valor entero. Con <0> termina: ");
    scanf ("%d", &num);
    if (!num)
        break;
}
```

Cuando se usa la sentencia **break** en una estructura anidada, la "liberación" afecta únicamente al nivel de la estructura más interna que la contenga.

Generalmente, se utiliza la sentencia **break** cuando hay dos condiciones para abandonar el lazo. Por ejemplo, el siguiente lazo debe finalizar cuando lea un caracter EOF o un caracter "nueva línea":

```
while ((ch = getchar ()) != EOF) {
    if(ch == '\n')
        break;
    putchar (ch);
}
```

Este lazo lee e imprime una sola línea de caracteres.

En un lazo se puede evitar la utilización de la sentencia **break**, colocando ambas condiciones para abandonar el lazo en un mismo lugar, de la siguiente manera:

```
while ((ch = getchar()) != EOF && ch != '\n')
    putchar (ch);
```

4.6.3. La Sentencia *continue*

La sentencia **continue** se puede utilizar con cualquiera de los tres tipos de lazos, pero no con la sentencia **switch**. Esta sentencia interrumpe el flujo del programa, para no ejecutar lo que está a continuación de ella hasta el fin del lazo, comenzando a ejecutar la siguiente iteración.

Si se reemplaza el **break** del ejemplo anterior por un **continue**, se obtiene:

```
while ((ch = getchar ()) != EOF) {
    if ((ch == '\n')
        continue;
    putchar (ch);
}
```

En este caso se imprime cada línea del texto ingresado sin cambio de línea, y el ingreso de la siguiente línea se hará a continuación, ya que **continue** hace que se ignore la impresión de los caracteres nueva línea; sin embargo, el lazo se abandonará únicamente cuando se ingrese un caracter EOF.

Para no usar la sentencia **continue** solo basta invertir el **test** de la sentencia **if** de la siguiente manera:

```
while ((ch = getchar ()) != EOF)
    if(ch != '\n')
putchar (ch);
```

Por otra parte, la sentencia **continue** puede acortar algunos programas, en especial si existen varias sentencias **if-else** anidadas.

Además, las sentencias **break** y **continue** se usan con menor frecuencia, porque un abuso de las mismas hace que los programas sean difíciles de entender, más propensos a errores y más difíciles de modificar.

4.6.4. La Sentencia *goto*

La sentencia **goto** hace que el control del programa salte a una nueva sentencia con la etiqueta adecuada.

La sentencia **goto** tiene dos partes: el **goto** y un nombre de etiqueta; la *etiqueta* se coloca al comienzo de la sentencia a donde se va a saltar, separándola de la misma por medio de dos puntos.

La estructura de la sentencia **goto** es:

```
goto etiqueta;
/* Sentencias. */
etiqueta : sentencia;
```

Los nombres de las etiquetas cumplen las mismas reglas que los nombres de las variables.

Por ejemplo:

```
tope : ch = getchar ();
/* Sentencias. */
if (ch != 's')
    goto tope;
```

Recomendación: Se debe EVITAR la sentencia ***goto***, si no es necesaria.

4.6.5. La Función *exit()*

La función estándar **exit()** obliga a salir o a terminar inmediatamente el programa, forzando la vuelta al sistema operativo. Esta función se encuentra en el archivo de cabecera "***stdlib.h***".

El prototipo de la función **exit()** es:

```
void exit (int codigo_vuelta);
```

donde:

- **codigo_vuelta**, es el valor que se devuelve al proceso de llamada, que normalmente es el sistema operativo.

    - Se usa un cero como código de vuelta, para indicar que se trata de una terminación normal del programa.

    - Se utilizan otros argumentos para indicar algún tipo de error.

Se utiliza **exit()** cuando no se satisface una condición obligatoria en la ejecución del programa, es decir cuando hay un error.

Por ejemplo, el siguiente programa determina si existe o no caracteres numéricos, para lo cual se utiliza la función **caracteres_numericos()** que devuelve verdadero si existe caracteres numéricos, y la función **proceso()** que es el proceso que se ejecuta si existe caracteres numéricos:

```
void main ()
```

```
{
    if (!caracteres_numericos())
        exit (1 );
    proceso ();
}
```

PROBLEMAS PROPUESTOS

1) Realizar un programa que genere una tabla de los cuadrados y cubos de los números desde 1 hasta un **tope** leído desde el teclado. Por ejemplo, si el **tope** es 3 la tabla será:

NUMERO	CUADRADO	CUBO
1	1	1
2	4	8
3	9	27

2) Realizar un programa para convenir temperaturas de grados Fahrenheit (**F**), desde 0 hasta 212 grados, a grados Celsius (**C**) de punto flotante con 3 dígitos de precisión, utilizando la fórmula:

   C = (5.0 / 9) * (F - 32)

   La salida deberá ser impresa en dos columnas justificadas a la derecha de los campos, cada una de 10 caracteres. Las temperaturas Celsius deberán ser antecedidas por un signo, tanto para valores positivos como negativos.

3) "***Un palíndromo***" es un número o una frase de texto, que se lee igual hacia adelante y hacia atrás. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos son palíndromos: 12321, 55555, 45554 y 11611. Escribir un programa que lea un entero de cinco dígitos y que determine si es o no un palíndromo.

4) Realizar un programa que ingrese desde teclado un número entero binario e imprimir su equivalente decimal. Validar que los dígitos del número sean binarios (solo dígitos **0** y **1**). Por ejemplo, el equivalente decimal del número 1011 binario es: 1*8 + 0*4 + 1*2 +1*1 = 8 + 0 + 2 + 1, es decir 11.

5) Escribir un programa que calcule e imprima el promedio de una secuencia de números enteros, suponiendo que el último valor leído mediante **scanf()** es el centinela 9999. El programa deberá leer un valor cada vez que **scanf()** sea ejecutado. Por ejemplo, una secuencia típica de entrada pudiera ser:

   10   8   11   7   9   9999

   El promedio debe calcularse de todos los valores que preceden a 9999. Además, el programa debe localizar el más pequeño de los números enteros.

6) Realizar un programa que ingrese desde teclado un número positivo **n**, y calcule la suma de los números impares desde 1 hasta **n**. Por ejemplo, calcular la suma de los primeros **n** números impares:

   1 + 3 + 5 +...+ (2 * n – 1)

   El programa debe terminar cuando se ingresa un número negativo o cero.

7) Escribir un programa que lea tres valores **float** diferentes a cero, que determine e imprima si pueden representar los lados de un triángulo. Además se debe indicar si esos valores pueden ser los lados de un triángulo rectángulo.

8) Realizar un programa que lea desde teclado una serie de caracteres hasta un fin de archivo <<Ctrl Z>> y que cuente los caracteres leídos, las palabras y las líneas del texto ingresado. Se definirá una palabra como una secuencia de caracteres sin espacios en blanco, sin tabulado y sin nueva línea.

9) Escribir un programa que lea el lado de un cuadrado y a continuación lo imprima en forma de un cuadrado hueco de asteriscos. El programa deberá funcionar para cuadrados de todos los tamaños entre 1 y 20 asteriscos. Por ejemplo, si se lee un tamaño de 5 asteriscos, deberá imprimir:

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

10) Escribir un programa que imprima la figura de un diamante como en el gráfico siguiente. Se puede utilizar funciones **printf()** que impriman ya sea un asterisco (*), o un espacio en blanco, minimizado el número de funciones **printf()**. Utilizar estructuras de repetición anidadas.

    El programa debe leer un número impar del rango 1 al 19 a fin de especificar el número de líneas del diamante. A continuación se deberá desplegar un diamante del tamaño apropiado. Por ejemplo, si el número de líneas es 9 la figura será:

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

11) Escribir un programa que imprima los siguientes patrones por separado, uno debajo de otro. Todos los asteriscos (*), deberán ser impresos por una sola función **printf()** de la forma **printf("*")**.

```
a)          b)            c)           d)

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

12) Escribir un programa que cifre los datos enteros de cuatro dígitos. El programa debe leer un entero de cuatro dígitos y cifrarlo como sigue: reemplazar cada dígito por el resultado de sumarle 7 y de determinar su residuo de 10. A continuación intercambiar el primer dígito con el tercero, y el segundo con el cuarto. Por último, imprimir el entero cifrado.

El programa debe introducir un entero de cuatro dígitos para cifrarlo, y luego descifrarlo para formar el número original.

13) Realizar un menú para cada uno de los siguientes enunciados:

a) Calcular e imprimir la suma de una secuencia de enteros, suponiendo que el primer entero leído con **scanf()** especifica el número de valores a introducir. Por ejemplo, una secuencia de entrada típica pudiera ser:

5 100 200 300 400 500

donde 5 indica que los 5 valores subsiguientes deberán de ser sumados.

b) Calcular e imprimir la suma de los enteros pares del 2 al valor **n** ingresado desde teclado.

c) Calcular e imprimir el producto de los enteros impares del 1 a **m** ingresado desde teclado.

14) Realizar un menú para cada uno de los siguientes enunciados:

a) Leer dos números del teclado, calcular la suma de los números y desplegar el resultado.

b) Leer dos números del teclado, determinar y desplegar cuál es el mayor de los dos números.

c) Leer desde el teclado una serie de números positivos, suponiendo que el valor del centinela es -1, para indicar la entrada de fin de datos. Luego determinar y desplegar la suma de la serie de números.

15) Realizar un programa que imprima una tabla de todos los equivalentes de los números decimales a números romanos, en el rango de 1 a **n** ingresado desde teclado.

16) Generar la siguiente "pirámide" de dígitos utilizando lazos anidados:

1
232
34543
4567654
567898765
67890109876

Realizar un programa para ingresar el número de líneas de la pirámide (mayor a 0), también determinar la fórmula que genere los dígitos correspondientes para cada línea y luego imprimir la pirámide.

17) Un conductor lleva registro de los kilómetros manejados y los galones utilizados en cada uno de los tanques. Desarrollar un programa que introduzca los kilómetros manejados y los galones utilizados en cada tanque, para calcular y desplegar los kilómetros por galón obtenidos 'de cada tanque. Después de procesar toda la información de entrada, el programa deberá calcular e imprimir los kilómetros por galón combinados, obtenido de todos los tanques. Por ejemplo:

Ingrese los galones usados <-1 para fin>: 15.3 <ENTER>
Ingrese los kilómetros conducidos: 300 <ENTER>
Los km/galón para este tanque fue: 19.608

Ingrese los galones usados <-1 para fin>: 8 <ENTER>
Ingrese los kilómetros conducidos: 140 <ENTER>
Los km/galón para este tanque fue: 17.5

Ingrese los galones usados <-1 para fin>: -1 <ENTER>
El promedio de los km/galón fue: 18.554

18) Una gran empresa química paga a su personal de ventas en base a comisiones. El personal de ventas recibe 200 dólares por semana más 9% de las ventas brutas de esa semana. Por ejemplo, una persona de ventas que vende 5000 dólares de productos químicos en una semana, recibe 200 dólares más 9% de 5000 dólares, o sea un total de 650 dólares. Desarrollar un programa que introduzca las ventas brutas de cada vendedor correspondiente a la última semana, calcule y despliegue las ganancias de dicho vendedor. Procesar las cifras vendedor por vendedor. Por ejemplo:

Ingrese las ventas <-1 para fin>: 5000 <ENTER>
El salario es: 650 dólares
Ingrese las ventas <-1 para fin>: 1000 <ENTER>

El salario es: 290 dólares
Ingrese las ventas <-1 para fin>: -1 <ENTER>

19) Desarrollar un programa que determine si un cliente de una tienda departamental ha excedido el límite de una cuenta de crédito. Para cada uno de los clientes están disponibles los siguientes datos:

a) Número de la cuenta de crédito.
b) Saldo al principio del mes.
c) Total de todos los créditos aplicados en el mes a la cuenta del cliente.
d) Límite permitido de crédito.

El programa deberá introducir cada uno de estos datos, luego calcular el nuevo saldo (saldo inicial+créditos), y determinar si el nuevo saldo excede el límite de crédito del cliente. Para aquellos clientes cuyo límite de crédito esté excedido, el programa deberá desplegar el número de cuenta del cliente, el límite de crédito, el nuevo saldo y el mensaje "Límite de crédito excedido". Por ejemplo:

Ingrese número de cuenta <-1 para fin>: 100 <ENTER>
Ingrese balance inicial: 539 <ENTER>
Ingrese total de créditos: 5000 <ENTER>
Ingrese crédito limite: 5500 <ENTER>
Cuenta: 100
Crédito límite: 5500
Balance: 5539
Límite de crédito excedido.

Ingrese número de cuenta <-1 par fin>: 200 <ENTER>
Ingrese balance inicial: 1000 <ENTER>

Ingrese total de créditos: 321 <ENTER>
Ingrese crédito límite: 1500 <ENTER>

Ingrese número de cuenta <-1 para fin>: -1 <ENTER>

20) Desarrollar un programa que determine el sueldo para cada uno de los empleados de una empresa. La empresa paga "tiempo normal" para las primeras 40 horas trabajadas de cada empleado y paga "sobre tiempo" de 1.5 veces para todas las horas trabajadas en exceso de 40 horas. Para introducir la información de cada uno de los empleados, se tiene una lista de los empleados de la empresa, el número de horas que cada empleado trabajó la última semana, y la tasa horaria de cada empleado. Determinar y desplegar el sueldo de cada uno de los empleados. Por ejemplo:

Ingrese # de horas trabajadas <-1 para fin>: 39 <ENTER>
Ingrese la tasa horaria (00.00 dólares): 10 <ENTER>
El salario es: 390 dólares

Ingrese # de horas trabajadas <-1 para fin>: 41 <ENTER>
Ingrese la tasa horaria (00.00 dólares): 10 <ENTER>
El salario es: 415 dólares

Ingrese # de horas trabajadas <-1 para fin>: -1 <ENTER>

21) El interés simple de un préstamo se calcula mediante la fórmula:

interes = capital * tasa * dias / 365

donde:

- **tasa**, es la tasa de interés anual, por lo que se divide para 365 (días).

Desarrollar un programa que introduzca el **capital** (principal), **tasa** y **dias** para varios préstamos diferentes, que calcule, y despliegue el interés simple para cada préstamo, mediante el uso de la fórmula anterior. Por ejemplo:

Ingrese el préstamo principal, <-1 para fin>: 1000 <ENTER>
Ingrese la tasa de interés: 0.1<<ENTER>
Ingrese el término del préstamo en días: 365 <ENTER>
El interés cargado es: USD. 100

Ingrese el préstamo principal, <-1 para fin>: 10000 <ENTER>
Ingrese la tasa de interés: 0.09 <ENTER>
Ingrese el término del préstamo en días: 1460 <ENTER>
El interés cargado es: USD. 3600

Ingrese el préstamo principal, <-1 para fin>: -1 <ENTER>

22) Una persona invierte **p** dólares en una cuenta de ahorros, que reditúa un interés del **r**%. Suponiendo que todo el interés se queda en depósito dentro de la cuenta, calcular e imprimir la cantidad de dinero en la cuenta durante **n** años.

Para la determinación del interés compuesto se utiliza la fórmula:

$a = p\,(1 + r)^n$

donde:

- **p**, es la cantidad original invertida (el principal).
- **r**, es la tasa anual de interés.
- **n**, es el número de años.
- **a**, es la cantidad del depósito al final del año **n**.

Realizar un programa para tasas de interés del 5%, 6%, 7%, 8%, 9% y 10%. Además el programa solo debe utilizar enteros para calcular el interés compuesto.

"*Sugerencia*": Tratar todas las cantidades monetarias como números enteros de centavos. A continuación "dividir" el resultado en su porción dólares y en su porción centavos, mediante el uso de las operaciones de división y de residuo respectivamente. Insertar un punto.

23) Realizar un programa que genere una tabla de intereses compuestos F/P, mediante la fórmula:

$$\frac{F}{P} = (1 + \frac{i}{100})^n$$

donde:

- **F**, representa el valor futuro de una cantidad dada de dinero.
- **P**, representa el valor actual.
- **i**, representa la tasa de interés anual, expresado como porcentaje.
- **n**, representa el número de años.

Cada fila en la tabla corresponde a un valor diferente de **n**, con **n** en el rango de 1 a 20.

Cada columna representa una tasa de interés diferente: 4 a 10 en incrementos de 0.5.

Además el programa debe poner los mensajes (etiquetas) apropiados en las filas y columnas.

24) Cuando se realiza en un banco un préstamo de **P** dólares, cada mes se deberá devolver **C** dólares hasta que se haya completado la cantidad total prestada. Parte del pago mensual serán intereses calculados como el **i** por ciento de la cantidad aún no pagada. El resto del pago servirá para reducir la cantidad adeudada.

    Escribir un programa que determine la siguiente información:

    a) La cantidad de interés pagado por mes.
    b) La cantidad de dinero aplicado a la reducción de la deuda total cada mes.
    c) La cantidad total de intereses que se lleva pagada al final de cada mes.
    d) La cantidad de deuda aún no pagada al final de cada mes.
    e) El número de pagos mensuales necesarios para devolver el préstamo.
    f) La cuantía del último pago, porque puede ser menor a **C**.

25) Realizar un programa que calcule la "***media ponderada***" de una lista de **n** números, utilizando la fórmula:

    $X_{media} = f_1x_1 + f_2x_2 + \cdots + f_nx_n$

    donde:
    - $\mathbf{f_i}$
    -
    - son el peso de cada cantidad y deben ser fraccionarios que cumplen las condiciones:

      $0 <= f_i <1$
      $f_1 + f_2 +...+ f_n = 1$

    NOTA: Validar el ingreso de los $\mathbf{f_i}$ para que cumplan las dos condiciones anteriores.

26) Realizar un programa que calcule la "***media geométrica***" de una lista de **n** números, utilizando la fórmula:

    $x_{media} = [x_1\, x_2 \ldots\, x_n]^{1/n}$

27) Determinar las raíces de la ecuación cuadrática:

    $ax^2 + bx + c = 0$

    utilizando la fórmula:

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

Indicar que tipo de raíces son las obtenidas, de acuerdo a las siguientes condiciones:

Si $b^2 - 4ac < 0$, las raíces son imaginarias.
Si $b^2 - 4ac = 0$, existe una sola raíz real.
Si $b^2 - 4ac > 0$, las raíces son reales.

28) "***Los números de Fibonacci***" son miembros de una serie en la que cada número es igual a la suma de los números anteriores, así:

$F1 = F_{i-1} + F_{i-2}$

donde:

- F_i, es el i-ésimo número de **Fibonacci**.

Los dos primeros números de la serie de **Fibonacci** son por definición iguales a 1. Por ejemplo, los cuatro primeros números de la serie serían:

$F_1 = 1$
$F_2 = 1$
$F_3 = F_2 + F_1 = 1 + 1 = 2$
$F_4 = F_3 + F_2 = 2 + 1 = 3$

Escribir un programa que determine los **n** primeros números de la serie de **Fibonacci**.

29) Escribir un programa que tenga un menú para cada uno de los siguientes literales:

a) Leer un entero no negativo, que calcule e imprima su factorial. "***El factorial***" de un entero no negativo **n** se escribe como **n!** y se define como sigue:

n! = n.(n - 1).(n - 2) … 1 (para valores de **n** mayor o igual que 1)

y

n! = 1 (para **n** = 0)

b) Calcular el valor de la constante matemática **e**, utilizando la fórmula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

c) Calcular el valor de **e**^{x}, utilizando la fórmula:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \qquad -\infty < x < \infty$$

30) Se puede calcular el **seno** de un ángulo **x** de forma aproximada, utilizando la siguiente serie infinita:

$$seno\ (x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots,$$

donde:

- **x**, se encuentra en radianes (1 radián = 180°).

Escribir un programa que lea el valor de **x** y calcule su **seno**. El programa realizarlo de dos formas diferentes:

a) Sumar los **n** primeros términos, en donde **n** es un entero positivo que se introduce desde el teclado.

b) Sumar términos de la serie hasta que el valor del término siguiente sea menor (en módulo) a un error seleccionado por el usuario. Por ejemplo, un error típico de 10^{-5}.

En este caso indicar el número de términos utilizados para conseguir la respuesta final.

31) El valor de **π** se puede calcular a partir de la siguiente serie infinita:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$$

Realizar un programa para imprimir una tabla que muestra el valor de **π**: aproximado a un término de esta serie, a dos, a tres, etc. ¿Cuántos términos de esta serie tendrá que utilizarse antes de que empiece a tener 3.14?, 3.141?, 3.1415?, 3.14159??.

32) "***Un número primo***" es una cantidad entera que es divisible solo por 1 y por sí mismo. Escribir un programa que imprima una lista con los **n** primeros números primos.

33) Escribir un programa que lea un valor entero positivo y determine lo siguiente:

a) Si el entero es un número primo.
b) Si el entero es un número de Fibonacci.

El programa debe repetirse hasta que se introduzca un cero como valor en la entrada.

34) Una "***terna pitagórica***" es el conjunto de los tres valores enteros correspondientes a los lados de un triángulo rectángulo. Estos tres lados deben de satisfacer la relación: de que la suma de los cuadrados de dos de los lados (catetos) es igual al cuadrado de la hipotenusa.

Realizar un programa que encuentre todos las ternas pitagóricas para **lado1**, **lado2** e hipotenusa, todos ellos no mayores de 500. Utilizar un lazo **for** de triple anidamiento, que pruebe todas las posibilidades.

35) Escribir un programa que imprima una tabla de los equivalentes binarios, octal y hexadecimal de los números decimales en el rango de 1 al 256.

36) Escribir un programa que introduzca el año en el rango de 1994 al 2020, y utilice un lazo para producir un calendario condensado e impreso de forma nítida. Considerar los años bisiestos.

37) Los estudiantes de un curso de programación rindieron **n** exámenes. Escribir un programa que acepte como entrada cada nombre de estudiante y sus notas de exámenes, que determine la media de cada estudiante, y escriba el nombre del estudiante, las notas de los exámenes y la media calculada. Además, calcular e imprimir la nota media de la clase.

Para calcular la media de cada estudiante se dará distinto peso a cada examen, entonces ingresar al inicio del programa el número (**n**) y pesos de los exámenes. Por ejemplo, que cada uno de los primeros cuatro exámenes contribuye con el 15 por 100 de la nota final, y que cada uno de los dos últimos contribuye con el 20 por 100.

38) Generar una gráfica en la pantalla y en la impresora de la siguiente función:

$y = e^{-0.1t} seno\ 0.5t$

Se debe utilizar un asterisco (*) para cada punto que aparezca en la gráfica. Hacer que la gráfica se imprima verticalmente, con un punto (un asterisco) por línea.

"*Sugerencia*": Determinar la posición del asterisco redondeando el valor de **y** al entero más cercano y utilizando una escala adecuada al número de caracteres por línea. Cada línea impresa debe consistir en un asterisco precedido por el número adecuado de espacios en blanco.

39) Escribir un programa que convierta una fecha introducida de la forma **dd-mm-aa**, en un entero que indique el número de días a partir del 1 de enero de 1960. Para esto utilizar las siguientes relaciones:

a) El **dia** del año en curso se puede determinar aproximádamente mediante:

   dia = (int) (30.42 * (mm - 1)) + dd

b) Si mm == 2 (febrero), incrementar el valor de **dia** en 1.

c) Si mm > 2 y mm < 8 (marzo, abril, mayo, junio o julio), decrementar el valor de **dia** en 1.

d) Si aa % 4 == 0 y mm > 2 (año bisiesto), incrementar el valor de **dia** en 1.

e) Incrementar el valor de **dia** en 1461 por cada ciclo completo de cuatro años a partir del 1-1-60.

f) Sumar 365 a **dia** por cada año a partir del último ciclo completo de cuatro años, sumar entonces 1 (por el último año bisiesto más reciente).

40) Escribir un programa que permita leer tres números y determine el valor máximo, indicando cuáles de los números ingresados corresponde a ese máximo.

41) Realizar un programa que cuente el número de cada tipo de calificación expresada en letras, que los estudiantes de un curso alcanzaron en un examen. El usuario debe ingresar las calificaciones, en letras, correspondientes a cada tipo, hasta que se ingrese fin de archivo <<Ctrl Z>>; en caso de ingresar una letra incorrecta se presenta un mensaje de error. Se tiene 5 tipos de calificaciones en orden descendente: 'A' o 'a', 'B' o 'b', 'C' o 'c', 'D' o 'd' y 'E' o 'e'.

Al finalizar el ingreso se escribirán los totales para cada calificación, y por último calcular la calificación promedio de la clase.

42) Una empresa paga a sus empleados como se muestra a continuación:

"*Gerentes*". Reciben un salario semanal fijo.

"*Trabajadores horarios*". Reciben un salario horario fijo por las primeras 40 horas de trabajo, y reciben 1.5 veces su sueldo horario para las horas extras trabajadas.

"*Trabajadores a comisión*". Reciben 250 dólares más 5.7% de sus ventas semanales brutas.

"*Trabajadores a destajo*". Reciben una cantidad fija de dinero por cada una de las piezas que produce cada trabajador a destajo, que trabaja solo un tipo de piezas.

Escribir un programa para calcular la nómina semanal de cada empleado, no se sabe por anticipado el número de empleados y cada tipo de empleado tiene su propio código de nómina, así:

1 los gerentes.
2 los trabajadores.
3 los trabajadores a comisión.
4 los trabajadores a destajo,

Utilizar una sentencia **switch** para calcular la nómina de cada empleado, basado en el código de nómina de dicho empleado.

43) Escribir un programa que permita leer el **nombre** y el **sueldo** de tipo **long**, de los empleados de una empresa y calcular el número de billetes de 100, 50, 20, 10 y 5 dólares que se debe retirar del banco para no tener que dar vuelto al empleado. El sueldo deberá ser múltiplo de 5. El ingreso de datos se termina cuando se digite <<ENTER>> en lugar de nombre. Listar en columnas el nombre, el sueldo, el número de billetes por tipo para cada empleado. Además listar el número total de billetes por tipo.

44) El equipo de fútbol L.D.U. ha tenido una buena campaña y desea premiar a sus jugadores con un aumento del salario para la siguiente temporada.

Los sueldos deben ajustarse de la siguiente tabla:

SUELDO ACTUAL (dólares)	AUMENTO (%)
0 – 2000	20
2001 – 3500	10
3501 – 8000	5
sobre 8000	nada

El equipo tiene como máximo un cuadro de 30 jugadores. Diseñar un programa que lea el nombre del jugador y su salario actual, y que a continuación imprima el nombre, el sueldo actual y el sueldo aumentado; al final de la lista debe proporcionar también el monto total de la nómina actual y el monto de la nueva nómina que incluye los aumentos mencionados.

45) Una empresa de ventas por correo vende cinco productos distintos, cuyos precios de menudeo se muestran en la tabla siguiente:

NUMERO DE PRODUCTO	PRECIO AL MENUDEO (dólares)
1	29.80
2	45.00
3	99.80
4	44.90
5	68.70

Escribir un programa que lea una serie de pares de números, como sigue:

1. Número del producto.
2. Cantidad vendida en un día.

El programa deberá utilizar una sentencia **switch** para ayudar a determinar el precio de menudeo de cada producto. Además, calcular y desplegar el valor total de menudeo, de todos los productos vendidos la semana pasada.

46) Realizar un programa que ingrese desde teclado dos números enteros positivos, para determinar si son amigos. Dos números enteros son amigos si la suma de los divisores del primer número es igual al segundo número; en forma similar debe cumplir para el segundo número.

47) Una compañía ha reducido a la mitad los límites de crédito de sus clientes para impedir que sus cuentas por cobrar (el dinero que debe) crezca demasiado. Por ejemplo, si un cliente particular tenía un límite de crédito de 2000 dólares el límite de crédito de este cliente es ahora de 1000 dólares. Escribir un programa que analice el estado de crédito de **n** clientes de esta empresa. Para cada cliente se le dará:

a) El número de cuenta del cliente.
b) El límite anterior de crédito del cliente.
c) El saldo actual del cliente (cantidad que el cliente le debe a la empresa).

El programa deberá calcular e imprimir el nuevo límite de crédito para cada cliente, determinar e imprimir qué clientes tienen saldos actuales que exceden sus nuevos límites de crédito.

Capítulo 5

PUNTEROS

Las razones más poderosas para usar los punteros son:

1. Proporcionar los medios por los cuales las funciones pueden modificar sus argumentos de llamada.

2. Soportar las rutinas de asignación dinámica del lengueje C, como listas enlazadas, árboles, etc.

3. Mejorar la eficiencia de ciertas rutinas mediante el uso de arreglos, cadenas, estructuras, etc.

Como se mencionó en la introducción, la teoría y ejemplos de punteros en esta obra corresponden a una arquitectura del computador de 16 bits (2 bytes), para simplificar la explicación y facilitar su entendimiento.

5.1. VARIABLE PUNTERO

Es una variable que almacena una dirección de memoria de otra variable, siendo esta dirección la posición interna de la variable en la memoria RAM del computador. Es decir, es una variable apuntando a otra, como se muestra en la Figura 5.1.

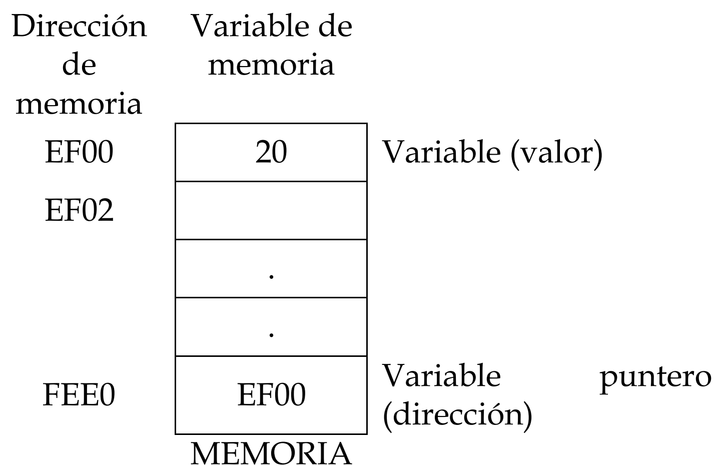


Figura 5.1. Variables en la memoria RAM del computador

5.2. DECLARACIÓN DE LA VARIABLE PUNTERO

La forma general para declarar una variable puntero es: (SCHILDT, 1994)

tipo *nombre;

donde:

- **tipo**, es cualquier tipo válido del lenguaje C.

- **nombre**, es el nombre de la variable puntero.

"**Tipo base del puntero**" define el tipo de variable a la que puede apuntar el puntero, especificándose a la vez el tamaño del dato. Debe anotarse que el puntero solo apunta al primer byte del dato. El tipo base del puntero se define en la declaración de la variable puntero.

El número de bytes ha utilizar de la variable puntero es 2, por ser un tipo entero sin signo representado en hexadecimal, por ejemplo, EF00. Pero en algunos compiladores como el lenguaje C++ tiene cuatro bytes, siendo un puntero largo, los dos punteros bytes corresponden al "segmento de memoria" y los dos restantes, al "desplazamiento" dentro del segmento, así: 5B5B:EF00.

La dirección nula es la macro NULL que se encuentra definida en el archivo de cabecera "***stdio.h***", y corresponde al valor en hexadecimal 0000.

Técnicamente, cualquier tipo de puntero puede apuntar a cualquier lugar de la memoria, sin embargo, toda la aritmética de punteros está hecha en relación a su "tipo base".

5.3. OPERADORES DE PUNTEROS

Existen dos operadores de punteros que son: dirección (**&**) e indirección (*****).

5.3.1. El operador de Dirección (&)

Es un operador unario que devuelve la dirección de memoria de su operando que debe ser una variable.

Por ejemplo, si se tiene la siguiente declaración:

```
int *m, cuenta = 20;
```

al asignar:

```
m = &cuenta;
```

m recibe "la dirección" de la variable **cuenta**.

Si **cuenta** está ubicada en la posición EF00, en la cual se guarda el valor 20, entonces **m** tiene el valor EF00.

En la Figura 5.2 se muestra las variables **cuenta** y **m** en la memoria RAM del computador.

Dirección de memoria	MEMORIA	Variable
EF00	20	cuenta
EF02		
	.	
	.	
FEE0	EF00	m

Figura 5.2. Variables **cuenta** *y* **m** *en la memoria*

5.3.2. El operador de Indirección (*)

Es un operador unario complemento de **&**, que devuelve el valor de la variable localizada en la dirección del operando que debe ser un puntero. Por ejemplo, si se tiene la siguiente declaración:

```
int *m, q, cuenta = 20;
```

al asignar:

```
m = &cuenta;
q=*m;
```

q recibe el "valor" (contenido) de la dirección **m**.

Esto es equivalente a realizar la sentencia:

```
q = cuenta;
```

Si **cuenta** está ubicada en la posición EF00, en la cual se guarda el valor 20, y **q** está en la dirección EF02; entonces **q** tendrá el valor de 20, que está almacenado en la posición EF00, que es la dirección asignada a **m**, como se muestra en la Figura 5.3.

Dirección de memoria	MEMORIA	Variable
EF00	20	cuenta
EF02	20	q
	.	
FEE0	EF00	m

Figura 5.3. Variables **cuenta, q** *y* **m** *en la memoria*

Las variables punteros deben apuntar siempre al "tipo base" correspondiente, pero puede haber errores si se asigna a un tipo base que no corresponda.

Por ejemplo, mediante un puntero entero asignar el valor de la variable de tipo flotante **x** a la variable de tipo flotante **y**.

```
/* PROG0501.C */

#include <stdio.h>

void main ()
{
  float x = 100, y;
  int *p;

  p = &x;
  y = *p;
  printf ("x = %.2f, y = %.2f\n", x, y);
}
```

Si el compilador no detecta este tipo de errores, la salida de este programa podría ser:

X = 100.00, y = 0.00

Pero el programa no produce el resultado deseado, porque no se asigna el valor de **x** a **y**, debido a que p se declara como un puntero a entero, por lo que solo se transfiere a **y** los 2 bytes menos significativos de los 4 bytes de información de **x**., como se muestra en la Figura 5.4.

Dirección de memoria	MEMORIA	Variable
EF00	100	x
EF02	?	y
	.	
FEE0	EF00	p

Figura 5.4. Variables **x**, **y** *y* **p** *en la memoria*

Además, se pueden producir errores por incompatibilidad entre el tipo de dato y el puntero utilizado, por lo que no se podría hacer la asignación:

p=&x;

5.4. OPERACIONES CON PUNTEROS

Las expresiones que involucran punteros se ajustan a las mismas reglas que cualquier otra expresión del lenguaje C.

5.4.1. Asignación de Punteros

Se puede asignar una dirección o un puntero solamente a una variable puntero.

Por ejemplo, el siguiente programa imprime la dirección y el valor de la variable **x**, mediante punteros.

```
/* PROG0502.C */

#include <stdio.h>

void main ()
{
  int x = 100;
  int *p1, *p2;

  p1 = &x;  /* Se asigna la dirección de x. */
  p2 = p1;  /* Se asigna el contenido del puntero p1. */
  printf ("Dirección de x = %p, Puntero p2 = %p\n", &x, p2);
  printf ("Contenido del puntero p2 = %d\n", *p2);
}
```

La salida de este programa con un puntero de 2 bytes, podría ser:

Dirección de x = EF00, Puntero p2 = EF00
Contenido del puntero p2 = 100

Como el operador **&** es el complemento del operador *****, entonces ***&p2** es igual a **p2**, que corresponde a la dirección de **x**; para **&*p2**, sucede igual. En el caso de tener ***&x**, es igual a **x**; pero en **&*x** produce un error.

5.4.2. Aritmética de Punteros

Toda la aritmética de punteros se lleva a cabo de acuerdo al tipo base del puntero, de forma que el puntero siempre apunte a un elemento adecuado del tipo base.

Existen solo dos operaciones aritméticas que se pueden usar con variables punteros: la suma y la resta. (SCHILDT, 1994)

1. **La suma**. Se puede hacer por medio de una suma de un puntero y un entero, o por medio del operador incremento al puntero. Por ejemplo, con la declaración:

```
int *p;
```

si **p** apuntaría a un entero cuya dirección es EF00, después de realizar:

```
p++;
```

p contiene EF02.

Cada vez que se incrementa un puntero, éste apunta a la posición de memoria del siguiente elemento de su tipo base, o sea el puntero se incrementa en la longitud del dato al cual apunta, en este caso un entero de dos bytes.

Si **p** tiene la dirección EF00, al realizar:

- **p++**, se actualiza la dirección de **p**, en este caso da como resultado EF02.
- **p+1**, se mantiene la dirección de **p** con la dirección EF00, pero **p+1** apunta a EF02.

Por ejemplo, en la Figura 5.5 se tendría:

```
int *p = EF00;
```

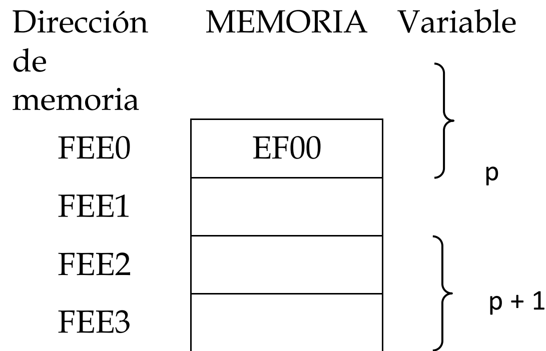


*Figura 5.5. Variable **p** en la memoria*

Ejercicio

En el siguiente programa imprimir la dirección de las variables **x** e **y**, asignar la dirección de **y** a un puntero para incrementarlo, y luego imprimir la dirección y contenido del puntero (que corresponde a la variable **x**).

```
/* PROG0503.C */

#include <stdio.h>

void main ()
{
  float x = 100, y = 200, *p;

  p = &y;
  p++;

  printf ("\nDirección de x = %p, Dirección de y = %p\n", &x, &y);
  printf ("Dirección del puntero = %p, Contenido del puntero = %.2f\n", p, *p);
}
```

La salida de este programa podría ser:

Dirección de x = EF04, Dirección de y = EF00
Dirección del puntero = EF04, Contenido del puntero = 100.00

NOTA: El crecimiento de la memoria en el segmento de pila es al revés, mayor a menor, por lo que cada vez que se declara una variable se almacena en celdas de direcciones menores.

2. **La resta**. Esta operación tiene la misma forma que la suma, pero en su lugar se resta.

5.4.3. Comparación de Punteros

Se pueden comparar dos punteros mediante los operadores de relación. Por ejemplo, si se declara:

```
int *p, *q;
```

se puede hacer la siguiente comparación:

```
if (p > q)
    printf ("p apunta a menor dirección de memoria que q\n");
```

Generalmente, la comparación de punteros se utiliza cuando dos o más punteros apuntan a un objeto común (a un mismo dato).

5.5. INICIALIZACIÓN DE PUNTEROS

Después de declarar un puntero, éste tiene un valor desconocido; por lo tanto, se debe inicializar el puntero a un valor conocido antes de usarlo.

Por convenio, cuando se inicializa un puntero se debe asignarle el valor NULL, para indicar que no apunta a una dirección de memoria específica. Esto se utiliza normalmente en asignación dinámica de memoria.

Por otro lado, es común inicializar las cadenas con un valor conocido. Por ejemplo:

```
char *p = "Hola que tal!";
```

"*Tabla de Cadenas Constantes*". Es creada por todos los compiladores de lengua C, que usa internamente el compilador para guardar las cadenas constantes utilizadas por el programa.

Por lo tanto, la sentencia anterior almacena la cadena constante "Hola que tal!" en la tabla de cadenas constantes, asignando su dirección al puntero **p**. Por esto, el puntero **p** puede utilizarse como una cadena.

Por ejemplo, al realizar la ejecución de las siguientes sentencias:

```
puts (p):
putchar (*p);
```

Imprimirá:

Hola que tal
H

Si a continuación se añade la primera sentencia:

```
p++;
puts (p):
putchar (*p);
```

Imprimirá:

ola que tal
o

Ejercicio

El siguiente programa imprimirá hacia atrás una cadena constante previamente definida.

```
/* PROG0504.C */

#include "stdio.h"

void main ()
{
  char *p = "Hola que tal!";
  char *aux;

  printf ("\nCadena original.\n");
  puts (p);

  for (aux = p; *aux; aux++);
  printf ("Cadena al rev,s.\n");
  for (--aux; aux >= p; aux--)
    putchar (*aux);
}
```

La salida de este programa será:

Cadena original.
Hola que tal!

Cadena al revés.
!lat euq aIoH

5.6. INDIRECCIÓN MÚLTIPLE (PUNTERO A PUNTERO)

Se puede hacer que un puntero apunte a otro puntero que a su vez apunte a un valor deseado. Por lo que se tiene dos tipos de indirección: (SCHILDT, 1994)

1. **Indirección simple**. El valor de un "puntero normal" es la dirección de la variable que contiene el valor deseado, como se muestra en la Figura 5.6.

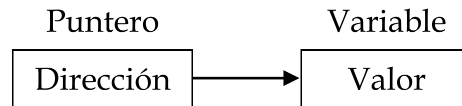


Figura 5.6. Indirección Simple

2. ***Indirección múltiple***. En "puntero a puntero" el primer puntero contiene la dirección del segundo puntero, que apunta a su vez a la variable que contiene el valor deseado, como se muestra en la Figura 5.7.

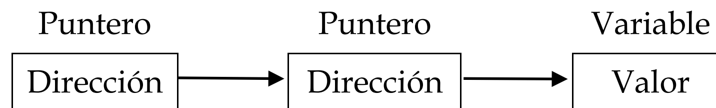


Figura 5.7. Indirección Múltiple

La indirección múltiple puede tener varios "puntero a puntero", pero existen pocos casos en los que se necesite más de un puntero a puntero. De hecho, la indirección excesiva es difícil de seguir y propensa a errores conceptuales.

Declaración de una variable puntero a puntero. Se añade ** al frente del nombre de la variable. Por ejemplo:

```
float **ptr;
```

donde:

- **ptr**, es un puntero a puntero de tipo **float**.

Para acceder al valor deseado indirectamente apuntado por un puntero a puntero, hay que poner dos veces el operador *.

Ejercicio

Imprimir el valor de la variable **x**, utilizando indirección múltiple.

```
/* PROG0505.C */

#include <stdio.h>

void main ()
{
  int x = 100;
  int *p, **q;

  p = &x;
  q = &p;
  printf ("x = %d\n", **q);
}
```

La salida de este programa será:

X = 100

En el ejercicio anterior, la impresión de la variable **x** también se puede realizar de las siguientes maneras:

```
printf ("x = %d\n", *p);
printf ("x = %d\n", x);
```

En la Figura 5.8 se muestra en la memoria las variables **x, p** y **q**.

Dirección de memoria	MEMORIA	Variable
EF00	100	x
	.	
FEE0	EF00	p
FEE2	FEE0	q

Figura 5.8. Variables* x *y* p *en la memoria

En la Figura 5.9 se muestra esquemáticamente la indirección múltiple de las variables **x**, **p** y **q**.

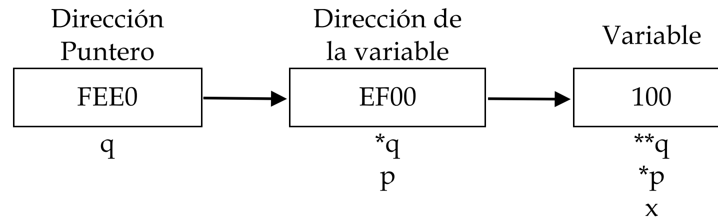


***Figura 5.9. Indirección múltiple de las variables* x, p *y* q**

Los valores que toman las variables **x**, **p** y **q**, son los siguientes:

q == FEE0
p == *q == EF00
x == *p == **q ==100

La aplicación más importante de la indirección múltiple se tiene cuando se quiere cambiar la dirección de un puntero en una función.

5.7. PROBLEMAS CON PUNTEROS

Cuando se comete un error en la utilización de un puntero, puede que cada vez que se realiza una operación utilizando ese puntero, se está leyendo o escribiendo en algún lugar desconocido de la memoria. Es decir:

- ***Si se lee*** de la dirección del puntero, lo peor que puede ocurrir es que se obtenga basura.
- ***Si se escribe*** en la dirección del puntero, puede suceder que se esté escribiendo en otras partes del código o de los datos.

Errores más comunes

1. "*Puntero no inicializado*". Este tipo de problema asigna el valor del dato a alguna posición desconocida de memoria, ya que el puntero nunca ha recibido un valor previamente.
   Este tipo de problema a menudo pasa inadvertido cuando el programa es pequeño, debido a que existen muchas posibilidades de que el puntero no contenga una dirección segura como: una dirección en el código, en el área de datos o en el sistema operativo. Sin embargo, a medida que el programa crece aumenta la posibilidad que el puntero apunte a una "dirección segura".
   La solución para este tipo de problemas es asegurarse siempre, que el puntero está apuntando a un dato válido antes de utilizarlo.

2. "*Desconocimiento del uso de punteros*". Se produce cuando se asigna valores de datos a los punteros en lugar de hacerlo con una dirección, lo que provoca que el programa termina abortando con un mensaje parecido a "Segmentation fault".

PROBLEMAS PROPUESTOS

1) Realizar un programa que ilustre la relación entre dos variables **u** y **v**, sus correspondientes direcciones y sus punteros asociados ***pu** y ***pv**.

   pu es un puntero a **u** y **pv** es un puntero a **v**. Por ejemplo, al ejecutar este programa podría producir la siguiente salida:

   u = 3, &u = EF8E, pu = EF8E, *pu = 3
   v = 3, &v = EF8C, pv = EF8C, *pv = 3

   La variable **u** representa el valor 3 que debe especificarse en la sentencia de declaración. Al puntero **pu** se le asigna la dirección de **u**, a **v** se le asigna el valor ***pu**, y a **pv** se le asigna la dirección de **v**.

   Además el programa debe tener los siguientes literales:

   a) Usar datos enteros, y asignar el valor inicial u=3.
   b) Usar datos en punto flotante, y asignar el valor inicial u=0.3.
   c) Usar datos en doble precisión, y asignar el valor inicial u=0.3x10^{45}.
   d) Usar caracteres, y asignar el valor inicial u='C'.

   Realizar un programa que inicialice la variable **v** con el valor de 5 y asignar la dirección de **v** a la variable puntero **pv**. Utilizar la función **printf()** para mostrar los valores actuales de ***pv** y **v**. Luego el valor de ***pv** es puesto a 0.

2) Utilizar la función **printf()**, para que muestre los nuevos valores de ***pv** y **v**.

   Al ejecutar el programa se generaría la siguiente salida:

   pv = 5, v = 5
   *pv = 0, v = 0

   Además el programa debe tener los siguientes literales:

   a) Usar datos enteros, y asignar el valor inicial v=5.
   b) Usar datos en punto flotante, y asignar el valor inicial v=1.5.
   c) Usar datos de doble precisión, y asignar el valor inicial v=1.2x10^{52}.
   d) Usar caracteres, y asignar el valor inicial v='A'.

3) Escribir un programa que ejecute con una sola sentencia lo indicado en cada literal. Suponiendo que han sido declaradas las variables enteras largas **valor1** y **valor2**, y que **valor1** se inicialice a 200000.

a) Declarar la variable **Ptr** que sea un puntero a una variable de tipo **long**.
b) Asignar la dirección de la variable **valor1** a la variable puntero **Ptr**.
c) Imprimir el valor de la variable apuntada por **Ptr**.
d) Asignar el valor de la variable apuntada por **Ptr** a la variable **valor2**.
e) Imprimir el valor de **valor2**.
f) Imprimir la dirección de **valor1**.
g) Imprimir la dirección almacenada en **Ptr**.

4) Elaborar un programa que lea una línea de entrada en una cadena de caracteres y cuente cuántos caracteres son dígitos, letras mayúsculas y letras minúsculas; luego imprimir los resultados. Usar teoría de punteros.

5) Realizar un programa que lea las letras del alfabeto inglés en una cadena de caracteres, hasta que encuentre un caracter diferente, luego convertir todas las letras a mayúsculas e imprimir la cadena indicando cuantos caracteres se leyeron. La función **toupper(ch)** convierte el caracter **ch** a mayúscula. Usar teoría de punteros.

6) Realizar un programa que lea una cadena de caracteres en mayúsculas y minúsculas, a continuación escribir las mayúsculas y las minúsculas intercambiadas, todos los dígitos reemplazados por ceros y el resto de los caracteres sustituidos por asteriscos. Usar teoría de punteros.

7) Realizar un programa que ingrese desde teclado un texto que se almacene en una cadena de caracteres para obtener su cadena inversa (hacia atrás). Luego imprimir las dos cadenas y sus direcciones. Usar teoría de punteros.

8) Realizar un programa que ingrese desde el teclado dos cadenas de caracteres para unirlas, la segunda cadena a continuación de la primera, modificándose la primera cadena. Imprimir las dos cadenas con los respectivos mensajes. Usar teoría de punteros.

9) Escribir un programa que convierta una cantidad romana a cantidad entera positiva. Diseñar el programa para que se ejecute repetidamente. Usar teoría de punteros.

10) Realizar un programa que introduzca una cadena de caracteres, para modificarla letra a letra restando 30 del valor numérico que se utiliza para representar cada letra en código ASCII. Escribir la cadena en forma codificada, luego decodificar la cadena y escribirla.

11) Elaborar un programa que lea desde teclado cadenas de caracteres hasta que se digite una cadena nula, en donde los caracteres de cada cadena están ordenados de tal manera que representan un número en notación científica.

Luego el programa debe determinar e imprimir cada número en punto flotante que corresponda a la cadena de caracteres.

Por ejemplo, la cadena:

1.52E+04

debe convertirse a:

15200

12) Realizar un programa para crear un menú con la sentencia **switch**, que tenga las siguientes opciones:

0. Salir.
1. Cuenta caracteres blancos.
2. Cuenta letras.
3. Cuenta dígitos.

El menú debe aparecer cada vez que se desea una opción.

Las opciones del menú realizan lo siguiente:

0. Terminar el programa y vuelve al sistema operativo.

1. Cuenta el número de espacios en blanco que tiene una cadena de texto ingresada desde teclado.

2. Cuenta el número de letras del alfabeto inglés (mayúsculas y minúsculas) que tiene una cadena de texto ingresada desde teclado.

3. Cuenta el número de dígitos que tiene una cadena de texto ingresada desde teclado.

13) Se quiere analizar una línea de texto determinando cada caracter a qué categoría pertenece. Contar el número de vocales, consonantes, dígitos, espacios en blanco y "otros" caracteres.

Los caracteres de nueva línea no son incluidos en la categoría "blancos", ya que no pueden estar dentro de una línea simple de texto, y las letras deben ser convertidas a letra mayúsculas.

Además, contar el número de palabras y el número total de caracteres de la línea de texto. Una palabra nueva puede ser reconocida por la ocurrencia de un caracter blanco seguido de otro caracter no blanco.

El programa debe leer una línea de texto y almacenarla en una cadena de caracteres, luego determinar el valor de cada contador después de que todos los caracteres hayan sido analizados.

14) "***El generador de pig latin***" es una forma codificada de escribir y de hablar que suelen usar los niños ingleses como juego. Una palabra en "pig latin" se forma transponiendo el primer sonido, generalmente la primera letra, de una palabra original al final de la misma y añadiendo al final la letra '**a**' _ Por ejemplo, la palabra "perro" se convierte en "erropa".

    Escribir un programa que se ingrese una cadena de caracteres y escriba su correspondiente texto en "pig latin". Suponer que el texto está contenido en una cadena de 80 caracteres, con un solo espacio en blanco entre cada dos palabras sucesivas. Además, considerar marcas de puntuación, letras mayúsculas y sonidos de letras dobles.

Capítulo 6

FUNCIONES

6.1. INTRODUCCIÓN

Una función es un bloque de código de programa autocontenido, diseñado para realizar una tarea determinada.

La razón principal para usar funciones es para evitar tediosas repeticiones de programación. Escribiendo una sola vez la función apropiada, se la puede emplear cualquier número de veces en un determinado programa, en diferentes situaciones, únicamente llamándola por su nombre.

La filosofía del diseño del lenguaje C está basada en el empleo de funciones, ya que un programa estructurado en funciones es más modular, por tanto más fácil de leer, modificar o arreglar.

Cuando una función es lo suficientemente general, se la puede emplear en diferentes programas, teniéndola a disposición en una librería. Para incluir la función en un programa se utiliza el comando de preprocesador **#include**.

En general, se plantean las funciones como "cajas negras", para lo cual se las define mediante su **entrada** y su **salida**, es decir, mediante la información que hay que suministrarlas y el producto recibido de ellas, respectivamente. Entonces, lo que sucede dentro de la función no es de nuestra incumbencia, a menos que dicha función sea diseñada por el usuario. Si se considera las funciones como "cajas negras", se puede centrar en el diseño global del programa en lugar de preocuparse por detalles.

Además, la función **main()** se puede ubicar en cualquier lugar del programa, pero es conveniente ubicarla al principio, para leer con facilidad el programa, porque la función **main()** contiene la parte principal del programa.

La forma general de una función es: (SCHILDT, 1994)

```
tipo nombre_función (lista de parámetros)
{
    /* Cuerpo de la función. */
}
```

donde:

- **tipo**, especifica el valor de cualquier tipo válido que retorna lafunción mediante la sentencia **return**. Si no se especifica ningún tipo, el compilador asume que la función retorna un valor entero.

- **nombre_función**, es el nombre de la función que debe ser un nombre descriptivo sobre lo que va a realizarse.

- **lista de parámetros**, es la lista de nombres de variables, separados por comas, con sus tipos asociados. Esas variables reciben los valores de los argumentos cuando se llama a la función. Por ejemplo:

```
void nombre (tipo var1, tipo var2, tipo varN)
{
    /* Sentencias. */
}
```

  Una función puede no tener parámetros, en cuyo caso la lista de parámetros estará vacía, pero deben mantenerse los paréntesis. Para indicar que no hay parámetros se puede poner la palabra clave **void**.

Ámbito de las funciones. Esto significa si un bloque de código tiene acceso (conoce) a otro bloque de código o a ciertos datos.

El código de una función es exclusivo de esa función y no es accesible por otra función, a menos que se haga una llamada a esa función. Es decir, el código y los datos que están definidos dentro de una función, no pueden interactuar con el código y los datos definidos dentro de otra función, porque las funciones tienen un ámbito diferente y las funciones en lenguaje C no son anidadas.

Todas las funciones en lenguaje C están al mismo nivel de ámbito, por lo que no se pueden definir "***funciones anidadas***". Entonces, el lenguaje C técnicamente no es un lenguaje estructurado en bloques.

Variables locales. Son aquellas que están definidas dentro de una función, y comienzan a existir cuando se entra a la función y se destruyen al salir de ella, por lo que las variables locales no pueden conservar sus valores después de haberse ejecutado la función, debido a que las variables se almacenan en el segmento de "pila".

6.2. TIPOS DE FUNCIONES

Las funciones en su mayoría son de tres tipos: (SCHILDT, 1994)

1. Las funciones "puras" son diseñadas específicamente para realizar operaciones con sus argumentos y devolver un valor en su nombre, basándose en esas operaciones.

   Por ejemplo, la función pura de biblioteca estándar **sqrt()**, que calcula la raíz cuadrada de sus argumentos.

2. Las funciones que manipulan la información y devuelven un valor por la función. Ese valor indica simplemente el éxito o el fallo de la manipulación, es decir, estas funciones producen acciones y suministran datos.

   Por ejemplo, la función estándar **fclose()**, que se usa para cerrar un archivo: devuelve un valor 0 si la operación de cierre tiene éxito; si no tiene éxito, la función devuelve un código de error.

3. Funciones que no tienen un valor de vuelta explícito, es decir, realizan una acción concreta y no devuelven un valor en su nombre. Todas las funciones que no devuelven valores deben ser declaradas de tipo **void**, por lo tanto se evita que se use esta función en una expresión.

   Por ejemplo, la función **exit()**, que termina la ejecución de un programa.

Además, algunas funciones retornan un valor que no es de mucho interés, por lo que no se tiene que usar necesariamente ese valor. Entonces, dicho valor no se asigna a ninguna variable para ignorarlo. Por ejemplo, la función **scanf()** devuelve el número de argumentos válidos leídos, que normalmente no es utilizado.

6.3. ARGUMENTOS Y PARÁMETROS DE FUNCIONES

6.3.1. Argumentos

Son los valores que se envían a la función en el momento en que se realiza su llamada, estos valores pueden ser de una constante, variable o expresión. Por ejemplo, la llamada de la función **calculo()** podría ser:

```
int numero = 10;
calculo (numero, 'A' , numero * 2);
```

Cuando se ejecuta esta ***sentencia de llamada a función***, el control del programa cambia a la función con el nombre especificado (en este caso **calculo()**) y ejecuta las sentencias indicadas allí, al terminar retorna el control de programa a la siguiente línea de la llamada a la función, si es de tipo **void**; o al punto de llamada, si es una función que retorna un valor.

6.3.2. Parámetros Formales

Es la lista de parámetros o variables que reciben los valores de los argumentos que se pasan a la función, por lo que deben ser declarados. Se comportan como otras variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir.

Al igual que con las variables locales, se pueden hacer asignaciones a los parámetros formales de una función o usarlos en cualquier expresión válida del lenguaje C.

Los parámetros formales son la interfaz entre el programa de llamada y la función. En este caso la función se usa como caja negra, conociéndose únicamente la entrada y salida de esa función mediante esos parámetros. Entonces estos parámetros sirven de comunicación con las funciones. Por ejemplo, la implementación de la función **calculo()** de la llamada anterior sería:

```
void calculo (int num, char ch, int prod)
{
    /* Sentencias. */
}
```

Los parámetros formales de esta función tomarían los valores: **num=10**, **ch='A'** y **prod=20**.

Los parámetros formales son del mismo tipo que los argumentos usados para llamar a la función, es decir, debe existir una correspondencia en tipo y número entre los argumentos y los parámetros. Si existe un error en los tipos, el compilador no mostrará un mensaje de error, pero se obtendrán resultados inesperados.

En general, se pueden pasar argumentos a una función de dos formas:

1. Llamada por valor.
2. Llamada por dirección.

6.4. LLAMADA POR VALOR

Este método, también llamado paso por valor, copia el valor de un argumento en el parámetro formal de la función, en el momento de la llamada. De esta forma, los cambios en los parámetros de la función no afectan a las variables (argumentos) que se usan en la llamada. Esto significa que no se pueden alterar las variables de los argumentos usadas para llamar a la función.

Por ejemplo, realizar una función **suma()** que reciba dos valores enteros, los sume e imprima su resultado:

- Implementación de la función:

```
void suma (int x, int y)
{
    printf ("Suma = %d\n", x + y);
}
```

- Llamada de la función:

```
int u = 2;
int v = 3;
suma (u, v);
```

Esta función imprimirá:

Suma = 5

Los valores de las variables **x** e **y** al ejecutar la función, se muestran en la Figura 6.1, luego que se termine de ejecutar la función desaparecen.

Dirección de memoria	MEMORIA (PILA)	Variable
FE02	2	u
FEE0	3	v
	.	
EF02	2	x
EF00	3	y

Figura 6.1. Valores de las variables* x *e* y *al ejecutar la función

6.5. LLAMADA POR DIRECCIÓN

En este método se copia la dirección del argumento en el parámetro formal, por lo que los parámetros deben ser variables punteros y los argumentos deben ser direcciones de memoria a variables o punteros.

Una llamada por dirección se obtiene pasando la dirección del argumento a la función, por lo que dentro de la función se usa la dirección real de las variables usadas en la llamada, entonces, los cambios hechos a los parámetros correspondientes afectan a dichas variables. Es decir, la llamada por dirección usa un puntero para cambiar el valor del argumento de la función.

Los punteros se pasan a las funciones como cualquier otro valor, por supuesto es necesario declarar los parámetros como tipo puntero. Por ejemplo, realizar una función **intercambio()** que intercambie el contenido de sus dos argumentos enteros:

- Implementación de la función:

```
void intercambio (int *x, int *y)
{
    int aux;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

- Llamada de la función:

```
intercambio (&u, &v);
```

Si las variables tienen los valores u=10 y v=20, luego de la llamada se intercambian los contenidos de estas variables a u=20 y v=10, como se muestra en la Figura 6.2.

Dirección de memoria	MEMORIA (PILA)	Variable
FEE2	20	u
FEE0	10	v
	.	
EF04	FEE2	x
EF02	EF00	y
EF00	10	aux

Figura 6.2. Valores de las variables **u** *y* **v** *luego de ejecutar la función*

De esta forma se intercambian los contenidos de las variables usadas en la llamada de la función, ya que se utilizan las direcciones de los argumentos y no sus contenidos.

"***Funciones de propósito general***". Una función de propósito general es aquella que se va a utilizar en muchas situaciones distintas, y por muchos programadores diferentes. Por lo que las funciones de propósito general no deben basarse en datos globales, ya que toda la información que necesiten estas funciones se la debe pasar mediante parámetros, para que no se produzca errores laterales y el código sean más legible.

6.6. PROTOTIPOS DE FUNCIONES

Para poder usar una función, primero hay que declararla, de esta manera se conoce: el tipo de dato que puede tomar la función, el número y los tipos de los argumentos. Esta declaración constituye el prototipo de la función, el mismo que permite que el lenguaje C lleve a cabo una comprobación del valor retornado de la función. Del número y tipos de argumentos. Por ejemplo: (SCHILDT, 1994)

- Elenguaje C puede encontrar o informar sobre conversiones de tipo ilegales entre el tipo de los argumentos usados en la llamada a la función y las definiciones de tipo de sus parámetros.

- El lenguaje C también detectará diferencias entre el número de argumentos usados en la llamada a la función y el número de parámetros de la misma.

La forma general de un prototipo de función es similar a la declaración de las variables, es decir, el especificador de tipo precede al nombre de la función con sus parámetros entre paréntesis, y termina con punto y coma, para indicar que es una "*sentencia de declaración*"; como se muestra a continuación:

tipo nombre_función (tipo par1, tipo par2, tipo parN);

Por ejemplo, el prototipo de la función **intercambio()** será:

void intercambio (int *x, int *y);

Si se tienen varias funciones con el mismo tipo de valor retornado, se pueden declarar como las variables, separadas con comas. Además, si se declaran las variables dentro de una función, solo pueden ser usadas dentro de la misma, por lo que se las debe declarar al principio del programa para que puedan ser usadas por cualquier función.

El uso de los nombres de los parámetros es opcional en el prototipo de la función. Pero, los nombres de los parámetros en la implementación de la función deben colocarse siempre y deben conservar el mismo nombre que los del prototipo, si ahí fueron especificados, aunque no tengan el mismo nombre de los argumentos.

Ejercicio

Realizar un programa completo que tenga la función **intercambio()**, la misma que intercambio el valor de sus dos argumentos enteros.

/* PROG0601.C */

```
#include "stdio.h"

/* Prototipo de la función. */
void intercambio (int *x, int *y);

void main ()
{
 int u, v;

 printf ("Ingrese el valor de u: ");
 scanf ("%d", &u);
 printf ("Ingrese el valor de v: ");
 scanf ("%d", &v);
 printf ("Valores originales\n");
 printf ("u = %d, v = %d\n", u, v);
 intercambio (&u, &v);
 printf ("Valores intercambiados\n");
 printf ("u = %d, v = %d\n", u, v);
}

/* Implementación de la función. */

void intercambio (int *x, int *y)
{
 int aux;

 aux = *x;
 *x = *y;
 *y = aux;
}
```

La salida de este programa podría ser:

Ingrese el valor de u: 10 <ENTER>
Ingrese el valor de v: 20 <ENTER>
Valores originales
u = 10, v = 20
Valores intercambiados
u = 20, v = 10

6.7. DEVOLUCIÓN DE UN VALOR POR LA FUNCIÓN: SENTENCIA *return*

Cuando no está declarado el tipo de una función, en algunos compiladores de lenguaje C, asigna automáticamente por omisión el tipo **int**.

Para devolver un tipo de dato diferente a **int**, hay que declarar la función con un especificador de tipo de acuerdo al tipo deseado. El tipo de la función debe estar especificado antes de hacer su llamada, porque tipos de datos diferentes tienen tamaños diferentes y representaciones internas diferentes:

Por ejemplo, el prototipo de la función **suma()**, que sume dos números de punto flotante y su resultado se almacena por la función será:

```
float suma (float a, float b);
```

La sentencia **return** tiene dos usos importantes:

1. ***Valores devueltos***

La sentencia **return** devuelve un valor por la función que la contiene; es decir, todas las funciones, excepto aquellas de tipo **void**, devuelven un valor, este valor se especifica explícitamente en la sentencia **return**.

La forma general de la sentencia **return** es:

```
return expresión;
```

Si la sentencia **return** no es especificada en la función, el compilador da un mensaje de advertencia, indicando que la función debe tomar un valor. Y si no se especifica el valor en la sentencia **return**, el valor devuelto por la función queda técnicamente indefinido, produciéndose un "*error*". Por lo que se debe usar la sentencia **return** sin la "*expresión*" en funciones de tipo **void**, cuando sea necesario.

En otras palabras, mientras que una función no se declare como **void**, puede ser usada como operando en cualquier expresión válida del lenguaje C. Sin embargo, una función no puede ser destino de una asignación.

Ejercicio

Realizar una función que sume dos valores de punto flotante, y que dicha función sea usada en una asignación, impresión o simplemente se la llame perdiéndose su valor.

```
/* PROG0602.C */

#include "stdio.h"

float suma (float a, float b);
```

```
void main ()
{
 float x = 10.5, y = 20.3, z;

 z = suma (x, y);              /* Asigna el valor de la función a z. */
 printf ("suma 1 = %.2f, suma 2 = %.2f\n",
        z, suma (20.2, y));  /* Imprime el valor de la función.    */
 suma (x, y);                  /* Se pierde el valor de la función.  */
}

float suma (float a, float b)
{
 return a + b;
}
```

La salida de este programa será:

suma 1 = 30.80, suma 2 = 40.50

NOTAS:

- **La entrada** de una función se realiza utilizando los argumentos.
- **La salida** de una función, se maneja por medio de la sentencia **return**, para almacenar el valor por la función.

2. ***Terminación de una función***

La sentencia **return** obliga a una salida inmediata de la función en la que se encuentra, haciendo que el control del programa vuelva al punto de llamada de la función.

Hay dos formas en que una función puede terminar su ejecución y volver al sitio en que se la llamó:

a) Si la función es de tipo **void**, se ejecuta la última sentencia de la función, al finalizar su bloque de código (en la llave **}**), pero si no es de este tipo se produce un error. No se utiliza mucho esta forma de terminación.

b) La mayoría de las funciones emplean la sentencia **return** para terminar la ejecución de la función, cuando se tiene que devolver un valor por la función o cuando se desea hacer un código más eficiente.

   Una función puede tener varias sentencias **return**, pero en la primera que se ejecute se termina la función.

Ejercicio

Realizar un programa que tenga una función, para calcular el valor absoluto de un número entero.

```
/* PROG0603.C */

#include <stdio.h>

abs (int x);

void main ()
{
  int x;

  printf ("Ingrese x: ");
  scanf ("%d", &x);
  printf ("Valor inicial: x = %d\n", x);
  printf ("Valor absoluto de: x = %d\n", abs (x));
}

abs (int x)
{
  if (x < 0)
    return -x;
  return x;
}
```

Una salida del programa sería:

Ingrese x: -2 <ENTER>
Valor inicial: x = -2
Valor absoluto de: x = 2

6.8. DEVOLUCIÓN DE UN PUNTERO POR LA FUNCIÓN

Para retornar punteros por la función se utiliza la sentencia **return**, porque los punteros no son ni enteros ni enteros sin signo; son las direcciones de memoria de un cierto tipo de datos expresado en entero hexadecimal.

Para indicar que una función retorna un puntero como valor, se debe colocar un asterisco (*) antes del nombre de la función, igual a lo que sucede en la declaración de una variable puntero.

Ejercicio

Realizar una función que devuelva un puntero a la primera ocurrencia de la variable caracter **c**, cuando ésta exista en la cadena **s**. Pero si no existe, la función debe devolver un puntero NULL. Esta función es equivalente a la función estándar **strchr()**.

Por ejemplo:

cadena s: "Quito es bonito"
caracter c: 'i'

La dirección de la cadena es: FFA4
El caracter **i** se encuentra en la dirección: FFA6

```
/* PROG0604.C */

#include <stdio.h>

char *encuentra (char *s, char c);

void main ()
{
  char s[80], c;

  printf ("Ingrese la cadena: ");
  gets (s);
  printf ("Ingrese el caracter: ");
  scanf ("%c", &c);
  printf ("La dirección de la cadena es: %p\n", s);
  printf ("El caracter %c se encuentra en la dirección: %p\n", c,
          encuentra (s, c));
}

char *encuentra (char *s, char c)
{
  while (*s && c != *s)
    s++;
  return (*s) ? s : NULL;
}
```

Una salida del programa podría ser:

Ingrese la cadena: Quito es bonito <ENTER>
Ingrese el caracter: i <ENTER>

La dirección de la cadena es: FFA4
El caracter i se encuentra en la dirección: FFA6

También se puede realizar la devolución de un puntero por la función, mediante "punteros genéricos" que apunten a cualquier tipo de datos. En este ejercicio, la declaración de la función **encuentra()** sería:

```
void *encuentra (char *s, char c);
```

Para realizar la llamada a la función **encuentra()**, se debe hacer un "cambio de tipo" (casting) a **char**, así:

```
char *p;
p = (char *) encuentra (s, c);
```

Ejercicio

Realizar un programa que contenga una función **buscar_sub()**, la misma que devuelva el puntero de la primera ocurrencia del comienzo de una subcadena dentro de una cadena, o que devuelva un puntero NULL si no se ha encontrado dicha ocurrencia. Esta función es equivalente a la función estándar **strstr()**.

Por ejemplo:

cadena s1: "Este es un ejemplo"
subcadena s2: "es"

La cadena comienza en la dirección : FFA4
La subcadena comienza en la dirección : FFA9

```
/* PROG0605.C */

#include "stdio.h"

void *buscar_sub (char *s1, char *s2);

void main ()
{
  char *ptr;
  char s1[80], s2[80];

  printf ("Ingrese la cadena   : ");
  gets (s1);
  printf ("Ingrese la subcadena: ");
```

```
  gets (s2);

  printf ("La cadena comienza en la dirección   : %p\n", s1);

  if (ptr = (char *) buscar_sub (s1, s2))
    printf ("La subcadena comienza en la dirección: %p\n", ptr);
  else
    printf ("No se encuentra\n");
}

void *buscar_sub (char *s1, char *s2)
{
  char *p, *p2, *q;

  p = s1;
  for (; *p; p++) {            /* Lazo para la cadena. */
    q = p;
    p2 = s2;
    while (*p2 && *p2 == *p) {  /* Lazo para la subcadena. */
      p++;
      p2++;
    }
    if (!*p2)
      return q;
    if (*(p - 1) == *(q - 1))  /* Si los caracteres anteriores son iguales. */
      p = q;
  }
 return NULL;
}
```

Una salida del programa podría ser:

Ingrese la cadena : Este es un ejemplo <ENTER>
Ingrese la subcadena: es <ENTER>
La cadena comienza en la dirección : FFA4
La subcadena comienza en la dirección: FFA9

6.9. PUNTEROS A FUNCIONES

Una función no es una variable, pero tiene una posición física en memoria cuya dirección puede ser asignada a un puntero.

La dirección de la función es el punto de entrada a la misma, por lo que puede usarse un "puntero a función", para llamar a la función. Esto se puede realizar por las siguientes razones: (SCHILDT, 1994)

- Cuando se compila cada función, el código fuente se transforma en código objeto y se establece un punto de entrada a la función.

- Cuando se llama a una función mientras se ejecuta el programa, se hace una llamada en lenguaje de máquina a ese punto de entrada.

Por lo tanto, si un puntero contiene la dirección del punto de entrada a la función, puede utilizarse dicho puntero para llamarla.

La dirección de la función se obtiene utilizando el nombre de la función sin paréntesis ni argumentos.

En la llamada de la función mediante su dirección, se deben especificar los argumentos, ya que una función tiene argumentos.

La declaración de un "puntero a función" es:

```
int (*p)(); /* La función apuntada debe devolver un dato de tipo entero */
```

Cuando una función retorna un valor, la llamada se debe realizar con el contenido del "puntero a función" y sus argumentos, **(*p)()**. Pero si la función no retorna ningún valor, solo se llama con su dirección y sus argumentos, **p()**.

Si se omiten los paréntesis en la declaración anterior:

```
int *p();
```

es una función que devuelve un puntero a entero.

Ejercicio

Realizar un programa que contenga una función **comprobar()**, la misma que determine si dos cadenas son iguales o no, usando como parámetros dos punteros a caracteres para las cadenas y un "puntero a función" de **strcmp()** para realizar la comparación de las dos cadenas. La función **strcmp()** se encuentra en el archivo de cabecera "***string.h***" y retorna 0 si son iguales las cadenas y un valor distinto de cero si son diferentes. El "puntero a función" sirve para llamar la función a la que apunta.

```
/* PROG0606.C */

# include "stdio.h"
```

```
# include "string.h"

void comprobar (char *a, char *b, int (*cmp) ());

void main ()
{
  char s1[80], s2[80];
  int (*p) ();

  puts ("Ingrese la primera cadena:");
  gets (s1);
  puts ("Ingrese la segunda cadena:");
  gets (s2);
  p = strcmp;   /* Dirección de la función strcmp(). */
  comprobar (s1, s2, p);
}

void comprobar (char *a, char *b, int (*cmp) ())
{
  puts ("Comprobando la igualdad de las cadenas");
  if (!(*cmp) (a, b))
    /* (*cmp)() es el contenido del "puntero a función" de strcmp(). */
    puts ("Las cadenas son iguales");
  else
    puts ("Las cadenas son diferentes");
}
```

Una salida del programa podría ser:

Ingrese la primera cadena:
Luis Eduardo <ENTER>
Ingrese la segunda cadena:
Luis <ENTER>
Comprobando la igualdad de las cadenas
Las cadenas son diferentes

La expresión **(*cmp)(a, b)** llama a la función **strcmp()** con los argumentos **a** y **b**, a través del puntero **cmp**. Siendo esta expresión el contenido del "puntero a función" de la función **strcmp()**.

También se puede llamar a la función **comprobar()** utilizando la dirección de la función **strcmp()** directamente, eliminando la necesidad de una variable puntero adicional, así:

comprobar (s1, s2, strcmp);

Las aplicaciones de punteros a funciones sirven para:

1. Pasar funciones arbitrarias como argumentos a otras funciones.

2. Mantener un arreglo de punteros a funciones con un índice a la función adecuada, y poderlos seleccionar de acuerdo a la necesidad.

Ejercicio

Realizar una sola función **comprobar()**, que compruebe la igualdad alfabética con la función estándar **strcmp()**, o la igualdad numérica con la función de usuario **numcmp()**. La función **numcmp()** debe tener 2 argumentos punteros a cadenas que se convertirán a números mediante la función estándar **atoi()**, la misma que se encuentra en la librería "***stdlib.h***". Y para determinar si es cadena numérica se debe utilizar la función estándar **isalpha()**, que se encuentra en la librería "***ctype.h***", la misma que devuelve un valor distinto de cero si el primer caracter es una letra, y 0 en otro caso. Para esta comprobación se debe realizar la llamada a la función **comprobar()** con una dirección a función diferente.

```
/* PROG0607.C */

# include "stdio.h"
# include "string.h"
# include "stdlib.h"
# include "ctype.h"

void comprobar (char *a, char *b, int (*cmp) ());
int numcmp (char *a, char *b);

void main ()
{
  char s1[80], s2[80];

  puts ("Ingrese la primera cadena:");
  gets (s1);
  puts ("Ingrese la segunda cadena:");
  gets (s2);
  if (isalpha (*s1) || isalpha (*s2)) {
    printf ("Para cadenas, ");
    comprobar (s1, s2, strcmp);
  }
  else
```

```
    comprobar (s1, s2, numcmp);
}

void comprobar (char *a, char *b, int (*cmp) ())
{
  printf ("Comprobando la igualdad\n");
  if (!(*cmp) (a, b))
    puts ("Son iguales");
  else
    puts ("Son diferentes");
}

int numcmp (char *a, char *b)
{
  puts ("De enteros");
  if (atoi (a) == atoi (b))
    return 0;
  return 1;
}
```

Una salida del programa podría ser:

ingrese la primera cadena:
Lindo <ENTER>
ingrese la segunda cadena:
Lindo <ENTER>
Para cadenas, Comprobando la igualdad
Son iguales

6.10. FUNCIONES RECURSIVAS

6.10.1. Definición

Una función recursiva es aquella que se llama a sí misma.

Una solución recursiva a un problema consiste en transformar el problema en uno nuevo que es similar al original, pero que de alguna manera es más simple.

Los lenguajes que soportan la recursividad dan al programador una herramienta poderosa para resolver ciertos tipos de problemas, reduciendo la complejidad y obviando los detalles del problema.

Se utiliza una solución recursiva en lugar de una no recursiva (iterativa) porque: (BECERRA, 1991)

- "*Una no recursiva*", a pesar de parecer más sencilla y más frecuente, no puede dar solución a todos los problemas.

- "*Una recursiva*", da un programa más sencillo y más elegante. Además, hay muchos problemas cuya respuesta se plantea solamente de forma recursiva.

Por ejemplo:

Resolver la función factorial mediante recursividad, para lo cual se necesita una fórmula en términos de sí misma:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n-1)!, & \text{si n} > 0 \end{cases}$$

Por ejemplo, calcular 4!:

Usando la segunda parte de la definición cuando n > 0, se tiene:

n = 4,   4! = 4 * (4 - 1)! = 4 * 3!,   no se sabe el valor de 3!
n = 3,   3! = 3 * (3 - 1)! = 3 * 2!,   no se sabe el valor de 2!
n = 2,   2! = 2 * (2 - 1)! = 2 * 1!,   no se sabe el valor de 1!
n = 1,   1! = 1 * (1 - 1)! = 1 * 0!,   se sabe el valor de 0!

"La condición de finalización" se obtiene cuando n==0 y n!==1, entonces 0!==1. Con esto se pueden completar los cálculos:

n = 0,   0! = 1
n = 1,   1! = 1 * 0! = 1 * 1 = 1
n = 2,   2! = 2 * 1! = 2 * 1 = 2
n = 3,   3! = 3 * 2! = 3 * 2 = 6
n = 4,   4! = 4 * 3! = 4 * 6 = 25

Entonces, 4! = 24.

Ahora se implementará la solución en forma computacional de la función factorial. Para calcular **n!** se construirá una función recursiva **nfact()**, donde se puede obtener el valor de **(n - 1)!** que se necesita en la fórmula. Naturalmente el parámetro actual **(n - 1)** de la llamada recursiva será diferente del parámetro de llamada original (**n**), y la llamada recursiva se hará desde dentro de la función.

La función recursiva **nfact()** que calcula **n!** para un entero no negativo **n** se presenta a continuación:

```
unsigned long nfact (int n)
{
    if (n == 0)
        return 1;
    else
        return n * nfact (n - 1);
}
```

Se utiliza el tipo **unsigned long** para retornar el valor del factorial por la función **nfact()**, porque así puede obtenerse rangos mayores al tipo **int** correspondiente al factorial de **n**.

Análisis de la función *nfact()*

Cuando se ejecuta una función, a cada parámetro formal y variable local se les asigna espacio de memoria en el segmento de pila, en el orden en que están declarados. Luego son accedidas hacia atrás a partir de una posición apuntada por un puntero llamado **tope** o **cabeza**. A continuación se muestra un ejemplo:

Implementación de la función:

```
float funcion (float x, float y)
{
    int i;
    /* Sentencias. */
}
```

En la Figura 6.3 se muestra las variables locales de la función ***funcion()***.

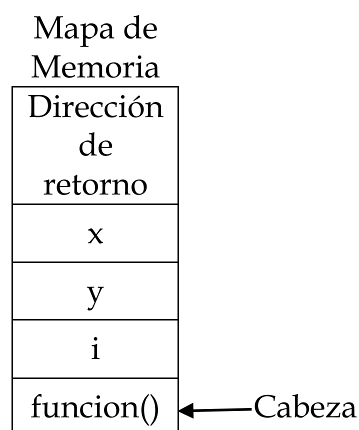


Figura 6.3. Variables locales de la función **funcion()**

Cuando se llama a una función, a los parámetros actuales se les asigna las siguientes posiciones libres, y la **Cabeza** se pone a la dirección del último parámetro o variable local asignados. Por lo tanto, se permite que el espacio necesitado para almacenar estos valores crezca conforme a las llamadas recursivas.

Después de ejecutada la función, se liberan estas posiciones para un uso posterior.

La asignación de memoria a las variables locales y parámetros de una función, se soportan mediante una pila en tiempo de ejecución, y la pila estará limitada por la cantidad de memoria disponible en el segmento de pila en tiempo de ejecución.

Por ejemplo

Para la implementación de la función recursiva **nfact()** sería:

```
unsigned long nfact (int n)
{
    if (n == 0)
        return 1;
    else
        return n * nfact (n - 1);
}           ↑________ D2
```

Y la llamada a esta función podría ser:

```
int n = 3;
printf ("%d! = %lu\n", n, nfact (n));
                        ↑________ D1
```

donde:

- **D1**, es la dirección de retorno en la llamada original.
- **D2**, es la dirección de retorno en la llamada recursiva.

En la primera llamada, en tiempo de ejecución, se pone en la pila tres posiciones:

1. Una para la dirección de retorno (**D1**) al programa de llamada.
2. Otra para el parámetro formal **n**.

3. Otra para el identificador de la función **nfact()** indeterminado, puesto que no se ha hecho ninguna asignación.

Estas tres posiciones de memoria, en la primera llamada recursiva, se muestran en la Figura 6.4

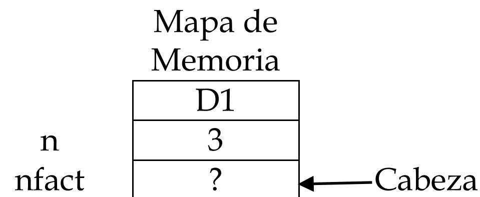


Figura 6.4. Valores de la pila en la primera llamada recursiva

Al ejecutar la función, se pregunta si n==0, como n==3, se bifurca por la alternativa **else**:

return n * nfact (n - 1);

Por lo tanto, **nfact()** se llama nuevamente desde dentro de la función con **nfact(3-1)**, cuya dirección de regreso es D2. De nuevo se asignan tres posiciones: D2, n==2, **nfact()** indeterminado. Este proceso continúa hasta que n==0, ejecutándose la sentencia de la alternativa verdadera: tomando **nfact()** el valor 1 (sentencia **return 1;**).

En la Figura 6.5 se tienen los valores de la pila, de las cuatro llamas recursivas y la terminación recursiva con el caso base n==0, lo que retornaría en **nfact()** el valor de 1 .

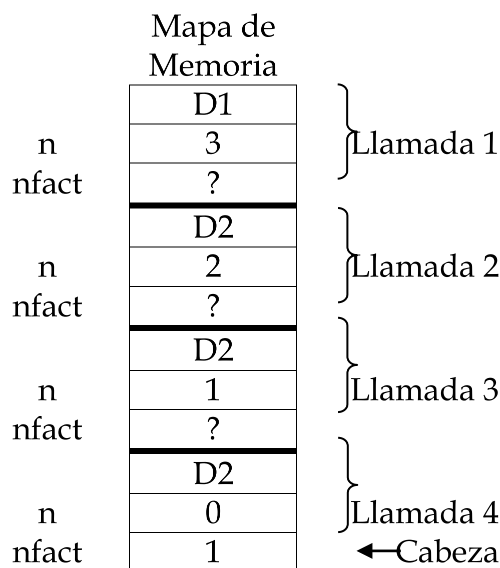


Figura 6.5. Valores de la pila de las cuatro llamadas recursivas

Se termina de ejecutar la función. Entonces, el último valor **nfact()**==1 se devuelve al lugar de llamada D2, luego de que el compilador saca de la pila la última asignación de memoria (llamada 4). En el lugar de llamada D2 se multiplica el valor devuelto de **nfact()**==1 por n==1 (de la llamada 3, **Cabeza** de la pila en este momento), almacenándose el valor 1 de la respuesta en **nfact()** (llamada 3), así:

```
return n * nfact (n - 1);
```

y la función se ha completado de nuevo. Esta sentencia **return** realiza los cálculos con los datos de la cabeza de la pila.

Este proceso continúa hasta que la pila se ha vaciado, devolviendo el valor de **nfact(n)**==6 en la llamada principal, completándose en su totalidad la función.

NOTA: El elemento sacado de la pila, da la dirección de retorno al punto de llamada, y también da el valor actual de la función. Pero, los cálculos se realizan con los datos de la cabeza de la pila.

Por otro lado, la cantidad de memoria aumenta cada vez que se profundiza en la recursión. Por ejemplo, con **nfact(30)** se llegaría a tener en un momento dado: 31 direcciones de vuelta, 31 variables **n** y 31 identificadores **nfact()**. Como se puede ver, el lenguaje C emplea gran cantidad de memoria, por lo que se debe programar recursivamente con precaución y solo en los casos en que una programación no recursiva sea inaadecuada.

Entonces, la selección entre la programación recursiva y no recursiva está usualmente determinada por la necesidad de almacenamiento temporal, por otro lado, las funciones que no tienen definiciones en sí mismas, se expresan siempre no recursivamente (iterativamente).

Ejercicio

Realizar un programa que contenga una función recursiva, para calcular el factorial de un número entero positivo ingresado por el teclado.

```
/* PROG0608.C */

#include <stdio.h>

/* Prototipo de la función nfact() */
unsigned long nfact (int n);

void main ()
{
```

```
  int n;

  printf ("Ingrese el número: ");
  scanf ("%d", &n);
  printf ("%d! = %lu\n", n, nfact (n));
}

unsigned long nfact (int n)
{
  if (n == 0)
    return 1;
  else
    return n * nfact (n - 1);
}
```

Una salida del programa podría ser:

Ingrese el número: 6 <ENTER>
6! = 720

6.10.2. Condiciones para crear las Funciones Recursivas

Para que una función recursiva sea efectiva, se debe considerar las siguientes cuestiones: (BECERRA, 1991)

1) Debe haber una salida no recursiva de la función, es decir, algún caso trivial o "base" en donde terminen las llamadas recursivas; si no, se tendría una serie infinita de llamadas recursivas.

   Por ejemplo, en la función **nfact()**, el caso base ocurre cuando n==0. Entonces, se asigna a **nfact()** el valor 1, que es el valor correcto de 0!.

2) Cada llamada a la función debe referirse a un caso más pequeño del problema, de tal manera que se llegue al caso base. Este sería el caso recursivo llamado "general", definido en términos de un caso cada vez más pequeño del mismo problema. La determinación del caso cada vez más pequeño, dependerá de la condición del problema para incrementar o decrementar el argumento.

   Por ejemplo, en la función **nfact()**, la llamada recursiva pasa el argumento n-1 a dicha función. Cada llamada recursiva siguiente envía un valor cada vez más pequeño del parámetro, hasta que el valor enviado es cero, que corresponde a **nfact()**==1.

NOTA: Cuando la función es de tipo **void** no es necesario que la llamada recursiva se realice con la sentencia **return**, porque no retorna un valor, y además el control del programa regresa a la siguiente sentencia de la llamada a la función.

Por ejemplo:

Realizar el análisis para multiplicar dos números naturales, e implementarla con una función recursiva.

Los números naturales **a** y **b** se pueden multiplicar así:

$$c = a * b$$

o también, así:

$$c = \underbrace{a + a + a + \ldots + a}_{b \text{ veces}}$$

que es equivalente a:

$$c = a * (b - 1) + a$$

Por ejemplo, multiplicar los números naturales a = 8 y b = 3:

a * b
8 * 3

8 * 2 + 8

8 * 1 + 8 + 8

8 * 0 + 8 + 8 + 8

$$\underbrace{0 + 8 + 8 + 8}_{24}$$

(b=3 veces)

En conclusión, para calcular la multiplicación se hará:

- c = a * b

Que es la llamada a la función para calcular la multiplicación, esto es **multi(a, b)**.

- c = a * (b - 1) + a

Que es la llamada recursiva, esto es **multi(a, b - 1) + a**.

"*Caso base*". El proceso recursivo termina cuando **b** es igual a cero, entonces la función retorna el valor de cero, así:

```
if (b == 0)
    return 0;
```

"*Caso general*". La llamada recursiva se da cuando **b** es diferente a cero, siendo **b - 1** un caso más pequeño que **b**, así:

```
return a + multi (a, b - 1);
```

Ejercicio

Realizar un programa que tenga una función recursiva, para calcular la multiplicación de dos números naturales.

```
/* PROG0609.C */

#include <stdio.h>

unsigned long multi (int a, int b);

void main ()
{
  int a, b;

  printf ("Ingrese el primer factor : ");
  scanf ("%d", &a);
  printf ("Ingrese el segundo factor: ");
  scanf ("%d", &b);
  printf ("%d * %d = %lu\n", a, b, multi (a, b));
}

unsigned long multi (int a, int b)
{
```

```
  if (b == 0)
    return 0;
  else
    return multi (a, b - 1) + a;
}
```

Una salida del programa podría ser:

Ingrese el primer factor : 8 <ENTER>
Ingrese el segundo factor: 3 <ENTER>
*8 * 3 = 24*

6.10.3. Diferencias entre Funciones Recursivas e Iterativas

1) Donde una rutina iterativa usa sentencias **while**, **for** o **do-while**, para controlar la repetición, una rutina recursiva usa **if-else** o **switch**.

2) Las funciones recursivas tienen frecuentemente más parámetros y menos variables locales que las funciones iterativas equivalentes. Porque los parámetros sirven para determinar el tamaño del problema, para realizar la recursividad y para llegar al caso base.

En versiones recursivas, un parámetro es necesario para permitir que su valor se modifique en cada llamada recursiva, y esto sirve para disminuir el tamaño del problema. Este valor se inicializa en la sentencia de llamada de la función, pero en una versión iterativa, pueden usarse variables locales para este propósito, inicializándolas al comienzo de la función e incrementándolas en cada iteración. Por ejemplo, realizar la función iterativa para calcular el factorial de un número entero positivo:

```
unsigned long nfact (int n)
{
    // Variables locales
    int i = 1;
    unsigned long fact = 1;

    for (; i <= n; i++)
        fact += i;
    return fact;
}
```

6.11. LISTA DE PARÁMETROS VARIABLES

Se puede especificar una función con un número variable de parámetros y tipos.

Para indicar al compilador que se va a pasar a una función un número y unos tipos desconocidos de parámetros, se debe terminar la declaración de sus parámetros con tres puntos. Pero cualquier función que use un número variable de argumentos, debe tener al menos un argumento conocido.

Por ejemplo, la siguiente declaración especifica que la función **ayuda()** tendrá al menos dos parámetros enteros conocidos y un número desconocido de parámetros de cualquier tipo:

ayuda (int a, int b, ...);

6.11.1. Paso de Argumentos de Longitud Variable a Funciones

Las macros **va_start()**, **va_arg()** y **va_end()** trabajan juntas, para permitir pasar a la función un número variable de argumentos.

Los prototipos de las macros son: (SCHILDT, 1994)

```
void va_start (va_list punt_arg, ult_param);
tipo va_arg (va_list punt_arg, tipo);
void va_end (va_list punt_arg);
```

donde:

- **va_list**, es un tipo de dato definido en el archivo de cabecera "***stdarg.h***".

El procedimiento general para crear una función que pueda tomar un número variable de argumentos es el siguiente:

1. La función debe tener al menos un parámetro conocido a la lista de parámetros variables, pero puede tener más.

2. El parámetro más a la derecha de la lista de "parámetros conocidos" se llama ***ult_param***.

3. Antes de que cualquiera de los parámetros de longitud variable puedan ser utilizados, el puntero ***punt_arg*** debe estar inicializado por medio de una llamada a la macro ***va_start()***, que es la dirección del primer parámetro variable que se encuentra después del último parámetro conocido ***ult_param***.

4. Después, los parámetros se devuelven a través de llamadas a la macro ***va_arg()*** de acuerdo al tipo que se especifica como parámetro de esta macro, siendo este ***tipo*** el siguiente del parámetro de la lista. Es decir, esta macro, en cada llamada, retorna el siguiente parámetro variable.

5. Finalmente, una vez que todos los parámetros han sido leídos y antes de volver de la función, se debe hacer una llamada a la macro ***va_end()*** para asegurar que la pila está reconstruida apropiadamente, entonces, si no se llama a ***va_end()***, es muy común que el programa falle.

NOTA: Los argumentos en la llamada de este tipo de función deben ser variables.

Ejercicio

Realizar una función **suma_serie()** con argumentos variables, para devolver la suma de la serie armónica. El primer argumento es conocido y debe contener el número de argumentos variables de los términos de la serie armónica a sumar. La fórmula de la serie armónica es:

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

```
/* PROG0610.C */

#include "stdio.h"
#include "stdarg.h"

double suma_serie (int num, ...);

void main ()
{
  double suma;
  double u = 1, v = 0.5, w = 0.33, x = 0.25, y = 0.2;

  /* Suma 5 términos de la serie. */

  suma = suma_serie (5, u, v, w, x, y);
  /* Los argumentos de la función deben ser variables. */
  printf ("La suma de la serie de %d términos es %.2f\n", 5, suma);
}

double suma_serie (int num, ...)
{
  double suma = 0.0, aux;
  va_list arg_ptr;  /* Obtiene un puntero "arg". */

  /* Inicializa "arg_ptr". */
  va_start (arg_ptr, num);

  /* Suma la serie. */
```

```
  for (; num; num--) {
    aux = va_arg (arg_ptr, double);
    suma += aux;
  }

  /* LLeva a cabo un cierre ordenado de la pila. */
  va_end (arg_ptr);

  return suma;
}
```

Una salida del programa podría ser:

La suma de la serie de 5 términos es 2.28

6.11.2. Función *vprintf()*

La función **vprintf()** es una función equivalente a **printf()**. La diferencia se encuentra en que la lista de argumentos se sustituye por un puntero **punt_arg** a una lista de argumentos. Este puntero tiene que ser de tipo **va_list** y está definido en el archivo de cabecera "***stdarg.h***".

El prototipo de la función es: (SCHILDT, 1994)

```
vprintf (char *formato, va_list punt_arg);
```

La función **vprintf()** reemplaza a la macro **va_arg()** de acuerdo a su aplicación, ya que esta función se utiliza solo para imprimir datos, a diferencia de la macro **va_arg()** que se utiliza para tomar datos en su nombre.

Ejercicio

Realizar una función llamada **mensaje()**, para imprimir una lista de argumentos.

```
/* PROG0611.C */

#include "a:\capitulo.6\mensaje.c"
#include <conio.h>

void main ()
{
  mensaje ("%s %s %d\n", "No se puede abrir el archivo:", "LLAMAR.C", 1);
}
```

La función **mensaje()** se encuentra almacenado en el archivo "a:\capitulo.6\mensaje.c".

```
/* MENSAJE.C */

#include "stdio.h"
#include "stdarg.h"
#include "conio.h"
#include "stdlib.h"

void mensaje (char *formato, ...);  /* Normalmente es la cadena de control. */
void pausa(void);

void mensaje (char *formato, ...)
{
  va_list punt;  /* Obtiene un puntero "arg". */

  /* Inicializa "punt" a que apunte al primer argumento a partir de la */
  /* cadena de "formato".                                              */
  va_start (punt, formato);

  /* Imprime el mensaje. */
  printf ("ERROR: ");
  vprintf (formato, punt);
  printf ("\n");

  /* Recostrucción apropiada de la pila. */
  va_end (punt);
  pausa ();
  exit (1);
}

void pausa(void)
{
  printf("Pulse una tecla para continuar...");
  getch ();
  printf ("\n");
}
```

La salida de este programa será:

ERROR: No se puede abrir el archivo: LLAMAR.C 1

Pulse una tecla para continuar...

6.12. DECLARACIÓN DE PARÁMETROS DE FUNCIONES CLÁSICAS FRENTE A LAS MODERNAS

El lenguaje C estándar ANSI soporta la declaración de parámetros en forma clásica y moderna, pero se recomienda la forma moderna. Aunque muchos programas antiguos utilizan la forma clásica.

"*La forma clásica*". Consiste en dos partes:

1) Una lista de parámetros, que van entre paréntesis y siguen al nombre de la función.

2) La declaración real de los parámetros va entre el paréntesis cerrado y la llave de apertura de la función. Se puede poner más de un parámetro después del tipo, igual que la declaración de cualquier variable. Como se muestra a continuación:

```
tipo nombre (param1, param2, paramN)
tipo param1;
tipo param2;
tipo paramN;
{
    /* Sentencias de la función. */
}
```

Por ejemplo, la declaración de la función **prueba()**, será:

```
float prueba (a, b, c)
int a, b;
char c;
{
    /* Sentencias de la función. */
}
```

La misma declaración de la función, pero en la forma moderna, será:

```
float (int a, int b, char c)
{
    /* Sentencias de la función. */
}
```

6.13. BIBLIOTECAS Y ARCHIVOS

Una vez que se ha escrito una función, se pueden hacer tres cosas con ella: (SCHILDT, 1994)

1) Ponerla en el mismo archivo que la función **main()**. En este caso, la compilación se realizará de la misma forma que la compilación de una sola función.

2) Ponerla en un archivo aparte con otras funciones compiladas por separado que se hayan escrito, para luego realizar un encadenamiento.

3) Ponerla en una biblioteca. En este caso, se utiliza la sentencia de preprocesador **#include**, para incluirla en el programa en donde se vaya usar.

Técnicamente, una biblioteca de funciones difiere de un archivo de funciones compiladas por separado, porque:

- Las rutinas de una biblioteca se unen al resto del programa, cargando y enlazando solo las funciones que el programa utiliza realmente.

- En un programa compilado por separado, todas las funciones de un archivo se cargan y se enlazan con el programa.

En el caso de la biblioteca estándar de lenguaje C nunca se deberá tener todas las funciones enlazadas en el programa, ya que el código objeto sería gigantesco.

PROBLEMAS PROPUESTOS

1. Escribir un programa que introduzca una serie de enteros y que los pase uno a la vez a la función **par()** que determina si el entero es par. Esta función deberá tomar un argumento entero y regresar en su nombre 1 si el entero es par, o 0 si no lo es.

2. Realizar una función que reciba un dato de tipo **char** y devuelva el caracter '0' si no es una letra. En caso de que el caracter sea una letra minúscula debe devolver la letra en mayúscula, caso contrario si el caracter es una letra mayúscula devuelve el mismo caracter.

   Luego utilizar esta función en un programa que permita ingresar una serie de caracteres hasta digitar un caracter '*' , e indicar el resultado correspondiente.

3. Escribir una función **Calidad_Puntos()** que reciba una nota de un alumno y regrese por la función: **4** si el promedio es entre 90 y 100, **3** si el promedio es entre 80 y 89, **2** si el promedio es entre 70 y 79, **1** si el promedio es entre 60 y 69 y **0** si el promedio es menor de 60.

4. Escribir la función **distancia_puntos()** que calcule la distancia entre dos puntos (x1, y1) y (x2, y2). Todos los valores de los puntos y el valor de regreso deberán ser de tipo **float**.

   Utilizando esta función realizar un programa que ingrese una serie de pares de puntos y determine la distancia entre ellos.

5. Elaborar una función **producto()** que reciba dos números de tipo **int**: **A** y **B**, y devuelva por la función un dato de tipo **int**, que contenga el resultado de multiplicar **A** por **B** por el método de sumas sucesivas.

   Leer en un programa un conjunto de dos números enteros positivos que serán enviados a la función **producto()**, luego utilizar el valor devuelto de la función para imprimir una tabla del producto de **A** por **B**, así:

A	B	PRODUCTO

6. Elaborar una función que reciba dos datos **ch1** y **ch2** de tipo **char** y devuelva un valor:

   verdadero, si ch1 <= ch2
   falso, si ch1 > ch2

Luego utilizar esta función en un programa que ingrese un conjunto de pares de caracteres hasta que se digite el caracter '*', y que muestre una tabla como la siguiente:

CARACTER1	CARACTER2	RESULTADO

7. Escribir una función **multiplo()** que determine en un par de enteros si el segundo de ellos es múltiplo del primero. La función debe tomar dos argumentos enteros y regresar 1 (verdadero) si el segundo es múltiplo del primero, y 0 (falso) de no ser así. Utilizar esta función en un programa que introduzca una serie de pares de enteros y liste el resultado de esa función.

8. Un "***número perfecto***" es aquel cuyos factores suman igual que dicho número, incluyendo el 1 pero excluyendo ese número entero como factor. Por ejemplo, 6 es un número perfecto porque 6 = 1 + 2 + 3. Tomando en cuenta esto escribir una función **perfecto()** que determine si su parámetro es un número perfecto. Luego utilizar esta función en un programa que determine e imprima todos los números perfectos entre 1 y 1000, con sus factores, para confirmar que el número de verdad sea perfecto.

9. Un número entero es "***primo***" si es divisible solo para 1 y para sí mismo. Por ejemplo, los números 2, 3, 5, 7 son primos, pero 4, 6, 8 y 9 no lo son. Con este concepto escribir una función que determine si un número es primo, y utilizarla en un programa que determine e imprima todos los números primos entre 1 y 10000.

   Nota: Inicialmente se pensaría que **n/2** es el límite superior para determinar si un número es primo, pero no es así, ya que para esto solo se necesita llegar hasta la raíz cuadrada de **n**.

10. Escribir una función que reciba el tiempo como tres argumentos enteros: horas, minutos y segundos, y regrese por la función el número de segundos transcurridos desde la hora 00:00:00. Utilizar esta función para calcular la cantidad de segundos entre dos tiempos.

11. Elaborar una función que reciba como argumentos la longitud de un péndulo **l** y la aceleración gravitacional **g**, para calcular el tiempo de oscilación **t**, mediante la siguiente fórmula:

$$t = \pi \sqrt{\frac{l}{g}}$$

Utilizar esta función en un programa que calcule el tiempo de oscilación de varios péndulos, mientras se lea en la longitud del péndulo un valor diferente de 0. Además, el programa debe determinar si **l** es mayor a 0, caso contrario si **l** es menor a 0 debe aparecer el mensaje: "*Valor de __ no permitido*", y volver a pedir más datos.

12. Elaborar un programa que calcule la altura máxima de un proyectil, la fórmula que determina la altura **h** es:

$$h = \frac{v^2 \sin^2(\alpha)}{2\,g}$$

donde:

- **v**, velocidad de proyección (m/seg).
- **α**, ángulo de proyección (radianes).
- **g**, aceleración gravitacional (m/seg^2).

Los datos a leer para calcular **h** son: velocidad de proyección, ángulo de proyección y aceleración gravitacional.

Los cálculos deben efectuarse en una función llamada **valor_h()**. Además, debe existir una función que calcule el seno de un ángulo, para no utilizar la función estándar **sin().**

Por último, el programa debe ingresar un conjunto de datos para generar una tabla con los datos ingresados y calculados.

13. Elaborar una función que reciba un dato de tipo **char**, que debe devolver un número:

0 si es una letra mayúscula.
1 si es una letra minúscula.
2 si es un dígito.
3 si es un caracter especial.

En un programa leer una serie de caracteres hasta ingresar un '*', que serán enviados a la función, luego utilizar el valor devuelto de la función para indicar el tipo de caracter.

14. Realizar un programa que lea dos fechas cualquiera con el formato año/mes/día, para calcular en una función el número de días que existe entre las dos fechas leídas y por último imprimir los datos leídos y calculado.

15. Elaborar una función lógica que reciba un valor **n** y determine si la siguiente relación es verdadera o no:

$$n + \sum_{i=2}^{i=n}(n - i + 2) = \frac{n^2}{2} + \frac{3n}{2} - 1$$

El programa principal debe leer el valor entero **n** que es enviado a esta función, luego escribir el mensaje apropiado para indicar si la relación es correcta 0 no.

16. Construir una función **hipotenusa()** que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando son conocidos los otros dos lados. La función debe tomar dos argumentos de tipo **double** y regresar la hipotenusa también como **double**. Utilizar esta función en un programa para determinar la longitud de la hipotenusa de varios triángulos, como se muestra a continuación:

Triángulo	Lado1	Lado 2	Hipotenusa
1	3.0	4.0	5.0
2	5.0	12.0	13.0
3	8.0	15.0	17.0

17. La función **sumar()** tiene 3 parámetros: los dos primeros son números enteros que van a ser sumados y el tercero corresponde al resultado de la suma. En el programa principal ingresar desde el teclado dos números para sumarlos en dicha función, y luego imprimir el resultado obtenido.

18. Escribir un programa que tenga una función que permita intercambiar los valores de tres variables **a**, **b** y **c**, si es necesario, de tal manera que se cumpla la relación:

    a <= b <= c

    El programa debe leer los datos **a**, **b** y **c** para llamar a la función, y escribir los resultados antes y después de llamar a la función.

19. Elaborar un programa que lea tres enteros positivos, los cuales representan las longitudes de los lados de un triángulo. Con estos enteros realizar los siguientes literales:

    a) Una función que determine si los tres valores leídos forman un triángulo.

    b) Una función que determine:

        - Si es un triángulo equilátero.
        - Si es un triángulo isósceles.
        - Si es un triángulo escaleno.

c) Una función donde se determine el área del triángulo, si cumple el literal a).

Además, el programa debe imprimir los resultados correspondientes.

20. Elaborar una función que calcule las raíces de la ecuación cuadrática:

$ax^2 + bx + c = 0$

utilizando la fórmula:

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

El programa principal lee los coeficientes de la ecuación, a continuación accede a la función para obtener la solución deseada, y finalmente escribe los valores de los coeficientes seguidos de los valores calculados de las raíces $\mathbf{x_1}$ y $\mathbf{x_2}$, de acuerdo a los siguientes tipos de raíces obtenidas:

Si $\mathbf{b^2 - 4ac < 0}$, las raíces son imaginarias.
Si $\mathbf{b^2 - 4ac = 0}$, existe una sola raíz real doble.
Si $\mathbf{b^2 - 4ac > 0}$, las raíces son reales.

Además, el programa principal debe escribir la ecuación cuadrática con los valores ingresados, y en caso de que se tenga el coeficiente **a** igual a 0 presente un mensaje de error.

21. Elaborar una función que reciba las variables **n** y **r**, para imprimir el número de combinaciones de **n** objetos tomados de **r** en **r**. El número de combinaciones de **n** objetos tomados de **r** en **r** se calcula con la siguiente fórmula:

$$\binom{n}{r} = \frac{n!}{r!\ (n-r)!}$$

Realizar un programa que lea los números **n** y **r** para llamar a la función y luego escribir los siguientes resultados:

El número de combinaciones de n: ___
objetos tomados de r: _ *en r:* ___
es: ___

Además, el programa debe tener una función adicional que calcule el factorial de cualquier número.

22. Realizar un programa que lea la fecha actual en el formato anio/mes/dia, luego enviar la fecha ingresada a una función para que imprima en palabras la fecha del día siguiente. Por ejemplo, si se lee:

    98/11/25.

    imprimirá:

    26 de noviembre de 1998

    Validar los datos de ingreso para que se encuentren dentro del rango correcto de los días, meses y años. Además, validar en el caso que sea el año bisiesto.

23. Diseñar un programa para leer un número entero largo entre 1 y 999999, luego enviar el número ingresado a una función para que imprima el número en letras.

24. Elaborar una función que reciba en una variable entera un número **n**, y devuelva en otra variable "verdadero" si el número recibido es primo, caso contrario devuelva "falso".

    En el programa leer el valor de **n** e indicar con un mensaje apropiado si **n** es primo o no.

25. Elaborar una función que reciba como argumento un dato de tipo **float** y devuelva tres variables: la primera que almacene la parte entera del número, la segunda que almacene como entero la parte decimal del número y la última que almacene el número de decimales que contiene el dato recibido por la función.

    En el programa se lee el dato de tipo **float** e imprime los resultados obtenidos de la función.

26. Realizar una función que calcule el tiempo total de un corredor en una competencia ciclística, los datos recibidos en la función son:

    - Tiempo utilizado hasta el momento (tiempo acumulado).
    - Tiempo utilizado en la última etapa.

    El programa lee los datos enviados a la función, considerando:

    a) Que los datos que expresan el tiempo total estén entre los siguientes límites:

       0 <= hora_total
       0 <= minuto_total <= 59
       0 <= segundo_tota1 <= 59

b) Que los datos que expresan el tiempo de la última etapa estén entre los siguientes límites:

0 <= hora_etapa <= 3
0 <= minuto_etapa <= 59
0 <= segundo_etapa <= 59

En caso de que no se cumpla alguna de las condiciones anteriores, se deben leer nuevos datos.

El programa principal debe calcular e imprimir el tiempo total de varios ciclistas, hasta ingresar un valor de 999 para la variable **tiempo_total**, luego el programa debe imprimir el número total de ciclistas a los cuales se les calculó el tiempo total.

27. La fórmula de la suma de **n** términos de una "***serie geométrica***" es:

$$gn = \frac{a - ar^n}{1 - r}$$

donde:

- **a**, es el primer término de la serie.
- **r**, es la razón de un término al anterior.

Elaborar una función que determine el número de términos necesarios, para obtener la suma que más se aproxime al valor recibido como parámetro, así como la suma aproximada, de la siguiente serie:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots$$

El programa debe leer desde el teclado el parámetro de la función, luego imprimir el número de términos y el valor de la suma cuando se cumplió la condición mencionada anteriormente.

28. Realizar un programa que lea el número de términos **n** para calcular la siguiente sumatoria:

$$suma = \sum_{i=1}^{i=n} \sqrt[3]{i!}$$

Elaborar dos funciones para la sumatoria y para el factorial.

29. Realizar un programa que escriba el triángulo de Pascal. La profundidad del triángulo está determinado por el valor **n** leído desde teclado. Por ejemplo, si el valor leído **n** es 4, el triángulo de Pascal debe imprimirse de la siguiente manera:

```
      1
    1   1
  1   2   1
1   3   3   1
```

Nota: El triángulo de Pascal anterior, es equivalente a:

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{2} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

30. Realizar una función llamada **xgets()** que realice las mismas funciones de la función de biblioteca estándar **gets()**. La función tiene como parámetro un puntero a la cadena a leer y en su nombre almacena un puntero a la dirección de la cadena leída. Luego de realizar la lectura salta a la siguiente línea.

    En un programa utilizar esta función para leer una cadena que será invertida, e imprimir las dos cadenas.

31. Realizar un menú con las siguientes opciones.

    a) Una función entera **Celsius()** que regresa el equivalente Celsius de una temperatura en Fahrenheit.

    b) Una función entera **Fahrenheit()** que regresa el equivalente en Fahrenheit de una temperatura en Celsius.

    c) Utilizar ambas funciones para escribir un programa que imprima tablas mostrando los equivalentes Fahrenheit de todas las temperaturas Celsius desde 0 hasta 100 grados, y los equivalentes Celsius de todas las temperaturas Fahrenheit entre 32 y 212 grados. Imprimir las salidas en un formato tabular nítido, que minimice el número de líneas de salida.

32. Realizar un programa para crear un menú que tenga las siguientes opciones:

0. Salir.
1. Cadena inversa.
2. Concatenar 2 cadenas.
3. Buscar subcadena.

El menú debe aparecer cada vez que se desea una opción y debe aceptar solo las opciones establecidas.

Las opciones del menú son funciones que realizan lo siguiente:

0. Termina el programa y vuelve al sistema operativo.

1. Ingresar una línea de texto para obtener la cadena inversa (hacia atrás) en la función **inversa()**. Esta función tiene dos parámetros: cadena original y cadena inversa y en su nombre regresa la dirección de la cadena inversa. Luego imprimir las dos cadenas con sus direcciones.

2. Ingresar dos cadenas para unirlas en la función **concat()**. Esta función tiene los dos parámetros correspondientes a las cadenas a unir, la segunda cadena se une a continuación de la primera.

3. Es una función llamada **buscar_sub()** que devuelve el índice del comienzo de una subcadena dentro de una cadena, si no se ha encontrado devuelve -1. La función tiene dos parámetros que son los punteros a las cadenas, y en su nombre regresa el índice. Por ejemplo, si se tiene:

   La cadena: "Lindos días"
   La subcadena: "dos"

   El índice que regresa es 3

33. Realizar un programa que tenga una función con un puntero a cadena de caracteres como parámetro, esta función retorna en su nombre la dirección de la cadena convertida de minúsculas a mayúsculas. El programa debe leer una cadena de caracteres que se envía a la función para imprimirla en mayúsculas.

34. Una aplicación de la función **floor()** es redondear un valor al entero más cercano, con la siguiente sentencia de asignación:

    y = floor (x + .5);

    que redondeará el número **x** al entero más cercano, y asignará el resultado a **y**.

Otra aplicación de la función **floor()** puede ser utilizada para redondear un número a una cantidad específica de lugares decimales. Por ejemplo, la sentencia de asignación:

y = floor (x * 10 +.5)/10;

redondea **x** a la posición de décimas, y la sentencia:

y = floor (x * 100 + .5)/ 100;

redondea **x** a la posición de las centésimas.

Escribir un programa que defina cuatro funciones para redondear un número **x** de varias formas:

a) Redondeo_Entero (numero)
b) Redondeo_Decima (numero)
c) Redondeo_Centesima (numero)
d) Redondeo_Milesima (numero)

Para cada uno de los varios valores leídos el programa debe imprimir el valor original, el número redondeado para el entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana, y el número redondeado a la milésima más cercana.

35. Elaborar una función que reciba varios datos de tipo **char** y devuelva un dato de tipo **float** que contenga el número formado por los datos recibidos. Por ejemplo, se envía los siguientes 4 datos: '-' , '4', '.' y '2', el número que debe devolverse será: -4.2.

    Si por algún motivo no se puede formar un número de tipo **float** con los datos recibidos, se debe devolver el número -1 para indicar en el programa con un mensaje.

    Realizar un programa para leer una variable entera que indica el número de caracteres a ser leídos, luego leer los caracteres, y por último utilizar la función con parámetros variables para realizar lo indicado.

36. Elaborar una función que reciba caracteres, en la cual los caracteres que no son dígitos deben ser rechazados, y con los caracteres aceptados, se debe generar un número **float** que corresponda a la unión de ellos. El número generado debe ser devuelto por la función. Por ejemplo, si los caracteres enviados a la función son: '-', '8', '4', 'A', '3', ';', '(', 'n', 'b', '.' y '1'; el número devuelto debe ser: -843.1.

Los caracteres punto y '-' pueden procesarse solamente una vez, pero el caracter menos se debe ingresar antes de encontrar el primer dígito. Por lo tanto, debe ser validado el ingreso.

En un programa leer caracteres hasta que encuentre un '*', y la función debe ser utilizada con parámetros variables para realizar lo indicado.

37. Escribir un programa que tenga una función que despliegue en el margen izquierdo de la pantalla, un cuadrado sólido en base a cualquier caracter recibido en un parámetro de tipo caracter, cuyo lado está especificado en otro parámetro de tipo entero. Por ejemplo, si el lado es 5 y el caracter es '#', la función mostrará:

```
#####
#####
#####
#####
#####
```

38. La función **calculo_carga()** determina la cuenta de cada cliente en un estacionamiento público, con una base de 2 dólares de estacionamiento mínimo por las primeras tres horas. El estacionamiento tiene 0.5 dólares adicionales por cada hora o fracción. La cuenta máximo para cualquier período de 24 horas es 10 dólares. Suponer que no existe ningún vehículo que se quede más de 24 horas a la vez.

Escribir un programa que calcule e imprima las cuentas por estacionamiento de cada cliente que ayer estacionaron sus automóviles en este garaje, para lo cual se deberá introducir las horas de estacionamiento para cada uno de los clientes.

El programa deberá imprimir los resultados en un formato tabular nítido, y deberá calcular e imprimir el total de los ingresos del día de ayer. Por ejemplo, la salida deberá aparecer en el formato siguiente:

Ord	Horas	Carga
1	1.5	2
2	4.0	2.5
3	24.0	10
TOTAL	29.5	14.5

39. Si el interés anual **i** se establece de diferentes frecuencias de composición: anual, semestral, trimestral o mensual; la cantidad futura de dinero acumulado después de **n** años viene dado por:

$$F = 12\,A \left(\frac{(1 + \, i \,/\, n)^{mn} - 1}{i} \right)$$

donde:

- **F**, es la cantidad futura.
- **A**, es la cantidad de dinero depositada cada mes.
- **i**, es la tasa de interés anual (expresado como decimal).
- **m**, es el número correspondiente de períodos por año, para las siguientes composiciones:

  m = 1, anual
  m = 2, semestral
  m = 4, trimestral
  m = 12, mensual

- Para la composición diaria, la cantidad futura es determinada por:

$$F = 12\,A\left(\frac{(1+\ i\ /\ m)^{mn} - 1}{(1+\ i\ /\ n)^{m/12} - 1}\right) \qquad ,donde\ m = 360$$

- Para la composición continua, la cantidad futura es determinada por:

$$F = 12\,A\left(\frac{e^{in}\ - 1}{e^{i/12}\ - 1}\right) \qquad ,donde\ m = 0$$

Desarrollar un programa para calcular la cantidad futura **F** como función de la tasa de interés anual **i** para los valores **A**, **m** y **n**; los datos **A** e **i** son leídos.

Cada una de las fórmulas estarán en las funciones **modo1()**, **modo2()** y **modo3()**; que tienen los parámetros: **i**, **m** y **n**.

Realizar los cálculos en una función **tabla()**, que debe tener el paso de un puntero a función para cada fórmula apropiada y los parámetros **A**, **m** y **n**.

Generar una tabla de valores futuros **F** para varios períodos de tiempo y diferentes frecuencias de composición. Mostrar la salida de la siguiente manera:

A = -										
i = -										
Período de tiempo (n) =	1	2	3	4	5	6	7	8	9	10
Frecuencia de composición:										

Anual	-	-	-	-	-	-	-	-	-	-
Semestral	-	-	-	-	-	-	-	-	-	-
Trimestral	-	-	-	-	-	-	-	-	-	-
Mensual	-	-	-	-	-	-	-	-	-	-
Diaria	-	-	-	-	-	-	-	-	-	-
Continua	-	-	-	-	-	-	-	-	-	-

40. En el siguiente programa escribir exactamente el resultado de la impresión:

```
#include "stdio.h"

void imprimir (int n);

void main ()
{
    int n;

    n = 2;
    imprimir (n);
    printf ("\n");
}

void imprimir (int n)
{
    if (n != 0) {
        printf ("A %d: ", n);
        imprimir (n - 1);
        printf ("B %d: ", n);
        imprimir (n - 1);
        printf ("C %d: ", n);
        imprimir (n - 1);
        printf ("D %d: ", n);
    }
    printf ("X %d: ", n);
}
```

41. En el siguiente programa escribir exactamente el resultado de la impresión:

```
#include "stdio.h"
#include "math.h"

float op (float x, int n);

void main ()
```

```
{
    printf ("Valor de la funclón:%8.1f\n", op (2.0, 10));
}

float op (float x, int n)
{
    printf ("Entra a op: x =%5.1f n =%3d\n", x, n);

    if (n == 0)
        return 1.0;
    else
        if (n % 2 != 0) /* Determina si n es lmpar. */
            return x * pow (op (x, n / 2), 2);
        else
            return pow (op (x, n / 2), 2);
    printf ("SaIe de op:\n");
}
```

42. Escribir una función recursiva denominada **potencia()** que tenga dos argumentos enteros: el primero es la base y el segundo es el exponente. El resultado de la potencia de los dos números será devuelto por la función. Suponer que el exponente es un entero mayor o igual a 1.

    En un programa leer dos números enteros positivos **a** y **b** que serán enviados a la función **potencia()**, luego utilizar el valor devuelto de la función para imprimir en la pantalla la potencia $\mathbf{a^b}$.

    Nota: No utilizar la función estándar **pow (base, exp)**.

43. Elaborar una función recursiva que reciba un número entero positivo o negativo, y lo imprima dígito a dígito en líneas diferentes.

    Por ejemplo, si la función recibe el numero -31702, debe imprimir lo siguiente:

    ```
    -
    3
    1
    7
    0
    2
    ```

    En el programa principal ingresar números enteros para imprimir el número y el resultado de la función, hasta digitar un cero.

44. Elaborar una función recursiva que reciba un valor **n** entero mayor o igual a cero, y devuelva la suma de los números desde 0 hasta **n**. Por ejemplo, si la función recibe el valor de 5, el valor devuelto será:

    1 + 2 + 3 + 4 + 5 = 15

    Realizar un programa que lea un valor entero mayor o igual a cero, e imprima el resultado de la función.

45. Realizar una función recursiva que reciba un número entero positivo **n**, y devuelva la suma de los números de 1 hasta **n** elevados al cuadrado. Por ejemplo, si **n** es igual a 5, debe devolver la siguiente sumatoria:

    $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$

    En el programa principal validar el ingreso de números enteros positivos hasta digitar un cero, y luego imprimir una tabla como la siguiente:

NUMERO	SUMA
5	55
4	30
6	91
.	
.	

46. Elaborar una función recursiva que reciba un valor entero mayor o igual a cero y devuelva la suma de sus dígitos.

    Realizar un programa que lea un valor entero mayor o igual a cero, e imprima el resultado de la función.

47. El máximo común divisor de dos enteros positivos es el entero más grande que divide de forma uniforme cada uno de los dos números. Es decir, el máximo común divisor, **mcd**, de dos enteros positivos **x** e **y** se define así:

$$mcd\ (x,y) = \begin{cases} y & ,si\ y \leq x\ y\ x\ \%\ y = 0 \\ mcd\ (y,x) & ,si\ x < y \\ mcd\ (y, x\ \%\ y) & ,en\ los\ demás\ casos \end{cases}$$

    Realizar un programa que tenga una función recursiva que calcule el **mcd** de dos números **x** e **y** ingresados desde el teclado. Validar los datos de entrada

48. Elaborar una función recursiva que reciba un número entero y lo devuelva invertido. Por ejemplo, si recibe el número 123, debe devolver el número 321.

Realizar un programa que lea valores enteros hasta que se ingrese un cero, e imprima el resultado de la función para cada número leído.

49. Determinar y especificar una manera recursiva de calcular el cociente entero de la división de dos enteros positivos o negativos, utilizando únicamente restas sucesivas. Escribir una función que implemente dicho método.

    Realizar un programa que lea dos valores enteros, e imprima el resultado de la función.

50. Dice la leyenda que en un templo del Lejano Este, los monjes estaban intentando mover una pila de discos de una estaca hacia otra. La pila inicial tenía 64 discos ensartados en una estaca y acomodados de la parte inferior a la superior en tamaño creciente. Los mojes intentaban mover la pila de esta estaca a la segunda con las limitaciones que exactamente un disco debe ser movido a la vez, y en ningún momento se puede colocar un disco mayor por encima de un disco menor. Existe una tercera estaca disponible para almacenamiento temporal de discos. Se suponía que cuando los monjes terminen su tarea llegará el fin del mundo.

    Desarrollar un programa para resolver este problema llamado "***Torres de Hanoi***" utilizando una función recursiva que imprima la secuencia precisa de las transferencias disco a disco entre dos estacas.

    El mover **n** discos de la estaca 1 a la estaca 3, ver Figura 6.6, puede ser visualizado en términos de solo mover **n - 1** discos, como sigue:

    1. Mover **n - 1** discos de la estaca 1 a la estaca 2, utilizando la estaca 3 como un área de almacenamiento temporal.
    2. Mover el último disco (el más grande) de la estaca 1 a la estaca 3.
    3. Mover los **n - 1** discos de la estaca 2 a la estaca 3, utilizando la estaca 1 como área de almacenamiento temporal.

    El proceso termina cuando la última tarea consiste en mover el disco **n - 1**, es decir, el caso base. Esto se lleva a cabo moviendo el disco sin necesidad de utilizar el área temporal de almacenamiento.

    La función recursiva tiene cuatro parámetros:

    1. El número de discos a moverse.
    2. La estaca en la cual se acumularán estos discos al inicio.
    3. La estaca a la cual esta pila de discos se moverá.
    4. La estaca a utilizarse como área de almacenamiento temporal.

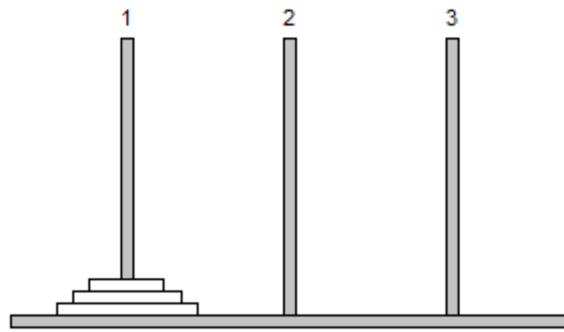


Figura 6.6. Torres de Hanoi

Capítulo 7

MODOS DE ALMACENAMIENTO

Los "modos de almacenamiento" o "especificadores de clase de almacenamiento", permiten determinar qué funciones conocen qué variables, hasta cuándo va a permanecer una variable en memoria para ser usada en el programa.

Cada variable tiene a más de su tipo de dato, un "modo de almacenamiento". Existen cuatro palabras clave en lenguaje C que se emplean para describir los modos de almacenamiento: (SCHILDT, 1994)

- auto (automática)
- extern (externa)
- static (estática)
- register (registro)

Definición del modo de almacenamiento. El "modo de almacenamiento" de una variable queda determinado por el ***lugar*** donde se declara y la ***palabra clave*** empleada, suponiendo que se usa alguna. Los "modos de almacenamiento" preceden al resto de la declaración de la variable, como se muestra a continuación: (SCHILDT, 1994)

palabra_clave tipo variable;

donde:

- **palabra__clave**, es el modo de almacenamiento a emplearse.

Propiedades del modo de almacenamiento. El "modo de almacenamiento" de una variable es responsable de dos propiedades distintas:

1) "*Alcance de una variable*". Se llama *alcance* de una variable a la mayor o menor extensión de la accesibilidad de la misma. Es decir, el "modo de almacenamiento" controla a las funciones determinando qué variables son accesibles, dónde son reconocidas, y en qué parte del programa se las pueden usar.

2) "*Tiempo que permanece en memoria*". El "modo de almacenamiento" determina cuánto **tiempo** va a persistir una variable en memoria.

Por lo tanto, el "modo de almacenamiento" de una variable determina su ***alcance*** y el ***tiempo*** que permanece la variable en la memoria del computador. El "modo de almacenamiento" queda a su vez determinado por el **lugar** donde se declara la variable y por la ***palabra clave*** asociada que se utiliza.

7.1. VARIABLES AUTOMÁTICAS: *auto*

Todas las variables declaradas *dentro de una función*, por defecto, son automáticas, a menos que se indique lo contrario. No obstante, si se desea que la variable sea automática, se emplea la palabra clave **auto**, anteponiéndola al tipo de la variable. Por ejemplo:

```
void funcion (void)
{
    auto int numero;
    /* Sentencias. */
}
```

"*Alcance*". Las variables automáticas tienen alcance local. La única función que conoce una variable de este tipo es aquella donde se la ha definido. Es decir, su alcance queda confinado al bloque en el cual se ha declarado la variable. Debido a que se declara las variables al comienzo del bloque de la función, el "alcance" cubrirá la función completa.

"*Tiempo*". Una variable automática se crea cuando se llama a la función que la contiene. Cuando esta función acaba su tarea y devuelve el control al programa de llamada, la variable desaparece. Su localidad de memoria se empleará en adelante para otros usos. Por supuesto, para no perder el valor de una variable de este tipo, se pueden usar argumentos para enviarle a otra función.

También se puede declarar una variable dentro de un subbloque. En este caso la variable sería conocida únicamente dentro del subbloque de la función. Por ejemplo, declarar la variable **num** dentro del siguiente bloque de código del lazo **for**:

```
for (i = 0; i < 100; i++) {
    int num;
    /* Sentencias. */
}
```

Estas variables se las llama "*variables locales*", porque son conocidas únicamente por las funciones que las contienen, y se crean en el *segmento de pila* en *tiempo de ejecución* del programa, al momento de la llamada de la función donde están declaradas. Por lo que estas variables, cuando no se inicializan, toman por defecto cualquier valor que se encuentra en memoria.

Como consecuencia de lo anterior, otras funciones pueden utilizar variables con el mismo nombre, las que se tratarán como variables independientes almacenadas en diferentes localidades de memoria. (SCHILDT, 1994)

7.2. VARIABLES EXTERNAS: *extern*

Una variable es externa cuando se declara fuera de las funciones, es decir, externamente a ellas. Entonces, al definir una variable externa en un archivo, se puede usar esta variable dentro del archivo, desde el lugar de la definición hasta el final del archivo. Si la variable se necesita en una función anterior a su declaración, sea en el archivo donde se la declaró o en otro, no se la declara nuevamente sino que se la define anteponiendo la palabra clave **extern** al tipo de variable. Esta palabra clave informa al computador que debe buscar la definición de la variable fuera de la función. (SCHILDT, 1994)

Se pueden omitir las definiciones **extern**, si las declaraciones originales de las variables externas aparecen antes de su utilización en el mismo archivo. Pero, si se omite la palabra **extern** en cualquier función, el computador considera que en esta función existe una variable distinta con el mismo nombre de la variable externa, porque se crea una nueva variable distinta y automática con el mismo nombre, y por consiguiente limitada en esta función. Conviene en estos casos etiquetar esta segunda variable con la palabra clave **auto**, para dejar claro que se ha hecho intencionalmente y no por descuido.

Por ejemplo, en el siguiente bosquejo de programa, se muestra la utilización de las variables externas:

```
void magia (void);  /* Prototipo de la función. */
int numero, num;   /* Variables externas. */

void main ()
{
    extern int numero;  /* Declaración como externa, opcional. */
    extern float prom;   /* Se declara como externa, obligatoria. */
    /* Sentencias. */
}

float prom;     /* Variable externa. */

void magia (void)
{                   /* La variable numero no se declara de ninguna forma. */
    int num; /* Variable local. */
    /* Sentencias. */
}
```

donde:

- Las variables externas **numero** y **prom** son conocidas por las funciones **main()** y **magia()**.

- La variable externa **num** es conocida solo por la función **main()**, y en la función **magia()** se crea otra variable local **num**, automática por defecto.

"*Alcance*". Las variables externas son conocidas por todas las funciones, y tiene alcance global. Porque la palabra clave **extern** permite que una función use una variable externa que haya sido declarado después de dicha función en el mismo archivo, o incluso en un archivo diferente; a pesar que los archivos deberán ser compilados por separado y luego enlazados.

"*Tiempo*". Una variable externa permanece siempre en memoria del computador durante toda la ejecución del programa, y al no pertenecer a ninguna función en concreto, no puede eliminarse al acabar cualquiera de las funciones.

Entonces, a estas variables se las llama "variables globales", porque son conocidas por todas las funciones, y se crean en el segmento de datos en tiempo de compilación del programa. Por lo que estas variables cuando no se inicializan, toman por defecto el valor de cero o nulo. (SCHILDT, 1994)

Ejercicio

En el siguiente programa, se desea que la variable ***unidades*** sea conocida por las funciones: **main()** y **critica()**. Es decir, la variable ***unidades*** se lee la primera vez por la función **main()** y las restantes lecturas por la función **critica()**, pero la función **main()** la utiliza para abandonar el lazo **while**.

```
/* PROG0701.C */

#include "stdio.h"

void critica ();        /* Prototipo de la función. */

void main ()
{
 extern int unidades;  /* Declaración obligatoria. */

 printf ("Ingreso de datos, termina con <0>.\n");
 scanf ("%d", &unidades);
 while (unidades != 0)
   critica ();

 printf ("Termina.\n");
}

int unidades;          /* Variable externa. */
```

```
void critica ()
{
  extern int unidades;  /* Declaración opcional. */

  printf ("Lo siento. Pruebe otra vez.\n");
  scanf("%d", &unidades);
}
```

7.3. VARIABLES ESTÁTICAS LOCALES: *static*

Las variables estáticas tienen la palabra clave **static** que se refiere a que estas variables permanecen en memoria todo el tiempo, mientras se encuentra en ejecución el programa, y son declaradas *dentro de una función* anteponiendo la palabra clave **static** al tipo de variable. Por ejemplo:

```
void funcion (void)
{
    static int suma;
    /* Sentencias. */
}
```

"*Alcance*". Las variables estáticas tienen el mismo alcance que de las variables automáticas.

"*Tiempo*". Una variable estática permanece siempre en memoria del computador durante toda la ejecución del programa, y no desaparece cuando la función que la contiene finaliza su ejecución, conservando su valor cuando la función vuelve a ser llamada. Es decir, las variables estáticas mientras permanecen en memoria se comportan igual que las variables externas.

Entonces, estas variables se las llama "*variables estáticas locales*", porque son conocidas únicamente por las funciones que las contienen, y se crean en el *segmento de datos* en *tiempo de compilación* del programa. Por lo que estas variables cuando no se inicializan, toman por defecto el valor de cero o nulo, el mismo que se asignará una sola vez el momento en que se compila el programa. (SCHILDT, 1994)

Ejercicio

En el siguiente programa se muestra una diferencia en la inicialización entre las variables automáticas y estáticas: la variable automática ***muere*** se inicializa cada vez que se llama a la función **prueba()**, en tanto que la variable estática ***vive*** se inicializa en la función **prueba()** una sola vez y cuando se "compila" el programa. Además, la variable ***vive*** se actualiza en cada llamada a la función.

```
/* PROG0702.C */

#include <stdio.h>

#define MAX 3

void prueba (void);

void main ()
{
  int cont;

  for (cont = 1; cont <= MAX; cont++) {
    printf ("Iteración %d:\n", cont);
    prueba ();
  }
}

void prueba (void)
{
  int muere = 1;      /* Se crea en el momento en que se llama a la función.  */
  static int vive =1; /* Se crea en el momento de compilación, manteniéndose. */

  printf("muere = %d y vive = %d\n", muere++, vive++);
}
```

La salida del programa sería:

Iteración 1:
muere = 1 y vive = 1
Iteración 2:
muere = 1 y vive = 2
Iteración 3:
muere = 1 y vive = 3

7.4. VARIABLES ESTÁTICAS EXTERNAS

Se puede declarar también una variable estática externa a las funciones, el resultado es la creación de una variable "estática externa".

La diferencia entre una variable externa ordinaria (global) y una variable externa estática reside en su "***alcance***": "*La variable externa ordinaria*" se puede utilizar en funciones de cualquier archivo, mientras que "*la variable estática externa*" puede

emplearse únicamente en funciones del mismo archivo que se encuentre debajo de su declaración; pero para usarla antes de su declaración, se debe definir como **extern**. (SCHILDT, 1994)

Se puede conseguir una variable estática externa colocando su declaración *fuera de la función*. Por ejemplo:

```
static int arco = 1;

void circulo (void)
{
    extern int arco;  /* Declaración opcional, en este caso. */
    /* Sentencias. */
}
```

Las variables estáticas permiten al programador ocultar partes de un programa cuando éste es grande y complejo. Además, permiten crear funciones generales que formen parte de una biblioteca, para su uso posterior.

Ejercicio

Realizar un programa que genere **num** números aleatorios, para lo cual se deben crear las funciones: **aleat()** y **saleat()**.

- **aleat()**, es una función "generadora de números aleatorios". El planteamiento comienza con un número que se llama "*semilla*", esta semilla se usa para producir un nuevo número aleatorio, el cual a su vez se utilizará como nueva semilla, y así sucesivamente. Para generar estos números aleatorios se usa la siguiente fórmula que tiene el valor de azar en el rango de -32768 a 32767:

  azar = (azar * 25173 + 13849) % 65536

  donde **azar**, de la segunda parte de la fórmula, es la semilla inicial.

- **saleat()**, es una función que permite "inicializar la primera semilla".

Además, se debe hacer que la variable azar sea una variable estática **externa** conocida por las funciones **aleat()** y **saleat()**.

El programa debe leer desde teclado la cantidad de números aleatorios (**num**) y la semilla inicial (**semilla**).

/* PROG0703.C */

```
#include <stdio.h>

int aleat (void);
void saleat (int x);

static int azar = 1;

void main ()
{
  int cont, num;
  int semilla;

  printf ("Ingrese la cantidad número aleatorios: ");
  scanf ("%d", &num);
  printf ("Ingrese un número como semilla: ");
  scanf ("%d", &semilla);
  saleat (semilla);          /* Pone una nueva semilla. */
  printf ("Números aleatorios generados:\n");
  for (cont = 1; cont <= num; cont++)
    printf ("%d\n", aleat ());
}

int aleat (void)
{
  /* Fórmula para que el valor de azar esté, en el rango -32768 a 32767. */
  azar = (azar * 25173 + 13849) % 65536;
  return azar;
}

void saleat (int x)
{
  azar = x;
}
```

Una salida del programa sería:

Ingrese la cantidad de números aleatorios: 4 <ENTER>
Ingrese un número como semilla: 1 <ENTER>
Números aleatorios generados:
-26514
-4449
20196
-20531

7.5. VARIABLES REGISTRO: *register*

Las "variables registro" se almacenan en los *registros de la CPU*, en donde son mucho más accesibles y se manipulan más rápidamente que en la memoria. La palabra clave **register** es una sugerencia al compilador, para que la variable almacenada en este modo sea usada eficientemente; colocándola en un registro de la CPU. (SCHILDT, 1994)

Por lo demás, las variables registro se comportan como las variables automáticas. Por ejemplo, como en la siguiente declaración:

```
void funcion (void)
{
    register int rapido;
    /* Sentencias. */
}
```

Los únicos tipos de datos que se almacenan en los registros de la CPU, son los tipos caracter y entero; por lo tanto, los tipos restantes solo se almacenan en la memoria.

Normalmente, las variables registro por ser de acceso rápido, se usan para el control de los lazos, así como en lugares donde se hagan muchas referencias a una misma variable.

Como las "variables registro" se guardan en los registros de la CPU, no tienen direcciones de memoria, por lo que no se puede usar el operador **&**; en el caso que existen registros diponibles.

La declaración de una variable registro es más una solicitud que una orden directa, porque el número de registros que están disponibles, que suelen ser bastante escasos, normalmente 4 o un poco más; por tanto, es posible que no se pueda atender a esa solicitud. En tal circunstancia, la variable se transforma en una variable automática ordinaria.

7.6. EL MODO DE ALMACENAMIENTO A EMPLEARSE

El modo de almacenamiento es casi siempre "automático", porque así se ha escogido como opción por defecto.

A primera vista el almacenamiento "externo" es bastante seductor, porque no se tendrá que preocuparse de utilizar en las funciones: argumentos, punteros y comunicaciones entre funciones. Pero, se debe tener en cuenta que una función puede cambiar los datos de las variables de otras funciones, sin ningún control; por lo que se deberá utilizar variables estáticas, para crear funciones independientes.

Además, no se podrían utilizar las funciones como cajas negras, ya que una de las reglas es mantener las tareas de cada función tan privadas como se pueda, compartiendo el mínimo número posible de valores y variables con otras funciones.

Por último, habrá ocasiones en que los restantes modos de almacenamiento serán útiles en ciertas aplicaciones específicas y de modo exclusivo.

NOTAS:

- Cuando no se inicializan las variables "*automáticas*" y "*registro*", toman basura que se encuentra en ese momento en la memoria del computador o en los registros de la CPU, respectivamente.

- Cuando no se inicializan las variables "*externas*" y "*estáticas*", toman el valor cero en números, y nulo en punteros y caracteres.

Las propiedades de los modos de almacenamiento se resumen en la Tabla 7.1. (DEITEL & DEITEL, 1995)

Tabla 7.1 Propiedades de los modos de almacenamiento

MODO DE ALMACENAMIENTO	PALABRA CLAVE	TIEMPO	ALCANCE	DECLARACIÓN EN FUNCIÓN
Automático	auto	Temporal	Local	Dentro
Registro	register	Temporal	Local	Dentro
Estático	static	Residente	Local	Dentro
Externo	extern	Persistente	Global (Todos los Archivos)	Fuera
Externo estático	static	Persistente	Global (A un Archivo)	Fuera

Ejercicio

Realizar un programa para simular un juego de azar: el lanzamiento de dos dados que tienen seis caras cada uno. Lo que se desea es un número aleatorio entre 1 y 6, pero la función **aleat()** del ejercicio anterior produce un número aleatorio en el rango de -32768 a 32767, por lo que se debe hacer los siguientes ajustes:

1) Convertir el número aleatorio a **float**, para tener parte decimal. Luego dividir el número aleatorio por 32768, y el resultado sería un número **x** en el rango: -1 <= x < 1.

2) Sumar 1, para que el número satisfaga la relación: 0 <= x < 2.

3) Dividir por 2, para obtener la relación: 0 <= x < 1.

4) Multiplicar por 6, para obtener la relación: 0 <= x < 6. Pero el cero no sirve como valor.

5) Sumar 1, en este momento la relación es: 1 <= x < 7. Pero, todavía se tiene una fracción decimal.

6) Truncar a entero, para tener el valor en el rango: 1 a 6.

7) Para generalizar el número de caras deseadas, basta con reemplazar el número 6 de la etapa 4) por ese número.

```
/* PROG0704.C */

#include <stdio.h>

#define ESCALA 32768.0

int cubilete (int lados);
int aleat ();
void saleat (unsigned x);

static int azar = 1;

void main ()
{
  int lados, dados, cont, tirada, semilla;

  printf ("Ingrese la semilla: ");
  scanf ("%d", &semilla);
  saleat (semilla);
  printf ("Indique el número de caras por dado. Terminar con <0>: ");
  scanf ("%d", &lados);
  while (lados > 0) {
    printf ("Cuántos dados?: ");
    scanf ("%d", &dados);
    for (tirada = 0, cont = 1; cont <= dados; cont++)
      tirada += cubilete (lados);  /* Cálculo total de la tirada. */
    printf ("Acaba de sacar un %d con %d dados de %d caras.\n",
          tirada, dados, lados);
    printf ("Cu ntas caras?. Pulse <0> para terminar: ");
    scanf ("%d", &lados);
```

```
 }
 printf ("\n");
}

int cubilete (int lados)
{
 float tirada;

 tirada = ((float) aleat () / ESCALA + 1.0) * lados / 2.0 + 1.0;
 return (int) tirada;
}

int aleat ()
{
 /* Fórmula para que el valor de azar está, en el rango -32768 a 32767. */
 azar = (azar * 25173 + 13849) % 65536;
 return azar;
}

void saleat (unsigned x)
{
 azar = x;
}
```

Una salida del programa sería:

Ingrese la semilla: 1 <*ENTER*>
Indique el número de caras por dado. Terminar con <0>: 6 <*ENTER*>
Cuántos dados ?: 2 <*ENTER*>
Acaba de sacar un 4 con 2 dados de 6 caras.
Cuántas caras?. Pulse <0> para terminar: 0 <*ENTER*>

PROBLEMAS PROPUESTOS

1) Escribir sentencias que asignen números enteros aleatorios a la variable **n** en los rangos siguientes:

    a) 1 <= n <=2
    b) 1<= n <= 100
    c) 0<= n <=9
    d) 1000 <= n <= 1112
    e) -1 <= n <=1
    f) -3 <= n< =11

2) Escribir una sola función que imprima un número al azar, para cada uno de los siguientes conjuntos de enteros:

    a) 2, 4, 6, 8, 10
    b) 3, 5, 7, 9, 11
    c) 6, 10, 14, 18, 22

3) Escribir un programa que simule lanzar una moneda, que en cada lanzamiento de la moneda imprima "***cara***" o "***cruz***", es decir, el programa deberá llamar a la función **Arriba()** que no tiene argumentos y que devuelva 0 para las caras, 1 para las cruces, luego imprimir los resultados. Además, el programa debe permitir lanzar la moneda 100 veces y contar el número de veces que aparece alguno de los dos lados de la moneda.

    Nota: El lanzamiento de la moneda debe ser de forma aleatoria.

4) Escribir un programa que codifique y decodifique una línea de texto.

    "*Para codificar una línea de texto*", proceder como sigue:

    a) Convertir cada caracter, incluído el espacio en blanco, en su equivalente ASCII.

    b) Generar un entero positivo aleatorio entre 0 y 127, con la siguiente fórmula:

        azar = azar * 25173 + 13849 % 65536

        donde:
        - -32768 <= azar <= 32767

        Añadir este entero aleatorio al equivalente ASCII de cada caracter de la línea de texto completa.

c) Suponer que N1 representa el menor valor permisible en ASCII y N2 es el mayor valor permisible en ASCII. Si el número obtenido en el paso b) excede de N2, se le resta N2 y se suma el resultado a N1. De igual forma el número codificado estará siempre entre N1 y N2, que representará siempre algún caracter ASCII.

d) Escribir los caracteres correspondientes a los valores ASCII codificados de la línea de texto.

"*Para decodificar una línea de texto*", el procedimiento es invertido, pero se debe asegurar que el entero positivo aleatorio es el mismo que se usó para codificar

Nota: Realizar funciones para la codificación, decodificación y generación del entero aleatorio.

5) Escribir un programa que simule el juego de "***adivine el número***". El programa escoge un entero al azar en el rango del 1 al 1000, para que sea adivinado, y a continuación el programa escribe:

Tengo un número entre 1 y 1000.
Puede Ud. adivinar mi número?
Por favor escriba su primer intento.

El jugador entonces escribe su primera estimación. El programa responde con una de las siguientes opciones:

1. *Excelente. Ud. adivinó el número!*
   Le gustaría jugar de nuevo (Y/N)?
2. *Para abajo. Intente de nuevo.*
3. *Para arriba. Intente de nuevo.*

Si la adivinanza del jugador es incorrecta, se deberá repetir hasta que el jugador obtenga el número correcto, para lo cual el programa debe indicar al jugador "***Para arriba***" o "***Para abajo***", como una ayuda a la contestación. Además, contar el número de veces que intenta adivinar el jugador y mostrar uno de los siguientes mensajes:

- Si el número es 10 o menor imprimir: "***O bien Ud. conoce el secreto o tiene buena suerte!***".

- Si el jugador adivina el número en 10 intentos imprimir: "***Ah ah! Ud. conoce el secreto!***".

- Si el jugador hace más de 10 intentos imprimir: "***Podría hacerlo mejor!***".

NOTA: La técnica de búsqueda empleada en este problema es conocida como "***búsqueda binaria***", que debe ser realizada en una función.

6) Utilizar las funciones estándares que generan números aleatorios, para generar un número del 1 al 4 y seleccionar una respuesta apropiada de cada una de las siguientes contestaciones:

   Respuestas a las contestaciones correctas:

1. Muy bien!
2. Excelente!
3. Buen trabajo!
4. No podrías hacerlo mejor!

   Respuestas a las contestaciones incorrectas:

1. No. Por favor intente de nuevo
2. Equivocado. Por favor una vez más.
3. No te rindas!
4. No. continúa intentando.

   Realizar un programa que utilice una estructura **switch** con funciones **printf()** para emitir las respuestas.

7) El juego de azar más popular de los casinos es el juego de dados conocido como "***craps***", las reglas del juego se muestran a continuación:

   Un jugador lanza 2 dados de 6 caras cada uno y las caras contienen 1, 2, 3, 4, 5, y 6 puntos. Una vez que los dados se hayan detenido, se calcula la suma de los puntos en las dos caras superiores, para establecer si se ha ganado o se ha perdido de acuerdo a lo siguiente:

   Existen dos formas de ganar:

   a) Obteniendo una puntuación de 7 u 11 en la primera lanzada.

   b) Obteniendo una puntuación de 4, 5, 6, 8, 9 o 10 en la primera lanzada, entonces dicha suma se convierte en el "punto" o en la "lanzada", y el jugador deberá continuar lanzando los dados para conseguir de nuevo el mismo "punto", antes de obtener un 7 en una lanzada posterior.

   Existen también dos formas de perder:

a) Lanzar los dados una vez y obtener 2, 3 o 12 (conocido como "craps"), es decir la casa "gana".

b) Obtener 4, 5, 6, 8, 9 o 10 en la primera lanzada y obtener un 7 en la lanzada posterior antes de repetir el "punto" original.

Diseñar una función para ejecutar un juego de "*craps*", y otra función que no tiene argumentos y devuelva el resultado de cada lanzada que es la suma de los dos dados (una cantidad entera con valor entre 2 y 12).

Realizar un programa que permita apuestas por lo que debe solicitar al jugador que introduzca una apuesta menor o igual a 10 dólares, que es su banco. Si el jugador gana, sumar la apuesta a la cantidad del banco e imprimir. Si el jugador pierde, reducir la apuesta a la cantidad del banco e imprimir. Verificar si el banco se ha convertido en 0, y si es así, imprimir el mensaje "***Lo siento. Ud. está quebrado!***". Además, el programa debe incluir un contador que determine el número total de victorias, para estimar la probabilidad de ganar en este juego de dados. Este valor, expresado en decimal, es igual al número de victorias dividido entre el número total de partidas jugadas. Si la probabilidad excede de 0.500, es favorable al jugador; de lo contrario es favorable a la casa.

Este programa requerirá de un generador de números aleatorios entre 1 y 6. La función de biblioteca para generar números aleatorios es **rand()**, que devuelve un entero entre 0 y (21^5 -1) (Esto es entre 0 y 32767). Luego convertir cada cantidad entera aleatoria en un número de punto flotante, **x**, que se encontrará entre 0 y 0.99999..., mediante la relación:

x = rand() / 32768.0

Para representar el resultado de un lanzamiento de un dado entre 1 y 6, se hará:

n = 1 + (int) (6 * x)

Para lanzar dos dados se repite este proceso dos veces y se suma los resultados.

La función de biblioteca **srand()** se utiliza para inicializar el generador de números aleatorios. Esta función requiere de un entero positivo, llamado **semilla**, que establece la secuencia de números aleatorios generados por **rand()**. Se generará una secuencia de números aleatorios diferente por cada semilla. Por comodidad, se puede incluir un valor de la semilla como constante simbólica dentro del programa.

8) Escribir un programa que ayudaría a un alumno de escuela primaria a aprender a multiplicar. Utilizar la función **rand()** para producir dos enteros positivos de un dígito o más y a continuación escribir una pregunta como la siguiente:

Cuanto es 4 veces 8?.

Luego el alumno escribe la respuesta. El programa verifica la respuesta del alumno. Si es correcta, imprime "***Muy bien!***" y a continuación solicita otra multiplicación. Si la respuesta es incorrecta, imprimará "***No. Por favor intente de nuevo.***" y a continuación permitirá que el alumno vuelva a intentar la misma pregunta en forma repetida, hasta que al final la conteste correctamente.

El programa tiene las siguientes características:

a) Contar el número de respuestas correctas e incorrectas escritas por el alumno. Una vez que el alumno escriba 10 respuestas, el programa deberá calcular el porcentaje de respuestas correctas. Si el porcentaje es menor de 75%, el programa deberá imprimir "***Por favor llame a su instructor para ayuda extra***" y termina el programa.

b) Permitir al usuario la introducción de una capacidad de nivel de grado. Un nivel de grado 1 significa que los problemas utilizan solo números de un dígito, un nivel de grado 2 significa la utilización de números hasta dos dígitos, etc.

c) Permitir al usuario escoger el tipo de problemas aritméticos. La opción 1 significa solo problemas de suma, 2 significa problemas de resta, 3 significa problemas de multiplicación, 4 significa problemas de división y 5 significa problemas entremezclados al azar, de todos los tipos anteriores.

9) Escribir un programa computacional que permita a una persona jugar a "***tres en raya***", de forma que la computadora pueda ser tanto el primero como el segundo jugador. Si la computadora es el primer jugador, el primer movimiento se generará al azar. Mostrar el estado del juego tras cada movimiento, e indicar cuando gana cada jugador.

10) Calcular la "***media ponderada***" de una lista de **n** números, utilizando la fórmula:

$$x_{media} = f_1x_1 + f_2x_2 + \dots + f_nx_n$$

donde:

- **fi,** son el peso de cada cantidad y han de ser fraccionarios que cumplen las siguientes condiciones:

  $0 <= f_i < 1$
  $f_1 + f_2 + \dots + f_n$

Realizar una función para calcular la media ponderada, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

El programa principal debe generar una tabla para valores de **i** de 1 a 10 de acuerdo a la siguiente cabecera:

i	f	x

11) Realizar una función para que calcule el "***producto acumulativo***" de una lista de **n** números, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas. El programa debe ingresar los datos y utilizar esta función para calcular el producto acumulativo.

12) Calcular la "***media geométrica***" de una lista de **n** números utilizando la fórmula:

$$x_{media} = [x_1 x_2 x_3 \dots x_n]^{1/n}$$

Realizar una función para la media geométrica y otra para la media aritmética, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

En el programa comparar los resultados de la media geométrica y los obtenidos con la media aritmética de los mismos datos. ¿Qué media es mayor?.

13) Calcular el "***seno de un ángulo x***" de forma aproximada sumando los **n** primeros términos de la serie infinita:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

donde:

- **x** se encuentra en radianes (1 radián = 180^0).

Realizar una función para calcular el seno de un ángulo, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

Escribir un programa que lea el valor de **x** y calcule su seno, sumando los **n** primeros términos, donde **n** es un número entero positivo.

14) Calcular y tabular una lista de "***números primos***". Un número primo es una cantidad entera que es divisible (sin resto) solo por 1 y por si mismo. Por ejemplo, 7 es un número primo, pero 6 no lo es.

Realizar una función para calcular un número primo, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

El programa debe calcular y presentar una lista con los **n** primeros números primos.

15) Calcular los "***pagos de un préstamo***" de **P** dólares realizado en un banco, con el reconocimiento de que se devolverán **A** dólares cada mes hasta que se haya completado la cantidad total prestada. Parte del pago mensual serán intereses calculados como el **i** por ciento de la cantidad aún no pagada. El resto del pago servirá para reducir la cantidad adeudada.

Escribir un programa que determine la siguiente información:

a) La cantidad de interés pagada por mes.
b) La cantidad de dinero aplicado a la reducción de la deuda total cada mes.
c) La cantidad total de intereses que se lleva pagada al final de cada mes.
d) La cantidad de deuda aún no pagada al final de cada mes.
e) El número de pagos mensuales necesarios para devolver el préstamo.
f) La cuantía del último pago (ya que puede ser menor que **A**).

Realizar funciones para cada literal, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

16) Realizar una función para calcular la "***media de los notas***" de los exámenes obtenidas por los estudiantes de una clase en seis exámenes de un curso de programación en lenguaje C, teniéndose cuidado con la elección de argumentos y si es necesario de variables externas.

Escribir un programa que acepte como entrada cada nombre de estudiante y sus notas, escribir a continuación en papel el nombre del estudiante, las notas de los exámenes y la media de cada estudiante, como se describe a continuación:

Nombre	Notas						Media
Acosta	15	15	16	15	15	15	15.17
Burbano	20	10	17	15	15	10	14.50
Dávila	20	8	18	15	16	15	15.33
Maldonado	15	15	12	15	12	15	14.00

17) Se desea encontrar el valor particular de x que haga máxima la siguiente función:

$y = x \cos(x)$

en el intervalo limitado por **x=0** y **x=π**. Para lo cual se necesita:

- Conocer con bastante exactitud el valor que maximice a **x**.
- Evaluar el menor número de veces posible la función **y = x cos(x)**.

Una manera de resolver este problema sería utilizando el "***esquema de eliminación***", que es un procedimiento altamente eficiente para todas las funciones que solo tienen un máximo (un "pico") dentro del intervalo de búsqueda, como se indica en la Figura 7.1:

Se empieza con dos puntos de búsqueda en el centro del intervalo de búsqueda, marcando una pequeña distancia entre ellos (en el orden del 0.00001), como se muestra a continuación:

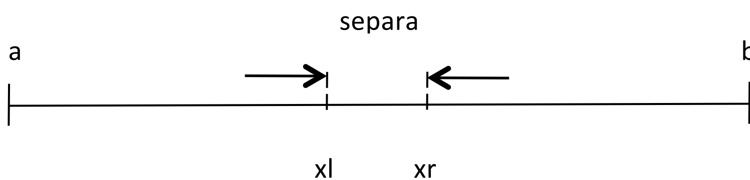


Figura 7.1. Intervalo de búsqueda

donde:

- **a**, extremo izquierdo del intervalo.
- **xl**, punto interior de búsqueda a la izquierda.
- **xr**, punto interior de búsqueda a la derecha.
- **b**, extremo derecho del intervalo.

Si se conocen **a**, **b** y **separa**, entonces los puntos interiores pueden calcularse como:

xl = a + .5 * (b – a - separa)
xr = a + .5 * (b - a + separa) = xl + separa

Luego se debe evaluar la función **y = x cos(x)** en **xl** y **xr** y se llamarán a estos valores **yl** e **yr**, que serán comparados, y se eliminará la porción del intervalo de búsqueda que contiene el valor más pequeño de **yl** e **yr** de acuerdo al siguiente procedimiento:

- Suponiendo que **yl** resulta ser mayor que **yr**, entonces el máximo está entre **a** y **xr**. Por tanto, se tomará el intervalo de búsqueda que va desde **x=a** hasta **x=xr**. Ahora se referirá al punto viejo **xr** como **b**, pues este es el extremo derecho del nuevo intervalo de búsqueda, y se generan otros dos nuevos puntos de búsqueda **xl** y **xr**. Estos puntos se localizan en el centro del nuevo intervalo de búsqueda, separados una distancia **separa**, como en el caso inicial.

- Suponiendo que **yl** resulta ser menor que **yr**, esto indicará que el nuevo intervalo de búsqueda se halla entre **xl** y **b**. Por tanto, se renombra el punto originalmente llamado **xl** como **a** y se genera dos nuevos puntos de búsqueda, **xl** y **xr**, en el centro del nuevo intervalo de búsqueda, como en el caso inicial.

- Si ocurre que ambos valores interiores son idénticos (lo cual puede suceder, pero es inusual), entonces el procedimiento de búsqueda se detiene y se asume que el máximo tiene lugar en el centro de los dos últimos puntos internos.

Se continúa generando un nuevo par de puntos de búsqueda en el centro de cada nuevo intervalo, comparando los respectivos valores de **y**, y eliminando una porción del intervalo de búsqueda hasta que el nuevo intervalo de búsqueda se hace menor que **3 * separa**.

Una vez acabada la búsqueda, tanto porque el intervalo de búsqueda se ha hecho lo suficientemente pequeño o porque los dos puntos interiores tienen valores idénticos de **y**, se puede calcular la localización aproximada del máximo como:

$x_{max} = 0.5 * (xl + xr)$

Realizar un programa para determinar el valor máximo correspondiente de la función, que se puede obtener como **f(xmax)=xmax*cos(xmax).** Crear una función para evaluar la función matemática **y = x cos(x)**, y la función para reducir el intervalo.

18) El método de "***esquema de eliminación***" del ejercicio anterior para maximizar una función puede modificarse fácilmente para "***minimizar***" una función de **x**. Este proceso de minimización puede proporcionar una técnica altamente eficaz para calcular las raíces de una ecuación algebraica no lineal.

    Suponer que se quiere encontrar el valor particular de **x** que hace la función **f(x)** igual a cero. Si se hace **y(x) - f(x)²**, entonces la función **y(x)** será siempre positiva, excepto para aquellos valores de **x** que sean raíces de la función dada, para los cuales **f(x)** y por lo tanto **y(x)** sean nulas. Por tanto, cualquier valor de **x** que minimice a **y(x)** es una raíz de la ecuación **f(x)=0**.

    Realizar el programa para minimizar una función dada. Usar el programa para obtener las raíces de las siguientes ecuaciones:

    a) $x + \cos(x) = 1 + \text{seno}(x)$, $\pi/2 < x < \pi$
    b) $x^5 + 3x^2 = 10$, $0 <= x <= 3$

19) Resolver ecuaciones algebraicas incluidas aquellas que no pueden resolverse utilizando métodos más directos. La solución se determina mediante un

procedimiento repetitivo de "***prueba y error***" que verifica un valor inicial supuesto. Por ejemplo la ecuación:

$x^5 + 3x^3 - 10 = 0$

se puede escribir de la siguiente forma:

$x = (10 - 3x^3)^{1/5}$

El procedimiento comenzará con un valor inicial **x**, que será sustituido en la parte derecha de la ecuación arreglada y a continuación calcular un nuevo valor de **x**. Si este nuevo valor es igual (o muy próximo) al valor anterior, se obtiene la solución de la ecuación. Caso contrario se sustituirá este nuevo valor en la parte derecha de la ecuación y se volverá a obtener un nuevo valor de **x**, y así sucesivamente. Continuará este procedimiento hasta que los valores sucesivos de **x** sean lo suficientemente próximos, o hasta que se exceda un número especificado de iteraciones.

Realizar un programa que tenga una función con los parámetros adecuados para calcular la solución de la ecuación.

20) Si se coloca una cantidad dada de dinero **A** en una libreta de ahorros a principio de cada año durante **n** años con un interés **i** anual, la cantidad de dinero que se acumulará tras **n** años viene dada por:

$F = A[(1 + i / 100) + (1 + i / 100)^2 + (1 + i / 100)^3 + \ldots + (1 + i / 100)^n]$

Escribir un programa que determine lo siguiente:

a) ¿Cuánto dinero se acumulará después de 25 años si se depositan 1000 dólares a comienzo de cada año y el porcentaje de interés compuesto es del 5 por 100 anual?

b) ¿Cuánto dinero se tiene que depositar al principio de cada año para acumular 100000 dólares después de 25 años, también suponiendo un interés compuesto del 5 por 100 anual?.

En cada caso, primero determinar la cantidad de dinero desconocido. Luego crear una tabla mostrando la cantidad total que se acumulará al final del año.

Escribir una función para obtener la exponenciación mediante la fórmula:

$y = x^n$

donde:

- **y** y **x** son variables de punto flotante y **n** una variable entera.

21) La fórmula adecuada para obtener intereses trimestrales en vez de anuales es:

$$F = A[(1 + i / 100m)^{m} + (1 + i / 100m)^{2m} + (1 + i / 100m)^{3m} + ... + (1 + i / 100m)^{nm}]$$

donde:

- **A**, cantidad dada de dinero a principio de cada año.
- **i**, interés anual.
- **n**, número de períodos de interés en años.
- **m**, número de períodos de interés por año.
- **F**, cantidad de dinero que se acumulará tras **n** años.

Escribir un programa con funciones que determine:

a) ¿Cuánto dinero se acumulará después de **n** años si se depositan **A** dólares a comienzo de cada año y el porcentaje de interés compuesto es del **i** por 100 anual?.

b) ¿Cuánto dinero se tiene que depositar al principio de cada año para acumular **F** dólares después de **n** años, también suponiendo un interés compuesto del **i** por 100 anual?.

En cada caso, primero determinar la cantidad de dinero desconocido, luego crear una tabla mostrando la cantidad total que se acumulará al final del año.

22) El costo de una hipoteca se determina de modo que el deudor paga a la institución prestamista, la misma cantidad de dinero al mes durante el tiempo de hipoteca. Este procedimiento se conoce como método de "***pago constante***". Al comienzo de la hipoteca la mayoría del pago mensual se destina al pago de los intereses y solo una pequeña fracción a reducir el pago del préstamo. Gradualmente, la cantidad se reduce, lo que provoca que el pago mensual de intereses decrezca.

Escribir un programa con funciones que pueda ser usado por una institución que preste dinero para suministrar la siguiente información a un cliente potencial:

- ¿Cuál será el pago mensual con el interés establecido?.
- ¿Qué cantidad del pago mensual se destina al pago de los intereses?.
- ¿Qué interés total ha pagado desde que se le prestó el dinero y cuánto adeuda al final de cada mes?.

Asumir que la cantidad del préstamo, el interés anual, y la duración del préstamo se especifican desde el teclado.

La cantidad pagada mensualmente se calcula con la fórmula:

$A = i\ P(1 + i)^n / [(1 + i)^n - 1]$

donde:

- **A**, pago mensual.
- **P**, cantidad total del préstamo.
- **i**, interés mensual, se puede expresar como decimal.
- **n**, número total de pagos mensuales.

El pago mensual de intereses se puede calcular con la fórmula:

$I = iB$

donde:

- **I,** pago mensual de interés.
- **B,** pago actual.

El pago actual es simplemente igual a la cantidad original del préstamo, menos la cantidad de pagos previos.

El pago mensual (la cantidad usada para reducir el pago) es:

$T = A - I$

donde:

- **T**, pago mensual.

El programa debe calcular el valor de una hipoteca de **P** dólares a **m** años con un interés anual del **i** por 100. Repetir para intereses anuales del 5 por 100 y al 9.5 por 100 en intervalos del 0.5.

23) Suponiendo que los pagos mensuales del costo de una hipoteca se calculan por el método del interés simple, esto es, que la misma cantidad se aplica para reducir el préstamo cada mes, se realiza así:

$T = P / n$

donde:

- **T**, pago mensual.
- **P**, cantidad total del préstamo.
- **n**, número total de pagos mensuales.

El interés mensual, sin embargo, dependerá de la cantidad del pago:

I = iB

donde:

- **I**, pago mensual de interés.
- **i**, interés mensual, se puede expresar como decimal.
- **B**, pago actual.

Por tanto, el pago mensual total, **A = T + I**, decrecerá cada mes según disminuya el pago.

Escribir un programa con funciones para calcular el costo de una hipoteca usando este método, para calcular el valor de una hipoteca de **A** a **m** años con un interés anual del **i** por 100.

24) "***La regla trapezoidal***" sirve para calcular el área encerrada por una curva, y consiste en dar una serie de puntos discretos $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ leídos que describen la curva **y = f(x)**, donde **x** está limitada por x_1 y x_n. Luego se aproxima el área encerrada por la curva dividiendo la curva en un número de pequeños rectángulos y calculando el área de esos rectángulos, donde la precisión de la solución mejora según aumenta el número de puntos. La fórmula sería:

$$A = (y_1 + y_2)(x_2 - x_1) / 2 + (y_2 + y_3)(x_3 - x_2) / 2 + ... + (y_{n_1} + y_n)(x_n - x_{n_1}) / 2$$

donde:

- $(y_i + y_{i+1}) / 2$, es la altura media de cada rectángulo.
- $(x_{i+1} - x_i)$, es la anchura de cada rectángulo.
- **i = 1,2, .., n**.

Escribir un programa que implemente una función para evaluar la fórmula matemática **y = f(x), y = x3**, entre los límites **x=1** y **x=4**. Resolver este problema para 16 puntos espaciados, luego con 61, y finalmente con 301 puntos. Comparar las respuestas.

25) En el método conocido como "regla trapezoidal" del ejercicio anterior, para calcular el área encerrada por una curva **y = f(x)**, donde una serie de valores tabulados $(x_1, y_1), (x_2, y_2), ..., (xn, yn)$ se usan para describir la curva, si los valores tabulados de **x** se espacian por igual, la ecuación sería:

$$A = (y_1 + 2y_2 + 2y_3 + 2y_4 + \dots + 2y_{n-1} + y_n)\, h / 2$$

donde:

- **h**, es la distancia entre dos valores sucesivos de **x**.

Otra técnica aplicable cuando hay un número impar de intervalos igualmente espaciados **h** es la "***regla de Simpson***". La ecuación para implementar la regla de Simpson es:

$A = (y_1 + 4y_2 + 2y_3 + 4y_4 + \ldots + 4y_{n-1} + y_n)\ h / 3$

Para un valor dado de **h**, este método proporciona mayor exactitud que la regla trapezoidal.

Escribir un programa usando ambas técnicas asumiendo un número impar de puntos espaciados por igual, para calcular el área encerrada por la curva:

$$y = e^{-x^2}$$

donde:

- **x**, va de 0 a 1.

Nota: Implementar cada método con una función separada y utilizar otra función para evaluar **y(x)**.

26) "***El método Montecarlo***" es otra técnica para calcular el área encerrada por una curva, que hace uso de números generados aleatoriamente. Suponiendo que la curva **y=f(x)** es positiva para cualquier valor de **x** entre los límites superior **x=b** e inferior **x=a**, y sea **y** el mayor valor de **y** dentro de estos límites. El método Montecarlo funciona así:

a) Empezar con un contador a cero.
b) Generar un número aleatorio, $\mathbf{r_x}$, de valor comprendido entre **a** y **b**.
c) Evaluar $\mathbf{y(r_x)}$.
d) Generar un segundo número aleatorio, $\mathbf{r_y}$, de valores entre 0 e **y***.
e) Comparar $\mathbf{r_y}$ con $\mathbf{y(r_x)}$. Si $\mathbf{r_y}$ es menor o igual que $\mathbf{y(r_x)}$, entonces este punto caerá en o bajo la curva dada. Así el contador se incrementa en 1.
f) Repetir los pasos de b) al e) un número grande de veces. Cada vez se denomina un ***ciclo***.
g) La fracción de puntos que caen en o bajo la curva **F** tras completar un número de ciclos, es calculada como el valor del contador dividido por el número total de ciclos.

El área se obtiene como:

$A = F\ y^* (b - a)$

Escribir un programa con funciones para calcular el área encerrada por la curva:

$$y = e^{-x^2}$$

entre los límites **a=0** y **b=1**, y determinar cuántos ciclos son necesarios para obtener el resultado con una precisión de tres cifras significativas.

27) Una variable aleatoria normalmente distribuida **x**, con media y "desviación estándar", se puede generar con la fórmula:

$$x = \mu + \sigma \frac{\sum_{i=1}^{N} r_i \quad - N / 2}{\sqrt{N / 12}}$$

donde:

- **r_i**, es un número uniformemente distribuido de valores comprendidos entre 0 y 1.

- **n=12**, es un valor que frecuentemente se selecciona.

La base de la fórmula es el "***teorema del límite central***", que establece que un conjunto de valores medios de variables aleatorias uniformemente distribuidas tienden a ser normalmente distribuidas para valores moderadamente grandes de **N**.

Escribir un programa que generará un número especificado de variables normalmente distribuidas con una media y una desviación estándar dadas. Sean el número de variables aleatorias, la media, y la desviación estándar datos de entrada del programa.

Generar cada variable aleatoria con una función que acepte la media y la desviación estándar como argumentos.

28) "***Los números de Fibonacci***" forman una serie en la que cada número es igual a la suma de los dos números anteriores, como se muestra a continuación:

$F_n = F_{n-1} + F_{n-2}$

donde:

- **F_n**, se refiere el número **n**-ésimo de Fibonacci.

Los dos primeros números son iguales a 1, esto es:

$F_1 = F_2 = 1$

Por tanto,

$F_3 = F_2 + F_1 = 1 + 1= 2$
$F_4 = F_3 + F_2 = 2 + 1 = 3$

$F_5 = F_4 + F_3 = 3 + 2 = 5$

y así sucesivamente.

Escribir un programa que incluya una función recursiva para determinar el valor **n**-ésimo número de Fibonacci, $\mathbf{F_n}$. El valor de **n** es ingresado por el usuario.

Capítulo 8

ARREGLOS

Un arreglo es una colección de datos del mismo tipo que se referencian por un mismo nombre, cuyos datos llamados "elementos" se distinguen entre sí con índices o direcciones de memoria; es decir, a un elemento específico de un arreglo se accede mediante índices o su dirección de memoria. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento. (BECERRA, 1991)

Los arreglos se clasifican por el número de índices de acceso a los elementos, por lo que pueden tener una o varias dimensiones: unidimensionales, bidimensionales y multidimensionales.

8.1. ARREGLOS UNIDIMENSIONALES

Un arreglo unidimensional es en esencia, una colección de datos del mismo tipo almacenados en posiciones contiguas de memoria, según el orden del índice.

Los arreglos tienen que declararse explícitamente, porque así el compilador puede reservar espacio de memoria, para almacenar los elementos. Por lo que estos arreglos se llaman "***arreglos estáticos***". (BECERRA, 1991)

La forma general de declaración de un arreglo unidimensional es:

```
tipo nombre_variable[tamaño];
```

donde:

- **tipo**, declara el tipo base del arreglo, que es el tipo de cada elemento del mismo.

- **tamaño**, indica el número de elementos que tendrá el arreglo.

- **nombre_variable**, es el nombre del arreglo.

Por ejemplo, a continuación se declara un arreglo **uva** con 10 elementos de tipo **float**:

```
float uva[10];
```

Los números empleados para identificar los elementos del arreglo se llaman "subíndices" o simplemente "índices". Estos índices deben ser enteros de constantes, variables o expresiones, y el primer valor de los índices comienza siempre por cero.

Entonces, los elementos del arreglo **uva** anterior serían:

uva[0], primer elemento del arreglo.
uva[1], segundo elemento del arreglo.
..
uva[9], décimo elemento del arreglo.

Cada uno de los elementos del arreglo **uva** debe contener un número en punto flotante. Por ejemplo, como se muestra en las siguientes asignaciones:

uva[5] = 32.56;
uva[6] = 1.2e-8;

Si el arreglo **uva** comienza en la posición de memoria EF00, como se muestra gráficamente en la Figura 8.1.

	uva[0]	uva[1]	uva[2]	uva[3]	uva[4]	uva[5]	uva[6]	uva[7]	uva[8]	uva[9]
Elementos						32.56	1.2e-8			
Dirección	EF00	EF04	EF08	EF0C	EF10	EF14	EF18	EF1C	EF20	EF24

Figura 8.1. Arreglo uva

La declaración de los arreglos puede ser de cualquier tipo de datos, como se muestra a continuación:

```
int patos[22];          /* Un arreglo para almacenar 22 enteros. */
char alfabeto[26];      /* Un arreglo para almacenar 26 caracteres. */
long int gordos[500];   /* Un arreglo para almacenar 500 enteros largos */
```

La cantidad de memoria requerida para almacenar un arreglo está directamente relacionada con su *tipo* y *tamaño*. Entonces, para un arreglo unidimensional, el tamaño total de bytes se puede calcular con la siguiente expresión:

Total_de_bytes = sizeof(tipo) * longitud_del_arreglo

El lenguaje C no comprueba los límites del arreglo, por lo que esos límites pueden ser desbordados en cualquier extremo del arreglo, pudiendo escribirse en alguna otra variable e incluso en el código del programa. Por lo tanto, el programador debe proporcionar una comprobación de los límites del arreglo.

También se puede realizar la declaración de los arreglos con constantes simbólicas. Por ejemplo:

```
#define N 15
```

```
int a[N];
char b[N+1];
float c[N];
```

Ejercicio

Realizar un programa que acepte números enteros como datos de entrada y calcule la *media* y el *valor máximo* de estos datos. El tamaño del arreglo será fijado por el programador, y deberá ser validado previamente en caso de que el número de datos supere el tamaño del arreglo. Además, se debe indicar la finalización de la entrada de los números, para lo cual uno de los métodos consiste en que el usuario ingrese un "valor especial" (**centinela**) que debe ser del mismo tipo que el resto de datos, y que anuncia el fin de la entrada de datos. Pero, el centinela debe ser de distinto valor que los datos ingresados, para evitar confusiones.

```
/* PROG0801.C */

#include "stdio.h"

# define STOP 0                    /* Signo de detención de entrada. */
# define NUM 50

void main ()
{
 int valor[NUM];
 int i, cont = 0, temp, maximo;
 float suma = 0, media;

 printf ("Comience a ingresar datos. <<Enter>> después de cada dato.\n");
 printf ("Teclee 0 para indicar el final de los mismos.\n");
 printf ("El número máximo de datos aceptable es %d.\n\n", NUM);

 printf ("dato %2d : ", cont + 1);
 scanf ("%d", &temp);                           /* Lee un número. */
 while (temp != STOP && cont < NUM) {
  /* Ve si hay que parar y también si queda sitio para el nuevo dato. */
  valor [cont++] = temp;         /* Guarda el dato y actualiza cont. */
  if (cont < NUM){
   printf ("dato %2d : ", cont + 1);
   scanf ("%d", &temp);                 /* Lee el siguiente valor. */
  }
  else
   printf ("\nNo caben más datos. !!!\n");
 }
```

```
  printf ("\nHa introducido: %d datos.\n", cont);
  printf ("Los datos leídos son:\n");
  for (i = 0; i < cont; i++)
    printf ("%-5d", valor[i]);

  /* Calcula la media. */
  for (i = 0; i < cont; i++)
    suma += valor[i];
  media = suma / cont;
  printf ("\n\nEl promedio de los números es: %.2f\n", media);

  /* Determina el número mayor. */
  for (maximo = valor[0], i = 1; i < cont; i++)
    if (valor[i] > maximo)
      maximo = valor[i];
  printf ("El número máximo es: %d\n", maximo);
}
```

Una salida del programa podría ser:

Comience a ingresar datos. <<Enter>> después de cada dato.
Teclee 0 para indicar el final de los mismos.
El número máximo de datos aceptable es 50.

dato 1: 76 <ENTER>
dato 2: 88 <ENTER>
dato 0: <ENTER>

Ha introducido: 2 datos.
Los datos leídos son:
76   88

El promedio de los números es: 82.00
El número máximo es: 88

Resumen. Un arreglo está compuesto por una serie de elementos del mismo tipo, que en su declaración debe indicarse que es una variable arreglo y especificarse el número de sus elementos. Además, el arreglo, lo mismo que cualquier variable, tiene su tipo y su modo de almacenamiento, como se indica a continuación:

- Los arreglos pueden tener los mismos tipos de datos que cualquier variable simple o "escalar".

- Los arreglos tienen los mismos modos de almacenamiento que una variable simple, como se muestra en el siguiente ejemplo:

```
/* Algunas declaraciones de arreglos. */

#inc|ude "stdio.h"

int temp[365];     /* Arreglo externo de 365 enteros. */

void main()
{
    float IIuvia[365];          /* Arreglo automático de 365 float. */
    static char codigo[12]; /* Arreglo static de 12 char. */
    extern int temp[];        /*' Opcional, arreglo de tipo externo. */
}
```

Como se ve, los corchetes **[]** vacíos identifican a la variable **temp** definida como un arreglo **extern**, y para las otras variables el número encerrado entre los corchetes indica cuantos elementos tiene cada arreglo.

8.1.1. Inicialización de Arreglos

Los arreglos se inicializan en el momento de declararlos.

La forma general de inicialización de un arreglo es similar a la de otras variables, así: (SCHILDT, 1994)

```
tipo nombre_arreglo[tamaño] = {lista_valores};
```

donde:

- **Lista_valores**, es una lista de constantes separadas por comas, cuyo tipo es compatible con **tipo**.

La primera constante se coloca en la primera posición del arreglo, la segunda constante se coloca en la segunda posición, y así sucesivamente.

Por ejemplo, para inicializar un arreglo de 6 elementos enteros con los números pares del 2 al 12, se haría:

```
int a[6] = {2, 4, 6, 8, 10, 12};
```

Por lo tanto, el arreglo **a** tendrá almacenado los siguientes datos:

```
a[0] == 2
```

```
a[1] == 4
a[2] == 6
a[3] == 8
a[4] == 10
a[5] == 12
```

Cuando no se inicializan los arreglos con modo de almacenamiento **extern** o **static**, pueden inicializarse automáticamente a 0 para los números, '\0' para los caracteres y NULL para los punteros. Es decir, si no son inicializados los arreglos **extern** y **static** se inicializan automáticamente a cero, mientras que los arreglos **auto** contendrán lo que hubiera en la memoria en donde son almacenados. Por ejemplo, en el siguiente programa:

```
void main()
{
    int auto[2];              /* Arreglo automático. */
    static int estatico[2];  /* Arreglo estático. */

    printf ("%8d %8d\n", auto[1], estatico[1]);
}
```

La salida podría ser:

20533   0

El número de constantes de la lista de inicializaciones puede tener: el mismo número y más pequeño, que el tamaño del arreglo:

a) "***Cuando la lista coincide con el tamaño del arreglo***", las constantes se asignan al arreglo una a una, respectivamente.

Ejercicio

Realizar un programa para inicializar un arreglo con el número de días de cada mes. Luego imprimir cada mes con su respectivo número de días.

```
/* PROG0802.C */

/* Días del mes. */

#include "stdio.h"

int dias[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
void main ()
{
  int indice;
  extern int dias[];                    /* Declaración opcional. */

  for (indice = 0; indice < 12; indice++)
    printf ("El mes %d tiene %d días.\n", indice + 1, dias[indice]);
}
```

La salida de este programa será:

El mes 1 tiene 31 días.
EI mes 2 tiene 28 días.
El mes 3 tiene 31 días.
El mes 4 tiene 30 días.
El mes 5 tiene 31 días.
El mes 6 tiene 30 días.
El mes 7 tiene 31 días.
El mes 8 tiene 31 días.
El mes 9 tiene 30 días.
El mes 10 tiene 31 días.
El mes 11 tiene 30 días.
El mes 12 tiene 31 días.

Este programa se equivoca cada 4 años (años bisiestos), y la declaración opcional: **extern int dias[]**; especifica los **[]** para indicar que **dias** es una variable externa de tipo entero a "*arreglo*".

b) "***Cuando la lista es más corta que el tamaño del arreglo***", se inicializan a 0 los restantes elementos, si el modo de almacenamiento es **static** o **extern**; pero si es **auto** estos elementos contendrán basura.

Ejercicio

Realizar un programa para inicializar un arreglo con el número de días de cada mes, siendo la lista de inicializaciones más corta que el número de meses del año. Luego imprimir cada mes con su respectivo número de días.

```
/* PROG0803.C */

/* Días del mes */

#include "stdio.h"
```

```
int dias[12] = {31, 28, 31, 30, 31, 30};

void main()
{
  int indice;
  extern int dias[];                /* Declaración opcional. */

  for (indice = 0; indice < 12; indice++)
    printf ("El mes %d tiene %d días.\n", indice + 1, dias[indice]);
}
```

Esta vez la salida sería:

El mes 1 tiene 31 días.
El mes 2 tiene 28 días.
El mes 3 tiene 31 días.
El mes 4 tiene 30 días.
El mes 5 tiene 31 días.
El mes 6 tiene 30 días.
El mes 7 tiene 0 días.
EI mes 8 tiene 0 días.
El mes 9 tiene 0 días.
El mes 10 tiene 0 días.
El mes 11 tiene 0 días.
El mes 12 tiene 0 días

NOTA: "***Cuando la lista es demasiado larga que el tamaño del arreglo***", se produce un ERROR, indicándose muchas constantes a inicializar.

8.1.2. Inicialización de Arreglos Indeterminados

Los arreglos indeterminados se denominan también arreglos "*sin tamaño*" o "*no delimitados*". El lenguaje C calcula automáticamente la dimensión de estos arreglos al inicializarlos.

Un "***arreglo indeterminado***" se tiene cuando en la sentencia de declaración de un arreglo no se especifica el tamaño, creándose automáticamente un arreglo para mantener todas las inicializaciones presentes. Es decir, de acuerdo a las constantes inicializadas se reserva el tamaño de memoria para dicho arreglo. (SCHILDT, 1994)

Ejercicio

Realizar un programa para inicializar un arreglo indeterminado con el número de días de cada mes, por lo tanto, se debe determinar automáticamente el tamaño del arreglo. Luego imprimir cada mes con su respectivo número de días.

```
/* PROG0804.C */

/* Días del mes */

#include "stdio.h"

int dias[] = {31, 28, 31, 30};

void main()
{
  int indice;
  extern int dias[];              /* Declaración opcional. */

  for (indice = 0; indice < sizeof dias / sizeof (int); indice++)
    printf ("El mes %d tiene %d días.\n", indice + 1, dias[indice]);
}
```

La salida de este programa será:

EI mes 1 tiene 31 días.
El mes 2 tiene 28 días.
El mes 3 tiene 31 días.
El mes 4 tiene 30 días.,

Hay dos puntos interesantes en este programa.

1. Si se emplea "*arreglos indeterminados*" para inicializar un arreglo, el compilador contará el número de constantes de la lista de inicializaciones, y ese será el tamaño del arreglo.

2. Para determinar automáticamente el tamaño del arreglo, se procede de la siguiente forma: dividir el número total de bytes del arreglo, **sizeof dias**, por el número de bytes del tipo de dato de cada elemento, **sizeof (int)**.

8.1.3. Punteros a Arreglos Unidimensionales

Los programas que utilizan punteros son más eficientes, porque los punteros usan directamente las direcciones de memoria, como usan las instrucciones de hardware. En particular, los punteros hacen más rápido el trabajo que los arreglos. Entonces, la notación con arreglos es un método disfrazado de empleo de punteros. (SCHILDT, 1994)

El nombre de un arreglo sin índice y sin corchetes es un puntero dirigido al primer elemento del arreglo. Por ejemplo, con las siguientes declaraciones:

```
int meses[12];
int *p;
```

se puede generar un puntero al primer elemento, usando el nombre del arreglo **meses**, así:

```
p = meses;
```

Por lo tanto, se cumple que:

```
p == &meses[0]
```

es cierto, y los punteros **meses** y **&meses[0]** representan la dirección de memoria del primer elemento del arreglo.

Además, los punteros **meses** y **&meses[0]** son constantes y no pueden cambiar su valor a lo largo del programa, porque indican dónde comienza a almacenarse el arreglo. Sin embargo, pueden ser asignados a una "variable puntero" como **p**.

El lenguaje C proporciona dos métodos de acceso a elementos de arreglos, que son:

1. Aritmética de punteros.
2. Indexación del arreglo.

Por ejemplo, para acceder a los elementos del arreglo meses mediante índices y punteros, se haría:

INDICES	PUNTEROS		
	Con el arreglo **meses**	Con el puntero **p**	
meses[0]	*meses	*p,	primer elemento
meses[1]	*(meses + 1)	*(p + 1),	segundo elemento
…			
meses[11]	*(meses + 11)	*(p + 11),	duodécimo elemento

En este caso la variable puntero **p** quedará apuntando al primer elemento. Sin embargo, se puede también acceder a los elementos incrementando la variable puntero en un lazo, así:

```
for (i = 0, p = meses; i < 12; f++, p++) {
      printf ("Ingrese el elemento %d: ", i);
      scanf ("%d", p);
}
```

Pero en este caso la variable puntero **p** quedará apuntando a la dirección siguiente del último elemento leído.

Ejercicio

Realizar un programa para inicializar dos arreglos de tipo entero y punto flotante. Luego asignar las direcciones de los dos arreglos a variables puntero, para imprimir las direcciones y contenidos de los elementos de los arreglos usando las variables puntero.

```
/* PROG0805.C */

#include "stdio.h"

void main ()
{
  int fechas[6] = {1, 2, 3, 4, 5, 6};
  int *punt, indice;
  float facturas[6] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
  float *pfloat;

  punt = fechas;          /* Asigna la dirección del arreglo al puntero. */
  pfloat = facturas;
  for (indice = 0; indice < 4; indice++)
    printf ("puntero + %d: %5p,  contenido: %d\n", indice, punt + indice,
          *(punt + indice));
  printf ("\n");
  for (indice = 0; indice < 4; indice++)
    printf ("puntero + %d: %5p,  contenido: %.1f\n", indice, pfloat + indice,
          *(pfloat + indice));
  printf ("\n");
}
```

La salida de este programa podría ser:

puntero + 0: FFEE, contenido: 1
puntero + 1: FFFO, contenido:2
puntero + 2: FFF2, contenido:3
puntero + 3: FFF4, contenido: 4
puntero + 4: FFF4, contenido: 5

puntero + 4: FFF4, contenido: 6

puntero + 0: FFDC, contenido: 1.1
puntero + 1: FFEO, contenido: 2.2
puntero + 2: FFE4, contenido: 3.3
puntero + 3: FFE8, contenido: 4.4
puntero + 4: FFE8, contenido: 5.5
puntero + 4: FFE8, contenido: 6.6

En este ejemplo, la primera línea de cada bloque escribe la dirección del comienzo de cada arreglo, que son los primeros elementos de los arreglos; las siguientes líneas en cada bloque dan el resultado de las direcciones y los contenidos de los siguientes elementos.

Entonces, cuando se accede a los elementos de un arreglo mediante punteros, la dirección cambia al siguiente "elemento", y no al siguiente "byte". Esta es una de las razones por la que se tiene que declarar a qué tipo de elementos se va a referir el puntero, ya que la dirección no es suficiente. Entonces, las siguientes igualdades son ciertas:

```
fechas + 2 == &fechas[2]    /* Misma dirección. */
*(fechas + 2) == fechas[2]   /* Mismo valor. */
```

Estas relaciones resumen la estrecha conexión sobre "arreglos" y "punteros", por lo que se puede usar un puntero para identificar a un elemento de un arreglo y acceder a su valor.

En esencia, son dos formas de nombrar la misma cosa, porque realmente el compilador convierte la notación de arreglos a punteros.

Por otro lado, no es lo mismo ***(fechas+2)** que ***fechas+2**, ya que el operador ***** tiene mayor jerarquía que el **+**, obteniéndose que:

```
*(fechas + 2)   /* Es el valor del tercer elemento de fechas. */
*fechas + 2     /* Suma 2 al valor del primer elemento. */
```

La relación entre "arreglos" y "punteros" permite que se pueda usar ambas notaciones indistintamente al escribir un programa. Un ejemplo típico es cuando se tiene una función con uno de sus parámetros como un arreglo.

8.1.4. Indexación de Punteros

Cualquier variable puntero que tenga la dirección del primer elemento de un arreglo unidimensional, puede indexarse como si estuviera declarada como un arreglo, esto se muestra a continuación:

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

p = a;
p[5] = 100;      /* Asignación usando índice. */
*(p + 5) = 100;  /* Asignación usando aritmética de punteros. */
```

Ambas sentencias de asignación ponen el valor 100 en el sexto elemento del arreglo **a**.

8.1.5. Paso de Arreglos Unidimensionales a Funciones

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

Esto significa que la declaración del parámetro de la función debe ser de un tipo puntero compatible al puntero del arreglo. Por ejemplo, si se tiene el siguiente esqueleto de programa:

```
void main ()
{
    int edad[50];

    convierte (edad);
    /* … */
}
```

Para que una función reciba un arreglo unidimensional, se puede declarar el parámetro de la función de tres formas: (SCHILDT, 1994)

1. Como un arreglo indeterminado.
2. Como un puntero.
3. Como un arreglo delimitado.

A continuación se explica cada una de estas formas, para lo cual el arreglo **edad** debe ser recibido en la función **convierte()**.

1. **Arreglo indeterminado**. Es una versión modificada de la declaración del arreglo; simplemente especifica que se va a recibir en la función la dirección del arreglo de un tipo dado de cierta longitud, así:

```
void convierte (int prima[]) /* prima[] es un puntero a arreglo. */
{
```

```
    /* … */
}
```

El arreglo **edad** tiene 50 elementos, pero **prima** es un puntero al primer elemento del arreglo **edad**.

La declaración del parámetro:

```
int prima[]
```

no crea un arreglo sino un puntero, porque en la llamada a la función:

```
convierte (edad);
```

el argumento **edad** es un "puntero" al primer elemento de los 50 elementos del arreglo; por lo tanto, la llamada de la función pasa un puntero a la función **convierte()**.

2. **Puntero**. La llamada de la función pasa a la misma una dirección que va a ser recibida en un puntero, así:

```
void convierte (int *prima)
{
    /* … */
}
```

Entonces, en la función **convierte()** se puede trabajar en el arreglo en forma indexada o con "aritmética de punteros".

Por lo tanto, son sinónimas las declaraciones de los siguientes parámetros:

```
int prima[]
int *prima
```

ya que ambas definen **prima** como un puntero a un entero. La principal diferencia es que **prima[]** apunta a un arreglo, y la otra es una variable puntero.

3. **Arreglo delimitado**. Consiste en la declaración estándar del arreglo como parámetro de la función, de la siguiente forma:

```
void conviene (int prima[50])
{
    /*  */
}
```

Este método no es aconsejable, porque da la idea de crear un nuevo arreglo **prima** de 50 elementos de tipo **int**, en el que solo se pasa un puntero a un arreglo.

Conclusión. El resultado de los tres métodos de declaración de los parámetros formales es idéntico, porque cada uno le indica al compilador que se va a recibir un puntero a entero. Además, en lo que a la función se refiere, no importa la longitud del arreglo porque el lenguaje C no comprueba sus límites.

NOTA: Cuando se emplea el nombre de un arreglo (que es un puntero) como argumento de una función, ésta usa el puntero real del arreglo que llamó a la función, por lo que se puede realizar cambios en el "*arreglo original*" del programa del cual partió la llamada.

En los anteriores esqueletos de programa, todas las operaciones que afecten a la variable **prima** afectarán también al arreglo **edad** de la función **main()**. Porque la llamada de la función **convìerte()** inicializa **prima** de forma que apunte a **edad[0]**. De esta forma, cambiar el elemento **prima[3]** es lo mismo que cambiar ***(prima+3)**, que a su vez, es lo mismo que cambiar **edad[3]**.

Por ejemplo, realizar dos funciones para escribir una cadena de caracteres en la pantalla, mediante la función **putchar()**: en una función el acceso de los elementos del arreglo se debe realizar con aritmética de punteros, y en la otra en forma indexada.

```
/* Acceso a los elementos del arreglo con aritmética de punteros. */

void putstring (char *c)
{
    while (*c)
        putchar (*c++);
    putchar ('\n');
}


/* Acceso a los elementos del arreglo en forma indexada. */

void putstring (char *c)
{
    register int t;

    for (t = 0; c[t]; ++t)
        putchar (c[t]);
    putchar ('\n');
}
```

Ejercicio

Escribir un programa que contenga una función con un parámetro arreglo para calcular la media de un arreglo de enteros, de la siguiente forma:

- La entrada, debe ser la dirección del arreglo y el número de elementos.
- La salida, es la media que se devuelve a través de la sentencia **return**.

```
/* PROG0806.C */

#include <stdio.h>

#define MAX 50

float media (int arreglo[], int n);

void main ()
{
  int n = 0;
  int a[MAX];

  printf ("Ingrese los datos del arreglo:\n");
  printf ("Termina con un dato 0.\n");
  do {
    printf ("Elemento %2d : ", n);
    scanf ("%d", &a[n]);
  } while (a[n] && ++n < MAX);

  printf ("La media es: %.2f\n", media (a, n));

}

/* Usa índices para calcular la media de un arreglo de n enteros. */

float media (int arreglo[], int n)
{
  int indice;
  long int suma;

  if (n > 0) {
    for (indice = 0, suma = 0; indice < n; indice++)
      suma += arreglo[indice];
    return (float) suma / n;              /* Devuelve un float. */
```

```
 }
 else {
   printf ("No hay arreglo.\n");
   return 0;
 }
}
```

Una salida del programa sería:

Ingrese los datos del arreglo:
Termina con un dato 0.
Elemento 0 : 4 <ENTER>
Elemento 1 : 8 <ENTER>
Elemento 2 : 2<ENTER>
Elemento 3 : 0 <ENTER>
La media es: 4.67

Otra alternativa para implementar la función **media()** del ejercicio inmediato anterior, utilizando punteros, sería:

```
float media (int *arreglo, int n)
{
    int indice;
    long int suma;

    if (n > 0) {
        for (indice = 0, suma = 0; indice < n; indice++)
            suma += arreglo ++;         /* suma += *(arreglo + indice); */
        return (float) suma / n;        /* Devuelve un float. */
    }
    else {
        printf ("No hay arreglo.\n");
        return 0;
    }
}
```

La llamada a la función **media()** no necesita ser cambiada, porque el nombre de un arreglo es un puntero y las siguientes declaraciones de los parámetros formales son idénticas:

```
int arreglo[]
int *arreglo
```

En la Figura 8.2 se tiene una representación gráfica de la memoria que indica cómo se accede al arreglo mediante punteros:

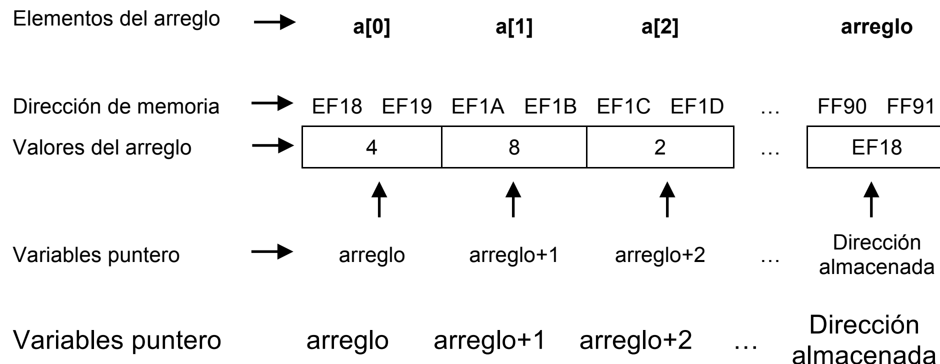


Figura 8.2. Forma de acceder al arreglo mediante punteros

arreglo = a; El puntero **arreglo** contiene la dirección EF18.
***arreglo** Es el valor contenido en la dirección EF18, en este caso 4.

arreglo+1 El puntero **arreglo+1** apunta a la dirección EF1A.
***(arreglo+1)** Es el valor contenido en la dirección EF1A, en este caso 8.

arreglo+2 El puntero **arreglo+2** apunta a la dirección EF1C.
***(arreglo+2)** Es el valor contenido en la dirección EF1C, en este caso 2.

También se puede acceder a los elementos de un arreglo mediante punteros, modificando la dirección del parámetro puntero de la función. Pero en este caso, la variable puntero quedará apuntando a la dirección siguiente del último elemento.

Ejercicio

Realizar un programa que tenga una función para convertir una cadena de caracteres a un número entero. Donde:

- La entrada, debe ser un puntero a cadena de caracteres.
- La salida, tendrá dos parámetros:
- El primero, será un puntero a entero que tomará el valor de la conversión a entero.
- El segundo, será el valor retornado por la función y tomará los siguientes valores:

1 Si la cadena se compone exclusivamente de caracteres numéricos precedidos de un signo menos o más (aunque no necesariamente), convenir la cadena en el valor numérico correcto.

0 Si lo anterior no se cumple, enviar un aviso de error.

```
/* PROG0807.C */

#include <stdio.h>

#define LEN 10      /* Longitud máxima de la cadena de dígitos. */
#define NONUM 0
#define SINUM 1

long convertir (char *string, long *intptr);

void main ()
{
  char string[LEN];
  long valor;

  printf ("Ingrese dígitos numéricos:\n");
  printf ("Termina digitando <<ENTER>>>:\n");
  while (*gets(string)){
    if (convertir (string, &valor))
      printf("Cadena %s convertida a entero: %ld\n", string, valor);
    else
      printf("Cadena %s no se puede convertir a entero\n", string);
    printf ("\nIngrese dígitos numéricos:\n");
    printf ("Termina digitando <<ENTER>>:\n");
  }
}

long convertir(char *string, long *intptr)
{
  int signo = 1;

  *intptr = 0;
  if (*string == '-' || *string == '+') {
    signo = (*string == '-') ? -1 : 1;  /* Pone el signo. */
    string++;
  }
  while (*string >= '0' && *string <= '9') {
    *intptr = 10 * (*intptr) + *string - '0';
    string++;
  }
  if (*string == '\0') {
```

```
    *intptr = signo * (*intptr);
    return SINUM;
  }
  else
    return NONUM;
}
```

Una salida del programa sería:

Ingrese dígitos numéricos:
Termina digitando <<ENTER>>:
123 <ENTER>
Cadena 123 convertida a entero: 123

Ingrese dígitos numéricos:
Termina digitando <<ENTER>>:
12aa <ENTER>
Cadena 12aa no se puede convertir a entero

Ingrese dígitos numéricos:
Termina digitando <<ENTER>>:
<ENTER>

8.2. ARREGLOS BIDIMENSIONALES

Un arreglo bidimensional es un arreglo de arreglos unidimensionales.

Por ejemplo, para declarar un arreglo bidimensional **cuadro** que tiene un arreglo principal de 3 elementos arreglos unidimensionales, que a su vez tienen 4 elementos enteros, se haría:

```
int cuadro[3][4];
```

También se puede visualizar a un arreglo bidimensional como una "***matriz***" con *filas* y *columnas*; el primer índice indica la fila y el segundo indica la columna. En donde, al variar el primer índice, se accede al arreglo a lo largo de una columna, y al variar el segundo índice, se accede por una fila.

Por ejemplo, para acceder al elemento de la fila 1 y columna 2 del arreglo **cuadro** se escribirá así:

```
cuadro[1][2]
```

Entonces, los elementos del arreglo bidimensional **cuadro** serían:

cuadro[0][0] cuadro[0][1] cuadro[0][2] cuadro[0][3]
cuadro[1][0] cuadro[1][1] cuadro[1][2] cuadro[1][3]
cuadro[2][0] cuadro[2][1] cuadro[2][2] cuadro[2][3]

Ejercicio

Realizar un programa que lea el número de filas y columnas de un arreglo bidimensional, que luego lea los elementos del arreglo, y por último, imprima los elementos en forma matricial fila por fila.

```
/* PROG0808.C */

#include "stdio.h"

#define MAX_FILA 100
#define MAX_COLUMNA 100

void main ()
{
  int a[MAX_FILA][MAX_COLUMNA];
  int i, j, fila, columna;

  do {
    printf ("Ingrese el número de filas   : ");
    scanf ("%d", &fila);
  } while (fila <= 0 || fila > MAX_FILA);

  do {
    printf ("Ingrese el número de columnas: ");
    scanf ("%d", &columna);
  } while (columna <= 0 || columna > MAX_COLUMNA);

  /* Ingreso de elementos. */

  printf ("\nIngreso de elementos:\n");
  for (i = 0; i < fila; i++)
    for (j = 0; j < columna; j++) {
      printf ("a[%2d, %2d] = ", i, j);
      scanf ("%d", &a[i][j]);
    }

  /* Imprimir en forma matricial. */
```

```
  printf ("\nElementos de la matriz:\n");
  for (i = 0; i < fila; i++) {
    for (j = 0; j < columna; j++)
      printf ("%5d", a[i][j]);
    printf ("\n");
  }
}
```

Una salida de este programa sería:

Ingrese el número de filas : 2 <ENTER>
Ingrese el número de columnas: 4 <ENTER>

Ingreso de elementos:
a[0, 0] = 1 <ENTER>
a[0, 1] = 2 <ENTER>
a[0, 2] = 3 <ENTER>
a[0, 3] = 4 <ENTER>
a[1, 0] = 5 <ENTER>
a[1, 1] = 6 <ENTER>
a[1, 2] = 7<ENTER>
a[1, 3] = 8 <ENTER>

Elementos de la matriz:
1 2 3 4
5 6 7 8

En la Figura 8.3 se representa gráficamente el arreglo **a**.

COLUMNAS

FILAS	**[0]**	**[1]**	**[2]**	**[3]**
[0]	1	2	3	4
[1]	5	6	7	8

***Figura 8.3. Arreglo* a**

8.2.1. Inicialización de un Arreglo Bidimensional

La inicialización de un arreglo bidimensional se realiza colocando las constantes de cada fila entre llaves, a su vez las filas están encerradas entre otro par de llaves más externas. Es decir, como las filas de un arreglo bidimensional son arreglos unidimensionales, las constantes dentro del primer par de llaves se asignan a la primera fila del **arreglo**, las del segundo par a la segunda fila, y así sucesivamente. (SCHILDT, 1994)

Las reglas que controlan el número de constantes cuando no coincide con el tamaño del arreglo, son las mismas que para arreglos unidimensionales, así:

1) Si hay "*demasiadas*" constantes se produce un error, y las que restan no servirán para inicializar la siguiente fila.

2) Si dentro de un par de llaves "*faltan*" constantes y el arreglo es **static** o **extern**, los elementos son inicializados por defecto al valor 0, y a cualquier valor si el arreglo es **auto**. Por ejemplo, en la declaración:

```
static int matriz[2][3] = {
     {5, 6},
     {7, 8}
}
```

se almacenará en el arreglo **matriz** de la siguiente forma:

```
5  6  0
7  8  0
```

Al colocar solo el par de llaves externas sin colocar las internas, y el número de constantes es igual a la dimensión del arreglo, la inicialización es correcta. Por el contrario, si se hubiera colocado menos constante, el arreglo se llena sin considerar la distribución en filas, terminando la inicialización del resto de elementos con ceros o basura. Por ejemplo, en la declaración:

```
static int matriz[2][3] = {5, 6, 7, 8};
```

se almacenará en el arreglo **matriz** de la siguiente forma:

```
5  6  7
8  0  0
```

Las reglas para inicializar un arreglo unidimensional cumplen para un arreglo bidimensional.

NOTA: Las reglas para inicializar un arreglo unidimensional cumple para un arreglo bidimensional.

Ejercicio

Realizar un programa meteorológico, el cual imprimirá la lluvia total caída cada año, la media anual y la media correspondiente a cada mes del año.

Para hallar la lluvia caída cada año se tiene que sumar los datos de cada fila, y para la media de lluvia de un determinado mes, sumar los datos de su columna correspondiente.

```
/* PROG0809.C */

/* Calcula totales anuales, promedio anual y promedio mensual de datos */
/* pluviométricos en un período determinado.                           */

#define DOCE 12      /* Número de meses del año. */
#define ANS 5        /* Número de años a tratar. */

#include "stdio.h"

void main()
{
  static float lluvia [ANS][DOCE] = {
    {8.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.2, 6.1, 7.4},
    {9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 5.2},
    {6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 4.2},
    {4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2, 0.4, 3.4, 3.5, 6.6},
    {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.2}
  };

  /* Inicialización de datos de lluvia en el período 1990-1994.        */

  int anio, mes;
  float subtotal, total;

  printf ("  AÑO       LLUVIA (cm.)\n\n");
  for (anio = 0, total = 0; anio < ANS; anio++) {
    /* En cada año suma la lluvia de todos los meses.                */
    for (mes = 0, subtotal = 0; mes < DOCE; mes++)
      subtotal += lluvia[anio][mes];
    printf ("%5d %12.1f\n", 1990 + anio, subtotal);
    total += subtotal;          /* Calcula lluvia total del período. */
  }

  printf ("\nEl promedio anual ha sido %.1f cm.\n\n",total / ANS);

  printf ("PROMEDIOS MENSUALES: \n\n");
  printf ("Ene  Feb  Mar  Abr  May  Jun  Jul  Ago  Sep  Oct  Nov  Dic\n");
  for (mes = 0; mes < DOCE; mes++) {
```

```
   /* Suma la lluvia de todos los a¤os en cada mes.                    */
   for (anio = 0, subtotal = 0; anio < ANS; anio++)
   subtotal += lluvia[anio][mes];
   printf ("%-5.1f", subtotal / ANS);
  }
  printf ("\n");
}
```

Una salida del programa sería:

AÑO LLUVIA (cm.)
1990 48.5
1991 41.9
1992 28.6
1993 32.2
1994 37.8

El promedio anual ha sido 38.0 cm.

PROMEDIOS MENSUALES

Ene	*Feb*	*Mar*	*Abr*	*May*	*Jun*	*Jul*	*Ago*	*Sep*	*Oct*	*Nov*	*Dic*
7.4	*7.2*	*4.1*	*3.0*	*2.1*	*0.8*	*1.2*	*0.3*	*0.5*	*1.9*	*3.6*	*6.1*

8.2.2. Inicialización de Arreglos Indeterminados Bidimensionales

En los arreglos indeterminados bidimensionales las filas son indeterminadas, omitiendo el índice de las filas, porque el arreglo bidimensional se almacena por filas, por lo que se debe especificar el número de elementos por cada fila, que corresponde al número de columnas del arreglo. Esto sirve para construir tablas de longitudes variables, ya que el compilador asignará automáticamente el espacio necesario para las constantes a ser inicializadas. Por ejemplo, para crear un arreglo de 2 columnas: la primera columna es un número entero y la segunda columna su cuadrado, se declararía así:

```
int cuad[][2] = {
     1, 1,
     2, 4,
     3, 9,
     4, 16
};
```

La ventaja de esta declaración es que la tabla puede alargarse o acortarse sin cambiar la dimensión del arreglo.

8.2.3. Punteros a Arreglos Bidimensionales

El nombre de un arreglo sin índices es un puntero a puntero dirigido al primer elemento del arreglo. Por ejemplo, con las siguientes declaraciones:

```
int caucho[4][2];   /* Arreglo int de 4 filas y 2 columnas. */
int *punt;          /* Puntero a entero. */
```

sólo para algunos compiladores de lenguaje C se puede generar un puntero que apunte al elemento de la primera fila y primera columna, usando el nombre del arreglo **caucho**, así:

```
punt = caucho;
```

Pero esto no es muy adecuado, porque se está asignando un "puntero a puntero" en un "puntero a entero", por consiguiente la mejor forma sería:

```
punt = &caucho[0][0];
```

Los elementos del arreglo bidimensional se almacenan en la memoria por filas, como arreglos de una dimensión, el orden de ellos se determina variando primero el índice de las columnas y luego el de las filas. En el anterior ejemplo se tendría:

```
caucho[0][0] caucho[0][1] caucho[1][0] caucho[1][1] …
```

Entonces, primero se coloca la primera fila, luego la segunda, la tercera, y así sucesivamente. Por tanto, las direcciones de cada elemento serán:

```
punt   == &caucho[0][0]   /* Fila 0, columna 0 */
punt+1 == &caucho[0][1]   /* Fila 0, columna 1 */
punt+2 == &caucho[1][0]   /* Fila 1, columna 0 */
punt+3 == &caucho[1][1]   /* Fila 1, columna 1 */
```

Teniendo en cuenta que: un arreglo bidimensional es un arreglo de punteros, que cada uno apunta a un arreglo unidimensional correspondiente a cada fila, y que el nombre de un arreglo es un puntero al primer elemento, se obtienen las direcciones a cada fila eliminando el segundo índice del arreglo bidimensional, así:

```
Primera fila:  caucho[0] == &caucho[0][0]
Segunda fila:  caucho[1] == &caucho[1][0]
Tercera fila:  caucho[2] == &caucho[2][0]
Cuarta fila:   caucho[3] == &caucho[3][0]
```

Esto permite que las funciones diseñadas para trabajar con arreglos unidimensionales, se puedan usar con arreglos bidimensionales.

En general, un arreglo **n** dimensional puede reducirse a arreglos **(n-1)** dimensionales. Estos nuevos arreglos pueden reducirse de nuevo utilizando el mismo método, el proceso termina cuando se produce un arreglo unidimensional.

Ejercicio

Realizar un programa para calcular el valor medio de los elementos de cada fila de una matriz. Usar la función **media()** que calcule la media de un arreglo unidimensional de enteros.

```
/* PROG0810.C */

#include <stdio.h>

/* Función unidimensional, arreglo bidimensional.              */

float media (int arreglo[], int n);

void main ()
{
  static int cosas [3][4] = {
    {2, 4, 6, 8},
    {100, 200, 300, 400},
    {10, 20, 60, 90}
  };
  int fila;

  for (fila = 0; fila < 3; fila++)
    printf ("La media de la fila %d es: %.2f\n", fila,
            media (cosas[fila], 4));

  /* cosas[fila] es un puntero a cada fila que es arreglo */
  /* unidimensional de 4 elementos.                      */
}

/* Calcula la media de un arreglo unidimensional. */

float media (int arreglo[], int n)
{
  int indice;
```

```
  long suma;

  if (n > 0) {
    for (indice = 0, suma = 0; indice < n; indice++)
      suma += (long) arreglo[indice];
    return (float) suma / n;
  }
  else {
    printf ("No hay arreglo.\n");
    return 0;
  }
}
```

La salida del programa será:

La media de la fila 0 es: 5.00
La media de la fila 1 es: 250.00
La media de la fila 2 es: 50.00

Al arreglo **cosas** se lo ha tratado por filas o como arreglos unidimensionales.

8.2.4. Indexación de Punteros en Arreglos Bidimensionales

El acceso a cualquier elemento de un arreglo bidimensional, se puede realizar de dos maneras: (SCHILDT, 1994)

1. ***Mediante indexación de un arreglo***. Para acceder al elemento de la fila **i** y columna **j** del arreglo bidimensional **a**, se utiliza los índices, así:

    a[i][j]

2. ***Mediante un puntero***. Para acceder al elemento de la fila **i** y columna **j** del arreglo bidimensional **a**, se tendría que:

    a[i][j] es equivalente a: *(*a + (i * longitud_fila) + j)

Por ejemplo, si se tiene la siguiente declaración del arreglo **a**:

int a[10][10];

el acceso al elemento de la fila 1 y columna 2 se haría de una de las dos maneras anteriores:

1) Mediante indexación de un arreglo: a[1][2]

2) Mediante un puntero: *(*a + 12)

Por último, no es posible aplicar la indexación a los arreglos bidimensionales, mediante un puntero con índices, existiendo una forma especial que se verá al final del tema "Arreglos asignados dinámicamente".

Por consiguiente para acceder al elemento de la fila **i** columna **j** del arreglo **a**, mediante un puntero **p**, teniendo las siguientes declaraciones:

```
int a[10][10];
int *p;
p = &a[0l[0];
```

se haría:

```
*(p + (i * 10) + j)
```

Ejercicio

Realizar un programa para leer líneas de texto hasta encontrar una línea nula y luego imprimir cada línea.

```
/* PROG0811.C */

#include "stdio.h"

#define MAX 100
#define LONG 80

void main (void)
{
 char texto[MAX][LONG];
 register int t, i;

 printf ("Introduzca una línea nula para terminar.\n");
 printf ("línea | texto\n");
 for (t = 0; t < MAX; t++) {
  printf ("%-5d | ", t);
  gets (texto[t]);
  if (!*texto[t])
   break;        /* Termina si línea es nula. */
 }
 printf ("\nTEXTO INGRESADO.\n\n");
```

```
 for (i = 0; i < t; i++)
   puts (texto[i]);
}
```

Una salida del programa sería:

Introduzca una línea nula para terminar.
línea | texto
0 | Luis Manuel <ENTER>
1 | Víctor Hugo <ENTER>
2 | Ana María <ENTER>
3 | <ENTER>

TEXTO INGRESADO.

Luis Manuel
Víctor Hugo
Ana María

8.2.5. Paso de Arreglos Bidimensionales a Funciones

Cuando se utiliza un arreglo bidimensional como argumento de una función, solo se pasa un puntero a puntero al primer elemento. Sin embargo, la función que recibe la dirección del arreglo bidimensional como parámetro, tiene que definir al menos la longitud de la segunda dimensión, para poderlo usar como matriz. (SCHILDT, 1994)

Esto se debe a que el compilador de lenguaje C necesita saber la longitud de cada fila para indexar el arreglo correctamente en notación matricial, porque sería imposible saber dónde empieza cada fila, como se muestra en el siguiente ejemplo de esqueleto de programa:

```
void cajon (int cosas[][4]);

void main ()
{
    static int cosas[3][4] = {
        {2, 4, 6, 8},
        {100, 200, 300, 400},
        {10, 20, 60, 90}
    };

    cajon (cosas);
    {
        /* … */
```

```
        }
    }
```

El parámetro de la función debe ser declarado como puntero a un arreglo indeterminado, para indicarle al compilador que los elementos están agrupados en filas de 4 columnas. Se puede especificar la primera dimensión del parámetro de la función, pero no es necesario.

No se puede declarar el parámetro de la función como puntero a puntero, pero si se declarara correctamente, en la función el arreglo bidimensional se accede mediante la teoría antes explicada.

Ejercicio

Realizar un programa para almacenar las notas de tres materias para cada estudiante, teniéndose un máximo de 50 estudiantes, sacar el promedio de cada estudiante e imprimir un reporte ordenado alfabéticamente como el siguiente:

		LISTA DE NOTAS			
ORD	NOMBRE	MATERIA1	MATERIA2	MATERIA3	PROMEDIO

```
/* PROG0812.C */

/* Base de datos de notas de alumnos. */

#include "stdio.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"
#include "conio.h"

#define MATERIAS 3
#define ALUMNOS  50
#define TAM      31

/* Prototios de las funciones. */

void ingreso_notas (char nom[][TAM], int notas[][MATERIAS] ,int *numero);
void mostrar_notas (char nom[][TAM], int notas[][MATERIAS], int numero);
void ordenar (char nom[][TAM], int notas[][MATERIAS] ,int numero);
void cambio (int x[], int y[]);
obt_nota (int num);

void main ()
```

```
{
 char nombres[ALUMNOS][TAM];
 int notas[ALUMNOS][MATERIAS];
 char ch;
 static int numero; /* Toma 0, al escoger inicialmente la opción M. */

 for (;;) {
  printf ("(I)ntroducir notas\n");
  printf ("(M)ostrar notas\n");
  printf ("(T)erminar\n");
  printf ("Ingrese la opción...");

  do {
   ch  = toupper (getch());

   /* La función toupper() convierte una letra a mayúscula .*/
  } while (ch != 'I' && ch != 'M' && ch != 'T');
  printf ("\n\n");

  switch (ch) {
   case 'I' : ingreso_notas (nombres, notas, &numero);
          break;
   case 'M' : mostrar_notas (nombres, notas, numero);
          break;
   case 'T' : exit (0);
  }
 }
}

/* Introducir las notas de los alumnos. */

void ingreso_notas (char nom[][TAM],  int notas[][MATERIAS], int *numero)
{
 int i, t;

  do {
   printf ("Introduzca el número de alumnos: ");
   scanf ("%d", numero);
  } while (*numero <= 0 || *numero > ALUMNOS);

  for (t = 0; t < *numero; t++) {
   fflush (stdin);
   printf ("\nIngrese el nombre : ");
   gets (nom[t]);   /* nom[t] es la dirección de cada fila. */
```

```
    printf ("\nAlumno: %s\n", nom[t]);
    for (i = 0; i < MATERIAS; ++i)
      notas[t][i] = obt_nota (i);

  }
}

/* Mostrar notas. */

void mostrar_notas (char nom[][TAM], int notas[][MATERIAS], int numero)
{
  int t, i;
  float prom;

  ordenar (nom, notas, numero);
  puts ("                              LISTA DE NOTAS");
  printf ("ORD NOMBRE                      MATERIA 1  MATERIA 2  MATERIA 3");
  puts ("   PROMEDIO\n");
  for (t = 0; t < numero; t++) {
    prom = 0;
    printf (" %2d %-30s ", t+1, nom[t]);
    for (i = 0; i < MATERIAS; i++) {
      prom += notas [t][i];
      printf ("%5d      ", notas[t][i]);
    }
    printf ("%5.2f\n", prom / numero);
  }
  printf ("\nDigite una tecla para continuar...");
  getch();
}

/* Ordena los nonbres y notas. */

void ordenar (char nom[][TAM], int notas[][MATERIAS] ,int numero)
{
  int i,  j, t;
  int aux[MATERIAS];
  char cad[TAM];
  for (i = 0; i < numero - 1; i++)
    for (j = i + 1; j < numero; j++)
      /* La funci¢n strcmp() compara dos cadena. */
      if (strcmp (nom[j], nom[i]) < 0) {
```

```
        /* Cambio de nombres. */
        /* La función strcpy() copia la segunda cadena a la primera. */
        strcpy (cad, nom[i]);
        strcpy (nom[i], nom[j]);
        strcpy (nom[j], cad);

        /* Cambio de notas de cada alumno. */
        cambio (aux, notas[i]);
        cambio (notas[i], notas[j]);
        cambio (notas[j], aux);
      }
}

/* Cambio de las notas. */

void cambio (int x[], int y[])
{
  int t;

  for (t = 0; t < MATERIAS; t++)
    x[t] = y[t];
}

/* Leer una nota. */

obt_nota (int i)
{
  char cad[80];

  printf ("Introduzca la nota %d: ", i + 1);
  gets (cad);
  return (atoi (cad));
}
```

La salida del programa va a depender de la opción del menú que se escoja: **I** para ingresar la información, **M** para mostrar la información y **T** para terminar el programa.

8.3. ARREGLOS DE PUNTEROS

Los punteros pueden estructurarse en arreglos como cualquier otro tipo de dato, es decir cada elemento del arreglo es un puntero.

La declaración para un arreglo de punteros es:

tipo *nombre[tamaño];

Por ejemplo, para declarar un arreglo de 10 punteros a enteros, se haría:

int *x[10];

En la Figura 8.4 se muestra gráficamente el arreglo **x**.

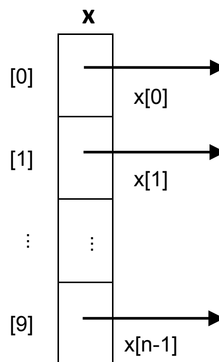


Figura 8.4. Arreglo x

Se ha obtenido 10 punteros, pero no existe memoria reservada para almacenar la información correspondiente a cada puntero. La forma más adecuada para reservar memoria es mediante la asignación dinámica que se verá en el siguiente tema. Por ejemplo, si se desea asignar la dirección de una variable entera llamada **var** al tercer elemento del arreglo de punteros **x**, se haría:

x[2] = &var;

y el acceso al valor de la variable **var** mediante el puntero **x[2]** sería ***x[2]**. Por ejemplo, si se desea almacenar el valor 15 en la variable **var** mediante el puntero **x[2]**, se haría:

*x[2] = 15;

8.3.1. Paso de un Arreglo de Punteros a una Función

Es el mismo método utilizado para otros arreglos, que consiste en llamar a la función con la dirección al primer elemento del arreglo, siendo esta dirección el nombre del arreglo sin índices.

Por ejemplo, a continuación se muestra la llamada y la implementación de la función **mostrar_arreglo()**, que imprime el contenido de las direcciones del anterior arreglo **x**:

- Llamada a la función:
    mostrar_arreglo (x); /* x es un puntero a un arreglo de punteros. */

- Implementación de la función:

```
void mostrar_arreglo (int *p[]) /* p es un arreglo de punteros. */
{
    int t;

    for (t = 0; t < 10; t++)
        printf ("%5d", *p[t]);
}
```

donde:

- **x**, es un puntero a un arreglo de punteros a enteros. Por lo que se debe declarar el parámetro **p** como un arreglo de punteros a enteros. También se puede declarar el parámetro de la función como un puntero a puntero así: **int **p**.

Para poder utilizar estas dos declaraciones (parámetros de la función), hay que crear el espacio de memoria dónde se almacenen los datos, ya que solo han sido declarados los punteros. Por lo tanto, se puede crear la memoria de dos formas:

1. Con cualquier función estándar de asignación dinámica de memoria.
2. Inicializando el arreglo de punteros (solo para cadenas).

NOTA: No se puede utilizar como parámetro de función un arreglo de punteros, cuando en la llamada a una función se envía como argumento la dirección de un arreglo bidimensional; aunque sea un puntero a puntero.

8.4. ARREGLOS ASIGNADOS DINÁMICAMENTE

Se asigna memoria dinámicamente mediante la función estándar **malloc()**, para trabajar con esa memoria como si fuera un arreglo.

La función **malloc()** devuelve un "*puntero genérico*" al primer byte de una región de memoria de la sección "montón", del tamaño en bytes determinado en el argumento. Si no existe suficiente memoria libre en el montón para satisfacer la petición, devuelve un puntero nulo, por lo que se debe verificar esta condición. Es decir, la comprobación de la función **malloc()** que devuelva un puntero válido, asegura que la petición de memoria ha tenido éxito, siendo necesario para prevenir un uso accidental de un puntero nulo. (SCHILDT, 1994)

El prototipo de la función **malloc()** está definida en la librería "***stdIib.h***", y es:

```
void *malloc (size_t tamaño);
```

donde:

- **tamaño**, es el número de bytes a reservar en el montón.

Al crear memoria mediante asignación dinámica, el "*puntero genérico*" apuntado al primer byte que devuelve la función, puede ser indexado como si fuera un arreglo de una sola dimensión.

Se debe utilizar el operador **sizeof** para especificar el tamaño del tipo de dato de cada elemento, garantizando la portabilidad de los programas a computadoras con tipos de tamaños diferentes. Por ejemplo, si se quiere reservar espacio de memoria para 10 elementos de tipo entero se haría:

```
int *p;
p = malloc (10 * sizeof (int));
```

Además, se debe garantizar la conversión del puntero obtenido por la función **maIloc()**, **void ***, a un puntero de cualquier tipo, mediante un "*casting*". En este caso para obtener un puntero de tipo entero, sería:

```
p = (int *) malloc (10 * sizeof (int));
```

Por último, para validar que el puntero obtenido de la función **malloc()** sea diferente de NULL, cuando no exista memoria suficiente para ser reservada, se haría:

```
if ((p = (int *) malloc (10 * sizeof (int))) == NULL) {
    printf ("ERROR, no existe suficiente memoria.\n");
    exit (1);
}
```

Ejercicio

Realizar un programa para obtener espacio de memoria dinámicamente, en la cual se almacene una cadena, luego ingresar una cadena e imprimirla al revés.

```
/* PROG0813.C */

#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void main ()
{
  char *ch_ptr;
  register int t;
```

```
  ch_ptr = (char *) malloc (80*sizeof(char));
  if (!ch_ptr) {        /* Verifica si se reservó memoria. */
    printf ("Fallo de petición de memoria.\n");
    exit (1);
  }

  printf ("Ingrese la cadena:\n");
  gets (ch_ptr);

  for (t = strlen (ch_ptr) - 1; t >= 0; t--)
    putchar (ch_ptr[t]);  /* putchar (*(ch_ptr + t)); */
  putchar ('\n');
}
```

Una salida del programa sería:

Ingrese la cadena:
Este es un ejemplo <ENTER>
olpmeje nu se etsE

Por otro lado, si en el programa no se han definido las dimensiones de un arreglo bidimensional, no se puede indexar directamente el puntero como si fuera un arreglo bidimensional. Para indexarlo se debe hacer uso de un truco: pasar el puntero como argumento a una función, de esta forma la función puede definir las dimensiones del arreglo bidimensional en el parámetro que recibe al puntero, lo que permitirá una indexación normal. (Esto es posible solo para lenguaje C)

Por ejemplo, si se reserva 20 elementos enteros:

```
int *p = (int *) malloc (20 * sizeof (int));
```

- La llamada a la función **manipuIar()** sería:

```
manipular (p);
```

- La cabecera de la función **manipular()** para usar el puntero como matriz de 5 elementos por fila (4 filas) sería:

```
void manipular (int mat[][5])
{
    /* … */
}
```

Ejercicio

Realizar un programa que construya una tabla de los números del 1 al 10 elevados a sus potencias: primera, segunda, tercera y cuarta, para lo cual en el programa principal se debe reservar memoria con la función **malloc()**. Luego en funciones independientes construir e imprimir la tabla pedida.

```
/* PROG0814.C */

/* Mostrar las potencias del 1 al 10. */

#include "stdio.h"
#include "stdlib.h"

int pot (int a, int b);
void tabla (int p[][4]);
void mostrar (int p[][4]);

void main ()
{
  int *p;

  p = (int *) malloc (40 * sizeof (int));
  if (!p) {
    printf ("Fallo de petición de memoria\n");
    exit (1);
  }

  /* p es simplementado como puntero. */
  tabla (p);
  mostrar (p);
}

/* Construir la tabla de potencias. */

void tabla (int p[][4])
{
  /* Ahora p trabaja como arreglo. */
  register int i, j;

  for (i = 1; i <= 10; i++)
    for (j = 1; j <= 4; j++)
      p[i - 1][j - 1] = pot (i, j);   /* p[i - 1][j - 1] = (int) pow (i, j); */
}
```

```
/* Mostrar la tabla de potencias. */

void mostrar (int p[][4])
{
  /* Ahora p trabaja como arreglo. */
  register int i, j;

  printf ("%10s %10s %10s %10s\n\n", "N", "N^2", "N^3", "N^4");
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 4; j++)
      printf ("%10d ", p[i][j]);
    printf ("\n");
  }
}

/* Elevar un número a la potencia especificada. */

/* Esta función se puede realizar con la función de librería pow(), */
/* que se encuentra en la librería "math.h".                        */
int pot (int a, int b)
{
  register int t = 1;

  for (; b; b--)
    t *= a;
  return t;
}
```

La salida de este programa será:

N	*N^2*	*N^3*	*N^4*
1	*1*	*1*	*1*
2	*4*	*8*	*16*
3	*9*	*27*	*81*
4	*16*	*64*	*256*
5	*25*	*125*	*625*
6	*36*	*216*	*1296*
7	*49*	*343*	*2401*
8	*64*	*512*	*4096*
9	*81*	*729*	*6561*
10	*100*	*1000*	*10000*

En este ejercicio se presenta un ***problema***: el puntero **p** que se pasa como argumento a las funciones es un puntero a entero, y el parámetro de las funciones que recibe al puntero es un puntero a puntero del arreglo bidimensional **p[][4].**

Por lo tanto, la alternativa más eficiente para obtener un arreglo dinámico bidimensional, sería declarando un puntero a puntero. Por ejemplo, para obtener el arreglo de elementos enteros **p** de **n** filas y **m** columnas, se declararía así:

```
int **p;
```

luego se debe crear el arreglo de punteros con la sentencia:

```
p = (int **) malloc (n * sizeof (int*));
```

donde:

- **int***, es el tamaño en bytes de cada puntero.

- **int****, es la conversión puntero a puntero a tipo int.

A continuación se debe crear la memoria para cada puntero del arreglo de punteros, con las sentencias:

```
for (i = 0; i < n; i++)
    p[i] = (int *) malloc (m * sizeof (int));
```

En la figura 8.5 se representa gráficamente el arreglo creado dinámicamente.

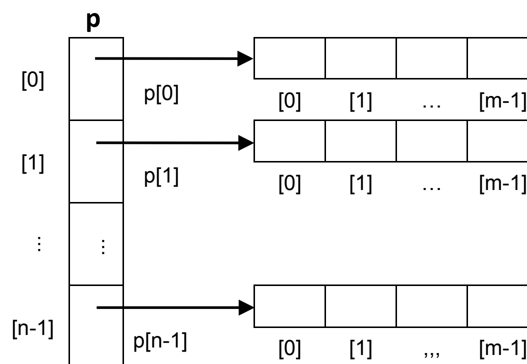


Figura 8.5. Arreglo* p *creado dinámicamente

Y para acceder a cada elemento del arreglo se lo hace en forma indexada, así: **p[i][j]**.

NOTA: La aplicación concreta de la asignación dinámica en arreglos se utiliza en arreglos a punteros.

El **problema** presentado en el ejercicio PROG0814.C se resuelve cambiando la declaración de **p** y la asignación dinámica, como se muestra a continuación:

```
int **p, i;

p = (int **) malloc (10 * sizeof (int*));

if (!p) {
    printf ("Fallo de petición de memoria\n");
    exit (1);
}

for (i = 0; i < 10; i++) {
    p[i] = (int *) malloc (4 * sizeof (int));
    if (!P[i]) {
        printf ("Fallo de petición de memoria\n");
        exit (1);
    }
}
```

Con estos cambios se tendría que escribir:

- Para la llamada a la función **tabla()**:

```
tabla (p);
```

- Para la implementación de la función **tabla()**:

```
void tabla (int *p[])
{
    /* … */
}
```

Liberación de Memoria: Función *free()*

La función **free()** es la opuesta de la función **maIloc()**, es decir, devuelve al sistema la memoria previamente asignada por **malloc()**. Una vez que la memoria ha sido liberada, puede ser reutilizada posteriormente llamando a **malloc()**. Esta función **free()** se encuentra disponible en el archivo de cabecera "***stdlib.h***", y su prototipo es:

```
void free (void *p);
```

donde:

- **p**, es la dirección del primer byte de la memoria a ser liberada, la cual fue asignada previamente por **malloc()**.

Nunca debe llamarse a **free()** con un argumento inválido (NULL), porque se destruirá la lista de memoria libre.

8.5. ARREGLOS MULTIDIMENSIONALES

El lenguaje C permite arreglos de más de dos dimensiones, siendo el límite exacto determinado por el compilador.

La forma general de la declaración de un arreglo multidimensional es:

tipo nombre[a] [b] [c]...[z];

Todo lo expuesto para los arreglos bidimensionales pueden extenderse a los de mayor número de dimensiones. Por ejemplo, un arreglo de tres dimensiones se declarará de la siguiente forma:

int salida[10] [20][30];

Este arreglo puede visualizarse como 10 matrices de 20*30 apiladas unas sobre otras. También puede considerase como un **arreglo** de ***arreglos*** de ***arreglos***, o sea, como un arreglo de 10 elementos cada uno de los cuales es un arreglo de 20 elementos, que a su vez son arreglos de 30 enteros.

La ventaja de este método es que se puede extender fácilmente a un mayor número de dimensiones, pero no es práctico.

Los arreglos de tres o más dimensiones no se utilizan a menudo, por la cantidad de memoria alta que se necesita para almacenarlos. Pues el almacenamiento requerido aumenta exponencialmente con el número de dimensiones, siendo el acceso más lento que en un arreglo unidimensional.

Cuando se pasan arreglos multidimensionales a funciones, se tienen que declarar todas las dimensiones excepto la primera dimensión. Pero, si se desea, puede incluirse la primera dimensión. (SCHILDT, 1994)

Por ejemplo, un arreglo **m** de cuatro dimensiones 3x4x5x6, se declararía así:

int m[3][4][5][6];

Una función que reciba al arreglo **m** podría ser:

```
func (int d[][4][5][6])
{
    /* … */
}
```

En forma similar, si se inicializan arreglos indeterminados multidimensionales, se deben especificar todas las dimensiones, excepto la localizada más a la izquierda, para permitir al compilador de lenguaje C indexar el arreglo adecuadamente.

PROBLEMAS PROPUESTOS

1) Elaborar un programa que sume y escriba el contenido de los primeros **n** elementos de un arreglo ingresados desde el teclado. El dato **n** debe ser leído desde el teclado y ser menor a 300.

2) Escribir un programa que lea **n** números y los almacene en un arreglo **a**, y que luego invierta su contenido, es decir: el contenido del elemento 0 debe intercambiarse con el contenido del elemento **n-1**, el contenido del elemento 1 debe intercambiarse con el contenido del elemento **n-2**, y así sucesivamente. El valor leído de **n** debe ser menor a 200

3) Escribir un programa que inicialice un arreglo con 99 respuestas de una encuesta, siendo cada una de ellas un número entre 1 y 9. El programa debe calcular el promedio aritmético, el valor medio y el valor que ocurre con mayor frecuencia de los 99 valores, cada uno de estos cálculos deben realizarse en funciones diferentes.

   El valor medio se determina ordenando el arreglo de respuestas en orden ascendente y seleccionando el elemento medio del arreglo ya ordenado. Si existe un número par de elementos en el arreglo, el valor medio deberá ser calculado como el promedio de los dos elementos medios del arreglo. Los valores que ocurran con la misma frecuencia deben ser indicados. Además, el programa debe tener una función para imprimir el arreglo.

4) Escribir un programa que simule el lanzamiento de dos dados, utilizando la función estándar **rand()** en el lanzamiento de cada dado. La suma de los dos valores variará desde 2 hasta 12, siendo 7 la suma más frecuente, y 2 y 12 las menos frecuentes.

   Además, utilizar un arreglo para llevar la cuenta del número de veces que aparece cada suma posible, luego imprimir los resultados en un formato tabular.

   También determinar si los totales son exitosos, es decir cuando se tiene un 7 (existiendo seis formas de hacerlo, por lo que, aproximadamente una sexta parte de todas las lanzadas deberán ser 7).

5) Escribir un programa que almacene el arreglo **numeros** de 10 elementos con enteros al azar desde 1 hasta 1000. Para cada uno de los elementos, imprimir el valor y el total acumulado del número de caracteres impresos, para lo cual se debe utilizar el especificador de formato **%n**. Además, imprimir el número total de caracteres para todos los elementos, incluyendo el valor actual, cada vez que éste sea impreso. La salida deberá tener el formato siguiente:

Total acumulado	Caracteres
342	3
1000	7
963	10
6	11

Total: 11

6) Una empresa le paga por comisión a su personal de ventas. Los vendedores reciben una cantidad entera de 200 dólares por semana más el 9% de sus ventas brutas de dicha semana. Por ejemplo, un vendedor que vende 3000 dólares en ventas brutas en una semana recibe 200 dólares más el 9% de 3000 dólares, o sea un total de 470 dólares.

    Escribir un programa que ingrese desde teclado las ventas de los vendedores, utilizando un arreglo que determine cuántos de los vendedores ganaron salarios en cada uno de los rangos siguientes:

    1. 200 - 299 dólares
    2. 300 - 399 dólares
    3. 400 - 499 dólares
    4. 500 - 599 dólares
    5. 600 - 699 dólares
    6. 700 - 799 dólares
    7. 800 - 899 dólares
    8. 900 - 999 dólares
    9. 1000 dólares o más

7) La "***desviación respecto a la media***", es la desviación de cada uno de los valores respecto a la media usando la fórmula:

    $d = x_i - media$

    donde:
    - x_i, representa cada una de las cantidades i = 1, 2, … n.
    - **media**, es la media calculada.

    Realizar un programa que lea una lista de **n** números de punto flotante, que los almacene en un arreglo unidimensional, y que calcule el valor de la media, mostrando la media calculada y las desviaciones respecto de ella.

Incluir dos funciones adicionales: una que lea los números para calcular la media y a la vez realice la suma, y la segunda que calcule la desviación respecto a la media.

8) Una pequeña aerolínea necesita un programa que asigne asientos en cada vuelo del único avión de la aerolínea (capacidad 20 asientos).

   El programa deberá mostrar el siguiente menú de alternativas:

   Por favor ingrese 1 para "fumadores".
   Por favor ingrese 2 para "no fumadores".

   Si la persona escribe 1, deberá asignarse un asiento en la sección de "fumar" (asientos 1 al 10); si la persona escribe 2, deberá asignar un asiento en la sección de "no fumar" (asientos 11 al 20). A continuación se deberá imprimir un pase de abordaje, indicando el número de asiento de la persona y si está en la sección de fumar o no del avión.

   Utilizar un arreglo unidimensional para representar el diagrama de asientos del avión, previamente inicializar todos los elementos del arreglo a cero para indicar que todos los asientos están vacíos. Conforme se asigne cada asiento, definir los elementos correspondientes del arreglo a 1 para indicar que dicho asiento ya no está disponible.

   El programa no deberá asignar nunca un asiento que ya haya sido asignado. Cuando esté llena la sección de "fumar", se deberá solicitar a la persona, si le parece aceptable ser colocada en la sección de "no fumar" (o viceversa). Si dice que sí, entonces efectuar la asignación apropiada de asiento; si dice que no, entonces imprimir el mensaje "*Próximo vuelo sale en 3 horas*".

9) "***Ordenar una lista de números***" de modo que se ordene en secuencia de valores decrecientes. Para ordenar se debe tener en cuenta que sólo se tendrá un arreglo unidimensional de **n** números, en el que se ordenará un elemento cada vez.

   La ordenación empezará recorriendo todo el arreglo para encontrar el mayor de los números. Este número será intercambiado con el primer elemento del arreglo, colocando así el mayor elemento en el principio de la lista. El resto de los **n-1** elementos del arreglo se recorrerán para encontrar el mayor, que se intercambiará con el segundo elemento, y así sucesivamente, hasta que el arreglo completo sea ordenado. La ordenación completa requerirá un total de **n-1** pasos a lo largo del arreglo, pero cada vez el análisis del arreglo a considerar es menor.

10) Leer **n** números y almacenarlos en un arreglo **a**, a continuación en el mismo arreglo se debe ordenarlo en forma ascendente de tal manera, que el primer elemento quede

en la posición 0 y luego el siguiente quede en la posición 1 y así sucesivamente hasta que en el último elemento, **n-1**, quede almacenado el mayor de todos los números.

El método utilizado se llama "***ordenación tipo burbuja***" u "***ordenación por hundimiento***", porque los valores más pequeños "flotan" de forma gradual hacia la parte superior del arreglo, como suben las burbujas de aire en el agua, en tanto que los valores más grandes se hunden hacia el fondo del arreglo. El cual funciona de la siguiente forma:

- Recorrer el arreglo varias veces, cada pasada consiste en comparar pares sucesivos de elementos, **a[i]** con **a[i+1]**, e intercambiar la información de los elementos si ellos no están en el orden apropiado, estableciendo al final de la pasada que en el último elemento del arreglo quedó el número mayor. Se sigue este mismo proceso para los elementos restantes, y en la última pasada el arreglo está clasificado ascendentemente.

- Como existen **n** elementos se necesita como máximo **n-1** comparaciones y **n-1** pasadas del arreglo para ordenarlo. En la primera pasada, el valor más grande está garantizado que se hundirá hasta el elemento más inferior del arreglo, **a[n-1]**. En la segunda pasada, el segundo valor más grande está garantizado que se hundirá a **a[n-2]**. Así sucesivamente hasta ordenarlo.

- Además, verificar al final de cada pasada, si se han hecho intercambios. Si no se han hecho intercambios, entonces los datos deben estar ya en orden apropiado y, por lo tanto, el programa debe darse por terminado. Si ha habido intercambios, entonces por lo menos se requiere de una pasada adicional.

11) Ordenar una lista de **n** números en punto flotante de modo que se pueda realizar cualquiera de las siguientes ordenaciones:

    a) Menor a mayor en magnitud
    b) Menor a mayor algebraicamente (por signo)
    c) Mayor a menor en magnitud
    d) Mayor a menor algebraicamente (por signo)

    Escribir un programa que lea un arreglo unidimensional de **n** elementos en punto flotante e incluir un menú que permita al usuario seleccionar una de las ordenaciones anteriores para ejecutar ese programa.

12) Elaborar un programa que lea **n** elementos y los almacene en un arreglo, en el que cada elemento debe ser un dígito de 1 a 9. El programa divide el contenido de cada elemento del arreglo por el elemento que tenga la más alta frecuencia dentro del mismo, cuando los números tengan la misma frecuencia, el contenido de cada elemento debe dividirse por el menor de ellos.

Por ejemplo, si **n=8** y el arreglo tiene los siguientes elementos:

2 2 4 4 5 7 4 7

El contenido de cada elemento debe dividirse por 4, ya que su frecuencia es la mayor, igual a 3.

13) Elaborar un programa que lea **n** números y los almacene en un arreglo, **n** debe cumplir que n <= 500. Los números son leídos desde el teclado hasta que se digite 'N' cuando aparece el mensaje:

"*Desea continuar (S/N)?*"

El programa debe comparar el contenido de cada uno de los elementos con el valor de 0. Si un elemento tiene un valor de 0, dicho valor debe desplazarse hacia la derecha del arreglo, permitiendo de esta manera que al final del proceso, todos los números diferentes de 0 estén almacenados al comienzo del arreglo en el mismo orden en el que se leyeron, y todos los números 0 estén almacenados al final del arreglo.

Por ejemplo, si se tiene el siguiente arreglo:

2 3 4 0 9 0 7 0 8 0

al final del proceso el mismo arreglo quedará:

2 3 4 9 7 8 0 0 0 0

14) Elaborar un programa que intercale ascendentemente dos arreglos, el arreglo **a** y el arreglo **b**, los arreglos intercalados deben quedar en el arreglo **c**. El número de elementos del arreglo **a** es **m** y el número de elementos del arreglo **b** es **n**, **m** y **n** deben ser menores de 100.

El proceso se debe efectuar en la medida en que se vayan almacenando los elementos del arreglo **c**; intercalándose ascendentemente, evitando de esta manera realizar un método de ordenamiento del arreglo **c**.

El programa debe leer los arreglos y escribir el contenido de cada uno de ellos luego del proceso.

15) Elaborar una función que reciba un arreglo de valores enteros y el número de elementos del arreglo, para devolver el mismo arreglo sin números duplicados y la frecuencia de cada uno de los elementos que quedan.

Realizar un programa que lea un arreglo e imprima los resultados de la función en una tabla, como la siguiente:

ELEMENTO	FRECUENCIA

16) Elaborar una función que reciba dos arreglos **a** y **b** que contienen una frase terminada en el caracter '**.**'. La función debe devolver el número de veces que se repite la palabra almacenada en el arreglo **b** dentro del arreglo **a**. Considerar que la palabra puede estar al inicio, al final o separada con uno o varios espacios en blanco.

    Realizar un programa que lea los dos arreglos e imprima el resultado de la función y los arreglos.

17) La "***media***" de una secuencia de números de punto flotante, $\mathbf{x_i}$, donde i=1,2,...,m, se define como:

$$\bar{x} = (x_1 + x_2 + \cdots + x_m) / m$$

La "***desviación***" respecto a la media es:

$$d_i = (x_i - \bar{x}), i = 1, 2, \dots m$$

Y la "***desviación estándar***" es:

$$s = \sqrt{(d_1^2 + d_2^2 + \cdots + d_m^2) / m}$$

Realizar un programa que lea **k** secuencias diferentes de números de punto flotante, para calcular en cada secuencia la media, la desviación estándar, el máximo y el mínimo algébrico. Además calcular la media global, la desviación estándar global, el máximo y el mínimo global, indicando la secuencia a la que pertenecen.

18) Elaborar un programa que genere la matriz identidad e imprima dicha matriz. Una matriz identidad es aquella en la que solo la diagonal principal tiene 1 y el resto 0. El orden de la matriz a generar es leído desde el teclado.

19) Elaborar una función que reciba una matriz, el número de fila y el número de columna, para escribir el mayor número almacenado en la matriz, su posición dentro de la misma, el número de la fila y el número de la columna.

    El programa debe leer el número de filas, el número de columnas y la matriz, luego escribir la matriz en forma matricial por filas, el número mayor y el lugar donde se encuentra dentro de la matriz.

20) Una "***ordenación tipo cubeta***" de un arreglo unidimensional de enteros positivos, se realiza mediante un arreglo bidimensional de enteros con filas con subíndices desde 0 hasta 9 y con columnas con subíndices desde 0 hasta **n-1**, donde **n** es el número de valores dentro del arreglo a ordenar. Cada fila del arreglo bidimensional se conoce como una cubeta. Escribir una función **ordenar_cubeta()** que tome un arreglo de enteros y el tamaño del arreglo como argumentos.

    El algoritmo respectivo es como sigue:

    a) Analice el arreglo unidimensional y coloque cada uno de sus valores en una fila del arreglo de cubeta basado en sus dígitos menos significativos. Por ejemplo, 97 se coloca en la fila 7, 3 se coloca en la fila 3, y 100 se coloca en la fila 0.

    b) Analice el arreglo de cubeta y copie los valores de regreso al arreglo original. El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.

    c) Repita este proceso para cada posición digital subsecuente (decenas, centenas, miles, etc.), y deténgase cuando se haya procesado el dígito más a la izquierda del número mayor.

    En la segunda pasada del arreglo, 100 se coloca en la fila 0, 3 se coloca en la fila 0 (sólo tiene un dígito), y 97 se coloca en la fila 9. El orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada, 100 se coloca en la fila 1, 3 se coloca en la fila 0 y 97 se coloca en la fila 0 (después de 3). La ordenación por cubeta garantiza tener todos los valores de forma correcta clasificados, una vez procesado el dígito más a la izquierda del número más grande. La ordenación por cubeta se termina cuando todos los valores se copian en la fila cero del arreglo bidimensional.

    El arreglo bidimensional de cubetas es diez veces del tamaño del arreglo entero a ordenarse pero da un mejor rendimiento. Por lo que esta técnica de ordenación permite un mayor rendimiento que una ordenación tipo burbuja, pero requiere de una capacidad de almacenamiento mucho mayor.

21) Una empresa tiene cuatro vendedores (1 a 4) que venden cinco productos diferentes (1 a 5). Una vez al día, cada vendedor emite un volante para cada tipo distinto de producto vendido, cada volante contiene:

    a) El número del vendedor.
    b) El número del producto.
    c) El valor total en dólares del producto vendido ese día.

Por lo tanto, cada vendedor entrega por día entre 0 y 5 volantes de ventas.

Escribir un programa que lea todos los volantes de las ventas del mes anterior, y que resuma las ventas totales por vendedor y por producto, donde todos los totales deberán almacenarse en un arreglo bidimensional. Luego imprimir los resultados en forma tabular, con cada una de las columnas representando a un vendedor en particular y cada una de las filas representando un producto en particular.

Además, obtener las ventas totales de cada producto y las ventas totales por vendedor, correspondiente al mes pasado. En la impresión en forma tabular se deberá incluir estos totales a la derecha de las líneas totalizadas y en la parte inferior de las columnas totalizadas.

22) La "***criba de Eratóstenes***" es un método para encontrar los números primos, que es cualquier entero que puede ser solo dividido entre sí mismo y entre 1. Esta criba funciona como sigue:

   a) Inicializar todos los elementos de un arreglo a 1, luego los elementos del arreglo con subíndices primos se conservarán en 1 y todos los otros elementos del arreglo se almacenarán con cero.

   b) Empezar con el subíndice 2 del arreglo (el subíndice 1 debe ser primo), todos los subíndices más allá de 2 son múltiplos de 2, estos elementos serán puestos en cero (subíndices 4, 6, 8, 10, etc.). Para el subíndice 3 del arreglo, todos los subíndices más allá de 3 son múltiples de 3, estos elementos serán puestos en cero (subíndices 6, 9, 12, 15, etc). Así sucesivamente.

Cuando se haya terminado este proceso, los elementos del arreglo que aún estén almacenados con 1, indicarán que el subíndice es un número primo.

Escribir un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 1 y **n** ingresado desde teclado. Ignorar el elemento cero del arreglo.

23) Realizar un programa que imprima el valor más pequeño de cada columna de una matriz de **m** filas y **n** columnas. Si hay más de una ocurrencia se debe reportar todos los valores. Imprimir el o los valores más pequeños y los índices donde se encuentran esos valores. El reporte tendrá la siguiente forma:

NUMERO	FILA	COLUMNA

Los valores **m** y **n** deben cumplir las siguientes condiciones m <= 50 y n <= 20. Además los datos que conforman cada línea del reporte, deben generarse en una función, cuyos parámetros son: la matriz, fila y columna.

24) Realizar un programa para generar una matriz **c** de números enteros de dimensión **nfilas** y **ncols**, donde cada elemento es el mayor de los correspondientes elementos en las matrices **a** y **b**, también de dimensiones **nfilas** y **ncols**. Representar cada matriz (cada arreglo) como un arreglo de punteros, por lo tanto usar notación de punteros para acceder a elementos individuales de las matrices.

    Como **a**, **b**, y **c** son definidas como arreglos de punteros, se debe reservar memoria para cada uno usando una función de asignación dinámica. Y las declaraciones de los parámetros formales dentro de las funciones deben representar punteros de punteros a arreglos.

25) Los "***gráficos tipo tortuga***", consiste en que una tortuga mecánica camina por la habitación bajo el control de un programa. La tortuga sujeta una pluma en dos posiciones posibles, arriba o abajo, cuando la pluma está abajo la tortuga traza formas conforme se mueve; cuando la pluma está arriba, la tortuga se mueve a su antojo libremente, sin escribir nada.

    Utilizar un arreglo de 50 por 50 que se inicialice a ceros. Llevar el control en todo momento de la posición actual de la tortuga, así como si la pluma en ese momento está arriba o abajo. Suponiendo que la tortuga siempre empieza a partir de la posición (0,0) en el piso, con su pluma arriba. El conjunto de comandos de la tortuga que el programa debe procesar, debe leerse de un arreglo que los contiene y son los siguientes:

COMANDO	SIGNIFICADO
1	Pluma arriba
2	Pluma abajo
3	Giro a la derecha
4	Giro a la izquierda
5, 10	Moverse hacia adelante 10 espacios (o un número distinto que 10)
6	Imprimir el arreglo de 50 por 50
9	Fin de los datos (valor centinela)

    Por ejemplo, si la tortuga está en algún lugar cerca del centro del piso, el programa dibujaría e imprimiría un cuadrado de 12 por 12. así:

2
5,12
3

5,12
3
5,12
3
5,12
1
6
9

Conforme la tortuga se mueve con la pluma abajo, definir los elementos apropiados del arreglo al valor 1. Cuando se da el comando 6 (imprimir), siempre que exista en el arreglo un 1, imprimir un asterisco, o cualquier otro caracter que se seleccione. Siempre que aparezca un 0, desplegar un espacio vacío.

Escribir un programa para poner en operación las capacidades gráficas de la tortuga, el cual dibuja varios gráficos de tortuga de formas interesantes.

26) La suma de dos matrices A y B es un tercera matriz C, que se calcula utilizando la relación:

C[i][j] = A[i][j] + B[i][j]

Considerar que todas las matrices contengan el mismo número de filas y columnas, y no deben exceder de 20 filas y 30 columnas

Realizar un programa que sume dos matrices de modo que se utilice un arreglo tridimensional en vez de tres arreglos bidimensionales. El primer índice se referirá a una de las tres matrices, el segundo al número de fila y el tercero al número de columna.

27) Se tiene una matriz A de enteros, con **m** filas y **n** columnas, y un vector X de enteros, con **n** elementos. Se desea generar un nuevo vector Y de enteros, que se forma realizando las siguientes operaciones:

Y[1] = A[1][1]*X[1] + A[1][2]*X[2] + … + A[1][N]*X[N]
Y[2] = A[2][1]*X[1] + A[2][2]*X[2] + … + A[2][N]*X[N]
…
Y[N] = A[M][1]*X[1] + A[M][2]*X[2] + … + A[M][N]*X[N]

Escribir un programa que ingrese A y X, y a continuación imprima la matriz A, el vector X y el vector Y.

28) La multiplicación de dos matrices se obtiene mediante el siguiente proceso: se tiene una matriz A de elementos de punto flotante, de **k** filas y **m** columnas, y B es una

matriz de elementos de punto flotante, de **m** filas y **n** columnas. Se desea generar una nueva matriz C, donde cada elemento es determinado por:

C[i][j] = A[i][1]*B[1][j] + A[i][2]*B[2][j] +… + A[i][m]*B[m][j]

donde:

- i = 1, 2, ..., k
- j =1, 2, ..., n

Para multiplicar las matrices debe cumplir que el número de columnas de la primera matriz sea igual al número de filas de la segunda.

Realizar un programa que lea las matrices en una función, y se imprima en forma matricial las tres matrices en otra función.

29) Realizar un programa que lea el orden **m** de una matriz cuadrada que debe cumplir la condición m <= 100, a continuación leer los elementos enteros para almacenarlos en la matriz.

El programa debe elevar la matriz a la potencia **p** que también debe ser leída. Por último, debe imprimirse la matriz leída y la matriz resultante.

Por ejemplo, el procedimiento para elevar una matriz cuadrada a la potencia **p** (p = 4) es el siguiente:

$a^4 = a * a * a * a$

Si se tiene un arreglo **a** de orden **(m, n)** y otro arreglo **b** de orden **(p, q)**. Para multiplicar las matrices debe cumplir que el número de columnas de la primera matriz sea igual al número de filas de la segunda, en este ejercicio n = p, y como las matrices son cuadradas cumple que m = n = p = q. El resultado de la multiplicación de la matriz **a** por la matriz **b** será almacenado en la matriz **c**:

c = a * b

Para calcular el valor del elemento **c[i, k]** se debe multiplicar la **fila-i** de la matriz a por la **columna-k** de la matriz **b**, así:

$$c[i,k] = \sum_{j=0}^{j=n-1} a[i,j] * b[j,k]$$

El programa debe tener una función para multiplicar dos matrices con seis parámetros: los tres arreglos **a**, **b** y **c**, y las variables **m**, **n** y **q**.

30) Una función matemática está representada por los siguientes conjuntos de valores tabulados de **x** e **y**.

Y_0 y_1 y_2 … y_n
X_0 x_1 x_2 … x_n

Se quiere obtener el valor de $\mathbf{y_i}$ para el valor deseado de $\mathbf{x_i}$, que cae entre dos de los valores tabulados del conjunto **y**.

Este problema se resuelve normalmente, mediante "*interpolación*", pasando un polinomio **y(x)** a través de **n** puntos: $y(x_0)=y_0$, $y(x_1)=y_1$, … $y(x_n)=y_n$ y después evaluando **y** para el valor deseado de **x**.

Uno de los métodos de realizar la interpolación es mediante la "***Forma Lagranje***" de la interpolación polinomial, con la fórmula:

$y(x) = f_0(x)*y_0 + f_1(x)*y_1 + \dots + f_n(x)*y_n$

$\mathbf{f_i(x)}$ es un polinomio tal que:

$$f_i(x) = \frac{(x - x_0)(x - x_1) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_n)}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)}$$

donde:

- $f_i(x_i) = 1$
- $f_i(x_j) = 0$, $\mathbf{x_j}$ es un valor tabulado de **x** distinto de $\mathbf{x_i}$. Por tanto, se asegura que $y(x_i) = y_i$.

Escribir un programa que lea **n** pares de datos, y obtenga un valor interpolado de **y** para uno o más valores especificados de **x**. Además, determinar cuántos pares tabulados de datos se requieren en cada cálculo para obtener un valor razonablemente ajustado de **y**.

31) Escribir un programa para procesar las notas de un grupo de estudiantes en un curso de programación de lenguaje C. Empezar especificando el número de notas del examen para cada estudiante (asumir que este valor es el mismo para todos los estudiantes de la clase). Después introducir el nombre de cada estudiante y las notas de los exámenes con pesos desiguales de las notas de los exámenes individuales, por lo que también deben ser ingresados. Por ejemplo, asumir que cada uno de los primeros cuatro exámenes contribuye con el 15 a 100 de nota final y cada uno de los dos últimos con el 20 por 100.

Almacenar los nombres de los estudiantes en un arreglo bidimensional de caracteres y las notas en un arreglo bidimensional de punto flotante. Hacer el programa lo más general posible.

A continuación calcular la media para cada estudiante, la media de la clase. Y la desviación de la media de cada estudiante respecto de la media general. Luego imprimir la ponderación de cada examen y etiquetar claramente la salida como se muestra en el siguiente ejemplo:

Nombre	Notas de exámenes (Porcentajes)						Promedio	Desviación
Andrade	45	80	80	95	55	75	71.00	7.64
Burbano	60	50	70	75	55	80	65.25	13.39
Carrión	80	90	60	50	85	85	76.00	2.64
Dávila	80	95	85	60	90	85	83.00	-4.36
Enríquez	80	95	80	95	90	90	88.50	-9.86
Figueroa	95	90	80	95	85	80	87.00	-8.36
Gómez	75	50	95	85	80	90	79.75	-1.11

Media de la clase = 78.64

32) Escribir un programa que genere una tabla de valores de la siguiente ecuación, la cual será almacenada en una matriz, para mostrar en papel una gráfica en función del tiempo:

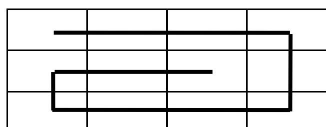
$$y = 2\,e^{-0.1t}\sin(0.5t)$$

donde:

- **t**, varía entre 0 y 60.

El parámetro de incremento **t** introducir desde el teclado.

33) Realizar un programa que lea el orden de una matriz **(m, n)** que debe cumplir la condición m <= 100 y n <= 200, a continuación leer los elementos enteros de la matriz en espiral como se muestra a continuación en la matriz con m = 3 y n = 4.



34) Realizar un programa que lea el orden de una matriz **(m, n)** que debe cumplir la condición m <= 100 y n <= 200, a continuación leer los elementos de tipo caracter de la matriz.

El programa debe ordenar la matriz de tal manera que todos los caracteres diferentes de blanco sean almacenados al comienzo de la matriz, e imprima las matrices leída y ordenada.

Por ejemplo para la matriz con m = 3 y n = 4 se tendrá:

f			
	t		a
u	s	r	

Matriz leída

a	f	r	s
t	u		

Matriz ordenada

35) Escribir un programa completo que realice la cuenta del número de vocales, constantes, dígitos, espacios en blanco y otros caracteres. Esto realizar de modo que se lea varias líneas de texto, que deben ser primero leídas y almacenadas. Además, determinar la media del número de vocales por línea, consonantes por línea y así sucesivamente.

Para almacenar las cadenas utilizar un arreglo de punteros, donde cada cadena es apuntada por un puntero, e introducir la cadena "FIN" para identificar la última línea de texto.

36) Escribir un programa completo que genera una tabla de intereses compuestos, mediante la siguiente fórmula:

$F/P = (1 + i/100)^n$

donde:

- **F**, representa el valor futuro de una suma dada de dinero.
- **P**, el valor actual.
- **i**, la tasa de interés anual, expresada como porcentaje.
- **n**, el número de años.

Cada fila en la tabla corresponde a un valor diferente de **n**, con **n** en el rango de 1 a 20 (20 filas) y cada columna representa una tasa de interés diferente. Incluir las tasas de interés de 4 a 11 por 100 en incrementos de 0.5 (un total de 15 columnas). Etiquetar las filas y las columnas de forma apropiada.

37) Si el interés anual **i** se establece de diferentes frecuencias de composición: anual, semestral, trimestral o mensual; la cantidad futura de dinero acumulado después de **n** años viene dado por:

$$F = 12A\left(\frac{(1 + i\,/\,n)^{mn} - 1}{i}\right)$$

donde:

- **F**, es la cantidad futura.
- **A**, la cantidad de dinero depositada cada mes.
- **i**, la tasa de interés anual (expresado como decimal).
- **m**, el número correspondiente de períodos por año, para la composición:

  m = 1, anual
  m = 2, semestral
  m = 4, trimestral
  m = 12, mensual

  - Para la composición diaria, la cantidad futura es determinada por:

$$F = 12A\left(\frac{(1 + i\,/\,m)^{mn} - 1}{(1 + i\,/\,m)^{m/12} - 1}\right), donde\ m = 360$$

  - Para la composición continua, la cantidad futura es determinada por:

$$F = A\left(\frac{e^{in} - 1}{e^{i/12} - 1}\right), donde\ m = 0$$

Desarrollar un programa para calcular la cantidad futura **F** como función de la tasa de interés anual **i** para los valores **A**, **m** y **n**; los datos **i**, **A** y **n** son leídos.

Como **n** es ingresado desde teclado, por lo que se debe usar un "arreglo dinámico".

Cada una de las fórmulas estarán en las funciones **modo1()**, **modo2()** y **modo3()**; que tienen los parámetros: **i**, **m** y **n**.

Realizar los cálculos en una función **tabla()**, que debe tener el paso de un puntero a función para cada fórmula apropiada (función) y los parámetros **A**, **m** y **n**.

Generar una tabla de valores futuros **F** para varios períodos de tiempo **n**, 1 <= n <= 50, (filas) y las diferentes frecuencias de composición (columnas). Además determinar los valores de **F** para tasas de interés de 1% a 20% al año de acuerdo a la frecuencia de composición.

38) Escribir un programa usando notación de punteros, que genere las siguientes tres columnas:

t	$ae^{bt}\sin(ct)$	$ae^{bt}\cos(ct)$

El programa debe tener dos funciones especiales, **f1()** y **f2()**, donde **f1()** evalúa la cantidad **ae^{bt} sin(ct)** y **f2()** la cantidad **ae^{bt} cos(ct)**. Leer los valores de **a**, **b** y **c**, en la función **main()**, entonces llamar a la función **obt_tabla()** que generará la tabla actual. Pasar **f1()** y **f2()** a **obt_tabla()** coma argumentos. Donde los valores de **t** son 1, 2, 3, …, 60.

39) La multiplicación de una matriz con un vector se obtiene mediante el siguiente proceso: se tiene una matriz A de elementos de punto flotante, de **n** filas y **m** columnas, y B es un vector de **m** elementos de punto flotante. Se desea generar un nuevo vector C de **n** elementos de punto flotante, donde cada elemento es determinado por:

C[i] = A[i][1]*B[1] + A[i][2]*B[2] +...+ A[i][m]*B[m]

donde:

- i = 1, 2,, n

Realizar un programa usando notación de punteros para leer la matriz en una función y el vector en otra función, luego imprimir la matriz en forma matricial en una función y los vectores en otra función.

40) Desarrollar un programa para "***barajar y repartir naipes***", que consiste en barajar un grupo de 52 naipes y a continuación repartir cada uno de los 52 naipes.

Utilizar un arreglo bidimensional de 4 filas y 13 columnas de nombre **caja**, para representar el grupo de naipes. Las filas corresponden a los palos: la fila 0 corresponde a los corazones, la fila 1 a los diamantes, la fila 2 a los tréboles y la fila 3 a las espadas. Las columnas corresponden a los valores nominales de los naipes: las columnas de 0 al 9 corresponden a los valores del ***As*** al 10 respectivamente, y las columnas 10 a la 12 corresponden al ***jack***, ***reina*** y ***rey***. Cargar el arreglo de cadenas **palo**, con cadenas de caracteres que representan los cuatro palos, y el arreglo de cadenas **cara**, con cadenas de caracteres que representen los trece valores nominales.

- "*Para barajar*" este grupo simulado de naipes puede ser como sigue: primero el arreglo **caja** se asigna a cero, a continuación, se selecciona aleatoriamente una **fila (0-3)** y una **columna (0-12)**. El número 1 se inserta en el elemento del arreglo **caja[fila][colum]** para indicar que este naipe será el primero que se repartirá del grupo barajado. Este proceso continúa con los números 2, 3,..,52 que se insertarán al azar en el arreglo **caja**, para indicar cuáles son los naipes que se le colocarán segundo, tercero, …, y cincuenta y dos en el grupo barajado. Conforme el arreglo **caja** se empiece a llenar con números de naipes, es posible que un naipe quede seleccionado dos veces, es decir **caja[fila][coIum]** al ser seleccionado resulte en no cero. Esta selección se ignora y se vuelve a repetir la selección de otras **filas** y **columnas** en forma aleatoria, hasta que se encuentre un naipe no seleccionado.

Eventualmente los números 1 hasta el 52 ocuparan las 52 filas del arreglo **caja**. Llegado a este punto el grupo de naipes ha sido totalmente barajado.

Si los naipes que ya han sido barajados se seleccionan repetidamente al azar, este algoritmo de barajar se ejecutaría en forma indefinida.

- "*Para repartir*" el primer naipe, se busca en el arreglo a caja[fila][co|um] = 1. El arreglo palo ha sido precargado con los cuatro palos, por lo que para obtener el palo, se imprime la cadena de caracteres **palo[fila]**. Similarmente, para obtener el valor nominal del naipe, se imprime la cadena de caracteres **cara[colum]**. También se imprime la cadena de caracteres " de " Al imprimir esta información en el orden adecuado, permite imprimir cada naipe de la forma "***Rey de Tréboles***", "***As de Diamantes***", y así en lo sucesivo.

Realizar un programa de tal forma que tenga una función de distribución de naipes que reparta dos manos de póker de cinco naipes, evaluar cada una de las manos y determinar cuál es la mejor mano. A continuación escribir las funciones adicionales siguientes:

a) Determinar si la mano contiene un par.
b) Determinar si la mano contiene dos pares.
c) Determinar si la mano contiene tres de un tipo (es decir, por ejemplo, tres jacks).
d) Determinar si la mano contiene cuatro de un tipo (por ejemplo, cuatro ases).
e) Determinar si la mano contiene un color (es decir, todos los cinco naipes del mismo palo).
f) Determinar si la mano contiene una flor imperial (es decir, cinco naipes de valores nominales consecutivos).

41) Modificar el programa desarrollado en el ejercicio anterior, de tal forma que pueda simular al tallador: la mano de 5 naipes del tallador se distribuye "tapada", de tal manera que el jugador no pueda verla. El programa deberá entonces evaluar la mano del tallador, y basado en la calidad de la misma, el tallador deberá de solicitar uno, dos o tres naipes adicionales para reemplazar el número correspondiente de naipes innecesarios de la mano original. El programa deberá entonces reevaluar la mano del tallador.

    Considerar de que se pueda manejar de forma automática la mano del tallador, pero el jugador pueda decidir que naipes de su mano reemplazar. El programa deberá entonces evaluar ambas manos y determinar quien gana.

42) Escribir un programa de juego de azar (juego de apuesta) en que simule un "***juego de blackjack***" entre dos jugadores, considerar que la computadora no será un

participante en el juego, simplemente dará las cartas a cada jugador y proveerá a cada jugador con una o más cartas adicionales cuando éste solicite.

Las cartas se dan en orden, primero una carta a cada jugador, después otra carta a cada uno. Las cartas adicionales deben ser solicitadas.

El objeto del juego es obtener 21 puntos, o tantos puntos como sea posible sin exceder de 21 en cada mano. Un jugador es automáticamente descalificado si las cartas en su mano exceden de 21 puntos. Las figuras cuentan 10 puntos y un ***As*** puede contar un punto u 11 puntos. Así un jugador puede obtener 21 puntos (blackjac) si tiene un ***As*** y una figura o un 10. Si el jugador tiene menos puntos con sus dos primeras cartas, puede pedir una carta o más, mientras su puntuación no pase de 21.

El programa requiere de números aleatorios que se obtienen con las funciones estándares y deben usarse para simular la salida de las cartas. Asegurar la inclusión de una condición para que la misma carta no sea dada más de una vez.

43) El juego de azar (juego de apuesta) de la "***ruleta***" se juega con una rueda que contiene 38 cuadros diferentes en su circunferencia. Dos de los cuadros, numerados con el 0 y 00, son verdes; 18 cuadros son rojos y 18 son negros. Se alternan los cuadros rojos y negros y están numerados de 1 al 36 en orden aleatorio.

Una pequeña bola gira dentro de la rueda, que como resultado termina quedando dentro de una ranura debajo de uno de los cuadros. El juego es apostar al resultado de los giros, de una de las siguientes maneras:

a) Seleccionado un cuadro rojo o negro, con una ventaja de 35 a 1. Así, si el jugador apuesta 1 dólar y gana, recibirá un total de 36 dólares: el original más otros 35 dólares.
b) Seleccionado un color, rojo o negro, con una paridad 1 a 1. Así, si el jugador elige rojo y apuesta 1 dólar, si la bola se para debajo de un cuadro rojo recibirá 2 dólares.
c) Seleccionado los números pares o impares (excluidos 0 y 00), con paridad 1 a 1.
d) Seleccionando los 18 números bajos o los 18 altos, con paridad 1 a 1.

El jugador perderá automáticamente si la bolita se para debajo de uno de los cuadros verdes (0 ó 00) .

Escribir un programa que simule el juego de la ruleta, el cual requiere de números aleatorios. Permitir que los jugadores seleccionen cualquier tipo de apuesta que deseen eligiéndola en un menú. Escribir el resultado de cada juego seguido por un mensaje apropiado que indique si el jugador ha ganado o ha perdido.

44) Se tiene un arreglo bidimensional de **n** filas y **m** columnas, donde cada fila representa un alumno y cada columna representa una calificación, en uno de los **m** exámenes que los alumnos pasaron durante el semestre. Las manipulaciones del arreglo se ejecutan mediante cuatro funciones:

- La función **minima_cal()** determina e imprime la calificación más baja de cualquier alumno para el semestre.
- La función **maxima_cal()** determina e imprime la calificación más alta de cualquier alumno para el semestre.
- La función **promedio()** determina e imprime el promedio para cada alumno en particular del semestre.
- La función **imp_arreglo()** imprime el arreglo bidimensional en un formato tabular.

Cada función recibe tres argumentos: el arreglo, el número de alumnos y el número de exámenes.

Escribir un programa de "arreglo de punteros a funciones", para utilizar una interfaz manejada por un menú, que tiene 4 opciones, como se muestra a continuación:

Ingrese una opción:

0 Imprimir el arreglo de calificaciones.
1 Encontrar la mínima calificación.
2 Encontrar la máxima calificación.
3 Imprimir el promedio para cada estudiante.
4 Fin del programa.

Tomar en cuenta que los "punteros a funciones" deben ser con el mismo tipo de regreso y que reciban argumentos del mismo tipo.

Almacenar los punteros a las cuatro funciones en un arreglo, y utilizar la elección que efectúe el usuario como subíndice del arreglo, para llamar a cada una de las funciones.

45) Escribir un programa de juego de azar (juego de apuesta) que simule el "***juego del BINGO***", el cual requiere de números aleatorios. Escribir cada combinación de letra-número según sea sacada (generada). Asegurarse que una combinación no se saca más de una vez, y que cada una de las letras de BINGO corresponde a un cierto rango de números, como se indica a continuación:

B: 1-15

I: 16-30
N: 31-45
G: 46-60
O: 61-75

Cada jugador tendrá una carta con cinco columnas, etiquetadas B-I-N-G-O. Cada columna contendrá cinco números, dentro de los rangos indicados arriba. Dos jugadores no pueden tener cartas iguales. El primer jugador que tenga una línea de números sacados (en vertical, horizontal o en diagonal) gana.

NOTA: A veces la posición central de cada carta se cubre antes de comenzar el juego (una jugada "gratis"), y también se juega a que deben salir todos los números de la carta para ganar.

46) Realizar una función recursiva que invierta los elementos de un arreglo de tipo **float**, en el mismo arreglo.

El programa debe leer el arreglo hasta que se digite el valor de cero a un elemento, e imprimir el arreglo original, el arreglo invertido y el número de elementos.

47) Elaborar una función recursiva que reciba un arreglo de enteros, su longitud y un dato **n** de tipo entero. La función debe devolver un valor -1 si el número **n** no está en el arreglo o la posición en la cual se encuentra **n** dentro del arreglo.

El programa debe leer el arreglo hasta que se digite el valor de cero a un elemento, también debe leer el valor a buscar. Luego imprimir el arreglo, el número de elementos y un mensaje adecuado para indicar si se encuentra el elemento dentro del arreglo.

NOTA: Usar el método de búsqueda binaria.

48) Escribir una función recursiva que ordene un arreglo de enteros utilizando el siguiente método:

a) Suponiendo que **mid** es el índice que direcciona la dirección del arreglo **a**.
b) Ordenar los elementos del arreglo desde **a[mid]** (inclusive) hasta el último.
c) Ordenar los elementos desde el principio hasta **a[mid-1]**.
d) Conformar un arreglo ordenado basado en los subarreglos anteriores.

El programa debe leer el arreglo hasta que se digite el valor de cero a un elemento, e imprimir el arreglo sin ordenar y ordenado.

49) Una "***ordenación de selección***" recorre un arreglo buscando el elemento más pequeño del mismo. Cuando encuentra el más pequeño, es intercambiado con el primer elemento del arreglo. El proceso a continuación se repite para el subarreglo que empieza con el segundo elemento del arreglo. Cada pasada del arreglo resulta en un elemento colocado en su posición correcta.

    Esta ordenación requiere hacerse **n-1** pasadas para un arreglo de **n** elementos, y en cada subarreglo se harán **n-1** comparaciones para encontrar el valor más pequeño. Cuando el subarreglo bajo proceso contenga un solo elemento, el arreglo habrá quedado terminado y ordenado.

    Escribir un progra que tenga una función recursiva **ordena_seleccion()** para ejecutar este algoritmo.

50) Una "***cadena palíndromo***" es una cadena que se escribe de la misma forma hacia adelante y hacia atrás. Por ejemplo, las cadenas "radar" y "Ana ama mamá Ana" son palíndromos.

    Escribir un programa que tenga una función **test_palindromo()**, que devuelva 1 si la cadena almacenada en el arreglo es un palíndromo y 0 si es lo contrario.

    La función deberá ignorar espacios y puntuaciones incluidas en la cadena.

51) La "***búsqueda lineal***" compara cada uno de los elementos del arreglo con el valor buscado. Dado que un arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentre, ya sea en el primer elemento como en el último. Por lo tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.

    El método de búsqueda lineal funciona bien para arreglos pequeños o para arreglos no ordenados.

    Realizar un programa que tenga una función recursiva **busqueda_lineal()** para ejecutar la búsqueda lineal del arreglo. La función deberá recibir como argumentos un arreglo entero y el tamaño del arreglo. Si se encuentra el valor buscado, la función devuelve el subíndice del arreglo, de lo contrario devuelve -1.

52) El método de "***búsqueda binaria***" se utiliza cuando el arreglo está ordenado. El algoritmo consiste en que después de cada una de las comparaciones: si son iguales, la clave de búsqueda ha sido encontrado y se regresa el subíndice del arreglo correspondiente a dicho elemento; si no son iguales, el problema se reduce a buscar en una mitad del arreglo, que corresponde a la primera parte del arreglo, si el valor buscado es menor que el elemento medio del arreglo, de lo contrario se buscará en la segunda parte. Si el valor buscado no se encuentra en el subarreglo especificado, el

algoritmo se repite en una cuarta parte del arreglo original. La búsqueda continúa hasta que el valor buscado es igual al elemento del medio del subarreglo, o hasta que el subarreglo ha quedado reducido a un elemento diferente al valor buscado, en este caso el valor buscado no ha sido encontrado.

Realizar un programa que tenga una función recursiva **busqueda_binaria()**, para ejecutar la búsqueda binaria del arreglo. La función deberá recibir como argumentos un arreglo entero y el subíndice inicial y final. Si el valor buscado es hallado, devuelva el subíndice del arreglo; de lo contrario, devuelva -1.

53) Realizar un programa para "***imprimir un arreglo***" mediante una función recursiva **imprimir_arreglo()**, que tiene un arreglo y el tamaño del arreglo como argumentos y que no devuelva nada. La función deberá dejar de procesar cuando reciba un arreglo de tamaño cero.

54) Realizar un programa para "***imprimir una cadena de atrás para adelante***" mediante una función recursiva ***cadena_inversa()***, que tome un arreglo de caracteres como argumento y que no regrese nada. La función deberá dejar de procesar y regresar cuando se encuentre el caracter nulo de terminación de la cadena.

55) Realizar un programa para "***encontrar el valor mínimo en un arreglo***" mediante una función recursiva **arreglo_minimo()**, que tiene un arreglo entero y el tamaño del arreglo como argumentos y regresa el elemento más pequeño del mismo. La función deberá detener su proceso y regresar cuando reciba un arreglo de un solo elemento.

56) El "***ordenamiento recursivo Quicksort***" es una técnica para ordenar un arreglo unidimensional. El algoritmo básico, es como sigue:

    a) ***Paso de partición***: tomar el primer elemento del arreglo no ordenado y determinar su posición final en el arreglo ordenado. Esto ocurre cuando todos los valores a la izquierda del elemento del arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores que el elemento. Teniendo ahora un elemento en su posición correcta y dos subarreglos no ordenados.

    b) ***Paso recursivo***: Ejecutar el paso 1 para cada uno de los subarregloas no ordenados.

Cada vez que en un subarreglo se ejecuta el paso 1, se coloca otro elemento del arreglo ordenado en su posición final, y se crean dos subarreglos no ordenados. Cuando un subarreglo está formado por un solo elemento deberá ser ordenado por lo que dicho elemento aparece en su posición final. Además, se debe analizar la posición final del primer elemento de cada subarreglo.

Escribir un programa que tenga una función recursiva para ordenar un arreglo entero unidimensional. La función deberá recibir como argumentos un arreglo entero, un subíndice inicial y un subíndice final. Otra función deberá ser llamada por la función recursiva para ejecutar el paso de partición.

57) La siguiente cuadrícula de unos y de ceros es un arreglo bidimensional, que representa un "***laberinto***".

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Los unos representan los muros del laberinto, y los ceros representan cuadros en las trayectorias posibles a través del mismo.

El algoritmo para caminar a través de un laberinto que garantiza encontrar la salida (suponiendo que una exista), y si no existe salida, se llevará de regreso a la posición inicial es: colocar la mano derecha sobre el muro a la derecha y empezar a caminar hacia adelante, sin retirar nunca la mano de la pared, si el laberinto gira a la derecha, seguir el muro a la derecha. de tal forma se llegará a la salida del laberinto. Pudiera existir una trayectoria más corta de la que se haya tomado pero está garantiza de que se saldrá del laberinto.

Escribir la función recursiva para caminar a través del laberinto, la función deberá recibir como argumentos un arreglo de cualquier ancho y altura, de **n** por **m** caracteres, que representa el laberinto, y la posición inicial del laberinto. Conforme la función intenta localizar la salida del laberinto, deberá colocar el caracter **x** en cada uno de los cuadros de la trayectoria. Y después de cada movimiento la función deberá mostrar el laberinto, para que el usuario pueda observar conforme resuelve el laberinto.

Escribir una función que tiene como argumento un arreglo de **n** por **m** y que produzca en forma aleatoria un laberinto. La función también deberá proporcionar las posiciones iniciales y finales del mismo.

Realizar un programa para generar varios laberintos generados al azar, e imprima cada movimiento de los laberintos.

Capítulo 9

CADENA DE CARACTERES

Una cadena en lenguaje C se define como un **arreglo** de tipo **char** que termina en un caracter nulo, '\0', que corresponde al ordinal 0 del código ASCII. Por esta razón, es necesario que el arreglo sea de un caracter más que la cadena más larga. Por ejemplo, la siguiente cadena contendrá un máximo de 30 caracteres:

```
char s[31];
```

9.1. VALORES DE CADENAS CONSTANTES

Una cadena constante es cualquier texto circundado por comillas. Los caracteres que estén encerrados entre las comillas, y junto con el caracter nulo, '\0', se almacenan en posiciones adyacentes de memoria en la "***tabla de cadenas constantes***", de forma que el compilador cuenta el número de caracteres, para conocer de antemano cuánta memoria va a necesitar en su almacenamiento. Por ejemplo la siguiente es una cadena constante:

"Buenos días"

En este caso no es necesario añadir el caracter nulo al final de la cadena constante, ya que el compilador de lenguaje C lo hace automáticamente.

Los caracteres que están encerrados entre las comillas actúan como un puntero al lugar donde se han almacenado éstos, es decir en la tabla de cadenas constantes. Esta situación es análoga al nombre de un arreglo, que es a su vez, un puntero al primer byte a la localización del arreglo. Por ejemplo, el siguiente programa muestra a las cadenas constantes como punteros:

```
/* Cadenas constantes como punteros. */

#incIude <stdio.h>
void main ()
{
    printf ("°%s, %p, %c\n", "Me", "gustas", *"tu");
}
```

La salida podría ser:

Me, 0F40, t

donde:

- El especificador de formato **%s** imprimirá la cadena **Me**.

- El especificador de formato **%p** imprimirá la dirección del primer car acter de la cadena "gustas", que se encuentra almacenada en la tabla de cadenas constantes, en este caso la dirección que se imprime es 0F40.

- ***"tu"** da como resultado el contenido de la dirección del primer carácter de la propia cadena, en este caso el caracter **t**.

9.2. DEFINICIÓN DE CADENAS DENTRO DE UN PROGRAMA

Existen muchas formas diferentes de definir una cadena dentro de un programa, las más importantes son: (SCHILDT, 1994)

1. Cadenas constantes simbólicas.
2. Inicialización de cadenas implementadas con arreglos.
3. Punteros de tipo **char** como cadenas.
4. Arreglo de cadenas de caracteres.

Las cuales serán explicadas en los siguientes subtemas:

9.2.1. Cadenas Constantes Simbólicas

Una "***cadena constante simbólica***" es una cadena constante a la que se le ha dado un nombre.

Se puede definir una "cadena constante simbólica" con el preprocesador **#define** o con el modificador **const**. Por ejemplo, las siguientes sentencias:

```
#define MU "dijo Juan"
printf ("\"¡Vuela, vuela, pajarito!\" %s.\n", MU);
```

Imprimirán:

"¡Vuela, vuela, pajarito!" dijo Juan.

El identificador MU es el puntero a la cadena constante "dijo Juan", esta cadena se encuentra almacenada en la tabla de cadenas constantes.

Se utiliza las cadenas constantes simbólicas principalmente como argumentos de las funciones **printf()** y **puts()**.

9.2.2. Inicialización de Cadenas Implementadas con Arreglos

Esta inicialización puede realizarse de dos maneras:

1. **Arreglo indeterminado**

Una forma de inicializar una cadena es inicializando el arreglo indeterminado con una cadena constante. Por ejemplo, la siguiente declaración inicializa el arreglo **m1** con el valor de la cadena indicada:

char m1[] = "limítese";

Esta forma de inicialización es simplemente una manera más corta de expresar la siguiente inicialización estándar de arreglos:

char m1 = {'l', 'i', 'm', 'í', 't', 'e', 's', 'e', '\0'};

En esta inicialización estándar se debe garantizar que se almacene el carácter nulo de fin de la cadena, porque sin él se tendría un arreglo de caracteres.

Se recomienda la primera forma de inicializar, aunque cualquiera sea la forma, el ordenador cuenta los caracteres y prepara un arreglo del tamaño correspondiente.

Al igual que en los arreglos, el nombre de la cadena **m1** es un puntero al primer elemento de la misma, como se indica a continuación:

 m1 == &m1[0]
*m1 == 'l'

Ejercicio

Realizar una tabla de mensajes de diferente tamaño, para imprimir el tamaño de estos mensajes.

```
/* PROG0901.C */

#include "stdio.h"

void main ()
{
  char e1[] = "Error de lectura.";
  char e2[] = "Error de escritura.";
  char e3[] = "No se puede abrir el archivo.";

  printf ("%s Tiene una longitud de %d.\n", e1, sizeof e1);
  printf ("%s Tiene una longitud de %d.\n", e2, sizeof e2);
  printf ("%s Tiene una longitud de %d.\n", e3, sizeof e3);
```

}

La salida de este programa será:

Error de lectura. Tiene una longitud de 18.
Error de escritura. Tiene una longitud de 20.
No se puede abrir el archivo. Tiene una longitud de 30.

2. Especificación explícita de tamaño del arreglo

Otra forma de inicializar una cadena, es mediante la especificación explícita de tamaño del arreglo, para así disponer de memoria. La declaración anterior se podría hacer así:

```
char m2[20] = "limítese";
```

La única precaución debe ser que el número de elementos declarados sea, como mínimo, uno más que la longitud de la cadena, para almacenar el carácter nulo. Es decir, si se asigna el tamaño a un arreglo se debe elegir un tamaño lo suficientemente grande para almacenar la información de la cadena.

Al igual que en los demás arreglos de tipo **static** o **extern**, cualquier elemento no utilizado será inicializado a cero; el cual en formato **char** corresponde al caracter nulo, '\0'. Por ejemplo, la declaración gráficamente de la cadena mascotas, se muestra en la Figura 9.1.

```
static char mascota[12] = "lindo pez";
```

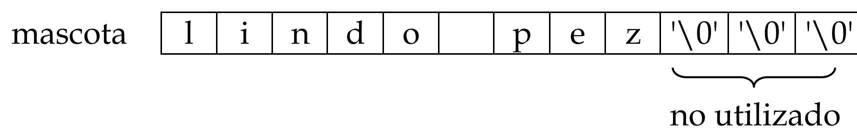


***Figura 9.1. Cadena* mascotas**

Además, para obtener una cadena nula es suficiente con almacenar el carácter nulo en el primer caracter de la cadena. Por ejemplo, en la siguiente asignación se obtiene nula la cadena **mascota**:

```
mascota[0] = '\0';    /* *mascota = '\0'; */
```

9.2.3. Punteros de Tipo *char* como Cadenas

Se puede utilizar notación de punteros para obtener una cadena: inicializando una variable puntero con la dirección de una cadena constante, o mediante la asignación

dinámica para la creación de la memoria donde se va almacenar la cadena. Por ejemplo, la inicialización de una cadena se haría:

```
char *m3 = "\nBasta por hoy";    /* m3 apunta a la cadena constante*/
```

que es lo mismo a:

```
static char m4[] = "\nBasta por hoy";
```

La declaración **char *m3 = "\nBasta por hoy"**; solo se utiliza cuando se inicializa una cadena, porque la memoria se reserva de acuerdo a la cadena constante a ser inicializada. Otra forma sería reservando memoria mediante asignación dinámica, así:

```
char *m5;
m5 = malloc (15 * sizeof (char));
strcpy (m5, "\nBasta por hoy");
/* strcpy(), función estándar para copiar cadenas. */
```

Semejanzas y Diferencias entre las declaraciones de *m3* y *m4*

Las semejanzas son:

a) Las declaraciones de **m3** y **m4** se han preparado como punteros a las cadenas correspondientes.

b) En ambos casos de **m3** y **m4**, la propia cadena determina la cantidad de almacenamiento que se reserva para ella misma, en la "tabla de cadenas constantes". Es decir, se está asignando a la variable puntero la dirección de la cadena constante que se encuentra en la "tabla de cadenas constantes".

Sin embargo, las dos formas de **m3** y **m4** no son completamente idénticas, teniéndose las siguientes diferencias.

a) "*En la forma de arreglo*". Se genera un arreglo de 14 elementos, cada elemento se inicializa a su caracter correspondiente y uno más para el '\0' final.

   A partir de ese momento, el compilador reconocerá a **m4** como sinónimo de la dirección del primer elemento del arreglo así: **&m4[0]**. Por lo que **m4** es un "puntero constante", y no puede ser cambiado porque implicaría alterar la localización (dirección) en que está almacenado el arreglo.

   Se puede utilizar operaciones como **m4+1** para identificar el siguiente elemento del arreglo, pero no se permite **++m4**, ya que el operador incremento solo puede emplearse con variables, y no con constantes.

b) "*En la forma de puntero*". También se genera 14 elementos para la cadena y uno más para el '\0' final. Además, se prepara una localización de memoria extra para la "variable puntero" **m3**, está variable apunta inicialmente al comienzo de la cadena, pero su valor se puede alterar. Entonces, es posible utilizar el operador incremento así: **m3++**, para que apunte al siguiente caracter.

Estas diferencias no son realmente importantes; todo depende de lo que se desee hacer.

Por ejemplo, si se realiza las declaraciones:

```
static char nena[]= "Papá querido" ;
char *nene = "Mamá querida";
```

La diferencia más significativa entre las dos declaraciones es que el puntero **nena** es una constante, mientras que el puntero **nene** es una variable. Entonces, en la práctica se tendría que:

1. Ambas formas pueden ser usadas con la adición de punteros, así:

```
for (i = 0; i < 4; i++)
    putchar (*(nena + i));
putchar ('\n');
for (i = 0; i < 4; i++)
    putchar (*(nene + i));
putchar ('\n');
```

La salida será:

Papá
Mamá

2. Únicamente la versión puntero puede utilizar el operador incremento, así:

```
while (*nene) /* Acaba al final de la cadena. */
    putchar (*(nene++));
```

La salida será:

Mamá querida

Entonces, si se desea asignar a **nene** la dirección de nena, se haría así:

nene = nena; /* nene apunta ahora al arreglo nena */

Pero no está permitido:

nena = nene;

La asignación **nene=nena**; no hace que desaparezca la cadena de caracteres "Mamá querida", simplemente cambia la dirección almacenada en **nene**, perdiéndose la dirección de acceso a esta cadena.

Además, se puede alterar las cadenas mediante la introducción de datos en el arreglo de caracteres. Por ejemplo, para cambiar a "Papá adorado" en la cadena **nena** se haría:

```
nena[5] = 'a';
nena[6] = 'd';
nena[7] = 'o';
nena[8] = 'r';
nena[9] = 'a';
```

Lo mismo se podría haber obtenido con aritmética de punteros.

NOTA: La declaración de cadenas mediante punteros a **char** sin inicialización o asignación dinámica no es recomendable, porque los punteros tienen una dirección desconocida, y esto puede alterar la información en esa dirección, o no se puede realizar la lectura al no existir memoria.

9.2.4. Arreglo de Cadenas de Caracteres

Un arreglo de cadenas de caracteres se utiliza cuando se tiene varias cadenas, y el acceso a las cadenas se realiza mediante un subíndice. Existen dos formas para manipular un arreglo de cadenas de caracteres:

1. **Arreglo de punteros a cadenas**

La memoria para almacenar una cadena se reserva mediante:

a) La inicialización del arreglo de punteros.
b) La asignación dinámica de memoria.

A continuación se describe cada una:

a) ***La inicialización del arreglo de punteros***. La memoria es reservada por las cadenas a ser inicializadas, por lo que el arreglo de punteros puede ser indeterminado o no. Por ejemplo, la declaración del **arreglo cad_punt**, que tiene cinco punteros a cadenas de caracteres, sería:

```
static char *cad_punt[5] = {
    "Números",
    "Múltiple",
    "Datos",
    "Pie de página",
    "Lenguaje C"
};
```

Como **cad_punt** es un arreglo de cinco punteros a cadenas de caracteres; cada puntero apunta a la dirección de su cadena constante correspondiente, la misma que está almacenada en la "tabla de cadenas constantes", como se muestra gráficamente en la Figura 9.2.

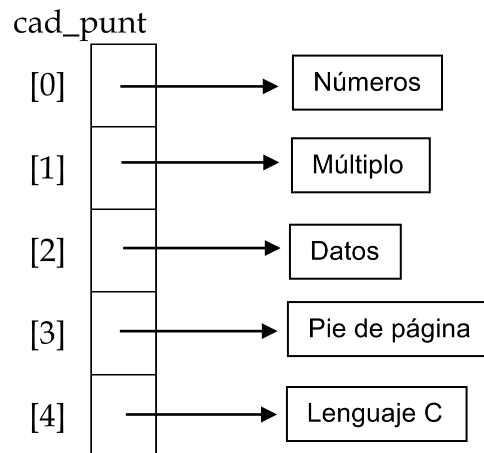


Figura 9.2. Arreglo de punteros **cad_punt**

En donde el primer puntero es **cad_punt[0]**, y apunta a la primera cadena "Números"; el segundo es **cad_punt[1]**, y apunta a la segunda cadena "Múltiple", y así sucesivamente. En concreto, cada puntero apunta al primer caracter de su cadena correspondiente.

Ejercicio

Realizar un programa para imprimir las cadenas inicializadas en el arreglo de punteros **cad_punt**.

/* PROG0902.C */

```
#include "stdio.h"

void main ()
{
  int i;
  static char *cad_punt[5] = {
    "Números",
    "Múltiplo",
    "Datos",
    "Pie de página",
    "Lenguaje C"
  };

  for (i = 0; i < 5; i++)
    puts (cad_punt[i]);
}
```

La salida de este programa será:

Números
Múltiplo
Datos
Pie de página
Lenguaje C

La inicialización para las cadenas sigue las reglas establecidas para arreglos: la parte entre comillas es equivalente a la inicialización de un arreglo de caracteres más el caracter nulo, entonces, la inicialización del arreglo de punteros tendría la forma:

```
static char *cad_punt[5] = {
    {...}, {...}, ..., {...}
};
```

donde, la primera pareja de llaves corresponde a un par de comillas, y se usa, por tanto, para inicializar el puntero al primer caracter de la cadena. La siguiente pareja de llaves inicializa el segundo puntero, etc.

Si no se inicializa el arreglo de punteros, se tiene solo la declaración de los punteros, sin el espacio de memoria para almacenar la información de las cadenas.

b) ***La asignación dinámica de memoria***. Se debe utilizar cualquier función de asignación dinámica, para obtener el espacio de memoria con el objeto de almacenar las cadenas.

Ejercicio

Realizar un programa para leer cadenas de caracteres desde el teclado, y almacenarlas en las direcciones de memoria apuntadas por los punteros de un "arreglo de punteros", para esto se debe reservar memoria con la función **malloc()**, en cada cadena de acuerdo a su tamaño. El tamaño (número de caracteres) de las cadenas se determina con la función **strlen()**. Por último, las cadenas ingresadas en el "arreglo de punteros" deben imprimirse.

```
/* PROG0903.C */

#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void main ()
{
 int i;
 char aux[80];
 static char *cad_punt[5];

 for (i = 0; i < 5; i++) {
   puts ("Ingrese una cadena:");
   gets (aux);

   if ((cad_punt [i] = malloc (strlen (aux) + 1)) == NULL) {
     printf ("Error, no hay suficiente memoria.\n");
     exit (1);
   }

   strcpy (cad_punt[i], aux); /* La función strcpy() copia una cadena en otra. */
 }

 puts ("\nCadenas ingresadas:");
 for (i = 0; i < 5; i++)
   puts (cad_punt[i]);
}
```

La salida de este programa podría ser:

Ingrese una cadena:
Lenguaje C <ENTER>
Ingrese una cadena:
Programación <ENTER>
Ingrese una cadena:
Objetos <ENTER>
Ingrese una cadena:
C++ <ENTER>
Ingrese una cadena:
Windows <ENTER>

Cadenas ingresadas:
Lenguaje C
Programación
Objetos
C++
Windows

Por último, un arreglo de punteros como el del anterior ejercicio:

```
static char *cad_punt[5];
```

declara un arreglo "no uniforme" o "no rectangular", en el que cada porción de memoria (cada cadena) está determinado por la cadena inicializada o la cantidad de memoria asignada dinámicamente, sin que se desperdicie memoria.

Por ejemplo, el siguiente "arreglo no rectangular" se podría graficar como se muestra en la Figura 9.3.

```
static char *cad_punt[5] = {
    "Números",
    "Múltiple",
    "Datos",
    "Pie de página",
    "Lenguaje C"
};
```

N	ú	m	e	r	o	s	\0						
M	ú	l	t	i	p	l	o	\0					
D	a	t	o	s	\0								
P	i	e		d	e		p	á	g	i	n	a	\0
L	e	n	g	u	a	j	e		C	\0			

Figura 9.3. Arreglo no rectangular **cad_punt**

Los arreglos de punteros frecuentemente se utilizan para mantener punteros a mensajes de error. Por ejemplo, para crear una función que muestre un mensaje, dado su número de código de error, se haría:

```
void err_sintaxis (int num)
{
    static char *error[3] = {
        "Error de lectura.\n",
        “Error de escritura.\n",
        "No existe el archivo.\n"
    };

    printf ("%s", error[num]);
}
```

2. **Arreglo bidimensional de tipo char**

Aquí se declara explícitamente el tamaño de las cadenas de caracteres, utilizando una declaración del tipo:

```
static char cad_punt[5][15];
```

Existe una diferencia entre la declaración de un "arreglo de punteros" a cadenas y esta segunda forma: en ésta se obtiene un "***arreglo rectangular***" con todas las filas de la misma longitud. Por ejemplo, el siguiente "arreglo rectangular” se podría graficar como se muestra en la Figura 9.4.

```
static char cad_punt[5][15] =
{
    "Números",
    "Múltiple",
    "Datos",
    "Pie de página",
    "Lenguaje C"
};
```

N	ú	m	e	r	o	s	\0	\0	\0	\0	\0	\0	\0	\0
M	ú	l	t	i	p	l	o	\0	\0	\0	\0	\0	\0	\0
D	a	t	o	s	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
P	i	e		d	e		p	á	g	i	n	a	\0	\0
L	e	n	g	u	a	j	e		C	\0	\0	\0	\0	\0

Figura 9.4. Arreglo rectangular **cad_punt**

Ejercicio

Realizar un programa para leer cadenas de caracteres desde el teclado, y almacenarlas en un arreglo de caracteres bidimensional. Luego imprimir las cadenas ingresadas en el "arreglo bidimensional".

```
/* PROG0904.C */

#include "stdio.h"

void main ()
{
  int i;
  char aux[80];
  static char cad_punt[5][81];

  for (i = 0; i < 5; i++) {
    puts ("Ingrese una cadena:");
    gets (cad_punt[i]);
  }

  puts ("\nCadenas ingresadas:");
  for (i = 0; i < 5; i++)
    puts (cad_punt[i]);
}
```

La salida de este programa podría ser:

Ingrese una cadena:
Lenguaje C <ENTER>
Ingrese una cadena:
Programación <ENTER>
Ingrese una cadena:
Objetos <ENTER>
Ingrese una cadena:
C++ <ENTER>
Ingrese una cadena:
Windows <ENTER>

Cadenas ingresadas:
Lenguaje C
Programación
Objetos

C++
Windows

NOTAS: Cuando se realiza el intercambio de cadenas, en el ordenamiento de éstas, se debe utilizar una de las siguientes alternativas:

- Si se usa "***arreglo de punteros***", se debe hacer intercambio de los punteros a las cadenas, por ser cada puntero una variable.

- Si se usa "***arreglo bidimensional de tipo char***", se debe utilizar las funciones de cadenas para intercambiar sus contenidos. Estas funciones se verán en el siguiente subtema.

Ejercicio

Realizar una función para buscar una cadena en un arreglo de punteros a cadenas, y que determine el índice donde se encuentra dicha cadena, si no se encuentra regresa -1.

En el programa de llamada a la función, el ingreso debe utilizar un puntero nulo para marcar el final del arreglo de punteros. Luego ordenar alfabéticamente el texto ingresado, buscar la cadena en el texto ordenado y por último imprimir el texto ordenado.

```
/* PROG0905.C */

#include "stdio.h"
#include "stdlib.h"   /* Para la función malloc (). */
#include "string.h"

#define TOTAL 30

buscar_cadena (char *nombres[], char *cadena);
void ordenar (char *nombres[], int numero);

void main ()
{
  static char *nombres[TOTAL];  /* Inicializa los punteros a nulos. */
  char cadena[81], aux[80];
  int t;

  printf ("Digite <<Enter>> para terminar.\n");
  printf ("Porque es una línea nula.......\n");
  printf ("l¡nea |  texto\n");
```

```
 for (t = 0; t < TOTAL; ++t) {
  printf ("%-5d | ", t);
  gets (aux);

  /* Reserva memoria. */

  if (*aux) {
   nombres[t] = malloc (strlen (aux) + 1);  /* ó malloc (sizeof aux + 1) */
   strcpy (nombres[t], aux);
  }
  else
   break;         /* Termina si línea es nula. */
 }

 printf ("\nIngrese la cadena a buscar: ");
 gets (cadena);

 ordenar (nombres, t);
 printf ("\n\nTEXTO INGRESADO:\n\n");

 for (t = 0; nombres[t]; t++)    /* Puntero nulo. */
  puts (nombres[t]);

 printf ("\nLa cadena se encuentra en la posición: %d\n",
      buscar_cadena (nombres, cadena));
}

buscar_cadena (char *nombres[], char *cadena)
{
 int t;

 for (t = 0; nombres[t]; ++t)       /* Si puntero es diferente de nulo. */

  if (!strcmp (nombres[t], cadena)) /* Las cadenas son iguales.      . */
   return t;
 return -1;
}

/* Ordena las líneas de texto. */

void ordenar (char *nombres[], int numero)
{
 int i, j;
 char *cad;
```

```
  for (i = 0; i < numero - 1; i++)
    for (j = i + 1; j < numero; j++)

      if (strcmp (nombres[j], nombres[i]) < 0) {
        cad = nombres[i];
        nombres[i] = nombres[j];
        nombres[j] = cad;
      }
}
```

Una salida de este programa sería:

Digite <<Enter>> para terminar.
Porque es una línea nula ....
línea I texto
0 I Víctor Hugo <ENTER>
1 I Ana María <ENTER>
2 I Pedro José <ENTER>
3 I <ENTER>

Ingrese la cadena a buscar: Ana María <ENTER>

TEXTO INGRESADO:

Ana María
Pedro José
Víctor Hugo

La cadena se encuentra en la posición: 0

9.3. FUNCIONES ESTÁNDARES DE CADENAS DE CARACTERES

Las funciones de cadenas de caracteres se encuentran definidas en el archivo de cabecera estándar "***string.h***". Las más comunes son:

La función *strlen()*. Devuelve el número de caracteres de la cadena apuntada por el argumento, como un valor entero sin signo. El caracter nulo no se contabiliza.

El prototipo de la función **strlen()** es: (SCHILDT, 1994)

size_t strlen (const char *cad);

La función **strlen()** devuelve un valor entero sin signo correspondiente al número de caractres de la cadena **cad**.

Por ejemplo:

```
printf ("%d", strlen ("largo"));
```

Imprime:

5

La función *strcat()*. Tiene dos punteros a cadenas de caracteres como argumentos, y añade una copia de la segunda cadena al final de la primera, y hace que esta versión combinada sea la nueva primera cadena. La segunda cadena no se altera.

El prototipo de la función **strcat()** es: (SCHILDT, 1994)

```
void *strcat (char *cad1, const char *cad2);
```

La función **strcat()** devuelve la cadena **cad1** que es la combinación de las cadenas **cad1** y **cad2**.

Por ejemplo:

```
static char *flor = "Las rosas";

strcat (flor, " son rojas")
puts (flor);
```

Imprime:

Las rosas son rojas

La función *strcmp()*. Tiene dos punteros a cadenas como argumentos, y compara caracter a caracter en las dos cadenas, hasta que se encuentre el primer par de caracteres diferentes, luego devuelve la diferencia ASCII de los mismos, y si las cadenas son iguales devuelve el valor 0. Si el valor determinado es diferente de 0, se tiene:

1. "*Un número negativo*" cuando la primera cadena es menor a la segunda desde un punto de vista alfabético.

2. "*Un valor positivo*" cuando el orden alfabético es correcto.

El prototipo de la función **strcmp()** es: (SCHILDT, 1994)

```
int strcmp (const char *cad1, const char *cad2);
```

La función **strcmp()** devuelve un valor entero que es el resultado de la comparación de las dos cadenas.

Por ejemplo:

```
strcmp ("Ana", "Ana"),            da el valor de: 0
strcmp ("Ana", "Anita"),          da el valor de: -8
strcmp ("manzanas", "manzana"),   da el valor de: 115
```

En el último ejemplo, se hace la diferencia de los caracteres 's' y nulo, en la octava comparación, así:

```
's' - '\0' = 115 - 0 = 115.
```

Entonces, con la sentencia:

```
printf ("%d", strcmp ("Ana", "Anita"));
```

Imprime:

-8

La función *strcpy()*. Tiene dos punteros a cadenas como argumentos, en la primera cadena se copia el contenido de la segunda cadena.

El prototipo de la función **strcpy()** es: (SCHILDT, 1994)

```
char *strcpy (char *cad1, const char *cad2);
```

La función **strcpy()** copia el contenido de la cadena **cad2** en la cadena **cad1** y devuelve un puntero a **cad1**.

Por ejemplo:

```
char cad[80];
strcpy (cad, "Hola");
printf ("%s", cad);
```

Imprime:

Hola

La función ***strchr()***. Devuelve un puntero a la primera ocurrencia del argumento caracter en el argumento cadena, caso contrario devuelve un puntero nulo.

El prototipo de la función **strchr()** es: (SCHILDT, 1994)

```
char *strchr (const char *cad, int car);
```

La función **strchr()** devuelve un puntero a la primera ocurrencia del carácter **car** en la cadena apuntada por **cad**.

Por ejemplo:

```
printf ("%p", strchr ("Esta es una prueba", 'u'));
```

Si la cadena comienza a almacenarse en la dirección EF00 imprime:

EF08

La función ***strstr()***. Devuelve un puntero a la primera ocurrencia del segundo argumento cadena en el primer argumento cadena, caso contrario devuelve un puntero nulo.

El prototipo de la función **strstr()** es: (SCHILDT, 1994)

```
char *strstr (const char *cad1, const char *cad2);
```

La función **strstr()** devuelve un puntero a la primera ocurrencia de la cadena apuntada por **cad2** en la cadena apuntada por **cad1**.

Por ejemplo:

```
printf ("%p", strstr ("Esta es una prueba", "es"));
```

Si la primera cadena comienza a almacenarse en la dirección EF00 imprime:

EF05

La función ***strtok()***. Esta función tiene dos argumentos punteros a cadenas, que devuelve un puntero a la siguiente palabra de la primera cadena, y los caracteres que constituyen la segunda cadena son los delimitadores que identifican la palabra. Cuando no hay ninguna palabra a devolver, la función devuelve un puntero nulo.

El prototipo de la función **strtok()** es: (SCHILDT, 1994)

```
char *strtok (char *cad1, const char *cad2);
```

Para que la cadena completa **cad1** pueda reducirse a sus palabras, la cadena **cad1** es utilizada como primer argumento en la primera llamada a la función **strtok()**. En las llamadas posteriores se utiliza un puntero nulo como primer argumento. Y en el segundo argumento se puede utilizar un conjunto diferente de delimitadores para cada llamada de la función **strtok()**.

La función **strtok()** modifica la cadena apuntada por **cad1**, ya que cada vez que se encuentra una palabra se pone un caracter nulo donde estaba el delimitador, de esta forma la función **strtok()** puede continuar avanzando por la cadena.

Por ejemplo, dividir en palabras la cadena "Este es un ejemplo, una aplicación" con espacios en blanco y comas como delimitadores:

```
char *p;

p = strtok ("Este es un ejemplo, una aplicación", " ");
while (p) {
     printf ("%s/", p);
     p = strtok ('\0', ", ");
}
```

Imprime:

Este/es/un/ejemplo/una/aplicación/

Ejercicio

Leer dos cadenas para calcular sus longitudes, verificar si son iguales, unir las dos cadenas, copiar el mensaje "Ejemplo de prueba", verificar si la letra 'o' está en la segunda cadena, y verificar si está la palabra "un" en la segunda cadena.

```
/* PROG0906.C */

#include "stdio.h"
#include "string.h"

void main ()
{
 char c1[80], c2[80];

 puts ("Ingrese primera cadena :");
```

```
  gets (c1);
  puts ("Ingrese segunda cadena :");
  gets (c2);
  printf ("\nLongitudes : %d, %d\n", strlen (c1), strlen (c2));
  if (!strcmp (c1, c2))
    printf ("Las cadenas son iguales\n");
  else
    printf ("Las cadenas no son iguales\n");
  strcat (c1, c2);
  printf ("Cadenas concatenadas: %s\n", c1);
  strcpy (c1, "Ejemplo de prueba");
  printf (c1);
  if (strchr (c2, 'o'))
    printf ("\nEl caracter <o> está  en la cadena: %s\n", c2);
  else
    printf ("\nEl caracter <o> no está  en la cadena: %s\n", c2);
  if (strstr (c2, "un"))
    printf ("La cadena <un> si se encuentra en la cadena: %s\n", c2);
  else
    printf ("La cadena <un> no se encuentra en la cadena: %s\n", c2);
}
```

Una salida del programa sería:

Ingrese primera cadena :
Este es un ejemplo, <ENTER>
Ingrese segunda cadena :
Ejemplo diferente <ENTER>

Longitudes : 19 , 17
Las cadenas no son iguales
Cadenas concatenadas: Este es un ejemplo, Ejemplo diferente
Ejemplo de prueba
El caracter <o> está en la cadena: Ejemplo de prueba
La cadena <un> no se encuentra en la cadena: Ejemplo diferente

9.4. ARGUMENTOS DE LA LÍNEA DE COMANDOS

"***Línea de comandos***" es cualquier comando ejecutable en el sistema operativo, incluyendo un programa compilado.

"***Los argumentos de la línea de comandos***" son items o parámetros adicionales de información de tipo cadena, que siguen al nombre del programa ejecutable (comando)

en la línea de comandos del sistema operativo, y son leídos para emplearlos dentro del programa. (SCHILDT, 1994)

Por ejemplo, si el programa compilado se tiene en un archivo **rifa**, la línea de comandos o ejecución sería:

A:\> rifa <*ENTER*>

La "*línea de comandos*" sirve para pasar información a la función **main()**. Por ejemplo, para ingresar al programa **rifa** la cadena "grande", se haría:

A:\> rifa grande <*ENTER*>

donde:

- **rifa**, es el nombre del programa ejecutable, que puede tener "path".
- **grande**, es el argumento de la línea de comandos.

Si se desea utilizar a varios argumentos separados con espacios en blanco, como una cadena de un solo argumento, se debe encerrar entre comillas a esos argumentos. Por ejemplo, la siguiente línea de comandos tiene un argumento:

A:\> rifa "muy grande" <*ENTER*>

Para recibir los argumentos de la línea de comandos existen dos parámetros especiales ya incorporados en la función **main()**: **argc** y **argv**, que son tradicionales, pero se les puede dar el nombre que se desee. Donde:

- **argc**, es una variable entera que contiene el número de argumentos de la línea de comandos.

  Vale 1 por lo menos, ya que el nombre del programa cuenta con el primer argumento.

- **argv**, es un arreglo de punteros a cadenas, cada elemento de éste apunta a un argumento de la línea de comandos.

  Todos los argumentos de la línea de comandos son cadenas, el sistema utiliza espacios en blanco o tabulados para separar los argumentos de la línea de comandos.

Entonces, la cabecera de la función **main()** sería:

```
void main (int argc, char *argv[])
{
```

```
        /* Sentencias. */
    }
```

Ejercicio

Imprimir los argumentos de la línea de comandos ingresados a la función **main()**.

```
/* PROG0907.C */

#include "stdio.h"

void main (int argc, char *argv[])
{
  int cont;

  for (cont = 1; cont < argc; cont++)
    puts (argv[cont]);
  printf ("\n");
}
```

Este programa se compila con el nombre PROGO907 y luego se ejecuta con la siguiente línea de comandos:

A:\> capitulo.9\prog0907 "ayuda por favor" <*ENTER*>

El programa imprimirá:

ayuda
por
favor

Entonces, el parámetro **argc** es igual a 4 y el arreglo de punteros **argv[]** apuntará a la siguiente información:

argv[0] == capitulo.9\prog0907
argv[l] == ayuda
argv[2] == por
argv[3] == favor

Si se ejecuta el programa con la línea de comandos:

A:\> capitulo.9\prog0907 "ayuda por favor" <*ENTER*>

El parámetro **argc** es igual a 2 e imprimirá:

ayuda por favor

En la mayoría de los casos se utiliza los argumentos de líneas de comandos para indicar un nombre de archivo o una opción, facilitando el uso del programa en archivos de procesamiento por lotes. Entonces, es usual que un programa que utilice argumentos en la línea de comandos, muestre un mensaje de error si no se introduce la información adecuada.

Ejercicio

Realizar un programa que cuente hacia atrás desde un valor especificado como segundo argumento en la línea de comandos, y que imprima la cuenta con un tercer argumento "ver". Luego hacer sonar un pitido cuando llegue a cero.

```
/* PROG0908.C */

/* Programa cuenta atras. */

#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "string.h"

void main (int argc, char *argv[])
{
 int ver, cuenta;

 if (argc < 2) {
   printf ("En la línea de comandos, ");
   printf ("no introduce la cantidad a contar.\n");
   exit (0);
 }
 if (argc ==  3 && !strcmp (argv[2], "ver"))
   ver = 1;
 else
   ver = 0;

 for (cuenta = atoi (argv[1]); cuenta; --cuenta)
 /* La función atoi() convierte una cadena de caracteres a número entero. */
   if (ver)
     printf ("%d\n", cuenta);
 putchar (7);   /* Pita. */
 printf ("FIN");
```

}

La línea de comandos para que cuente 5 veces sería:

A:\> capitulo.9\prog0908 5 "ver" <*ENTER*>

El programa imprimirá:

5
4
3
2
1
FIN

PROBLEMAS PROPUESTOS

1) Escribir un programa que lea una serie de cadenas de caracteres, e imprima las cadenas que empiezan con la letra 'b'.

2) Escribir un programa que lea una serie de cadenas de caracteres, e imprima solo aquellas cadenas que terminan con las letras "ón".

3) Escribir un programa que introduzca **n** cadenas que representen valores enteros, las convierta a enteros, sume esos valores enteros, e imprima el total de los **n** valores.

4) Escribir un programa que introduzca **n** cadenas que representen valores en punto flotante, las convierta a valores **double**, sume esos valores, e imprima el total de los **n** valores.

5) Escribir un programa que introduzca una cadena de caracteres llamada TEXTO, para calcular la frecuencia relativa de cada letra del alfabeto inglés en la cadena. Luego imprimir la cadena y una tabla con cada letra y su frecuencia.

6) Escribir un programa que introduzca varias líneas de texto, para determinar todas las ocurrencias de cada letra del alfabeto inglés en las líneas de texto. Las letras mayúsculas y minúsculas deberán ser contadas juntas. Almacenar en un arreglo los totales de cada letra, y una vez que se hayan determinado los totales, imprimirlos en forma tabular. Utilizar la función **strchr()**.

7) Escribir un programa que introduzca varias líneas de texto y un caracter de búsqueda, para determinar todas las ocurrencias del caracter en las líneas de texto. Utilizar la función **strchr()**.

8) Escribir un programa que tenga dos funciones para comparar dos cadenas introducidas por el usuario. La primera función utiliza la función estándar **strcmp()**, que recibe las cadenas, y la segunda utiliza la función estándar **strcmp()**, que debe recibir a más de las cadenas el número de caracteres a compararse. El programa deberá introducir las cadenas e indicar si la primera cadena es menor que, igual que o mayor que la segunda cadena.

9) Escribir un programa que introduzca desde el teclado una línea de texto y una cadena de búsqueda, para localizar en la línea de texto la primera ocurrencia de la cadena de búsqueda, y asignar la posición a la variable **buscar_Ptr** de tipo **char***. Si se encuentra la cadena de búsqueda imprimir el resto de la línea de texto, empezando con la cadena de búsqueda. A continuación localizar en la línea de texto la siguiente ocurrencia de la cadena de búsqueda. Si encuentra una segunda ocurrencia, imprimir el resto de la línea de texto, empezando con la segunda ocurrencia. Utilizar la función **strstr()**.

El programa debe introducir varias líneas de texto y una cadena de búsqueda, para utilizar la función **strstr()** que determine todas las ocurrencias de la cadena en las líneas de texto. Imprimir todo el texto.

Sugerencia: La segunda llamada a **strstr()** deberá contener **buscar_Ptr+1** como primer argumento.

10) Una "***quintilla jocosa***" es un verso cómico de cinco líneas, en las cuales la primera y segunda línea riman con la quinta y la tercera rima con la cuarta. Utilizando la comparación de cadenas y conociendo el número de caracteres, escribir un programa que produzca "quintillas al azar".

11) Diseñar un programa el cual tenga una función que acepte como parámetro una cadena arbitraria de caracteres y que determine si se trata de una cadena palíndromo o no. Un palíndromo es una cadena de caracteres que se lee igual hacia adelante que hacia atrás, considerando que en los textos más complejos se ignoran los espacios y los signos de puntuación. Por ejemplo, las siguientes cadenas son palíndromos: "OSO" y "Ana ama mamá Ana".

    El programa debe ingresar varias cadenas dinámicamente, para determinar cuáles son cadenas palíndromos.

12) Escribir un programa que introduzca una línea de texto, divida la misma con la función **strtok()**, y extraiga las palabras en orden inverso.

13) Escribir un programa que introduzca varias líneas de texto y utilice la función **strtok()**, para contar el número total de palabras. Suponer que las palabras están separadas por espacios en blanco, signos de puntuación o por caracteres de nueva línea.

14) Escribir un programa que cifre frases de la lengua inglesa y las convierta en latín infantil. "***El latín infantil***" es una forma de lenguaje codificado utilizado a menudo para diversión. Existen muchas variantes en los métodos utilizados para formar frases en latín infantil, el siguiente algoritmo simplifica la formación de las frases:

    Para formar una frase en latín infantil a partir de una frase en lengua inglesa, dividir la frase en palabras, utilizando la función **strtock()**. Para traducir cada palabra inglesa en una palabra en latín infantil, colocar la primera letra de la palabra inglesa al final de la palabra y añadir las letras "ay". De ahí la palabra "jump" se conviene en "umpjay" y la palabra "the" se convierte en "hetay". Los espacios en blanco entre palabras se conservan como están. Suponer lo siguiente: la frase inglesa está formada de palabras separadas por espacios en blanco, no hay signos de

puntuación y todas las palabras tienen dos o más letras. La función **imp_palabra_latin()** deberá desplegar cada palabra.

Sugerencia: Cada vez que se encuentre un distintivo (palabra) en una llamada a **strtok()** pasar el puntero del distintivo a la función **imp_palabra_latin()**, e imprimir la palabra en latín infantil.

15) Escribir un programa que introduzca un número telefónico como cadena, en la forma +(493)2 555-555. El programa deberá utilizar la función **strtok()** para extraer el signo + (00), y el código de país, el código de área, los tres primeros dígitos del número telefónico y los últimos tres dígitos del número telefónico; como palabra. Los seis dígitos del número telefónico deberán ser concatenados en una cadena.

    El programa deberá convertir la cadena del código de país, el código de área a **int**, y convertir la cadena del número telefónico a **long**. El signo +, el código de país, el código de área y el número telefónico; deberán ser impresos.

16) En un programa diseñar una función que acepte como parámetro una cadena de caracteres arbitrarias y regrese al programa principal una cadena que solo contenga los caracteres alfabéticos de la cadena original. Todos los espacios en blanco, signos de puntuación, números y caracteres especiales deben haber sido eliminados.

17) Escribir un programa que utilice generación de números aleatorios para crear oraciones. El programa deberá utilizar cuatro arreglos de punteros a **char** llamados artículo, sustantivo, verbo, y preposición. El programa deberá crear una oración seleccionando una palabra al azar de cada uno de los arreglos, en el orden siguiente: **articulo**, **sustantivo**, **verbo**, **preposicion**, **articulo** y **sustantivo**. Conforme se seleccione cada palabra, deberá ser concatenada con las palabras anteriores en un arreglo lo suficiente extenso para contener a toda la oración. Las palabras deberán estar separadas por espacios. Cuando se extraiga la oración final, deberá iniciar con una letra mayúscula y terminar con un punto. El programa deberá generar 20 oraciones de este tipo.

    Los arreglos deberán ser llenados como sigue: el arreglo **articulo** deberá contener los artículos "el", "uno", "un" "alguno", y "cualquiera"; el arreglo **sustantivo** deberá contener los nombres "muchacho", "muchacha", "perro", "pueblo", y "carro"; el arreglo **verbo** deberá contener los verbos "conducido", "jugado", "corrido", "caminado", y "saltado"; el arreglo **preposicion** deberá contener las preposiciones "hacia", "desde", "arriba", "bajo", y "sobre". En todos los casos deberán tener los femeninos y plurales.

    Producir una corta historia que esté formada con varias de estas oraciones al azar.

18) Se lee dos cadenas de caracteres que contienen nombres de personas, cada uno de estos nombres está separado por una coma. Suponiendo que cada nombre aparece una sola vez en cualquier cadena. Construir un programa que lea las dos cadenas e imprimir la unión de los nombres contenidos en ellas, sin que se repitan. Por ejemplo:

    Si la entrada es:

    Juan, Pedro, Luis, José, Alberto
    Manuel, Carmen, Luis, Alberto, Pedro

    La salida será:

    Juan, Pedro, Luis, José, Alberto, Manuel, Carmen

19) Examinar una entrada compuesta de una serie de caracteres de longitud **n** (letras del alfabeto solamente) y generar como 'resultado una serie de valores numéricos, uno para cada caracter de entrada. Cada posición de la serie resultante deberá ser ocupada por un número, que represente la cuenta de los caracteres que separan a este de la posición correspondiente hacia la izquierda, en que encuentra el más próximo caracter similar a este de la serie. No deberá presentarse ninguna posición mayor a 9, por lo cual cualquier caracter que se encuentre en este caso producirá un cero en la serie de salida. Por ejemplo:

Si la entrada es:	A	A	B	C	D	B	E	F	F	E	G	B	G	A	C
La salida será:	0	1	0	0	0	3	0	0	1	3	0	6	2	0	0

    Formular un programa para resolver este problema de distancia de caracteres, utilizando cadenas de carcteres.

20) Escribir un programa que lea una línea de texto, y lo almacene en un arreglo dinámico. Además, el programa debe tener una función con un puntero a cadena de caracteres como parámetro, esta función retorna la dirección de la cadena invertida: el último caracter es el primero y así sucesivamente.

    Luego en el programa principal ingresar cadenas hasta digitar una cadena nula, y a continuación con la función verificar si cada cadena es igual a su invertida (cadenas palíndromas) .

21) Realizar un programa que tenga 3 funciones:

- Una para convertir cadenas de caracteres a mayúsculas y eliminar caracteres especiales. Es decir, el texto debe tener palabras solo con letras y dígitos separadas con un solo espacio en blanco.
- Otra para buscar una cadena en un arreglo de punteros a cadenas, que debe tomar en su nombre la posición de la cadena encontrada.
- La última para ordenar en orden alfabético las cadenas del punto anterior.

En el programa de llamada utilizar un puntero nulo para marcar el final del ingreso de las cadenas, luego ingresar la cadena a buscar. Todas las cadenas ingresadas deben ser convertidas a mayúsculas, así como eliminados los caracteres especiales. Imprimir la lista de cadenas ordenadas alfabéticamente, y la posición si se encuentra la cadena buscada o un mensaje sino se encuentra la cadena dentro del arreglo.

22) Realizar un programa para ordenar una lista de cadenas en orden alfabético o en orden alfabético inverso. Usar arreglo de punteros a cadenas, e incluir un menú que permita al usuario seleccionar qué ordenación se realiza cada vez que se ejecuta el programa.

23) Considerar las siguientes monedas extranjeras y su equivalencia en dólares USA:

MONEDA ACTUAL	EQUIVALENTE DE LA MONEDA	
Libra británica:	0.6	libras por dólar USA
Dólar canadiense:	1.3	dólares por dólar USA
Guilder holandés:	2.0	guilders por dólar USA
Franco francés:	6	francos por dólar USA
Lira italiana:	1250	liras por dólar USA
Yen japonés:	140	yenes por dólar USA
Peso mexicano:	1600	pesos por dólar USA
Franco suizo:	1.4	francos por dólar USA
Marco alemán:	1.7	marcos por dólar USA

Escribir un programa con menús, que acepte dos monedas extranjeras y devuelva el valor de la segunda moneda por cada unidad de la primera moneda.

NOTA: Usar solo notación de punteros.

24) Considerar la siguiente lista de países y sus capitales:

Argentina	Buenos Aires
Brasil	Brasilia
Chile	Santiago
Colombia	Bogotá
Ecuador	Quito
Inglaterra	Londres
Francia	París
Israel	Jerusalén
Italia	Roma
Venezuela	Caracas
Estados unidos	Washington

Realizar un programa que acepte el nombre de un país como entrada y escriba su correspondiente capital y viceversa. Diseñar el programa de modo que se ejecute repetidamente, hasta que se introduzca la palabra "Fin".

25) Escribir un programa para procesar las notas de un curso, en el cual se debe indicar el número de materias que toma cada estudiante para determinar el arreglo dinámicamente donde se almacenarán las notas. Además, se debe utilizar un arreglo de punteros para almacenar los nombres de los estudiantes.

La información de cada estudiante es el nombre y las notas (sobre 20), que debe ser ingresada hasta tener un puntero nulo en el arreglo de punteros de los nombres.

Imprimir una tabla alfabéticamente como en el siguiente ejemplo de 3 alumnos y 2 notas:

ORD	NOMBRE	MATERIA1 NOTA1	MATERIA2 NOTA2	PROMEDIO/ ALUMNO	OBSERVACION
1	Pérez Carlos	16	15	15.5	--
2	Rosales Patricio	12	12	12.0	MAL
3	Salazar Juan	16	12	14.0	--

Promedio del curso: 13.83

La columna de la OBSERVACION consiste en imprimir el mensaje "MAL", cuando el promedio de las notas del estudiante es menor al promedio general del curso, caso contrario imprimir una línea.

26) Escribir un programa completo que realice la cuenta de vocales, consonantes, dígitos, espacios en blanco y otros caracteres. Esto realizar de modo que se lea varias líneas

de texto, que deben ser primero leídas y almacenadas. Para identificar la última línea de texto introducir la cadena "FIN".

Luego determinar la media del número de vocales por línea, consonantes por línea y así sucesivamente.

Para almacenar las cadenas utilizar un arreglo de punteros, donde cada cadena es apuntada por un puntero.

27) Realizar un programa para leer cadenas en un arreglo de punteros a cadenas hasta que se digite una cadena nula, cada cadena es de longitud variable. Luego contabilizar el número de vocales, consonantes, dígitos, espacios en blanco y caracteres especiales (otros caracteres) para cada cadena. Esta contabilización debe ser almacenado en un arreglo bidimensional.

Luego determinar la media del número de vocales por línea, consonantes por línea, y así sucesivamente. También contabilizar el número total de vocales, consonantes, dígitos, espacios en blanco y caracteres especiales del texto. Finalmente realizar un reporte como el siguiente:

	Vocales #	Consonantes #	Dígitos #	Espacios en blanco #	Otros #
Línea 0:					
Línea 1:					

Promedios por línea:
Línea 0:
Línea 1:

Número total por tipo de caracter:

Vocales: :
Consonantes :
Dígitos :
Espacios en blanco :
Otros :

28) Escribir un programa que tenga una función que acepte tres enteros, indicando el día, mes, y año, y muestre el correspondiente día de la semana, el día del mes, el mes y el año de modo más legible. Por ejemplo, si se introduce los datos 11 24 98, producirá la salida: martes, 24 de noviembre, 1998.

El día de la semana de la fecha especificada se puede determinar convirtiendo la fecha ingresada: mes, día y año (**dd mm aa**), en el número de días relativo a alguna "fecha base", y siempre que se conozca el día de la semana de la "fecha base". Por ejemplo, se podría elegir el lunes 1 de enero de 1900 como "fecha base", entonces se puede convertir cualquier fecha entre el 1 de enero de 1900 y el 31 de diciembre de 2020 a su correspondiente día de la semana.

El cálculo se realiza usando las siguientes reglas:

a) Determinar el día aproximado del año en curso como:
   ndias = (long) (30.42 * (mm - 1)) + dd
b) Si mm == 2 (febrero), incrementar el valor de **ndias** en 1.
c) Si (mm > 2) y (mm < 8) (marzo, abril, mayo, junio o julio), decrementar el valor de **ndias** en 1.
d) Si (aa % 4 == 0) y (mm > 2) (año bisiesto), incrementar el valor de **ndias** en 1.
e) Determinar el número de ciclos completos de cuatro años detrás de la "fecha base" mediante (aa/4). Por cada ciclo completo se añade 1461 a **ndias**.
f) Determinar el número de años completos después del último ciclo de cuatro años mediante (aa%4). Por cada año completo sumar 365 días a **ndias**. Entonces sumar 1, porque el primer año después del ciclo de cuatro años es un año bisiesto.
g) Si ndias > 59 (si la fecha es cualquier día después del 28 de febrero de 1900), decrementar el valor de **ndias** en 1, ya que 1900 no es un año bisiesto. (Notar que el primer año de cada siglo no es bisiesto, pero el cuarto año después de éste sí lo es).
h) Determinar el día correspondiente de la semana especificada como dia = (ndias%7).

Los nombres de los días de la semana se pueden inicializar en un arreglo de punteros a cadenas, comenzando desde "domingo" porque domingo corresponde a dia == 0 y la fecha base es lunes. Igualmente, los nombres de los meses se pueden inicializar en un arreglo de punteros a cadenas.

También, el programa debe determinar el número de días entre dos fechas, asumiendo que ambas fechas son posteriores a la "fecha base" del 1 de enero de 1900.

29) Algunos estudiosos creen que existe evidencia sustancial indicando que Christopher Marlowe, de hecho fue el que escribió las obras maestras atribuidas a Shakespeare. Los investigadores han utilizado computadoras para localizar similitudes en los textos de estos dos autores. Este ejercicio examina tres métodos para "analizar texto", utilizando una computadora.

a) Escribir una función que lea varias líneas de texto e imprima una tabla indicando el número de instancias de cada letra del alfabeto en dicho texto. Por ejemplo, la frase:

Ser, o no ser: eso es la cuestión:

contiene: ninguna 'a', ninguna 'b', una 'c', etc.

b) Escribir una función que lea varias líneas de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras que aparecen en el texto. Por ejemplo, la frase:

Ser, o no ser: eso es la cuestión:

contiene:

Longitud de palabra	Ocurrencias
1	1
2	3
2	3
3	0
4	0
5	0
6	0
7	0
8	1

c) Escribir una función que lea varias líneas de texto e imprima una tabla que indique el número de ocurrencias de cada palabra en el texto. La impresión deberá ser en orden alfabético. Por ejemplo, las líneas:

Ser, o no ser: eso es la cuestión:
La mejor forma de ser feliz.

contiene las palabras: "cuestión", una vez, "de" una vez, etc.

30) El "***procesamiento de palabras***" es el tratamiento de la manipulación de cadenas en los procesamientos de textos. Una función importante de los sistemas de procesamiento de palabras es el "tipo justificado", la alineación de las palabras, tanto en los márgenes izquierdo como derecho de una página. El tipo justificado puede ser llevado a cabo insertando uno o más caracteres en blanco entre cada una de las palabras de una línea, de tal forma que la palabra más a la derecha se alinee con el margen derecho.

Escribir un programa que lea varias líneas de texto e imprima este texto en formato de tipo justificado. Suponer que el texto debe ser impreso en papel de un ancho de 8 1/2 pulgadas, y que se deben dejar márgenes de una pulgada, tanto en el lado derecho como izquierdo de la página impresa. Suponer que la computadora imprime 10 caracteres por cada pulgada horizontal. Por lo tanto, el programa deberá imprimir 6 1/2 pulgadas de texto, es decir, 65 caracteres por línea.

31) En la correspondencia de negocios "las fechas se imprimen en varios formatos" diferentes. Dos de los formatos más comunes son:

11/21/2008,
Noviembre 21, 2008

Escribir un programa que lea una fecha en el primer formato y la imprima en el segundo.

32) Para evitar que sea alterada una cantidad de un cheque, la mayor parte de los sistemas computarizados de escritura de cheques emplean una técnica conocida como "***protección de cheques***". Ya que muchos sistemas computarizados de escritura de cheques no incluyen la cantidad del cheque en palabras.

Los cheques diseñados para impresión por computadora contienen un número fijo de espacios, en el cual la computadora puede imprimir una cantidad. Suponer que un cheque de nómina contiene ocho espacios en blanco, en el cual la computadora se supone imprimirá la cantidad de la nómina mensual.

Si la cantidad es grande, entonces todos los ocho espacios quedarán llenos. Por ejemplo:

1	.	2	3	0	.	0	0	chequeo del monto)
1	2	3	4	5	6	7	8	(posición de números)

Por otra parte, si la cantidad es menor de 1000 dólares, entonces varios de los espacios quedarían en blanco. Por ejemplo, a continuación se tiene tres espacios en blanco:

9	9	.	0	0			
1	2	3	4	5	6	7	8

Si un cheque se imprime con espacios en blanco, es más fácil que alguien pueda alterar la cantidad del cheque. A fin de evitar que se modifique un cheque, muchos

sistemas de escritura de cheques "presentan asteriscos", para proteger la cantidad, como sigue:

*	*	*	9	9	.	0	0
1	2	3	4	5	6	7	8

Escribir un programa que introduzca una cantidad en dólares a imprimirse en un cheque, y a continuación imprimir formato protegido de cheque con asteriscos anteriores, si ello fuera necesario. Suponer que para la impresión de una cantidad están disponibles ocho espacios.

33) Un método tradicional de seguridad para "***protección de cheques***" que impidan la modificación de las cantidades de los cheques, requiere que la cantidad del cheque se escriba tanto en números (de acuerdo al programa anterior) como en palabras, es decir, escribir el equivalente en palabras de una cantidad de cheque. Si alguien es capaz de modificar la cantidad numérica del cheque, resulta en extremo, difícil modificar la cantidad en palabras.

Escribir un programa que introduzca una cantidad de cheque numérico y escriba el equivalente en palabras de dicha cantidad. Por ejemplo, la cantidad 112.00 deberá quedar escrita como:

ciento doce con 00/100 dólares

34) Quizás el más famoso de todos los sistemas de codificación es el "***código Morse***", para uso en el sistema telegráfico. El código Morse asigna una serie de puntos y rayas a cada letra del alfabeto, a cada dígito y a unos cuantos caracteres especiales (como el punto, coma, punto y coma; y dos puntos). La separación entre palabras se indica por un espacio, o por la ausencia de un punto o de una raya.

La versión internacional del código Morse aparece en la siguiente tabla:

Carácter	Código	Carácter	Código
A	.-	T	-
B	-…	U	..-
C	-.-.	V	…-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	….		
I	..	Dígitos	
J	.---	1	.----

K	-.-	2	..---
L	.-..	3	…--
M	--	4	….-
N	-.	5	….
O	---	6	-….
P	.--.	7	--…
Q	--.-	8	---..
R	.-.	9	----.
S	…	0	-----

Escribir un programa que lea una frase en español y que cifre la frase en código Morse. También escribir un programa que lea una frase en código Morse y la convierta en el equivalente en español. Utilizar un espacio en blanco entre cada letra codificada Morse y tres espacios en blanco entre cada palabra codificada en Morse.

35) Escribir un programa de "***conversiones métricas***" que permita al usuario especificar como cadenas los nombres de las unidades (es decir, centímetros, litros, gramos, etc. para el sistema métrico y pulgadas, cuartos, libras, etc. Para el sistema inglés) y deberá responder a preguntas sencillas como:

¿Cuántas pulgadas están en 2 metros?
¿Cuántos litros están en 10 cuartos?

También el programa deberá poder reconocer conversiones inválidas. Por ejemplo, la pregunta:

¿Cuántos centímetros tiene en 5 Kilogramos?

no tiene sentido, porque "centímetros" es unidad de longitud, en tanto que "kilogramos" es una unidad de peso.

36) La cobranza de cuentas vencidas es el proceso de efectuar demandas repetidas o insistentes por carta a un deudor, en un intento de cobrar una deuda.

Generar automáticamente "***cartas de cobranza***", en grado creciente de severidad, conforme la cuenta se hace más vieja, La teoría es que mientras más vieja sea la cuenta, más difícil será su cobranza, por lo tanto, las cartas correspondientes deberán ser más y más amenazadoras.

Escribir un programa que contenga los textos de **n** (ingresado desde teclado) cartas de cobranza de severidad creciente. El programa deberá aceptar como entrada de cada carta:

a) El nombre del deudor.

b) La dirección del deudor.
c) La cuenta del deudor.
d) La cantidad adeudada.
e) El atraso en la cantidad adeudada (es decir, un mes de vencida, dos meses de vencida, etc.).

Utilizar el atraso en el pago para seleccionar uno de los **n** textos de mensaje, y a continuación imprimir la carta de cobranza insertando donde resulte apropiada la otra información proporcionada por el usuario.

37) "***El generador de pig latin***" es una forma codificada de escribir y de hablar que suelen usar los niños ingleses como juego. Una palabra en "pig latin" se forma transponiendo el primer sonido, generalmente la primera letra, de una palabra original al final de la misma y añadiendo al final la letra 'a'. Por ejemplo, la palabra "perro" se convierte en "erropa".

Escribir un programa que se ingrese varias cadenas en un arreglo de punteros a cadenas y escriba su correspondiente texto en "pig latin". Las cadenas tienen diferente longitud, con un solo espacio en blanco entre cada dos palabras sucesivas.

Además, considerar marcas de puntuación, letras mayúsculas y sonidos de letras dobles.

38) Escribir un programa que calcule recursivamente la suma de n números en punto flotante. Introducir el valor de n como parámetro de la línea de comandos y a continuación introducir los datos a ser sumados.

Usar las funciones de biblioteca **atoi()** y **atof()**, para convertir los parámetros de la línea de comandos en enteros y valores de punto flotante, respectivamente.

39) Escribir un programa que calcule recursivamente los primeros **n** términos de la serie:

$y = 1 - x + x^2/2 - x^3/6 + x^4/24 + \quad + (-1)^n x^n/n!$

Introducir el valor de **n** como parámetro de la línea de comandos. Usar la función de biblioteca **atoi()** para convertir el parámetro de la línea de comandos en entero.

Capítulo 10

ESTRUCTURAS

10.1. ESTRUCTURAS

10.1.1. Introducción

La estructura permite representar distintos tipos de datos bajo un mismo nombre, y además permite al usuario crear nuevos formatos. Estos datos se almacenan en variables que componen la estructura y se llaman "miembros" o "campos" de la estructura.

Existen tres condiciones para manipular una estructura: (SCHILDT, 1994)

1. Preparación del patrón de la estructura, es decir, preparar un patrón, formato, plantilla o forma para la estructura.

2. Declaración de variables estructuras, que se ajusten a dicho patrón.

3. Acceso a miembros de la estructura, es decir, acceder a los distintos componentes individuales de una variable de tipo estructura.

1. Patrón de la estructura

El "***patrón de la estructura***" describe la estructura o la forma de la misma.

La forma general de la definición del patrón de una estructura es:

```
struct etiqueta {
    tipo nombre_variable;
    tipo nombre_variable;
    …
} variables_estructura;
```

donde:

- **struct**, especifica como estructura lo que va a continuación de esta palabra reservada.

- **etiqueta**, es la etiqueta o rótulo opcional, que permite referirse posteriormente a la estructura de una manera abreviada, y es también el "especificador de tipo" de la estructura, porque solo se ha definido la forma de los datos y no la variable estructura.

- **Lista de miembros de la estructura**, que se encierra entre llaves, y es en donde cada miembro queda descrito en su propia declaración. Los

miembros pueden ser de cualquiera de los tipos de datos que ya se han estudiado, incluyendo otras estructuras.

- **variables_estructura**, son las variables de las estructuras de este tipo de patrón. Se puede omitir, o bien la "etiqueta", o bien las "variables_estructura", pero no ambas a la vez.

- **Punto y coma**, indican la finalización de la estructura, porque la definición de una estructura es una "*sentencia de declaración*".

Para que el patrón quede disponible para todas las funciones que se encuentren a continuación de la definición en el programa, se lo coloca fuera de las funciones (externamente). También se puede definir el patrón en el interior de una función, en este caso se lo puede utilizar únicamente dentro dicha función.

El nombre de la ***etiqueta*** es opcional, pero se lo emplea obligadamente si se define el patrón de la estructura en un lugar y sus variables en otro.

Por ejemplo, se presenta a continuación la definición del patrón **biblio** de una estructura, este patrón describe una estructura formada por dos arreglos de caracteres y una variable de tipo **float**:

```
struct biblio {
    char titulo[31];
    char autor[31];
    float precio;
};
```

2. Declaración de variables estructuras

La declaración o creación de una "***variable estructura***", se realiza mediante la palabra **struct**. Y como se vio anteriormente, esta misma palabra se utiliza también para definir el "patrón de la estructura", que indica el tipo de estructura.

Por ejemplo, la declaración de una variable estructura de tipo **biblio**, es:

```
struct biblio libro;
```

Una vez ejecutada esta sentencia, el computador crea una variable **libro** de tipo estructura **biblio**, ya que la declaración **struct biblio** juega el mismo papel que los tipos estándares **int**, **float**, etc.; pero de un tipo complejo de variable. En la Figura 10.1 se muestra gráficamente la estructura **libro**.

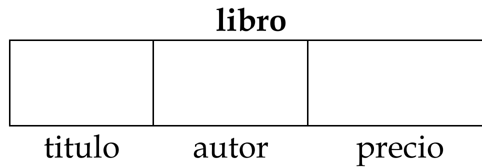


Figura 10.1. Estructura **libro**

El compilador de lenguaje C dispone automáticamente de la suficiente memoria para almacenar todas las variables que componen una variable estructura. Por ejemplo, en la variable **libro**, se tiene:

titulo :31 bytes
autor :31 bytes
precio: 4 bytes

Total : 66 bytes

Por lo tanto, se pueden declarar varias variables de tipo **struct biblio**, y también punteros a este tipo de estructura. Por ejemplo:

```
struct biblio uno, bar, *ptlibro;
```

donde:

- **uno** y **bar**, son variables que tendrán cada una de ellas los miembros: **titulo**, **autor** y **precio**.

- **ptlibro**, es un puntero que puede apuntar a las variables **uno** y **bar** o a cualquier otra estructura **biblio**.

Se puede también declarar una o más variables en el momento que se define una estructura. Es decir, el "patrón de estructura" y la "declaración de la variable estructura" pueden definirse simultáneamente, y así se evita el uso de etiquetas.

Por ejemplo la siguiente declaración.

```
struct biblio {
    char titulo[31];
    char autor[3l]; "
    float precio;
} articulo, folleto;
```

define un tipo estructura **biblio**, y declara las variables **articulo** y **folleto** de dicho tipo.

Si solo se necesita una variable estructura, no es necesaria la etiqueta, es decir, no se necesitan nuevas declaraciones de estructuras. Como se muestra a continuación:

```
struct {          /* Sin etiqueta. */
    char titulo[31];
    char autor[31];
    float precio;
} articulo;
```

Se debe anotar que la forma con "etiqueta" es más manejable si se utiliza más de una vez el mismo "patrón de estructura".

3. **Acceso a miembros de la estructura**

Para "***acceder a los miembros de una estructura***" se emplea el operador punto (**.**).

La forma general de acceso a un miembro de una estructura es:

nombre_estructura.nombre_miembro

Por ejemplo, al miembro **precio** de la variable estructura **libro** se asigna 85, entonces:

```
libro.precio = 85;
```

precio es un miembro de la estructura **libro**, y se puede utilizar **libro.precio** exactamente igual que cualquier otra variable de tipo **float**. El acceso a los demás miembros es similar. Por ejemplo, copiar en los miembros **libro.tituIo** y **libro.autor** los siguientes valores:

```
strcpy (libro.titulo, "De la tierra a la luna");
strcpy (libro.autor, "Julio Verne");
```

En esencia: **.tituIo**, **.autor**, y **.precio** juegan el papel de subíndices en la estructura de tipo **biblio**, y se refieren siempre a los miembros de dicha estructura.

10.1.2. Asignación de Estructuras

Se puede asignar la información contenida en una estructura a otra estructura del mismo tipo. Es decir, en lugar de asignarse los valores de todos los miembros por separado, se utiliza una sola sentencia de asignación entre las dos estructuras.

Por ejemplo, la asignación:

```
folleto = libro;
```

es equivalente a realizar:

```
strcpy (folleto.titulo, libro.titulo);
strcpy (folleto.autor, libro.autor);
folleto.precio = libro.precio;
```

Es decir, se asigna cada miembro de la estructura **libro** a la estructura **folleto** de una sola vez.

10.1.3. Inicialización de una Estructura

En la inicialización de una "variable estructura", se debe tener en cuenta dónde se define la "variable", y no dónde está definido el "patrón".

Además, para inicializar una estructura se coloca la declaración de cada miembro en cada línea. Sin embargo, lo único que se necesita para separar la inicialización de un miembro de otro son comas. (SCHILDT, 1994)

Por ejemplo, para inicializar la estructura **libro** se procede así:

```
static struct biblio libro = {
    "De la tierra a la luna",
    "Julio Verne",
    85
};
```

10.1.4. Arreglo de Estructuras

Para trabajar con varias estructuras del mismo tipo, se usa un arreglo de estructuras, siendo éste su uso más común. En un arreglo de estructuras, cada elemento es una estructura.

Los dos puntos más importantes de un arreglo de estructuras son:

1. La declaración de un arreglo de estructuras.

2. La identificación de los miembros en un arreglo de estructuras.

1. Declaración de un arreglo de estructuras

El proceso de declaración de un "arreglo de estructuras" es completamente análogo al de cualquier tipo de arreglo. Es decir, se debe definir primero el patrón de la estructura y luego declarar una variable arreglo de dicho tipo.

Por ejemplo, para declarar **libros** como un arreglo de 100 elementos, en donde cada elemento del arreglo es una estructura de tipo **biblio**, se indica a continuación:

```
struct biblio libros[100];
```

2. Identificación de los miembros en un arreglo de estructuras

Para identificar los miembros en un "arreglo de estructuras" se aplica la misma regla que se emplea para estructuras individuales: se escribe el nombre de la estructura (elemento del arreglo de estructuras) seguido del operador punto y del nombre del miembro. Por ejemplo:

libros[0].titulo, es el "titulo" asociado con el primer elemento estructura del arreglo.

libros[9].precio, es el "precio" asociado con el décimo elemento estructura del arreglo.

En la Figura 10.2 se muestra gráficamente el arreglo de estructuras **libros**.

libros

	titulo	autor	precio
libros[0]			
libros[1]			
	⋮	⋮	⋮
libros[99]			

Figura 10.2. Arreglo de estructuras **libros**

Si se desea leer todo el arreglo **libros**, se escribiría:

```
for (i = 0; i < 100; i++){
     gets (Iibros[i].titu|o);
     gets (Iibros[i].autor);
     scanf ("%f", &libros[i].precio);
}
```

NOTA: En algunos compiladores de Linux, no se puede leer directamente un miembro de una estructura de tipo **float**. Por lo tanto, la solución sería leer en

una variable de tipo **float** y luego almacenar el dato leído en el miembro **float** de la estructura.

Ejercicio

Diseñar un programa que realice un inventario de libros, donde la información ingresada de cada libro es: título, autor y precio. Además, se desea obtener una lista completa de los libros clasificados por título, autor y precio. Por ejemplo:

TITULO	AUTOR	PRECIO (USD)

```
/* PROG1001.C */

/* Inventario de libros. */

#include <stdio.h>

#define MAXTIT 31      /* Longitud máxima del título.      */
#define MAXAUT 31      /* Longitud máxima del autor.       */
#define MAXLIB 100     /* Número máximo de libros.         */
#define STOP ""        /* Cadena nula, finaliza entrada.   */

struct biblio {     /* Patrón de la estructura.         */
  char titulo [MAXTIT];
  char autor [MAXAUT];
  float precio;
};

void main ()
{
  struct biblio libros[MAXLIB];      /* Arreglos de estructuras biblio. */
  int cont = 0;
  int indice;
  float aux_float;

  printf ("Pulse <<Enter>> a comienzo de línea de título para");
  printf (" parar.\n\n");
  printf ("Introduzca el título del libro: ");

  while (strcmp (gets (libros[cont].titulo), STOP) &&
        cont < MAXLIB) {
    printf ("Introduzca el autor          : ");
    gets (libros[cont].autor);
```

```
    printf ("Introduzca el precio        : ");
    scanf ("%f", &aux_float);
    libros[cont++].precio = aux_float;
    while (getchar () != '\n');    /* Limpia línea de entrada. */
    if (cont < MAXLIB)
      printf ("\nIntroduzca el siguiente título: ");
  }

  printf ("\nLISTA DE LIBROS.\n\n");
  printf ("TITULO                    AUTOR                    ");
  puts ("PRECIO (S/.)");
  printf ("------------------------------------------------------------");
  puts ("------------");
  for (indice = 0; indice < cont; indice++)
    printf ("%-30s %-30s %10.2f\n", libros[indice].titulo,
          libros[indice].autor, libros[indice].precio);
}
```

La salida de este programa podría ser:

Pulse <<Enter>> a comienzo de línea de título para parar.
Introduzca el título del libro: Platero y Yo <ENTER>
Introduzca el autor :Juan Jiménez <ENTER>›
Introduzca el precio : 89 <ENTER>
Introduzca el siguiente título: <ENTER>

LISTA DE LIBROS

TITULO   AUTOR   PRECIO (UDS)

Platero y Yo   Juan Jiménez   89.00

Se debe tomar en cuenta que la sentencia:

```
while (getch () != '\n');   /* Limpia la línea de entrada. */
```

elimina el caracter nueva línea de la cola de entrada, con ello permite que la siguiente lectura con **gets()** disponga de un comienzo adecuado. Es decir, esta sentencia se utiliza porque la función **scanf()** ignora espacios en blanco y caracteres nueva línea.

Por ejemplo, si se teclea para el precio ***89*** <ENTER>, se transmite la secuencia de los caracteres '8', '9', '\n'. Luego la función **scanf()** recoge los caracteres '8', '9'; pero deja el caracter '\n' esperando a la próxima sentencia de lectura, que sería **gets(libros[cont].titulo)**. Por consiguiente, **gets()** leería el caracter '\n' como primer

caracter, siendo una cadena nula, que es una señal de STOP para terminar el lazo y por consiguiente el programa.

10.1.5. Arreglos dentro de Estructuras

Un miembro de una estructura puede ser un arreglo, que es utilizado como un miembro de un tipo de dato estándar. Por ejemplo, considerar la siguiente estructura:

```
struct {
    int a[10][10];
    float b;
} y;
```

Para referenciar el elemento entero de la fila 1 y la columna 9 del arreglo miembro **a** en la estructura **y**, se escribe:

```
y.a[1][9] = 500;
```

En la Figura 10.3 se muestra gráficamente la estructura **y**.

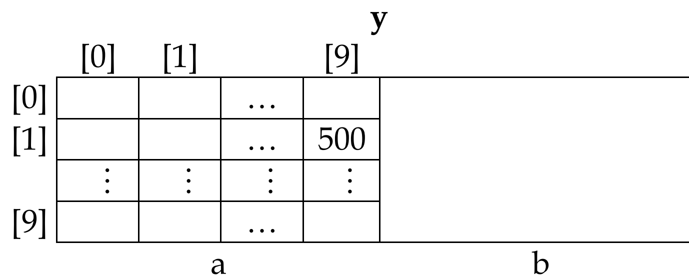


Figura 10.3. Estructura **y**

Si se desea leer toda la estructura **y**, se escribiría:

```
for (i = 0; i < 10; j++)
    for (j = 0; j < 10; i++)
        scanf ("%d", &y.a[i][j]);
scanf ("%af", &y.b);
```

Los subíndices escritos a la derecha del operador punto (**.**) se aplican a los datos individuales, en tanto que los subíndices a la izquierda del operador se aplican a los elementos del arreglo de estructuras.

10.1.6. Estructuras Anidadas

Son estructuras contenidas en otras, es decir cuando una estructura es miembro de otra estructura. La estructura anidada se coloca en el patrón, declarándola de forma similar que una variable de tipo estándar. Por ejemplo, las siguientes estructuras anidadas:

```
struct nombres {                /* Primer patrón de estructura. */
    char nom[21];
    char apeII[21];
};

struct empleado {               /* Segundo patrón de estructura. */
    struct nombres conductor; /* Estructura anidada. */
    char comida[21];
    char trabajo[21];
    float gana;
};
```

A continuación, declarar la variable **empleos**, que es un arreglo de doce estructuras anidadas de tipo **empleado**:

```
struct empleado empleos[12];  /* Son 12 estructuras anidadas. */
```

Como en el patrón **empleado** se tiene el miembro **conductor**, que es una variable de tipo **struct nombres**, se debe definir previamente el patrón de **struct nombres**.

Para acceder a los miembros de una estructura anidada, se utiliza dos veces el operador punto (**.**) de la siguiente manera: primero, se escribe el nombre de la estructura externa, a continuación se escribe su miembro (estructura interna), y por último el miembro de la estructura interna; separados cada uno por el operador punto. Por ejemplo, para acceder al miembro **nom** de la estructura **empleos[0]**, se haría:

```
empleos[0].conductor.nom
```

Si se desea leer todo el arreglo de estructuras anidadas **empleos**, se haría:

```
for (i = 0; i < 12; i++){
    gets (empleos[i].conductor.nom);
    gets (empIeos[i].conductor.apeII);
    gets (empIeos[i].comida);
    gets (empIeos[i].trabajo);
    scanf ("%f". &empIeos[i].gana);
}
```

El estándar ANSI especifica que las estructuras pueden anidarse al menos hasta 15 niveles.

Inicialización de estructuras anidadas

La inicialización de estructuras anidadas se realiza en forma similar a la inicialización de una estructura. Por ejemplo, para inicializar la estructura **Ruiz** de tipo **empleado**, se haría:

```
static struct empleado Ruiz = {
{
    {"Pedro","Ruiz"}, /* Estructura interna. */
    "zanahorias",
    "chofer de bus",
    300.00
};
```

10.1.7. Punteros a Estructuras

En lenguaje C se permite tener punteros a estructuras, igual que se permite punteros a cualquier otro tipo de variables.

Se utiliza punteros a estructuras por tres razones: (SCHILDT, 1994)

1. Facilitar el manejo de las estructuras, al igual que sucedía con los punteros a arreglos.

2. Para pasar punteros a estructuras como argumentos de una función. Es decir, generar una llamada a una función, con un argumento de una estructura por "***dirección***" para poder cambiar su contenido. (Se verá más adelante en este capítulo.)

3. Para realizar representaciones de datos mediante asignación dinámica de memoria, que son estructuras que contienen punteros a otras estructuras. Es decir, crear listas enlazadas, árboles y otras estructuras de datos dinámicas utilizando el sistema de asignación dinámica. (No será tratado en este texto.)

Declaración de un puntero a estructura

La declaración de un puntero a estructura se lo hace de la misma forma que la utilizada en otras declaraciones de punteros ya vistas. Es decir, en primer lugar se coloca la palabra clave **struct**, después la etiqueta del patrón y por último un ***** seguido del nombre del puntero. Por ejemplo, en la siguiente declaración:

```
struct empleado *este, empleos[12];
```

El puntero **este** puede apuntar a cualquier estructura de tipo **empleado**. Por ejemplo, con el arreglo de estructuras **empleos** declarada anteriormente, se tendría:

```
este = empleos;
/* Inicializa un puntero a la primera estructura del arreglo. */
```

Al sumar 1 a **este** el resultado apuntaría al siguiente elemento del arreglo de estructuras, porque se suman a la dirección del puntero **este** los 88 bytes que ocupa cada estructura **empleado**: 21 para el **nom**, 21 para **apell**, 21 para **comida**, 21 para **trabajo** y 4 para **gana**. Entonces, para acceder a los elementos del arreglo de estructuras mediante aritmética de punteros, se haría:

este apunta a **empleos[0]**
este+1 apunta a **empleos[1]**

Para encontrar la dirección de una variable estructura se coloca el operador **&** antes del nombre de la estructura. Por ejemplo, para asignar la dirección de la primera estructura de **empleos** en la variable puntero **este**, se haría:

```
este = &empleos[0];
```

Acceso a los miembros de la estructura mediante un puntero

Para acceder a un miembro de una estructura mediante un puntero, existen dos métodos: (SCHILDT, 1994)

1. **Se utiliza un nuevo operador llamado *flecha (->)*** de la siguiente manera: un "puntero a estructura" seguido del operador flecha, funciona exactamente igual que un nombre de estructura seguido del operador punto (**.**). Este método es más común. Por ejemplo, si se declara:

   ```
   struct empleado *ptr, manual;
   ```

   y luego se asigna la dirección de la estructura **manual** al puntero **ptr**:

   ```
   ptr = &manual;
   ```

   Entonces, el acceso a los miembros de la estructura **manual** mediante el puntero **ptr**, sería:

   ```
   ptr->conductor.nom
   ptr->conductor.apell
   ptr->comida
   ```

```
ptr->trabajo
ptr->gana
```

El acceso a los miembros de un arreglo de estructuras con un puntero, se realiza en forma similar. Por ejemplo, imprimir el arreglo de estructuras **empleos**, mediante el puntero **este**:

```
for (i = 0, este = empleos; i < 12; i++, este++) {
    puts (este->conductor.nom);
    puts (este-›conductor.apelI);
    puts (este-›comida);
    puts (este->trabajo);
    printf ("%f", este->gana);
}
```

donde:

- **este**, es un puntero.
- **este->gana**, es un "miembro de la estructura" apuntada por el puntero **este**, que es simplemente una variable de tipo **float**.

2. **Se utiliza el operador *indirección (*)***. Este segundo método especifica el contenido del "puntero a estructura". Por ejemplo, con la declaración:

```
struct empleado *ptr, manual;
```

si se asigna:

```
ptr = &manual;
```

se cumplirá que la siguiente comparación es verdadera:

```
*ptr == manual      /* estructura completa. */
```

Esto se debe a que **&** y ***** son dos operadores recíprocos, por lo tanto se cumple lo siguiente:

```
(*ptr).gana == manual.gana   /* miembro gana. */
```

En la expresión **(*ptr).gana**, es necesario el uso de paréntesis, porque el operador punto tiene mayor precedencia que el operador *****.

En resumen, si se define **ptr** como puntero a la estructura **manual**, se cumple la siguiente equivalencia:

manual.gana == (*ptr).gana == ptr->gana

NOTA: Nótese que cuando se accede a un miembro de una estructura a través de un puntero a estructura, se debe usar el operador flecha en lugar del operador punto.

Ejercicio

El siguiente programa inicializa estructuras anidadas de los tipos **nombres** y **empleado** descritas anteriormente, y luego imprime las direcciones y miembros de dichas estructuras, mediante el operador flecha y el operador de indexación.

```
/* PROG1002.C */

/* Puntero a estructura. */

#include "stdio.h"

#define LEN 21

struct nombres {
  char nom[LEN];
  char apell[LEN];
};

struct empleado {
  struct nombres conductor;
  char comida[LEN];
  char trabajo[LEN];
  float gana;
} empleos[2] = {
    {
      {"Pedro", "Ruiz"},
      "Zanahorias",
      "Chofer de bus",
      300000.00
      },
    {
      {"José,","Díaz"},
      "salmón ahumado",
      "programador",
      250000.00
    }
  };
```

```
void main ()
{
  struct empleado *este;
  extern struct empleado empleos[];   /* Declaración opcional. */

  printf ("Dirección 1: %p;   2: %p\n", &empleos[0], &empleos[1]);
  este = &empleos[0];
  printf ("Puntero   1: %p;   2: %p\n\n", este, este+1);
  printf ("este->gana vale: %.2f;   (*este).gana vale: %.2f\n",
  este->gana ,(*este).gana);
  este ++;
  printf ("este->comida es: %s;   este->nombres.apell es: %s\n",
  este->comida, este->conductor.apell);
}
```

La salida del programa podría ser:

Dirección 1: 00AA; 2: 0102
Puntero 1 : 00AA; 2: 0102

*este->gana vale: 300.00; (*este). gana vale: 300.00*
este->comida es: salmón ahumado; este->nombres.apell es: Díaz

10.1.8. Paso de Estructuras a Funciones

Los argumentos que se pueden pasar a una función son "valores" simples de tipo **char**, **int**, **float** o **puntero**, además pueden ser estructuras o punteros a estructuras.

Existen cuatro métodos para enviar a una función información acerca de una estructura, los cuales se describen a continuación: (SCHILDT, 1994)

10.1.8.1. Paso de Miembros de una Estructura a Funciones

Cuando se pasa un miembro de estructura a una función, se está realmente pasando el valor de ese miembro a la función, porque los miembros de la estructura son variables simples, como los tipos **int**, **char**, **float**, **double** o **puntero**, a menos que sean miembros complejos como un arreglo de caracteres.

Ejercicio

Realizar un análisis financiero, el cual suma las cantidades depositadas por un cliente en su cuenta corriente y libreta de ahorros, que están en diferentes bancos. La

suma se efectúa en una función con paso de parámetros de los miembros de la estructura.

```
/* PROG1003.C */

/* Paso de miembros de una estructura a función. */

#include "stdio.h"

struct fondos {
  char *corriente;
  float c_corriente;
  char *ahorro;
  float c_ahorro;
};

float suma (float x, float y);

void main ()
{
  struct fondos garcia = {
    "Banco Pacífico",
    102343.00,
    "Banco Pichincha",
    423982.00
  };

  printf ("García tiene un total de: %.2f dólares.\n",
        suma (garcia.c_corriente, garcia.c_ahorro));
}

/* Función que suma dos números float. */
float suma (float x, float y)
{
  return x + y;
}
```

La salida de este programa será:

García tiene un total de: 526.32 dólares.

La función **suma()** no reconoce si los argumentos enviados son miembros de una estructura o no, simplemente requiere que sean del tipo **float**.

Por supuesto, si se desea que una función modifique el valor de un miembro de una estructura del programa de llamada, se debe realizar un paso de parámetros por dirección, enviándose la dirección de dicho miembro. Por ejemplo, la llamada a una función **modifica()** que modifique el estado de la cuenta de ahorro del Sr. García, sería:

```
modifica (&garcia.c_ahorro);
```

10.1.8.2. Paso de Estructuras Completas a Funciones

Cuando se utiliza una estructura como argumento de una función, se pasa la estructura íntegra mediante el uso del método estándar de llamada por "valor". Entonces, todos los cambios realizados en los miembros de esa estructura dentro de la función llamada, no afectan a la estructura utilizada como argumento. Esto evita pasar a la función miembro a miembro de la estructura, en el caso que tenga muchos miembros.

Existe un importante inconveniente en el paso de estructuras a funciones: el tiempo que demanda ingresar y extraer todos los elementos de estructura en el segmento de la pila, especialmente cuando son muchas las estructuras o si algunos de los miembros de las mismas son arreglos. Entonces, el tiempo de ejecución es cada vez más lento, por lo que es preferible pasar a las funciones pocas estructuras con pocos miembros.

Ejercicio

Realizar un análisis financiero, el cual suma las cantidades depositadas por un cliente en su cuenta corriente y libreta de ahorros, que están en diferentes bancos. La suma se efectúa en una función con paso de parámetros de la estructura completa.

```
/* PROG1004.C */

/* Paso de la estructura completa a función. */

#include "stdio.h"

struct fondos {
  char *corriente;
  float c_corriente;
  char *ahorro;
  float c_ahorro;
};

float suma (struct fondos dinero);

void main ()
{
```

```
  struct fondos garcia = {
    "Banco Pacífico",
    102343.00,
    "Banco Pichincha",
    423982.00
  };

  printf ("García tiene un total de: %.2f dólares.\n", suma (garcia));
}

/* Función que suma dos números float. */

float suma (struct fondos dinero)
{
  return dinero.c_corriente + dinero.c_ahorro;
}
```

La salida de este programa será idéntica al anterior:

García tiene un total de: 526.32 dólares.

Debe tenerse en cuenta, que el tipo de parámetro de la función **suma()** tendrá que corresponder con el tipo de argumento de la llamada a la función.

10.1.8.3. Paso de la Dirección de la Estructura a Funciones

Cuando se pasa a una función un puntero a estructura, solo se guarda la dirección de la estructura en el segmento de la pila, y no la estructura completa. El paso de la dirección de la estructura a una función, tiene dos ventajas:

1. La llamada a la función es muy rápida, porque solo se pasa la dirección de la estructura.

2. Cambia los contenidos de los miembros reales de la estructura de llamada.

Ejercicio

Realizar un análisis financiero, el cual suma las cantidades depositadas por un cliente en su cuenta corriente y libreta de ahorros, que están en diferentes bancos. La suma se efectúa en una función con paso de parámetros de la dirección de la estructura.

```
/* PROG1005.C */
```

```
/* Paso de las direcciones de estructuras a función. */

#include "stdio.h"

struct fondos {
  char *corriente;
  float c_corriente;
  char *ahorro;
  float c_ahorro;
};

float suma (struct fondos *dinero);

void main ()
{
  struct fondos garcia = {
    "Banco Pacífico",
    102343.00,
    "Banco Pichincha",
    423982.00
  };

  printf ("García tiene un total de: %.2f dólares.\n", suma (&garcia));
}

/* Función que suma dos números float. */

float suma (struct fondos *dinero)
{
  return dinero->c_corriente + dinero->c_ahorro;
}
```

La salida de este programa será idéntica a la anterior:

García tiene un total de: 526.32 dólares.

La función **suma()** tiene el parámetro **dinero** que es un puntero de tipo estructura **fondos**, que sirve para recibir en la función la dirección de la estructura **garcia**. Entonces, para acceder a la estructura **garcia** mediante el puntero **dinero**, se debe utilizar el operador **->** para acceder a los miembros **c_corriente**, **c_ahorro** y al resto de miembros de dicho estructura.

NOTAS:

- El nombre de la estructura no es sinónimo de una dirección, por lo que se debe emplear el operador **&** para indicar su dirección.

- Se debe utilizar el operador punto para acceder a los miembros de la estructura cuando se trabaja directamente sobre ella, y cuando se tiene un puntero a una estructura se debe utilizar el operador ***flecha***.

10.1.8.4. Paso de un Arreglo de Estructuras a Funciones

Este método se aplica a un arreglo de estructuras. Como ya se conoce que el nombre de un arreglo es la dirección a su primer elemento, puede ser pasado a una función, igual que cualquier otro arreglo.

Ejercicio

Realizar un análisis financiero, el cual suma las cantidades depositadas por dos clientes en sus cuentas corrientes y libretas de ahorros, respectivamente, que están en diferentes bancos. La suma se efectúa en una función con paso de parámetros de un arreglo de estructuras, usando punteros.

```
/* PROG1006.C */

/* Paso de un arreglo de estructuras a función. */

#include "stdio.h"

struct fondos {
  char *corriente;
  float c_corriente;
  char *ahorro;
  float c_ahorro;
};

float suma (struct fondos *dinero);

void main ()
{
  struct fondos garcias[2] = {
    {
      "Banco Pacífico",
      102343.00,
      "Banco Pichincha",
      423982.00
    },
```

```
    {
      "Banco Rumiñahui",
      97657.00,
      "Banco Continental",
      176018.00
    }
  };

  printf ("Garcías tiene un total de: %.2f dólares.\n", suma (garcias));
}

/* Función que suma números float. */

float suma (struct fondos *dinero)
{
  int i;
  float total;

  for (i = 0, total = 0; i < 2; i++, dinero++)
    total += dinero->c_corriente + dinero->c_ahorro;

  return total;
}
```

La salida de este programa será:

Garcías tiene un total de: 800 dólares.

El nombre del arreglo **garcias** es un puntero a arreglo, que apunta al primer elemento del mismo, correspondiente a la estructura **garcias[0]**. Así, el puntero **dinero** en la función **suma()** se asigna inicialmente la dirección:

```
dinero == &garcias[0]
```

Otra alternativa para implementar la función **suma()** del ejercicio inmediato anterior, utilizando indexación de los elementos del arreglo, sería:

```
/* Función que suma números float. */
float suma (struct fondos dinero[])
{
      int i;
      float total;

      for (i = 0, total = 0; i < 2; i++)
```

```
            total += dinero[i].c_corriente + dinero[i].c_ahorro;
        return total;
    }
```

A continuación se plantea un ejercicio que utiliza la mayoría de los conceptos vistos hasta este momento.

Ejercicio

Realizar un programa para crear una lista de correos, que utilice un arreglo de estructuras para guardar la información de cada persona, la misma que incluye el nombre, la calle, la ciudad, la provincia y el código postal.

El programa debe tener un menú para introducir los datos de una persona, borrar un nombre y listar el archivo. Además, el programa asume que un elemento del arreglo no está en uso si el campo nombre está vacío.

```
/* PROG1007.C */

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

#define MAX 100

struct dir {
  char nombre[20];
  char calle[15];
  char ciudad[15];
  char provincia[15];
  unsigned long int codigo;
};

void inic_lista (struct dir info_dir[]), entrar (struct dir info_dir[]);
void borrar (struct dir info_dir[]), listar (struct dir info_dir[]);
int menu (void), buscar_libre (struct dir info_dir[]);

void main (void)
{
  int opcion;
  struct dir info_dir[MAX];

  inic_lista (info_dir); /* Inicializar el arreglo de estructuras. */
  for (;;) {
    opcion = menu ();
```

```
    switch (opcion) {
      case 0 : exit (0);
               break;
      case 1 : entrar (info_dir);
               break;
      case 2 : borrar (info_dir);
               break;
      case 3 : listar (info_dir);
    }
  }
}

/* Inicializar la lista (arreglo de estructuras) poniendo un    */
/* caracter nulo en el primer byte de cada nombre.              */

void inic_lista (struct dir info_dir[])
{
  register int t;

  for (t = 0; t < MAX; ++t)
    info_dir[t].nombre[0] = '\0';
}

/* Muestra los mensajes de las opciones y devuelve la opción   */
/* del usuario.                                                 */

menu (void)
{
  char s[80];
  int c;

  printf ("0   Salir.\n");
  printf ("1   Introducir un nombre.\n");
  printf ("2   Borrar un nombre.\n");
  printf ("3   Listar archivo.\n");
  do {
    printf ("\nIntroduzca una opción... ");
    gets (s);
    c = atoi (s);
  } while (c < 0 || c > 3);
  return c;
}

/* Introducir los datos en cada estructura en la lista. */
```

```
void entrar (struct dir info_dir[])
{
  int sitio;
  char s[80];

  sitio = buscar_libre (info_dir); /* Busca el sitio de inserción. */
  if (sitio == -1) {
    printf ("lista llena");
    return;
  }
  printf ("Introduzca el nombre    : ");
  gets (info_dir[sitio].nombre);
  printf ("Introduzca la calle     : ");
  gets (info_dir[sitio].calle);
  printf ("Introduzca la ciudad    : ");
  gets (info_dir[sitio].ciudad);
  printf ("Introduzca la provincia : ");
  gets (info_dir[sitio].provincia);
  printf ("Introduzca el código    : ");
  gets (s);
  info_dir[sitio].codigo = strtoul (s, '\0', 10);
  /* La función strtoul() convierte una cadena apuntada por s en un */
  /* unsigne long.                                                   */
 }

/* Buscar una estructura no usada. */

buscar_libre (struct dir info_dir[])
/* Obtiene la posición a insertar, pero si la lista est  llena da -1. */
{
  register int t;

  for (t = 0; info_dir[t].nombre[0] && t < MAX; ++t);
  if (t == MAX)
    return - 1;  /* No hay sitio libre. */
  return t;
 }

/* Eliminar una dirección. */

void borrar (struct dir info_dir[])
{
  register int sitio;
```

```
  char s[80];

  printf ("Introduzca número de registro < 0 a 99>: ");
  gets (s);
  sitio = atoi (s);
  if(sitio >= 0 && sitio < MAX )
    info_dir[sitio].nombre[0] = '\0';
}

/* Mostrar la lista en la pantalla. */

void listar (struct dir info_dir[])
{
  register int t;

  puts ("\n\nLISTADO DE PERSONAS\n");
  printf ("NOMBRE          CALLE       CIUDAD       PROVINCIA");
  puts ("     CODIGO");
  for (t = 0; t < MAX; ++t)
    if (info_dir[t].nombre[0]) {
      printf ("%-20s", info_dir[t].nombre);
      printf ("%-15s", info_dir[t].calle);
      printf ("%-15s", info_dir[t].ciudad);
      printf ("%-15s", info_dir[t].provincia);
      printf ("%-10lu\n", info_dir[t].codigo);
    }
  printf ("\n\n");
}
```

La salida del programa dependerá del siguiente menú: 1 Introducir un nombre, 2 Borrar un nombre, 3 Listar archivo y 0 para terminar el programa.

10.1.9. Nuevos formatos de datos

Uno de los usos más importantes de las estructuras es la creación de nuevos formatos de datos, porque resultan mucho más eficientes que utilizar arreglos o estructuras simples. Estos nuevos formatos se obtienen con la palabra clave **typedef**, que se será estudiada más adelante.

Estos formatos son tales como las pilas, colas, árboles binarios, tablas y grafos, los mismos que se construyen a partir de estructuras encadenadas. Cada estructura encadenada contiene uno o más datos y, además, uno o un par de punteros que apuntan a otras estructuras del mismo tipo. Estos punteros sirven para encadenar una estructura

a la siguiente en tiempo de ejecución, y facilitan un camino que permita un rastreo por la estructura global.

10.2. CAMPOS DE BITS

Los "campos de bits" están basados en la estructura y sirven para acceder a un bit individual dentro de un byte. Esto puede ser útil por distintas razones:

1. Si es limitada la memoria, las variables lógicas pueden almacenarse en un mismo byte, porque éstas solamente pueden ser unos y ceros, representadas por un bit.

2. Porque ciertos dispositivos transmiten la información codificada en bits dentro de bytes.

3. Porque ciertas rutinas de cifrado necesitan acceder a los bits dentro de los bytes.

Aunque todas estas tareas se pueden realizar utilizando los bytes y los operadores a nivel de bits, los "campos de bits" aportan eficiencia del código.

"***Los campos de bits***" son realmente un tipo especial de elementos de estructura que definen su tamaño en bits.

La forma general de definición de los "campos de bits" es: (SCHILDT, 1994)

```
struct etiqueta {
    tipo nombre_1 : longitud;
    tipo nombre_2 : longitud;
    tipo nombre_N : longitud;
} lista_variables;
```

Cada "campo de bits" debe declararse como **int**, **unsigned** o **signed**, y el largo de cada campo depende del compilador, siendo en este caso de 1 a 16 bits de largo. Entonces, los "campos de bits" de longitud 1 deben declararse como **unsigned**, ya que un solo bit no puede tener signo.

El acceso a cada "campo de bits" es exactamente igual al acceso a un miembro de la estructura. Para esto se utiliza el operador punto (**.**), pero si se referencia la estructura a través de un puntero, se usa el operador flecha (**->**).

Los "campos de bits" se utilizan frecuentemente cuando se analiza la entrada de un dispositivo hardware. Por ejemplo, el puerto de entrada de un adaptador de comunicaciones serie puede devolver el "byte de estado" organizado de la siguiente forma:

BIT	Significado del Bit Cuando está Activado
0	Cambio en la línea, listo para enviar
1	Cambio en datos listos
2	Detección de final
3	Cambio en la línea de recepción
4	Listo para enviar
5	Datos listos
6	Llamada telefónica
7	Señal recibida

Se puede representar esta información en un "byte de estado" mediante los siguientes "campos de bits":

```
struct tipo_estado {
    unsigned delta_cts  : 1;
    unsigned delta_dsr : 1;
    unsigned tr_final    : 1;
    unsigned delta_rec  : 1;
    unsigned cts          : 1;
    unsigned dsr          : 1;
    unsigned ring                 : 1;
    unsigned linea                : 1;
} estado;
```

Para permitir que un programa determine el envío o la recepción de datos, se puede hacer lo siguiente:

```
estado = obt_estado_puerto ();     /* Determina el estado del puerto. */
if (estado.cts)
    printf ("Listo para enviar.\n")
if (estado.dsr)
    printf ("Datos listos.\n");
```

Si no se desea realizar el envío de datos, se asignaría en el campo **cts** el 0, así:

```
estado.cts = 0;
```

No es necesario especificar los bits que no se van a usar, en este caso el campo se utiliza en la estructura "sin nombre" como relleno. Por ejemplo, si solo interesan los bits **cts** y **drs**, se podría declarar la estructura **tipo_estado** como sigue:

```
struct tipo_estado {
```

```
        unsigned      : 4;
        unsigned cts : 1;
        unsigned dsr: 1;
    } estado;
```

Esta declaración usa como relleno un campo de 4 bits sin nombre, en estos cuatro bits no se puede almacenar nada. En la Figura 10.4 se muestra en memoria la variable **estado**:

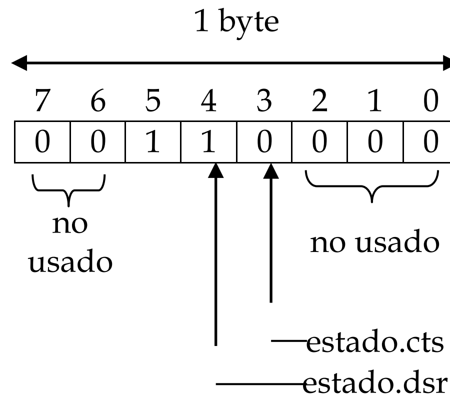


Figura 10.4 . Memoria de la variable **estado**

Además, cuando no alcanza un campo en la unidad de almacenamiento, generalmente de 2 bytes, se almacenan los campos de bits faltantes en otra unidad de almacenamiento.

Por otro lado, un "campo de bits" sin nombre con ancho cero, se utiliza para alinear el siguiente "campo de bits" en el límite de la nueva unidad de almacenamiento (al último de la unidad de almacenamiento). Por ejemplo, la definición de la siguiente estructura, alinea **dsr** con el límite de la siguiente unidad de almacenamiento:

```
    struct tipo_estado {
        unsigned      : 4;
        unsigned cts : 1;
        unsigned      : 0;
        unsigned dsr: 1;
    } estado;
```

Por último, se pueden mezclar elementos de estructuras normales con elementos de "campos de bits". Por ejemplo, para definir un registro de empleados que utilice solo un byte, la declaración de los siguientes tres elementos de sus cinco elementos de información:

- El estado del empleado (bajo o activo)
- Si el empleado es fijo o por horas.

- El porcentaje de deducciones de impuesto.

se tendría la implementación de la estructura de los empleados como sigue:

```
struct empleado {
    struct dir direccion; /* Estructura dirección del ejercicio PROG1007.C. */
    float gana;
    unsigned baja : 1;          /* Estado de baja o activo. */
    unsigned por_horas : 1;     /* Fijo o por horas. */
    unsigned deducciones : 1;   /* Deducciones de impuestos. */
};
```

Pero, sin utilizar los campos de bits, la información de los tres últimos elementos habría que almacenarse en 3 bytes, aumentando en dos bytes su almacenamiento.

Entonces, los campos de bits permiten una mejor utilización de la memoria, al almacenar datos en el mínimo número de bits requeridos. La razón de esto es que el lenguaje C proporciona la capacidad de especificar o definir el número de bits en el cual se almacenan un miembro **unsigned** o **int** de una ***estructura*** o de una ***union***.

Restricciones de los "campos de bits".

1. No se puede determinar la dirección de una variable de campos de bits.

2. No se puede construir arreglos de variables de campos de bits.

3. No se puede solapar los límites enteros.

4. No se puede saber, de máquina a máquina, si los campos se dispondrán de derecha a izquierda o viceversa. Es decir, cualquier código que use campos de bits puede tener algunas dependencias con la máquina.

Ejercicio

Realizar un programa para almacenar e imprimir las 52 cartas de un naipe en un formato de dos columnas. El grupo de cartas se debe representar como un arreglo de estructuras de "*campos de bits*".

La siguiente definición contiene tres campos de bits utilizados, para representar una carta de un grupo de 52 cartas:

```
struct carta_bit {
    unsigned cara  : 4;   /* cara está almacenado en 4 bits. */
    unsigned palo  : 2;   /* palo está almacenado en 2 bits. */
```

```
    unsigned color : 1;    /* color está almacenado en 1 bit. */
};
```

La constante entera representa el ancho del campo y el número de bits en el cual queda almacenado el miembro, esta constante debe ser un entero entre 0 y el número total de bits utilizados para almacenar un **int** o un **unsigned** en el sistema utilizado.

El número de bits se basa en el rango deseado de valores correspondiente a cada miembro de estructura:

- El miembro **cara** almacena valores entre 0 (As) y 12 (Rey), por lo que se necesitan 4 bits (valor entre 0 y 15).
- El miembro **palo** almacena valores entre 0 y 3 (0=Diamantes, 1=Corazones, 2=Tréboles y 3= Espadas), por lo que se necesitan 2 bits (valor entre 0 y 3).
- El miembro **color** almacena valores entre 0 (Rojo) y 1 (Negro), por lo que se necesita 1 bit (valor entre 0 y 1).

El programa también debe crear el arreglo **caja**, que contiene 52 estructuras de tipo **struct carta_bit**. La función **caja_llena()** inserta las 52 cartas en el arreglo **caja**, y la función **cantidad()** imprime las 52 cartas.

```
/* PROG1008.C */

#include <stdio.h>

struct carta_bit {
  unsigned cara : 4;
  unsigned palo : 2;
  unsigned color: 1;
};

void caja_llena (struct carta_bit *wcaja);
void cantidad (struct carta_bit * wcaja);

void main ()
{
  struct carta_bit caja[52];

  caja_llena (caja);
  cantidad (caja);
}

void caja_llena (struct carta_bit *wcaja)
{
```

```
  int i;

  for (i = 0; i < 52; i++) {
    wcaja[i].cara = i % 13;
    wcaja[i].palo = i / 13;
    wcaja[i].color = i / 26;
  }
}

/* Imprime en 2 columnas: índice i las cartas del 0 a 25 e */
/* índice j las cartas del 26 a 51.                        */

void cantidad (struct carta_bit * wcaja)
{
  int i, j;

  for (i = 0, j = i + 26; i < 26; i++, j++) {
    printf ("carta:%3d palo:%2d color:%2d  -  ",
          wcaja[i].cara, wcaja[i].palo, wcaja[i].color);
    printf ("carta:%3d palo:%2d color:%2d\n",
          wcaja[j].cara, wcaja[j].palo, wcaja[j].color);
  }
}
```

La salida del programa sería:

carta: 0 palo: 0 color: 0 - carta: 0 palo: 2 color: 1
carta: 1 palo: 0 color:0 - carta: 1 palo: 2 color: 1
carta: 2 palo: 0 color: 0 - carta: 2 palo: 2 color: 1
carta: 3 palo: 0 color:0 - carta: 3 palo: 2 color: 1
carta: 4 palo: 0 color:0 - carta: 4 palo: 2 colar: 1
carta: 5 palo: 0 color: 0 - carta: 5 palo: 2 color: 1
carta: 6 palo: 0 color: 0 - carta: 6 palo: 2 color: 1
carta: 7 palo: 0 color:0 - carta: 7 palo: 2 color: 1
cana: 8 palo: 0 color:0 - cana: 8 palo: 2 color: 1
carta: 9 palo: 0 color:0 - carta: 9 palo: 2 color: 1
carta: 10 palo: 0 color: 0 - carta: 10 palo: 2 color: 1
carta: 11 palo: 0 color: 0 - carta: 11 palo: 2 color: 1
cana: 12 palo: 0 color: 0 - carta: 12 palo: 2 color: 1
carta: 0 palo: 1 color: 0 - carta: 0 palo: 3 color: 1
carta: 1 palo: 1 color: 0 - caña: 1 palo: 3 color: 1
carta: 2 palo: 1 color:0 - carta: 2 palo: 3 color: 1
carta: 3 palo: 1 color:0 - cana: 3 palo: 3 color: 1
cana: 4 palo: 1 color:0 - carta: 4 palo: 3 color: 1

cana: 5 palo: 1 coIor:0 - carta: 5 palo: 3 color: 1
carta: 6 palo: 1 color: 0 - carta: 6 palo: 3 color: 1
carta: 7 palo: 1 color: 0 - carta: 7 palo: 3 color: 1
cana: 8 palo: 1 color: 0 - carta: 8 palo: 3 color: 1
cana: 9 palo: 1 color: 0 - carta: 9 palo: 3 color: 1
carta: 10 palo: 1 color: 0 - carta: 10 palo: 3 color: 1
carta: 11 palo: 1 color: 0 - carta: 11 palo: 3 color: 1
carta: 12 palo: 1 color: 0 - carta: 12 palo: 3 color: 1

10.3. UNIONES

Una unión es una porción de memoria que es compartida por dos o más variables diferentes, pero no simultáneamente.

Las uniones se preparan de forma muy semejante a las estructuras; existiendo el patrón y las variables de las uniones correspondiente a ese patrón. También se puede definir una variable **union** junto con el patrón, o bien utilizando una etiqueta correspondiente a la **union**. Es decir, todo lo expuesto para las "estructuras" respecto a la declaración, asignación, acceso a miembros, y paso de argumentos a funciones, es similar para las "uniones".

La forma general de definición de la **union** es: (SCHILDT, 1994)

```
union etiqueta {
    tipo nombre_variable;
    tipo nombre_variable;
    …
} variables_union;
```

Por ejemplo, a continuación se presenta un patrón de una **union**, con la etiqueta **toma**:

```
union toma {
    int numero;
    float grande;
    char letra;
};
```

La definición de variables de la union de tipo toma sería:

```
union toma ficha;          /* Variable union de tipo toma. */
union toma guarda[10];   /* Arreglo de 10 variables union. */
union toma *pu;             /* Puntero a una variable toma. */
```

También se puede declarar una variable de tipo **union**, colocando su nombre al final de la definición del patrón.

En la variable **ficha**, tanto el entero **numero**, el punto flotante **grande** y el caracter **letra**, comparten la misma porción de la memoria. En la Figura 10.5 se muestra como es compartida esa porción de la memoria, por los miembros **numero**, **grande** y **letra** de la variable unión **ficha**:

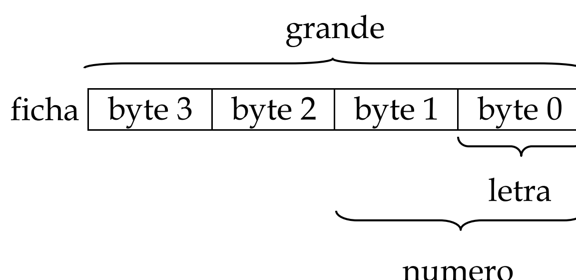


Figura 10.5. Unión **ficha** *compartida la memoria por sus miembros*

El compilador prepara espacio suficiente para almacenar el dato más grande de la **union**. En el ejemplo de la unión **toma**, el dato de mayor tamaño entre los miembros especificados depende del sistema que se está estudiando, en este caso es 4 bytes. Entonces, el número de bytes requeridos para las variables uniones declaradas anteriormente, sería:

- En la variable **ficha**, 4 bytes.
- En el arreglo **guarda** de 10 elementos, 4 bytes para cada elemento.

Para acceder a un miembro de una **union** se utiliza la misma sintaxis utilizada para las estructuras, mediante los operadores punto y flecha. Entonces, el acceso a un miembro de una **union** permite especificar el tipo de dato que se está utilizando en ese momento. Por ejemplo, asignar valores a cada miembro de la unión **ficha**:

```
ficha.numero = 23;  /* Se guarda 23 en ficha usando 2 bytes. */
ficha.grande = 2;     /* Borra el 23, guarda 2 usando 4 bytes. */
ficha.letra = 'h';       /* Borra el 2, guarda 'h' usando 1 byte. */
```

Se debe tener en cuenta que solo se guarda un valor en cada momento, no se pueden almacenar varios valores a la vez, incluso si hubiera espacio suficiente para ello.

Entonces, se debe llevar en cuenta el tipo de dato que se está usando en la **union**, porque se puede cometer errores si el contenido es distinto al tipo que se está usando.

Se emplea el operador **->** con uniones, de igual manera como se hacía con las estructuras. Por ejemplo, acceder al miembro **numero** de la unión **ficha**, mediante un puntero:

```
int x;
pu = &ficha;
x = pu->numero;    /* Equivale a x = ficha.numero; */
```

NOTAS:

- Si se está trabajando directamente con la **union**, se usa el operador punto.

- Si se accede a la variable **union** a través de un puntero, se utiliza el operador flecha.

Aplicaciones

1. Se utiliza una **union** para la creación de una tabla que guarde una mezcla de diferentes tipos. Esta tabla se crea mediante un arreglo de uniones del mismo tamaño, en las que cada una de ellas contiene datos de un tipo distinto.

2. El uso de una **union** ayuda a la creación de códigos independientes de la máquina, porque el compilador sigue la pista de los tamaños actuales de las variables **union**.

3. Las uniones se utilizan frecuentemente donde se necesiten conversiones de tipos, como en el paso de información a las funciones o en el acceso de los datos en memoria.

Ejercicio

Realizar un programa que tenga una función llamada **escribir()** que escriba un número entero, byte a byte. El argumento de la función debe ser un puntero a **union** que tenga los siguientes miembros: un número de tipo **int** y un arreglo de 2 elementos de tipo **char**.

El programa debe leer el número entero en una variable **union**, y en la función se accede a este número, elemento por elemento de tipo **char**, para imprimirlo byte a byte.

```
/* PROG1009.C */

#include "stdio.h"

union pw {
 int i;
 char c[2];
};
```

```
void escribir (union pw *palabra);

void main ()
{
  union pw num;

  printf ("Ingrese un número entero: ");
  scanf ("%d", &num.i);
  printf ("la salida byte a byte ser :\n");
  printf ("byte MS|byte mS \n");
  escribir (&num);
  putchar ('\n');
}

void escribir (union pw *palabra)
{
  printf ("%-8c", palabra->c[1]);  /* Escribe la primera mitad. */
  printf ("%-8c", palabra->c[0]);  /* Escribe la segunda mitad. */
}
```

La salida del programa podría ser:

Ingrese un número entero: 12353 <ENTER>
La salida byte a byte será:
byte MS|byte Ms
0 A

Se obtiene este resultado porque el número 12353 se almacena en memoria en binario, como se muestra en la Figura 10.6.

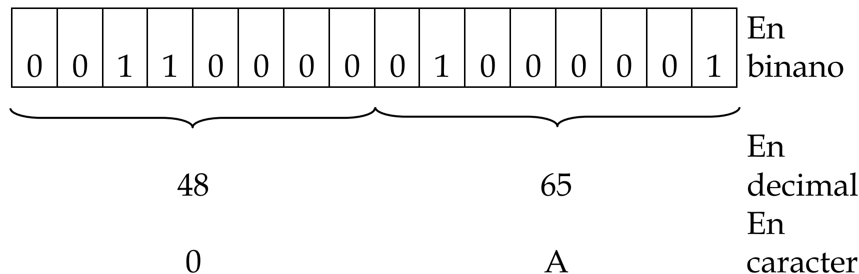


Figura 10.6. Representación gráfica del número 12353

Además, una **union** puede ser miembro de una estructura y una estructura puede ser miembro de una **union**. También, las estructuras y las uniones pueden ser mezcladas libremente con arreglos. Por ejemplo, en las siguientes declaraciones:

```
union identifica {      /* Puede representar el color o la talla. */
    char color[12];
    int talla;
};

struct ropa {
    char fabricante[20];
    float costo;
    union identifica descripcion;
} camisa, blusa;
```

Las variables **camisa** o **blusa** contendrán los siguientes miembros: la cadena **fabricante**, el valor de punto flotante **costo**, y una unión **descripcion** que puede representar el **color** o la **talla**.

Para acceder a los miembros anidados de la variable unión **camisa**, se haría:

```
camisa.descripcion.color
camisa.descripcion.talla
```

Por último, en una variable **union** sólo es posible inicializar su primer miembro con un valor del mismo tipo. Por ejemplo, al inicializar la estructura **pantalon**:

```
static struct ropa pantalon = {"LEVIS", 150000.00, "azul"};
```

la cadena "azul" es el valor inicial al primer miembro color de la unión **pantalon**.

10.4. ENUMERACIONES

Una enumeración es un conjunto de constantes enteras positivas con nombre, que especifica todos los valores válidos que una variable de este tipo puede tener.

Es decir, cada una de las constantes de la enumeración corresponden a un valor entero, por lo que pueden usarse en cualquier expresión entera. El tipo enumeración fue incluido por el estándar ANSI. Las enumeraciones se definen de forma parecida a las estructuras: con la palabra clave **enum** que indica el comienzo de un tipo enumerado.

La forma general de definición de una **enum** es: (SCHILDT, 1994)

```
enum etiqueta {lista_enumeraciones} lista_variables;
```

donde:

- **etiqueta** y **lìsta_variables**, son opcionales como en las "estructuras".

De igual forma que en las estructuras, se usa el nombre de la etiqueta para declarar variables del tipo **enum**. Por ejemplo, una enumeración de las monedas de USA sería:

```
enum moneda {penique, niquel, diez_centavos, cuarto, medio_dolar, dolar};
enum moneda dinero;
```

Dada la definición de la enumeración **moneda** y la declaración de la variable **dinero** de tipo **enum**, las siguientes sentencias son válidas:

```
dinero = diez_centavos;       /* Asignación. */
if (dinero == cuarto)         /* Comparación. */
    printf ("Es un cuarto de dólar.\n");
```

Cada constante recibe un valor entero que es mayor en uno que el valor de la constante anterior. El valor de la primera constante de la enumeración es 0. Por ejemplo, la siguiente sentencia:

```
printf ("%d %d", penique, diez_centavos);   /* Impresión. */
```

imprimirá:

0 2

Se puede especificar el valor de una o más constantes utilizando un inicializador: poniendo después de la constante un igual y el valor entero. A las constantes que siguen a una inicializada, se les asigna valores mayores en uno que el valor previo inicializado, pero solamente si esta constante no es inicializada. Por ejemplo, la enumeración:

```
enum moneda {penique, niquel, diez_centavos=10, cuarto=25,
                    medio_dolar, dolar=100};
```

Tendrán los siguientes valores para las constantes enteras de la enumeración **moneda**:

```
penique         == 0
niquel          == 1
diez_centavos   == 10
cuarto          == 25
medio_dolar     == 26
dólar           == 100
```

Lectura y escritura de enumeraciones

No se pueden ni leer ni escribir directamente las enumeraciones, porque las constantes son solo nombres para enteros; no son enteros ni cadenas. Esto hace que la creación del código de entrada y salida sea un poco tediosa. En el ejemplo anterior:

- ***Para leer***, la enumeración **moneda** sin inicializar, y almacenar su correspondiente valor en la variable **dinero**, se haría:

```
scanf ("%d", &entero);

switch (entero) {
case 0 : dinero = penique;
    break;
    case 1:  dinero = niquel;
            break;
    case 2:  dinero = diez centavos;
            break;
    case 3:  dinero = cuarto;
            break;
    case 4:  dinero = medio_dolar;
            break;
    case 5:  dinero = dolar;
}
```

- ***Para imprimir***, en palabras la enumeración **moneda** que contiene la variable **dinero**, se haría:

```
switch (dinero) {
    case penique:          printf ("penique");
                          break;
    case niquel:           printf ("níqueI”);
                          break;
    case diez_centavos:  printf ("diez centavos");
                          break;
    case cuarto:           printf ("cuarto");
                          break;
    case medio_doIar:     printf ("medio dólar");
                          break;
    case dólar:            printf ("dólar");
}
```

Es preferible declarar un arreglo de cadenas y utilizar el valor de la enumeración como un índice para traducir un valor de enumeración en su cadena correspondiente; pero esto se hará solamente si no se han usado inicializaciones de

constantes, ya que un arreglo de cadenas se indexa siempre desde 0. Por ejemplo, la impresión anterior sería:

```
char *nombre[] = {
    "penique",
    "niquel",
    "diez_centavos",
    "cuarto",
    "medio_doIar",
    "dolar"
};

printf ("%s", nombre[dinero]);
```

Por último, se pueden emplear las enumeraciones para crear nuevos tipos de datos. Por ejemplo, para crear el tipo de dato **bool** que represente los valores lógicos: **true** y **false**, se haría:

```
enum bool {false, true};
```

Y la declaración de una variable de tipo **bool**, sería:

```
enum bool boolean;
```

Ejercicio

Realizar un programa que use una **union** para pasar información a una función. El programa también debe tener una función para evaluar la fórmula **y = x^n**, donde **x** e **y** son valores en punto flotante y **n** un entero o un punto flotante:

- Si n es entero, puede evaluarse la fórmula multiplicando **x** por sí misma **n** veces.
- Si n es un valor de punto flotante, entonces se puede evaluar con la fórmula: **log y = n log x**, que es lo mismo a **y = e^(n log x)**. En este caso **x** debe ser una cantidad positiva, ya que no existe el logaritmo de cero o de cantidades negativas.

La **union** contiene los dos posibles exponentes en la expresión **fexp** de tipo **float** y **nexp** de tipo entero.

Una estructura debe contener: un miembro de punto flotante x que es la base, un miembro indicador de tipo enumerativo (con **exp_int** que es el exponente entero, y con **exp_float** que es el exponente punto flotante), y **exp** que es el miembro **union** que corresponde al exponente.

```
/* PROG1010.C */

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

enum tipo_exp {exp_float, exp_int};

union nvalor {
  float fexp;            /* Exponente de punto flotante. */
  int nexp;              /* Exponente de entero.        */
};

struct valores {
  float x;               /* Valor para calcular la potencia. */
  enum tipo_exp indicador; /* Indicador de tipo de exponente.  */
  union nvalor exp;       /* union conteniendo el exponente.  */
};

float potencia (struct valores a);

void main ()
{
  struct valores a;
  int i;
  float n, y;

  /* Introducir los datos. */
  printf ("Ingrese el valor de x: ");
  scanf ("%f", &a.x);
  printf ("Ingrese el valor de n: ");
  scanf ("%f", &n);

  /* Determinar el tipo de exponente. */
  i = (int) n;
  a.indicador = (i == n) ? exp_int : exp_float;
  if (a.indicador == exp_int)
    a.exp.nexp = i;
  else
     a.exp.fexp = n;

  /* Elevar x a la potencia adecuada y mostrar el resultado. */
  if (a.indicador == exp_float && a.x < 0.0) {
    printf ("\nERROR, no se puede elevar un número no positivo a una ");
    printf ("potencia en punto flotante\n");
```

```
  }
  else {
    y = potencia (a);
    printf ("\ny = %.4f\n", y);
  }
}

/* Realiza la exponenciación. */

float potencia (struct valores a)
{
  int i;
  float y = 1;

  if (a.indicador == exp_int)     /* Exponente entero. */
    if (a.exp.nexp == 0)
      y = 1.0;                  /* Exponente cero.   */
    else {
      for (i = 1; i <= abs (a.exp.nexp); i++)
        y *= a.x;
      if (a.exp.nexp < 0)
        y = 1. / y;             /* Exponente entero negativo.   */
    }
  else                          /* Exponente en punto flotante. */
    if (a.x)
      y = exp (a.exp.fexp * log (a.x));
    else
      y = 0;
  return y;
}
```

La salida del programa podría ser:

Ingrese el valor de x: 5.1 <ENTER>
Ingrese el valor de n: 2.4 <ENTER>

y = 49.9078

Si se realiza otra corrida con valores enteros, se tiene:

Ingrese el valor de x: 5 <ENTER>
Ingrese el valor de n: 4 <ENTER>

y = 625.0000

10.5. TIPOS DEFINIDOS POR EL USUARIO: *typedef*

typedef permite adicionar un nuevo nombre para un tipo de dato ya existente, con un nombre arbitrario otorgado por el usuario; es decir, realmente no se crea un nuevo tipo de dato. **typedef** se parece a la sentencia de preprocesador **#define** en este aspecto; sin embargo, tiene tres diferencias:

1. Al contrario que **#define**, **typedef** está limitado a otorgar únicamente nombres simbólicos a tipos de datos.

2. **typedef** se ejecuta por compilador, no por preprocesador.

3. Dentro de sus límites, **typedef** es más flexible que **#define**.

La forma general de la sentencia **typedef** es: (SCHILDT, 1994)

typedef tipo nombre;

donde:

- **tipo**, es cualquier tipo de dato ya existente.

- **nombre**, es el nuevo nombre que se adiciona para este tipo; no es un reemplazo del nombre ya existente.

Por ejemplo, para definir el nuevo nombre REAL del tipo **float**, se haría:

typedef float REAL;

A partir de este momento, se puede usar el identificador REAL para definir variables de tipo **float**, porque es el otro nombre del tipo **float**. Por ejemplo, como se muestra a continuación:

REAL x, y[25], *pt;

Además, como ya está definido REAL, puede usarse en otra sentencia **typedef**.

Por ejemplo:

typedef REAL MODIFICADO;

donde:

- MODIFICADO, es el otro nombre de REAL que a su vez es el otro nombre de **float**.

El alcance de esta definición depende de la localización de la sentencia **typedef**.

Si la definición se realiza dentro de una función, el alcance queda confinado a la misma. Si la definición es externa a la función, el alcance es global.

Se acostumbra a usar letras mayúsculas para estas definiciones, con el fin de visualizar que el nombre del tipo es en realidad una abreviatura simbólica.

Las definiciones anteriores con **typedef** podrían realizarse de igual forma con un **#define**. Sin embargo, la definición presentada a continuación no puede hacerse con **#define**:

```
typedef char *STRING;
```

donde:

- STRING, se convierte en un identificador de puntero a **char**.

Por tanto, se puede realizar la siguiente declaración:

```
STRING nombre, signo;
```

que significa:

```
char *nombre, *signo;
```

El uso más común de la sentencia **typedef**, es la creación de nuevos nombres para los tipos de las estructuras. Por ejemplo, para definir el tipo estructura COMPLEX, que represente números complejos, se haría así:

```
typedef struct {
    float real;
    float imagi;
} COMPLEX; /* Aquí COMPLEX es el nuevo nombre del tipo. */
```

Y la declaración de la variable complejo de tipo COMPLEX, sería:

```
COMPLEX complejo;
```

También se puede definir el tipo estructura COMPLEX de la siguiente forma:

```
typedef struct datos COMPLEX;
struct datos {
    float real;
```

```
    float imagi;
};
```

En la declaración anterior, se debería declarar **struct datos** antes del **typedef**, porque todo identificador debe estar declarado antes de ser usado, pero solo en este caso es una excepción.

La declaración de una variable **num_com** y del arreglo **lista_num** de 100 estructuras de tipo **struct datos**, de la definición anterior, sería:

```
COMPLEX num_com, lista_num[100];
```

También se puede definir un puntero a estructura de tipo COMPLEX, así:

```
typedef COMPLEX *PUNTERO;
```

Y la declaración de una variable puntero de tipo COMPLEX, sería:

```
PUNTERO punt;
```

Por último, también se puede emplear la sentencia **typedef** para crear nuevos tipos de datos. Por ejemplo, crear el tipo de dato BOOLEAN para representar los valores lógicos: **true** y **false**, que se haría así:

```
typedef enum bool BOOLEAN;
enum bool {false, true};
```

Y la declaración de una variable tipo BOOLEAN sería:

```
BOOLEAN sw;
```

La variable **sw** solo tomará los valores **false** y **true**. Por ejemplo, en la siguiente asignación se debe hacer un "casting" al tipo BOOLEAN:

```
sw = (BOOLEAN) (numero > 0);    /* numero, es de tipo int. */
```

Razones para usar typedef

1. Para crear nombres convenientes y reconocibles para los tipos de datos que se utilizan a menudo. Es decir, ayuda en la documentación del código, usando nombres que sean más descriptivos que los tipos de datos estándar.

2. Para describir tipos de datos complicados. Por ejemplo la siguiente declaración:

```
typedef char *FRP()[5];
```

hace que **FRP()** sea un tipo en el que una función devuelva un puntero a un arreglo de tipo **char** de cinco elementos.

3. Para hacer que los programas sean más transportables, sin dependencias con la máquina. Esto se hace únicamente cambiando la definición en el **typedef**.

   Por ejemplo cuando un programa necesita usar números de 16 bits, que podría ser el tipo **short** o podría ser el tipo **int**. Entonces, para transportar de un sistema a otro, se crea un archivo **#include** introduciendo la siguiente definición:

   ```
   typedef short DOSBYTES;
   ```

   Ahora se podrá usar el tipo DOSBYTES en el sistema que define las variables **short** de 16 bits, y cuando se cambie al sistema en el que se necesite el tipo **int**, simplemente se cambia la definición del archivo **#include** por la siguiente:

   ```
   typedef int DOSBYTES;
   ```

   Ejercicio

Realizar un programa para leer la información de **n** personas. Los datos de una persona están dados por el nombre y un código de tipo entero. El programa también debe ordenar a las personas alfabéticamente, para realizar un reporte como el siguiente:

ORD	NOMBRE	CODIGO

```
/* PROG1011.C */

/* Datos de las personas. */

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

#define MAX 50

typedef struct datos EMPLEADO;
struct datos {
  char nombre[20];
  int codigo;
};
```

```
void ordenar (EMPLEADO lista[], int num);
void leer (EMPLEADO lista[], int *n);
void imprimir (EMPLEADO lista[], int num);

void main ()
{
  EMPLEADO lista[MAX];
  int n;

  leer (lista, &n);
  ordenar (lista, n);
  imprimir (lista, n);
}

void leer (EMPLEADO lista[], int *n)
{
  int i;

  do {
    printf ("Ingrese el número de personas: ");
    scanf ("%d", n);
  } while (*n <= 0 || *n > MAX);

  for (i = 0; i < *n; i++) {
    while (getchar () != '\n');  /* Vacía el buffer de entrada. */
    printf ("\nIngrese el nombre: ");
    gets (lista[i].nombre);

    do {
      printf ("Ingrese el código: ");
      scanf ("%d", &lista[i].codigo);
    } while (lista[i].codigo <= 0);
  }
}

void ordenar (EMPLEADO lista[], int num)
{
  int i, j, menor;
  EMPLEADO aux;

  for (i = 0; i < num - 1; i++) {
    menor = i;
    for (j = i + 1; j < num; j++)
      if (strcmp (lista[j].nombre, lista[menor].nombre) <= 0)
```

```
        menor = j;
    if (i != menor) {
      aux = lista[i];
      lista[i] = lista[menor];
      lista[menor] = aux;
    }
  }
}

void imprimir (EMPLEADO lista[], int num)
{
  int i;

  printf ("\nORD NOMBRE              CODIGO");
  printf ("\n------------------------------\n");
  for (i = 0; i < num; i++)
    printf ("%03d %-20s %5d\n", i + 1, lista[i].nombre, lista[i].codigo);
}
```

Una salida del programa sería:

```
Ingrese el número de personas: 2 <ENTER>

Ingrese el nombre: Víctor Hugo <ENTER>
Ingrese el código: 111 <ENTER>

Ingrese el nombre: Ana María <ENTER>
Ingrese el código: 222 <ENTER>

ORD NOMBRE           CODIGO
---------------------------------------
001 Ana María            111
002 Víctor Hugo          222
```

PROBLEMAS PROPUESTOS

1) Se tiene la siguiente información de una persona: nombre y fecha de nacimiento. Realizar un programa que lea la información de varias personas, hasta ingresar una cadena nula en lugar del nombre. Luego imprimir un reporte en orden cronológico, con la siguiente cabecera:

Nombre	Edad

2) Realizar una función lógica que muestre el siguiente mensaje, validando el ingreso únicamente para las letras S o N:

   ¿Desea continuar (S/N)?

   En un programa utilizar esta función para ingresar los meses del año, y determine la frecuencia de los meses ingresados de acuerdo al reporte:

Meses	Frecuencia

   Usar la siguiente enumeración:

   enum meses {Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
   Agosto, Septiembre, Octubre, Noviembre, Diciembre};

3) Elaborar un programa para crear un directorio telefónico ordenado en forma alfabética, los campos de cada persona del directorio constan del nombre y número telefónico. El incremento de personas se debe realizar mientras se digite 'S', cuando aparezca el mensaje:

   ¿Desea ingresar otro (S/N)?

   Además, el programa debe realizar la búsqueda de un número telefónico mediante su nombre y por último imprimir el directorio completo.

   NOTA: Usar un arreglo dinámico para almacenar la información, considerando que al incrementar las personas aumenta el arreglo.

4) La información de los alumnos de un curso es una estructura de dos campos: un código lógico para indicar si aprueba o no el curso y el nombre del alumno. Realizar un programa que lea la información desde el teclado e imprima un reporte en orden alfabético como el siguiente:

TERCER NIVEL

	APROBADOS
ORD	NOMBRE
1	.
.	.

	REPROBADOS
ORD	NOMBRE
1	.
.	.

NOTA: Usar arreglos dinámicos para almacenar y ordenar la información.

5) Realizar un programa que ingrese una serie de **n** elementos de tipo entero para que se almacenen en un arreglo dinámico, luego debe imprimir la serie en el mismo orden ingresado. Por último, determina la frecuencia de cada elemento que debe ser mostrada en orden ascendente de acuerdo a los elementos ingresados. Por ejemplo:

   Si la serie ingresada es:

   6 2 1 4 1 2 4 1 4

   La frecuencia de elementos ordenados será:

Elementos	Frecuencia
1	3
2	2
4	3
6	1

   Utilizar una estructura para determinar la frecuencia de cada elemento con la siguiente información: elemento y frecuencia.

6) Una estructura para un estudiante tiene los siguientes campos: el nombre y las notas de las materias (Química, Algebra, Física y Cálculo). Realizar un programa que ingrese la información de cada estudiante, hasta digitar <<ENTER>> por un estudiante o un máximo de 50 estudiantes, y luego imprimir un reporte en orden alfabético como el siguiente:

ORD	NOMBRE	QUIMICA	ALGEBRA	FISICA	CALCULO

NOTAS:
- Realizar funciones para el ingreso, ordenamiento, intercambio y reporte.
- Usar punteros para el paso de parámetros a las funciones.

7) Sean **nombre**, **sexo** y **anios** tres variables que contienen el nombre, sexo y años de pertenencia de cada uno de los miembros de un club. Masculino o Femenino se denota mediante 'M' y 'F', respectivamente.

Diseñar un programa mediante arreglos de estructuras, que ingrese los datos hasta leer por un miembro como una cadena nula, y genere un reporte ordenado alfabéticamente, como se muestra a continuación:

LISTA DE MIEMBROS DEL CLUB

ORD	NOMBRE	ANTIGUEDAD (Años)	SEXO
1	Fiallos Verónica	3	Femenino
2	Dávila Pedro	5	Masculino
3	Gordón Lorena	1	Femenino

Además, buscar la persona que tiene la mayor antiguedad e indicar si es masculino o femenino.

NOTA: Usar la declaración de la estructura con **typedef**.

8) Realizar un programa que lea desde el teclado la información para cada alumno, con la siguiente estructura:

```
typedef enum meses NOTAS_MESES;
enum meses {COE1_1, COE25_1, COE1_2, COE25_2};
typedef enum materias NOTAS_MATERIAS;
enum materias (Quimica, Algebra, Fisica, Calculo y Matematicas );
typedef struct alumno DATOS;
struct alumno {
    char Nombre[31];
    int Carnet;
    float Notas[5][4];
};
```

Determinar la lista de los alumnos que tienen el mejor promedio en cada materia. Por ejemplo:

NOMBRE	MATERIA	PROMEDIO
Játiva Pedro	Química	18.3
Guzman José	Cálculo	19.4

9) Escribir un programa que introduzca un valor de tipo **char**, **short**, **int**, y **long**, y que almacene los valores en las variables unión de tipo **union integer**, que tenga como miembros **char c**, **short s**, **int i**, y **long l**. Cada variable de la unión deberá ser impresa como un **char**, un **short**, un **int** y un **long**.

10) Escribir un programa que introduzca un valor del tipo **float**, **double** y **long double**, y que almacene los valores en variables unión de tipo **union Punto_flotante**, que tenga como miembros **float f**, **double d**, y **long double l**. Cada variable de la unión deberá imprimirse como un **float**, un **double** y un **long double**.

11) Una compañía de seguros ofrece tres tipos de pólizas: vida, automóvil y casa. Para los tres tipos de pólizas es necesario tener el nombre del asegurado, su dirección, la cantidad asegurada, el pago de la póliza mensual que es el 3% de la cantidad asegurada y un número de póliza. Considerar el siguiente cuadro para cada póliza:

POLIZA DE SEGUROS	SE NECESITA ADICIONALMENTE
Vida (V)	- Fecha de nacimiento del asegurado (año, mes y día). - Nombre del beneficiario.
Automóvil (A)	- Número de placa. - Provincia. - Modelo del auto. - Año del auto. - Alarma.
Casa (C)	- Año de construcción de la casa. - Seguridades existentes. (rejas, alarmas, perro, vigilante).

Realizar un programa que lea la información para cada cliente y la almacene en un arreglo, hasta digitar 'N' cuando aparezca la pregunta:

¿Desea continuar S/N?

El programa también debe realizar un reporte de todos los clientes.

NOTAS:
- Usar uniones como estructura de datos.
- Validar el ingreso, en donde sea necesario.

12) En una empresa se tiene tres tipos de empleados: a contrato, a nombramiento y a contrato por obra. Para emitir el rol de pagos se necesitan el nombre y el sueldo a recibir de acuerdo al siguiente cuadro:

EMPLEADO	DATOS ADICIONALES	SUELDO A RECIBIR
Contrato (C)	- Sueldo - Retenciones	Sueldo - Retenciones
Nombramiento (N)	- Sueldo - Bonificaciones - Retenciones	Sueldo + Bonificaciones - Retenciones
Contrato por obra (O)	- Contrato - Número de meses	Contrato / Número de meses

Realizar un programa que lea la información para cada empleado, hasta digitar 'N' cuando aparezca:

¿Desea continuar S/N?

El programa también debe realizar un reporte de todos los empleados.

NOTA: Usar una unión como estructura de datos.

13) En un sistema de cuentas de clientes, cada cliente tiene la siguiente información: nombre del cliente, dirección con calle, ciudad y provincia (una estructura anidada), número de cuenta, estado de la cuenta (al día, atrasada, delincuente), saldo anterior, pago actual, nuevo saldo y fecha de pago (estructura anidada). Cada registro de cliente será una estructura y el conjunto completo de estos registros estarán almacenados en un arreglo llamado **cliente**, donde cada elemento del arreglo será una estructura independiente que será leída desde el teclado.

Introducir un número de cuenta para un cliente particular y transferirlo con el arreglo de registros a la función llamada **buscar()**. Dentro de esta función el número de cuenta será comparado con el número de cuenta almacenado dentro de cada registro, hasta que se produzca una coincidencia o hasta que se haya buscado en toda el arreglo de registros.

Si se produce una coincidencia, se devuelve a **main()** un puntero a ese elemento del arreglo que es la estructura que contiene el registro del cliente deseado, y los contenidos del registro son mostrados en la pantalla.

Si no se produce coincidencia después de la búsqueda de todo el arreglo, la función devuelve el valor NULL o cero a **main()**. El programa muestra entonces un mensaje de error pidiendo al usuario que reintroduzca el número de cuenta. Esta búsqueda continuará hasta que se introduzca el valor cero como número de cuenta.

El programa también debe mostrar un menú para imprimir el informe que a continuación se determina:

a) Estado de todos los clientes (generado por el programa).
b) Estado solo de clientes atrasados y delincuentes.
c) Estado solo de clientes delincuentes.

14) Una compañía alquila tres tipos de transporte: coches, barcos y aviones. Para el cobro del alquiler de los tres tipos de transportes es necesario el nombre del cliente y el monto mínimo o ***base*** de alquiler de acuerdo al siguiente cuadro:

TRANSPORTE	DATOS ADICIONALES	COSTO DE ALQUILER
Coche (C)	- Kilometraje	Kilometraje * base
Barco (B)	- Longitud (Millas) - Tonelaje	Longitud * base * Tonelaje
Avión (A)	- Número de motores - Número de asientos - Recorrido (Millas)	(Número motores + Número asientos) * base + Recorrido

Realizar un programa que lea la información para cada cliente, hasta digitar 'N' cuando aparezca:

¿Desea continuar S/N?

El programa también debe realizar un reporte de todos los empleados.

NOTA: Usar una unión como estructura de datos.

15) Escribir un programa que lea la información para varias personas, donde cada persona es una estructura con los datos: nombre y dirección. La estructura en la cual se va a almacenar toda la información es un arreglo de punteros a estructuras. Donde, la dirección es una unión conteniendo los miembros **direc_oficina** y **direc_casa**, cada miembro de la unión debe ser a su vez una estructura que consta de

dos arreglos de 80 caracteres llamados **calle** y **ciudad**, respectivamente. Imprimir una lista ordenada alfabéticamente indicando la dirección si es de la casa u oficina.

16) Se desea generar un directorio telefónico que será almacenado en un arreglo de punteros a estructuras, siendo cada estructura la información para una persona con el nombre, teléfono y dirección. Donde, la dirección es una unión conteniendo los miembros **direc_oficina** y **direc_casa**, cada miembro de la unión debe ser a su vez una estructura que consta de dos arreglos de 80 caracteres llamados **calle** y **ciudad** respectivamente. Se tiene otro miembro en la estructura primaria, que es un caracter indicador que debe ser 'o' o 'c', para indicar qué tipo de dirección está actualmente almacenada en la unión.

    Diseñar un programa que pregunte al usuario qué tipo de dirección tiene cada cliente, después que muestre la dirección apropiada con su correspondiente etiqueta (dirección oficina y dirección casa), conjuntamente con el resto de la información.

17) Hay tres métodos comúnmente usados para el "***cálculo de la depreciación anual***", como son el método de "línea recta", el método de "balance doblemente declinante", y el método de "suma de los dígitos del año", aplicados, por ejemplo, a un edificio o a una máquina:

    - En el "***método de línea recta***", el valor original del objeto se divide por su vida (número total de años). El cociente resultante será la cantidad en la que el objeto se deprecia anualmente. La depreciación anual es la misma cada año. Por ejemplo, si un objeto se deprecia 8000 dólares en 10 años, entonces la depreciación anual será 8000/10 = 800 dólares. Por tanto, el valor del objeto habrá disminuido en 800 dólares cada año.

    - En el "***método de balance doblemente declinante***", el valor del objeto disminuye cada año en un porcentaje constante. Por tanto, la verdadera cantidad depreciada, en dólares, variará de un año al siguiente. Para obtener el factor de depreciación se divide por 2 la vida del objeto, este factor se multiplica por el valor del objeto al comienzo de cada año (y no el valor original del objeto) para obtener la depreciación anual. Por ejemplo, si un objeto se deprecia 8000 dólares en 10 años, el factor de depreciación será 2/10 = 0.2. Por tanto, la depreciación el primer año será 0.2*8000=1600 dólares, la depreciación el segundo año será 0.2*(8000-1600)=1280 dólares, y así sucesivamente.

    - En el "***método de suma de los dígitos del año***", el valor del objeto irá disminuyendo en un porcentaje que es diferente cada año. El factor de depreciación será una fracción cuyo denominador es la suma de los números de 1 a **n** (1+2+...n), en donde **n** representa la vida del objeto. Para el primer año el numerador será **n**, para el segundo año será **(n-1)**, y así sucesivamente. La depreciación anual se obtiene multiplicando el factor de depreciación por el valor

original del objeto. Por ejemplo, si un objeto se deprecia 8000 dólares en 10 años. Por tanto, la depreciación el primer año será (10/55)*8000=1454.55 dólares, la depreciación el segundo año será (9/55)*8000=1309.09 dólares, y así sucesivamente.

Escribir un programa utilizando una **union** que permita seleccionar algunos de estos métodos para cada conjunto de cálculos. El proceso comenzará leyendo el valor original (sin depreciar) del objeto, la vida del objeto (el número de años en los que se depreciará) y un entero que indica qué método se utilizará. La depreciación anual y, el valor remanente (no depreciado) del objeto se calculará a continuación y se escribirá para cada año.

18) "***El generador de pig latin***" es una forma codificada de escribir y de hablar que suelen usar los niños ingleses como juego. Una palabra en "pig latin" se forma transponiendo el primer sonido, generalmente la primera letra, de una palabra original al final de la misma y añadiendo al final la letra 'a'. Tomar en cuenta marcas de puntuación, letras mayúsculas y sonidos de letras dobles. Por ejemplo, la palabra "perro" se convierte en "erropa".

Escribir un programa que acepte múltiples líneas de texto, y se represente cada línea de texto con una estructura separada. Incluir los siguientes tres miembros en cada estructura:

a) La línea de texto original.
b) El número de palabras dentro del texto.
c) La línea de texto modificada (el "pig latin" equivalente del texto original).

19) Escribir un programa que acepte la siguiente información para cada equipo de la liga de béisbol o fútbol americano:

a) Nombre del equipo, incluido la ciudad
b) Número de victorias.
c) Número de derrotas.

Añadir la siguiente información para un equipo de béisbol:

a) Número de bolas bateadas con éxito.
b) Número de carreras.
c) Número de errores.
d) Número de juegos extra.

Añadir la siguiente información para un equipo de fútbol americano:

a) Número de empates.

b) Número de tantos.
c) Número de goles de campo.
d) Número de contra ataques.
e) Total de yardas ganadas (total de la temporada).
f) Total de yardas cedidas a los oponentes.

Introducir esta información para todos los equipos de la liga. Después ordenar y escribir la lista de equipos de acuerdo con su registro de victorias/derrotadas, usando las técnicas de ordenación. Almacenar la información en un arreglo de estructuras, donde cada elemento del arreglo contiene la información para representar la información de un equipo. Hacer uso de una **union** para representar la información variable (béisbol o fútbol) que se incluye como parte de la estructura. La **union** debe a su vez contener dos estructura, una para las estadísticas relacionadas con el béisbol y otra para las estadísticas del fútbol.

Además, el programa debe incluir una variable de enumeración para distinguir entre béisbol y fútbol.

20) La información de un cliente de una cuenta de banco es una estructura como la siguiente:

a) Nombre
b) Calle
c) Ciudad/Provincia/Código
d) Número de cuenta
e) Estado de la cuenta (un caracter indicando: al día, retrasado o delincuente).

Crear un programa que permita crear una lista de personas, para añadir nuevos elementos a la lista, o borrar elementos de la lista. El programa deberá utilizar la gestión de memoria dinámica, para almacenar la información apuntada por un puntero a puntero.

El programa también tendrá menús para facilitar su uso, y se incluirá una disposición para mostrar la lista después de cualquier cambio hecho a la lista y la selección de cualquier elemento del menú.

21) Realizar un programa para la simulación de barajar y distribuir cartas de alto rendimiento, en el que el grupo de cartas se representa como un arreglo de estructuras.

El programa debe tener una función **caja_IIena()**, que inicializa el arreglo de estructuras de tipo **carta** y también tiene dos punteros a cadenas **cara** y **palo**; el arreglo se inicializa en orden desde "*As*" hasta "*Rey*" de cada uno de los palos. La estructura de tipo **carta** se pasa a la función **barajar()**, donde se pone en operación el

algoritmo de "***alto rendimiento de barajar***". La función **barajar()** tiene como argumento un arreglo de 52 estructuras de tipo **carta**, para cada una de las cartas es tomado al azar un número entre 1 y 51, y dos punteros a arreglos de cadenas: **wcara** que tiene la inicialización de los números de las cartas y **wpalo** que tiene inicializado cada palo. A continuación, en el arreglo son intercambiadas la estructura actual **carta** y la estructura seleccionada al azar **carta**. En una sola pasada de todo el arreglo se llevan a cabo un total de 52 intercambios, y el arreglo de estructuras queda barajado. Por último se tiene la función **cantidad()** para distribuir las cartas barajadas en una sola pasada del arreglo.

22) Realizar un programa para la simulación de "***barajar y distribuir cartas de alto rendimiento***" de acuerdo al ejercicio anterior, en el cual el grupo de cartas se representa como un arreglo de estructuras de "*campo de bits*" para almacenar el grupo de cartas.

La siguiente definición contiene tres campos de bits, utilizados para representar una carta de un grupo de 52 cartas:

```
struct carta_bit {
    unsigned cara : 4;    /* cara está almacenado en 4 bits. */
    unsigned palo : 2;    /* palo está almacenado en 2 bits. */
    unsigned color : 1;   /* color está almacenado en 1 bits. */
};
```

La constante entera representa el ancho del campo, que es el número de bits en el cual queda almacenado el miembro; ésta constante debe ser un entero entre 0 y el número total de bits utilizados para almacenar un **int** o un **unsigned** en el sistema utilizado.

El número de bits se basa en el rango deseado de valores correspondiente a cada miembro de estructura:

- El miembro **cara** almacena valores entre 0 (As) y 12 (Rey), por lo que se necesita 4 bits (valor entre 0 y 15).
- El miembro **palo** almacena valores entre 0 y 3 (0=Diamantes, 1=Corazones, 2=Tréboles y 3= Espadas), por lo que se necesita 2 bits (valor entre 0 y 3).
- El miembro **color** almacena valores entre 0 (Rojo) y 1 (Negro), por lo que se necesita 1 bit (valor entre 0 y 1).

El programa debe crear el arreglo **caja**, que contiene 52 estructuras **struct carta_bit**. La función **caja_llena()** inserta las 52 cartas en el arreglo caja, y la función **cantidad()** imprime las 52 cartas. El miembro **color** se incluye para tener la posibilidad de indicar el color de la carta en un sistema que permita despliegues en color.

Este programa debe barajar las cartas utilizando el algoritmo de barajar de alto rendimiento, luego imprima el grupo de cartas resultante en un formato de dos columnas, que anteceda cada carta con su color.

23) Realizar un programa que introduzca para varios estudiantes el nombre y la fecha de nacimiento, y los almacene en un arreglo. Luego el programa mostrará para cada estudiante el nombre, el día de nacimiento (día de la semana), la fecha de nacimiento y la edad de cada estudiante (en años).

Cada fecha de nacimiento constará de tres números enteros: mes, día y año de nacimiento (el año se ingresa desde 1990 con cuatro dígitos). Estos tres enteros se almacenarán en campos de bits dentro de una sola palabra de 16 bits, así:

```
typedef struct {
    unsigned mes : 4;     /* de 4 bits, cuyo valor varía de 0 a 15. */
    unsigned dia : 5;     /* de 5 bits, cuyo valor varía de 0 a 31. */
    unsigned anio : 7;    /* de 7 bits, cuyo valor varía de 0 a 127. */
};
```

Además, generar un menú que permita al usuario seleccionar cualquiera de las siguientes características:

a) Mostrar la edad de un estudiante cuyo nombre y fecha de nacimiento se introduzca como entrada.
b) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario.
c) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario o menor.
d) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario o mayor.

24) Escribir un programa completo que realice la cuenta de vocales, consonantes, dígitos, espacios en blanco y otros caracteres. Esto realizar de modo que se lea varias líneas de texto, que deben ser primero leídas y almacenadas en un arreglo.

Luego determinar la media del número de vocales por línea, consonantes por línea y así sucesivamente.

La línea de texto de 80 caracteres debe almacenarse en un arreglo de 70 bytes. (Asumir caracteres ASCII de 7 bits). Para hacer esto, usar el operador de desplazamiento a nivel de bits de manera que un grupo de ocho caracteres se almacene en siete elementos consecutivos del arreglo (7 bytes). Cada elemento del arreglo contendrá un caracter completo, más un bit del octavo caracter.

Además, el programa debe incluir la posibilidad de mostrar los contenidos del arreglo de 70 bytes (usando constantes hexadecimales) en la forma comprimida y en la forma no comprimida.

25) Escribir un programa que desplace una variable entera 4 bits hacia la derecha. El programa deberá imprimir el entero en bits antes y después de la operación de desplazamiento.

26) Escribir un programa que acepte un número entero como entrada y muestre un menú que permita realizar cualquiera de las siguientes operaciones:

   a) Mostrar el número equivalente del complemento a uno.
   b) Realizar una operación de enmascaramiento y mostrar el número equivalente del resultado.
   c) Realizar una operación de desplazamiento de bits y mostrar el número equivalente del resultado.
   d) Salir.

   Si se selecciona la operación de enmascaramiento. pedir al usuario el tipo de operación (AND, OR y XOR a nivel de bits) y un valor (número) para la máscara. Si se selecciona la operación de desplazamiento, pedir al usuario el tipo de desplazamiento (derecha o izquierda) y el número de bits.

   La cantidad de entrada (número) pueda ser una constante decimal. hexadecimal u octal. Empezar mostrando un menú para permitir al usuario elegir el tipo de número (el sistema de numeración deseado) antes de introducir el valor real. Después mostrar el valor en los otros dos sistemas de numeración y su correspondiente patrón de bits. Una vez que la cantidad ha sido introducida y mostrada, generar el menú anterior pidiendo el tipo de operación deseada. Si se selecciona una operación de enmascaramiento, introducir la máscara como una constante hexadecimal u octal. Mostrar el resultado de cada operación en decimal, hexadecimal, octal y binario.

   El programa debe mostrar los patrones binarios además de los valores. Usar una función separada para mostrar los patrones binarios.

27) Para "***imprimir un entero sin signo***" **unsigned** en su representación binaria en grupos de 8 bits cada uno, se utiliza el operador AND a nivel de bits para combinar el **valor** con una "variable de máscara", **mascara**, que es inicializada con 1<<15 (10000000 00000000). El operador de desplazamiento a la izquierda desplaza el valor 1 de la posición inferior (más a la derecha) hacia el bit de orden superior (más a la izquierda) en la variable **mascara**, y rellena con bits 0 a partir de la derecha. La impresión:

```
putchar (valor & mascara ? '1' : '0');
```

determina si deberá de imprimirse un '1' o un '0' para el bit actual más a la izquierda del valor.

Cuando se ejecuta **valor & mascara**, todos los bits, a excepción del bit de orden superior, en la variable **valor**, son enmascarados (ocultos), porque cualquier bit manipulado por AND con 0 da como resultado 0. Si el bit más a la izquierda es 1, **valor & mascara** se evalúa a 1, y 1 se imprime, de lo contrario se imprime 0. Luego la variable **valor** es después desplazada un bit a la izquierda, mediante la expresión **valor<<=1**. Estos pasos se repiten para cada uno de los bits en la variable **valor**. Por último, para imprimir en grupos de 8 bits, luego de cada octavo bit impreso se deja un espacio en blanco.

El programa imprimirá números enteros de 4 bytes en su representación binaria en grupos de 8 bits.

NOTA: A menudo el operador AND a nivel de bits se utiliza con un operando conocido como una máscara, un valor entero con bits específicos establecidos a 1. Las máscaras se utilizan para ocultar algunos bits en un **valor**, mientras otros bits se seleccionan.

28) Escribir un programa que ilustre equivalencia entre:

a) Desplazar un número binario **n** posiciones a la izquierda y multiplicarlo por 2^n.
b) Desplazar un número binario **n** posiciones a la derecha y dividirlo por 2^n (o equivalentemente a multiplicarlo por 2^{-n}).

Elegir el número binario inicial con cuidado de modo que no se pierdan bits como resultado de las operaciones de desplazamiento. (Para el desplazamiento a la izquierda, elegir un número relativamente pequeño de modo que tenga varios ceros en las posiciones de más a la izquierda. Para el desplazamiento a la derecha, elegir un número relativamente grande, con ceros en las posiciones más a la derecha.)

Utilizar el operador de desplazamiento en la función **potencia()**, que tiene dos argumentos enteros **numero** y **expo** para calcular:

numero * 2^{exp}

El resultado deberá ser impreso como entero y como bits.

29) El operador de desplazamiento a la izquierda puede ser utilizado para empacar dos valores de caracteres en una variable entera sin signo de 2 bytes. Escribir un programa que introduzca dos caracteres del teclado y que los pase a la función **empacar_caracteres()**. Para empacar dos caracteres en una variable entera **unsigned**, asignar el primer caracter a la variable **unsigned**, desplazar la variable a la izquierda en 8 posiciones de bits, y combinar la variable **unsigned** con el segundo caracter utilizando el operador XOR a nivel de bits.

    El programa deberá extraer los caracteres en su formato de bits, antes y después de haber sido empacados en el entero **unsigned**, para probar que los caracteres de hecho han sido empacados correctamente en la variable **unsigned**.

    Utilizar el operador de desplazamiento a la derecha, el operador AND a nivel de bits y una máscara, en la función **desempacar_caracteres()** que retorna el entero unsigned empaquetado por la función **empacar_caracteres()** y lo desempaqueta en dos caracteres. Para desempacar dos caracteres de un entero **unsigned** de 2 bytes, combinar el entero **unsigned** con la máscara 65280 (11111111 00000000) y desplazar hacia la derecha el resultado en 8 bits. Asigne el valor resultante a una variable **char**. A continuación combinar el entero unsigned con la máscara 255 (00000000 11111111). Asignar el resultado a otra variable **char**.

    El programa deberá imprimir el entero **unsigned** en bits, antes de ser desempacado, y a continuación imprimir los caracteres en bits para confirmar que fueron desempacados correctamente.

30) En el programa anterior empacar 4 caracteres en un entero de 4 bytes. También, volver a escribir la función **desempacar_caracteres()** para desempacar 4 caracteres. Crear las máscaras que se necesite para desempacar los 4 caracteres desplazando hacia la izquierda el valor 255 en la variable de enmascaramiento en 8 bits 0,1,2 o 3 veces, (dependiendo del byte que se está desempacando).

31) Escribir un programa que invierta el orden de los bits de un valor entero sin signo. El programa deberá introducir el valor proveniente del usuario y llamar a la función **Reversa_Bits()**, para imprimir los bits en orden inverso. Imprimir el valor en bits tanto como después de la inversión de bits, para confirmar que los bits hayan sido invertidos correctamente.

Capítulo 11

E/S POR ARCHIVOS

11.1. INTRODUCCIÓN

El lenguaje C no contiene ninguna sentencia de E/S, por el contrario, todas las operaciones de E/S tienen lugar a través de llamadas a la biblioteca estándar. Además, los datos son siempre transferidos en su representación binaria interna o en formato de texto normal. En este libro solo se estudiará el sistema de E/S para compiladores en el sistema operativo Windows.

Los conjuntos de funciones de E/S definidos por el lenguaje C son: (SCHILDT, 1994)

1. El sistema de E/S definido por el estándar ANSI, también denominado "sistema de archivos con buffer" o con "formato de alto nivel".

2. El sistema de E/S tipo UNIX, a veces referido como "sistema de archivos sin buffer" o "sin formato", fue creado para los primeros compiladores de lenguaje C, que fueron desarrollados bajo UNIX.

3. Funciones de E/S de bajo nivel que operan directamente sobre el hardware de la computadora.

En este capítulo solo se estudiará el sistema de E/S ANSI, el mismo que define un conjunto completo de funciones de E/S que se pueden usar para leer y escribir cualquier tipo de datos.

Todas las declaraciones de los prototipos de las funciones, así como los distintos tipos y constantes que se utilizan en las funciones de biblioteca de E/S con buffer de lenguaje C, se encuentran en el archivo de cabecera "***stdio.h***".

Además, la mayoría de las funciones del sistema de E/S UNIX con formato buffer, tenían el prefijo **f**, por lo que el comité ANSI decidió mantener este convenio por interés de continuidad.

11.2. FLUJOS Y ARCHIVOS

El sistema de archivos de E/S de lenguaje C ANSI está compuesto por varias funciones interrelacionadas, que proporciona al programa de lengueje C, una interfaz consistente e independiente del dispositivo al que se está accediendo. Esto ofrece un nivel de abstracción muy útil entre el programa y el dispositivo que se está usando. A esta abstracción se le llama **flujo** y al dispositivo real **archivo**.

11.2.1. Flujos

El sistema de archivos de E/S del lenguaje C está diseñado para trabajar con una amplia variedad de dispositivos, incluyendo terminales y controladores de discos y de cintas. Aunque cada dispositivo es diferente, el sistema de archivos con buffer transforma cada uno de ellos en un dispositivo lógico llamado "***flujo***" o "***buffer***".

Todos los flujos se comportan de forma similar, debido a que estos son independientes del dispositivo, entonces, la misma función puede leer o escribir en un archivo de disco o en otro dispositivo.

El compilador de lenguaje C convierte la entrada o salida en un flujo de fácil manejo, ya que solo se necesita pensar en términos de flujos y utilizar únicamente un sistema de archivos para realizar todas las operaciones de E/S.

Existen dos tipos de flujos: de texto y binario; que serán descrito a continuación:

1. Flujo de texto

Un "flujo de texto" es una secuencia de caracteres en código ASCII. El estándar ANSI permite, pero no exige, organizar un flujo de texto en líneas terminadas por un caracter de salto de línea. De hecho, la mayoría de los compiladores de lenguaje C no hacen que los flujos de texto terminen con caracteres de salto de línea.

En un flujo de texto, pueden ocurrir ciertas conversiones de caracteres, si el entorno del sistema lo requiere. Por ejemplo, en el sistema operativo Windows, en la mayoría de las implementaciones en modo texto, se tendría las siguientes conversiones:

- ***En la lectura***, los caracteres "retorno de carro/salto de línea" se convierten a caracteres de "salto de línea".

- **En la escritura**, ocurre lo contrario; los caracteres de "salto de línea" se convierten a "retorno de carro/salto de línea".

Otra conversión se da en el tipo de datos entre binario a texto y viceversa, ya que los datos siempre se almacenan en forma de texto (conversión binario a texto) y se leen de acuerdo al tipo de variable en donde se van almacenar (conversión de texto a binario).

Por esto, no puede haber una relación de uno a uno entre los caracteres que se escriben o se leen, y los que aparecen en el dispositivo externo. Entonces, debido a las posibles conversiones, puede que el número de caracteres escritos o leídos, no coincida con el dispositivo externo.

2. **Flujo binario**

   Un "Flujo binario" es una cadena de bytes almacenados en forma binaria, con una correspondencia de uno a uno con los del dispositivo externo, esto es, no se realizan conversiones de caracteres ni de tipos de datos. Además, el número de bytes escritos o leídos es el mismo que en el dispositivo externo.

11.2.2. Archivos

En lenguaje C, un ***archivo*** puede ser: un archivo de disco, un terminal o una impresora. etc.. y estos son manejadas en forma de archivos, los mismos que tienen dos características:

1. **Asociar un flujo con un archivo específico**. Se obtiene al realizar una operación de apertura. Una vez que el archivo está abierto se crea el flujo o buffer, y la información puede ser intercambiada entre éste y el programa. Entonces por cada archivo abierto existe un flujo.

   No todos los archivos tienen las mismas posibilidades de lectura y escritura. Por ejemplo, un archivo de disco permite el acceso directo, mientras que un teclado no. Es decir, todos los flujos son iguales pero no todos los archivos lo son, ya que unos pueden ser leídos, otros escritos o ambas cosas.

   Por otro lado, la información de disco se escribe o se lee sector a sector, ya que éste es la parte más pequeña accesible de un disco, que normalmente corresponde a 512 bytes. Para leer y escribir en disco, se debe tener en cuenta lo siguiente:

   - ***Al leer de disco***, se lee por lo menos un sector de datos, incluso si solo se necesita un byte. Estos datos se colocan en una región de memoria denominada ***buffer***, hasta que puedan ser usados en el programa.

   - ***Al almacenar en disco***, la información se colocará en un buffer hasta que se acumule la suficiente información para llenar un sector; en ese momento se escribe en el disco.

   Al principio de la ejecución de un programa en lenguaje C, se abren tres flujos de texto predefinidos que se refieren a los dispositivos de E/S estándar, conectados al sistema; los cuales de muestran en la Tabla 11.1.

Tabla 11.1. Flujos de texto de los dispositivos de E/S estándar

FLUJO	DISPOSITIVO
stdin	Teclado
stdout	Pantalla
stderr	Pantalla

2. **Disociar un archivo de un flujo específico**. Se obtiene con una operación de cierre. Si se cierra un archivo abierto para operaciones de salida, se escribe en ese archivo el contenido de su flujo asociado, si existiera. A este proceso se le conoce normalmente como "***vaciado del flujo***" y garantiza que no quede información accidentalmente en el buffer.

   Todos los archivos se cierran automáticamente cuando un programa termina; normalmente cuando la función **main()** devuelve el control al sistema operativo, o cuando existe una llamada a la función **exit()**. Si se interrumpe la ejecución del programa, los sistemas operativos actuales se encargan de cerrar los archivos, pero en otros casos puede darse que los archivos no se cierren.

11.3. PUNTERO A ARCHIVO

El "puntero a archivo" es un puntero a una información que define varias cosas sobre él: el nombre, el estado y la posición actual del archivo.

En esencia, el "puntero a archivo" identifica un archivo en disco específico y utiliza el flujo asociado para dirigir el manejo de las funciones de E/S con buffer. Es decir, el "puntero a archivo" es el hilo común que unifica el sistema de E/S con buffer.

El "puntero a archivo" es una variable puntero de tipo FILE que se define en el archivo de cabecera "***stdio.h***", que sirve como enlace para realizar la lectura o escritura en los archivos.

Una variable de tipo "puntero a archivo" se obtiene con la siguiente sentencia de declaración:

FILE *pt;

Indicador de posición del archivo. Indica la posición dentro del archivo de disco, que se inicializa al comienzo o al final del archivo en el momento de abrir el mismo. Además, según se lea o se escribe los datos del archivo, el indicador de posición se incrementa al siguiente dato.

11.4. APERTURA DE UN ARCHIVO: *fopen()*

La función **fopen()** abre un flujo para asociarlo a un archivo, para que pueda ser utilizado. Después esta función devuelve el "puntero a archivo" asociado con ese archivo.

El prototipo de la función **fopen()** es: (SCHILDT, 1994)

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

donde:

- **nombre_archivo**, es un puntero a una cadena de caracteres que representa un nombre válido del archivo, y puede incluir una especificación de directorio.

- **modo**, es un puntero a una cadena de caracteres que determina como se abre el archivo; los modos de apertura de un archivo pueden ser: en modo texto o en modo binario, tanto para escritura, como para lectura o ambas, que se muestran en la Tabla 11.2.

Tabla 11.2. Modos de apertura de un archivo

MODO	SIGNIFICADO
r	Abre un archivo de texto para lectura
w	Crea un archivo de texto para escritura
a	Abre un archivo de texto para añadir
rb	Abre un archivo binario para lectura
wb	Crea un archivo binario para escritura
ab	Abre un archivo binario para añadir
r+	Abre un archivo de texto para lectura/escritura
w+	Crea un archivo de texto para lectura/escritura
a+	Añade en un archivo de texto en modo lectura/escritura
r+b ó rb+	Abre un archivo binario para lectura/escritura
w+b ó wb+	Crea un archivo binario para lectura/escritura
a+b ó ab+	Añade en un archivo binario en modo lectura/escritura

La función **fopen()** devuelve un "puntero a archivo", cuyo valor nunca debe ser alterado. Si se produce un error cuando se intenta abrir un archivo, la función **fopen()** devuelve un puntero NULL.

Por lo tanto, para abrir un archivo se debe verificar si el "puntero a archivo" es diferente de NULL, evitando cualquier error, como puede ocurrir si el disco está protegido contra escritura o si está lleno. Por ejemplo, para abrir un archivo de texto llamado PRUEBA para escritura, podría procederse así:

```
FILE *pt;
if ((pt = fopen ("prueba", "w")) == NULL) {
    printf ("No se puede abrir el archivo: PRUEBA.\n");
    exit (1);
}
```

Observaciones:

Si se utiliza la función **fopen()** para abrir un archivo, se presentan las siguientes observaciones:

1. "Para escritura", cualquier archivo que ya existe se borra y se crea uno nuevo, y si no existe también se crea uno nuevo.

2. "En lectura", solo se pueden abrir archivos que ya existen, si el archivo no existe, la función **fopen()** devuelve un error.

3. "Para lectura/escritura", si ya existe el archivo no será borrado, si no existe será creado.

4. "Para añadir" información al final del archivo, se debe utilizar el modo "a". El estándar ANSI especifica que pueden estar abiertos por lo menos ocho archivos en cualquier momento. Sin embargo, la mayoría de los compiladores de lenguaje C permiten un número mayor.

11.5. CIERRE DE UN ARCHIVO: *f*close()

La función **fclose()** cierra un flujo que fue abierto mediante una llamada a **fopen()**, almacenando en el archivo toda la información que todavía se encuentre en el buffer, y realizando un cierre formal del archivo a nivel del sistema operativo. La función **fclose()** también libera el flujo asociado con el archivo, dejándolo libre para su reutilización.

Un error en el cierre de un flujo puede generar todo tipo de problemas, incluyendo la pérdida de datos, la destrucción de archivos y posibles errores intermitentes en el programa.

Existe un límite del sistema operativo respecto al número de archivos abiertos simultáneamente, por lo que es necesario cerrar un archivo antes de abrir otro.

El prototipo de la función **fclose()** es: (SCHILDT, 1994)

int fclose (FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por la llamada a **fopen()**.

La función **fclose()** devuelve un valor cero si la operación de cierre ha tenido éxito.

Generalmente, la función **fclose()** solo falla cuando un dispositivo externo de memoria se ha retirado antes de tiempo o cuando no queda espacio libre en dicho dispositivo.

11.6. ESCRITURA DE UN CARACTER: *putc()*

La función **putc()** escribe un caracter en un archivo que haya sido abierto en modo escritura mediante la función **fopen()**.

El prototipo de la función **putc()** es: (SCHILDT, 1994)

int putc (int car, FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**, que indica a la función **putc()** en qué archivo de disco debe escribir.

- **car**, es el caracter que se va a escribir.

La función **putc()** devuelve el caracter escrito si la operación de escritura tiene éxito, en otro caso devuelve EOF.

NOTAS:

- El estándar ANSI define dos funciones equivalentes que escriben un caracter en un archivo: **putc()** y **fputc()**. Además, técnicamente **putc()** está implementada como una macro. Estas dos funciones idénticas, solo existen para preservar la compatibilidad con versiones antiguas del lenguaje C. Por lo tanto, ambas funciones pueden utilizarse indistintamente.

- Por razones históricas, **car** se define de tipo **int** para compaginar con la comparación de EOF que es -1, pero solo se utiliza el byte menos significativo. Igual sucede para el valor retornado por la función.

11.7. LECTURA DE UN CARACTER: getc()

La función **getc()** lee un caracter de un archivo abierto en modo lectura mediante la función **fopen()**.

El prototipo de la función **getc()** es: (SCHILDT, 1994)

```
int getc (FILE *pt);
```

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

La función **getc()** retorna el caracter leído del archivo, o retorna una marca EOF cuando se haya alcanzado el final del archivo.

Entonces, la lectura de un archivo de texto puede realizarse hasta que se encuentre la marca de fin de archivo, con las siguientes sentencias:

```
while ((car = getc (pt)) != EOF)
    putchar (car);     /* Escribe cada caracter en pantalla. */
```

NOTAS:

- El estándar ANSI define dos funciones equivalentes que leen un caracter de un archivo: **getc()** y **fgetc()**. Estas dos funciones idénticas solo existen para preservar la compatibilidad con versiones antiguas del lenguaje C. Por lo tanto, ambas funciones pueden utilizarse indistintamente.

- Debido a razones históricas **getc()** devuelve un entero, pero el byte más significativo es cero.

Las funciones **fopen()**, **getc()**, **putc()** y **fclose()** constituyen el conjunto mínimo de rutinas de tratamiento de archivos.

Ejercicio

Realizar un programa para leer caracteres desde el teclado y almacenarlos en un archivo de disco, hasta que el usuario teclee el caracter '*'. El nombre del archivo de disco debe ser ingresado desde la línea de comandos y el nombre del programa se llamará PROG1101.

Por ejemplo, en la línea de comandos se pondría el nombre del programa ejecutable (PROGl101.EXE) y el nombre del archivo que almacena los caracteres (PRUEBA.TXT), de la siguiente manera:

A:\> CAPITULO.11\PROG1101 PRUEBA.TXT <ENTER>

```
/* PROG1101.C */

#include "stdio.h"
#include "stdlib.h"

void main (int argc, char *argv[])
{
 FILE *pt;
 char ch;

 if (argc != 2) {
   printf ("Ingrese: <Programa> <Archivo>\n");
   exit (1);
 }

 if ((pt = fopen (argv[1], "w")) == NULL) {
   printf ("No se puede abrir el archivo: %s.\n", argv[1]);
   exit (1);
 }

 printf ("Ingrese caracteres hasta digitar '*'.\n\n");

 while ((ch = getchar ()) != '*')
   putc (ch, pt);

 fclose (pt);
}
```

Para que el programa pueda ser ejecutado, primero se debe obtener el programa .EXE, y luego ejecutarlo desde la línea de comandos, así:

A:\> CAPITULO.11\PROG1101 PRUEBA.TXT <ENTER>

Una salida del programa sería:

Ingrese caracteres hasta digitar ''.*
Este texto es un ejemplo, <ENTER>
que sera' verificado en el siguiente programa. <ENTER>
<ENTER>

En el archivo PRUEBA.TXT se almacenará el texto anterior ingresado desde teclado.

Ejercicio

Realizar un programa para leer cualquier archivo de texto, y su contenido sea impreso en la pantalla. El nombre del archivo de disco debe ser ingresado desde la línea de comandos, de la siguiente manera:

A:\> CAPITULO.11\PROG1102 PRUEBA.TXT <ENTER>

donde,

- PROG1102 es el nombre del programa y PRUEBA.TXT es el nombre del archivo a leer.

```
/* PROG1102.C */

#include "stdio.h"
#include "stdlib.h"

void main (int argc, char *argv[])
{
 FILE *pt;
 char car;

 if (argc != 2) {
  printf ("Ingrese: <Programa> <Archivo>\n");
  exit (1);
 }

 if ((pt = fopen (argv[1], "r")) == NULL) {
  printf ("No se puede abrir el archivo: %s.\n", argv[1]);
  exit (1);
 }

 printf ("Contenido del archivo de texto: %s.\n\n", argv[1]);

 while ((car = getc (pt)) != EOF)
  putchar (car);
 fclose (pt);
}
```

Para que el programa pueda ser ejecutado, primero se debe obtener el programa .EXE, y luego ejecutarlo desde la línea de comandos, así:

A:\> CAPITULO.11\PROG1102 PRUEBA.TXT <ENTER>

Una salida del programa seria:

Contenido del archivo de texto: PRUEBA. TXT.

Este texto es un ejemplo,
que será verificado en el siguiente programa.

11.8. FIN DE ARCHIVO: *feof()*

La función **feof()** determina cuando se ha alcanzado el fin del archivo, al leer un archivo de datos binario.

El prototipo de la función **feof()** es: (SCHILDT, 1994)

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

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

La función **feof()** devuelve verdadero si se ha alcanzado el final del archivo, en otro caso devuelve 0.

Por ejemplo, para leer todo el archivo binario apuntado por **pt**, es decir, para leer el archivo hasta que se encuentre el final del mismo y mostrar su contenido en la pantalla, se haría así:

```
while (!feof (pt))
    puchar (getc (pt));
```

Se puede aplicar este método tanto a archivos binarios como a archivos de texto, hasta que se encuentre el caracter EOF, que está definido en el archivo de cabecera "***stdio.h***" con el valor -1. Esto sirve para trasladar bloques de datos sin importar el modo de almacenamiento.

Cuando se abre un archivo para datos binarios, se puede leer un valor entero igual al de la marca EOF. Esto podría hacer que en la lectura indique una condición de fin de archivo, aún cuando el fin físico del mismo no se haya alcanzado.

Ejercicio

Realizar un programa para copiar cualquier archivo binario o de texto a otro archivo. Pero los archivos deberán abrirse en modo binario, y operarse como archivos de texto. La función **feof()** deberá comprobar el fin del archivo origen.

```
/* PROG1103.C */

#include "stdio.h"
#include "stdlib.h"

void main (int argc, char *argv[])
{

  FILE *origen, *destino;

  if (argc != 3) {
    printf("Ingrese: <Programa> <Origen> <Destino>\n");
    exit(1);
  }

  if ((origen = fopen (argv[1], "rb")) == NULL) {
     printf("No se puede abrir el archivo origen: %s.\n", argv[1]);
     exit(1);
  }

  if ((destino = fopen(argv[2], "wb")) == NULL) {
    printf("No se puede abrir el archivo destino: %s.\n", argv[2]);
    exit(1);
  }

  while (!feof (origen))
    putc (getc (origen), destino);

  printf ("Copia el archivo: %s, al archivo %s.\n", argv[1], argv[2]);

  fclose (origen);
  fclose (destino);
}
```

Para que el programa pueda ser ejecutado, primero se debe obtener el programa PROG1103.EXE, y luego ejecutarlo desde la línea de comandos, así:

```
A\> CAPITULO.11\PROG1103 PROG1103.EXE NUEVO.EXE <ENTER>
```

Una salida del programa sería:

Copia el archivo: prog1103.exe, al archivo nuevo.exe.

11.9. INICIALIZAR EL INDICADOR DE POSICIÓN: *rewind()*

La función **rewind()** posiciona el "indicador de posición del archivo" al principio del mismo. Es decir, "rebobina" el archivo.

El prototipo de la función **rewind()** es: (SCHILDT, 1994)

void rewind (FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

11.10. ESCRITURA Y LECTURA DE CADENAS: *fputs()* y *fgets()*

Las funciones **fputs()** y **fgets()** escriben y leen cadenas de caracteres sobre archivos.

Los prototipos de las funciones **fputs()** y **fgets()** son: (SCHILDT, 1994)

int fputs (const char *cad, FILE *pt);
char *fgets (char *cad, int longitud, FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

- La función **fputs()** funciona como **puts()** pero escribe la cadena apuntada por **cad** en el flujo especificado por **pt**. Si se produce un error devuelve EOF.

- La función **fgets()** lee una cadena apuntada por **cad** del flujo especificado por **pt**, hasta que se lee un caracter salto de línea o hasta que se hayan leído ***longitud-1*** caracteres.

  Si se lee un caracter de salto de línea, éste forma parte de la cadena leída (a diferencia de **gets()**). La cadena resultante se termina en un caracter nulo.

  La función **fgets()** devuelve un puntero a **cad** si se ha ejecutado correctamente, y un puntero NULL si se produce un error.

Ejercicio

Realizar un programa para leer cadenas desde el teclado y escribirlas en un archivo llamado TEXTO.TXT, hasta ingresar una cadena nula. Debido a que la función **gets()** no guarda el caracter de salto de línea, se debe añadir uno antes de escribir la cadena en el archivo, con el fin de que el archivo sea almacenado por líneas, para que se pueda leer más fácilmente. Luego mostrar el contenido del archivo que se acaba de crear; para

realizar esto el programa debe inicializar el indicador de posición del archivo después de finalizar su escritura y posteriormente utiliza **fgets()** para leerlo de nuevo.

```
/* PROG1104.C */

#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void main ()
{

  FILE *pt;
  char cad[80];

  /* Abrir el archivo en modo lectura/escritura. */
  if ((pt = fopen ("texto.txt", "w+")) == NULL) {
     printf("No se puede abrir el archivo: TEXTO.TXT.\n");
     exit(1);
  }

  printf ("Introducir cadenas con <ENTER> termina.\n");
  while (*gets (cad)) {
    strcat (cad, "\n");   /* Añade a la cadena el caracter nueva línea. */
    fputs (cad, pt);
  }

  /* Inicializa el indicador de posición del archivo al principio del mismo. */
  rewind (pt);

  printf ("Texto ingresado:\n");

  /* Para eliminar la última lectura que queda en el buffer y no se repita */
  /* en la impresión, se realiza una lectura antes de ingresar al lazo.    */
  fgets (cad, 79, pt);

  while (!feof (pt)) {
    printf ("%s", cad);
    fgets (cad, 79, pt);
  }

  fclose (pt);
}
```

Una salida del programa sería:

Introducir cadenas con <ENTER> termina.
Este es un ejemplo, <ENTER>
para ingresar cadenas por líneas. <ENTER>
<ENTER>
Texto ingresado:
Este es un ejemplo,
para ingresar cadenas por líneas.

11.11. ELIMINACIÓN DE ARCHIVOS: *remove()*

La función **remove()** borra el archivo especificado en su argumento, que previamente debe estar cerrado.

El prototipo de la función **remove()** es: (SCHILDT, 1994)

int remove (char *nombre_archivo);

donde:

- **nombre_archivo**, es el nombre del archivo en disco.

La función **remove()** devuelve cero si tiene éxito, y un valor distinto de cero si falla.

Ejercicio

Realizar un programa para borrar el archivo especificado en la línea de comandos, pero antes de esto permitir rectificar esta decisión.

```
/* PROG1105.C */

#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "conio.h"

int main (int argc, char *argv[])
{
  char cad;

  /* Comprobación doble antes del borrado. */

  if (argc != 2) {
    printf ("Ingrese: <Programa> <Archivo>\n");
```

```
    exit(1);
  }

  printf ("Desea borrar el archivo: %s? (S/N): ", argv[1]);
  cad = getche ();

  if (toupper (cad) == 'S')
    if (remove (argv[1])) {
      printf ("\nNo se puede borrar el archivo.\n");
      return 1;
    }
    else
      return 0;   /* Le indica al Sistema Operativo el éxito del borrado. */
}
```

Para que el programa pueda ser ejecutado, primero se debe obtener el programa PROG1105.EXE, y luego ejecutarlo desde la línea de comandos, así:

A:\> CAPITULO.11\PROG1105 NUEVO.EXE <ENTER>

Una salida del programa sería:

Desea borrar el archivo: nuevo.exe? (S/N): s <ENTER>

11.12. VACIAR EL CONTENIDO DE UN FLUJO: *fflush()*

La función **fflush()** vacía el contenido de un flujo especificado en el argumento, pero previamente esta función escribe todos los datos almacenados en el buffer sobre el archivo asociado con su argumento.

El prototipo de la función **fflush()** es: (SCHILDT, 1994)

int fflush (FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

La función **fflush()** devuelve 0 si se tiene éxito, en otro caso devuelve EOF.

Por ejemplo, para vaciar el flujo de la entrada estándar (teclado) se utiliza la siguiente sentencia:

fflush (stdin);

Si se llama a **fflush()** con un puntero NULL se vacían los buffers de todos los archivos abiertos.

11.13. DETERMINACIÓN DE ERRORES: ferror()

La función **ferror()** determina si se ha producido un error en una operación de lectura o escritura sobre un archivo.

El prototipo de la función **ferror()** es: (SCHILDT, 1994)

int ferror (FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

La función **ferror()** devuelve verdadero si se ha producido un error durante la última operación sobre el archivo, caso contrario devuelve falso.

Se debe tener en cuenta que en cada operación sobre un archivo se actualiza la condición de error, por esto se debe llamar a **ferror()** inmediatamente después de cada operación de este tipo, si no se hace así el error no podrá ser detectado.

Ejercicio

Realizar un programa que lea un archivo de texto, y que convierta las letras minúsculas a letras mayúsculas. El texto resultante debe ser almacenado en otro archivo de texto. Además se debe realizar la comprobación de errores.

```
/* PROG1106.C */

#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"

#define ENTRADA 0
#define SALIDA 1

void err (int e);

void main (int argc, char *argv[])
{
  FILE *entrada, *salida;
  char car;
```

```
if (argc != 3) {
  printf ("Ingrese: <Programa> <Entrada> <Salida>\n");
  exit (1);
}

if ((entrada = fopen (argv[1], "rb")) == NULL) {
  printf ("No se puede abrir el archivo: %s.\n", argv[1]);
  exit(1);
}

if ((salida = fopen (argv[2], "wb")) == NULL) {
  printf ("No se puede abrir el archivo: %s.\n", argv[2]);
  exit(1);
}

printf ("Copia el archivo: %s, al archivo %s.\n", argv[1], argv[2]);

while (!feof (entrada)) {
  car = getc (entrada);
  if (ferror (entrada))
    err (ENTRADA);

  if (car >= 'a' && car <= 'z')
    car = toupper (car);
  else
     switch (car) {
       case ' á': car = 'A';
                 break;
       case 'é': car = 'E';
                 break;
       case 'í: car = 'I';
                 break;
       case 'ó': car = 'O';
                 break;
       case 'ú': car = 'U';
                 break;
       case 'ñ': car = 'Ñ';
     }

  putc (car, salida);
  if (ferror (salida))
    err (SALIDA);
}
```

```
 fclose (entrada);
 fclose (salida);
}

void err (int e)
{
 if (e == ENTRADA)
   printf ("Error en la entrada.\n");
 else
   printf ("Error en la salida.\n");
 exit(1);
}
```

Para la utilización de este programa, se debe especificar los nombres de los archivos de entrada y salida en la línea de comandos, así:

A:\> CAPITULO.11\PROG1106 PRUEBA.TXT SALIDA.TXT <ENTER>

Una salida del programa sería:

Copia el archivo: prueba.txt, al archivo saIida.txt.

Para verificar el contenido del archivo SALIDA.TXT, utilizar la siguiente línea de comandos:

A:\> CAPITULO.11\PROG1102 SALIDA.TXT <ENTER>

La salida sería:

Contenido del archivo de texto: SALIDA.TXT

ESTE TEXTO ES UN EJEMPLO,
QUE SERA VERIFICADO EN EL PROGRAMA SIGUIENTE.

11.14. FLUJO DE DATOS BINARIOS

11.14.1. Lectura y Escritura: *fread()* y *fwrite( )*

Las funciones **fread()** y **fwrite()** pueden leer y escribir cualquier tipo de información, siempre y cuando el archivo se haya abierto para operaciones con datos binarios.

Es decir, estas funciones sirven para leer y escribir tipos de datos que ocupan más de un byte, o sea, permiten la lectura y escritura de bloques de cualquier tipo de datos.

Los prototipos de las funciones **fread()** y **fwrite()** son: (SCHILDT, 1994)

```
size_t fread (void *buffer, size_t numero_bytes, size_t cuenta, FILE *pt);
size_t fwrite (void *buffer, size_t numero_bytes, size_t cuenta, FILE *pt);
```

donde:

- **buffer**:

  "En **fread()**", es un puntero genérico a una región de memoria donde se almacenan los datos leídos del archivo.

  "En **fwrite()**", es un puntero genérico a la información que va a ser escrita en el archivo.

  El buffer puede ser, y normalmente es, simplemente la dirección de memoria utilizada para almacenar una variable.

- **numero_bytes**, es el número de bytes a leer o a escribir.

- **cuenta**, determina cuántos elementos se van a leer o escribir, cada uno de tamaño **numero_bytes**.

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

La función **fread()** devuelve el número de elementos leídos, este valor puede ser menor que cuenta: si se produce un error, o si se encuentra el final del archivo antes de completar la lectura.

La función **fwrite()** devuelve el número de elementos escritos, este valor será igual a **cuenta** a menos que se produzca un error.

Por ejemplo, la escritura de un número de punto flotante en un archivo de disco asociado con el "puntero a archivo" **pt**, se haría así:

```
float promedio = 123.45;
fwrite (&promedio, sizeof (float), 1, pt);
```

Se pueden ignorar los valores devueltos por **fread()** y **fwrite()**, pero en un caso real se deben comprobar estos valores por si se producen errores.

Una de las aplicaciones más útiles de **fread()** y **fwrite()** es la lectura y escritura de tipos de datos definidos por el usuario. Por ejemplo, dada la estructura:

```
struct tipo__estruc {
```

```
        char nombre[30];
        unsigned long codigo;
    } cliente[10];
```

La siguiente sentencia escribe el contenido de los 5 primeros elementos del arreglo **cliente** en el archivo que apunta **pt**:

```
fwrite (cliente, sizeof (struct tipo_estruc), 5, pt);
```

Ejercicio

Realizar un programa para escribir el contenido de un arreglo de estructuras en un archivo PERSONAS.DAT, usando una sola función **fwrite()**. Luego leer el archivo elemento por elemento de tipo estructura, usando la función **fread()** para mostrar su contenido en la pantalla.

La información de la estructura está formada por el nombre y el código de una persona.

```
/* PROG1107.C */

#include "stdio.h"
#include "stdlib.h"

#define MAX 10

typedef struct tipo_estruc DATOS;
struct tipo_estruc {
  char nombre[30];
  unsigned long codigo;
};

void main ()
{
  FILE *pt;
  DATOS cliente[MAX], un_cliente;
  int i;

  /* Abrir para escritura y lectura. */
  if ((pt = fopen ("personas.dat", "wb+")) == NULL) {
    printf ("No se puede abrir el archivo: DATOS.DAT.\n");
    exit (1);
  }
```

```
  printf ("Introduzca en el nombre <<ENTER>> para terminar.\n");
  for (i = 0; i < MAX; i++) {
    fflush (stdin);
    printf ("Ingrese el nombre: ");
    gets (cliente[i].nombre);
    if (!*cliente[i].nombre)
      break;        /* Termina si el nombre es nulo. */
    printf ("Ingrese el código: ");
    scanf ("%lu", &cliente[i].codigo);
  }

  /* Almacena el arreglo de estructuras completo en el archivo. */
  fwrite (cliente, sizeof (DATOS), i, pt);

  rewind (pt);

  /* Lee el archivo elemento por elemento y lo imprime en patalla. */
  printf ("\nContenido del archivo.\n");
  printf ("%-30s %-10s\n", "NOMBRE", "CODIGO");
  fread (&un_cliente, sizeof (DATOS), 1, pt);
  while (!feof (pt)) {
    printf ("%-30s %-10lu\n", un_cliente.nombre, un_cliente.codigo);
    fread (&un_cliente, sizeof (DATOS), 1, pt);
  }

  fclose (pt);
}
```

Una salida del programa sería:

```
Introduzca en el nombre <<ENTER>> para terminar.
Ingrese el nombre: Luis Enrique <ENTER>
Ingrese el código: 111 <ENTER>
Ingrese el nombre: Víctor Hugo <ENTER>
Ingrese el código: 222 <ENTER>
Ingrese el nombre: <ENTER>

Contenido del archivo.
NOMBRE          CODIGO
Luis Enrique      111
Víctor Hugo       222
```

11.14.2. E/S de Acceso Directo: *fseek()*

La función **fseek()** sitúa el "indicador de posición del archivo" en cualquier parte del archivo, es decir, con esta función se pueden realizar operaciones de lectura y escritura directa sobre un archivo binario.

El prototipo de la función **fseek()** es: (SCHILDT, 1994)

```
int fseek (FILE *pt, long numero_bytes, int origen);
```

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

- **numero_bytes**, es el número de bytes a partir del parámetro origen que tomará la nueva posición.

- **origen**, es una de las siguientes macros definidas en el archivo de cabecera "***stdio.h***":

ORIGEN	NOMBRE DE LA MACRO
Principio del archivo	SEEK_SET = 0
Posición actual	SEEK_CUR = 1
Fin del archivo	SEEK_END = 2

- **origen** debe ser SEEK_SET, para situar el "indicador de posición del archivo" a **numero_bytes** desde el principio del archivo.

- **origen** debe ser SEEK_CUR, para situar el "indicador de posición del archivo" a **numero_bytes** a partir de la posición actual.

- **origen** debe ser SEEK_END, para situar el "indicador de posición del archivo" a **numero_bytes** a partir del final del archivo.

La función **fseek()** devuelve 0 cuando se ha tenido éxito y un valor distinto de cero cuando se produce un error.

Por ejemplo, si se desea leer el byte 100 del archivo asociado con el "puntero a archivo" **pt**, se haría:

```
fseek (pt, 99L, SEEK_SET);
car = getc (pt);
```

Además, se puede utilizar **fseek()** para recorrer el archivo en múltiplos del tamaño de cualquier tipo de datos (los elementos almacenados deben ser del mismo tipo). Esto

se hace multiplicando el tamaño de los datos por el número del elemento que se quiere alcanzar. Por ejemplo, para posicionarse en el elemento 10 (décimo elemento) del ejercicio del literal anterior, se procedería así:

```
fseek (pt, 9 * sizeof (DATOS), SEEK_SET);
```

La función **fseek()** no se debe utilizar con archivos de texto, ya que las traducciones de caracteres producen errores de localización.

Ejercicio

Buscar en un archivo el byte que se especifica en la línea de comandos junto con el nombre del archivo. Luego mostrar el contenido del byte.

```
/* PROG1108.C */

#include "stdio.h"
#include "stdlib.h"

void main (int argc, char *argv[])
{
  FILE *pt;

  if (argc != 3) {
    printf ("Ingrese: <Programa> <archivo> <byte>\n");
    exit (1);
  }

  if ((pt = fopen (argv[1], "r")) == NULL) {
    printf ("No se puede abrir el archivo: %s.\n", argv[1]);
    exit(1);
  }

  if (fseek (pt, atol (argv[2]), SEEK_SET)) {
    printf ("Error en posicionamiento.\n");
    exit(1);
  }

  printf ("El byte %lu es %c.\n", atol (argv[2]), getc (pt));

  fclose (pt);
}
```

Para la utilización de este programa, se debe especificar el nombre del archivo y el número del byte a buscar en la línea de comandos, así:

A:\> CAPITULO.11\PROG1108 PERSONASDAT 10 <ENTER>

Una salida del programa podría ser:

El byte 10 es u.

Otra aplicación de la función **fseek()** sería la inspección de disco, es decir, permitir examinar el contenido de cualquier archivo que se elija, tanto en código ASCII como en forma hexadecimal.

Ejercicio

Realizar un programa para examinar el contenido de cualquier archivo a elegir, tanto en ASCII como en hexadecimal. Para lo cual se puede ver el archivo en "sectores" de 128 bytes a medida que se mueve por el archivo en cualquier dirección. La salida que se muestra es similar en estilo al formato usado por DEBUG con la orden D (volcado de memoria).

```
/* PROG1109.C */

#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"

#define TAM 128

void mostrar (int num_leidos);

char buf[TAM];

void main (int argc, char *argv[])
{
 FILE *pt;
 int sector, num_leidos;

 if (argc != 2) {
  printf ("Ingrese: <Programa> <archivo>\n");
  exit (1);
 }

 if ((pt = fopen (argv[1], "rb")) == NULL) {
```

```
    printf ("No se puede abrir el archivo: %s.\n", argv[1]);
    exit(1);
  }

  do {
    printf ("\nIntroduzca sector, para salir valores negativos: ");
    scanf ("%ld", &sector);
    if (sector >= 0) {
      if (fseek (pt, sector * TAM, SEEK_SET))
        printf ("Error de b£squeda.\n");
      if ((num_leidos = fread (buf, 1, TAM, pt)) != TAM)
        printf ("Se alcanzado el fin del archivo.\n");
       mostrar (num_leidos);
    }
  } while (sector >= 0);
  fclose (pt);
}

/* Mostrar el contenido del archivo. */

void mostrar (int num_leidos)
{
  int i, j;

  for (i = 0; i < num_leidos / 16; i++) {
    for (j = 0; j < 16; j++)
      printf ("%3X", buf[i * 16 + j]);
    printf ("  ");
    for (j = 0; j < 16; j++) {
      /* La función isprintf() devuelve cierto si el caracter es imprimible, */
      /* caso contrario falso. Se encuentra en el archivo "ctype.h".        */
      if (isprint (buf[i * 16 + j]))
        printf ("%c", buf[i * 16 + j]);
      else
        printf (".");
    }
    printf ("\n");
  }
}
```

Para la utilización de este programa, se debe especificar el nombre del archivo en la línea de comandos, así:

```
A:\> CAPITULO.11\PROG1109 PRUEBA.TXT <ENTER>
```

Una salida del programa podría ser:

Introduzca sector, para salir valores negativos: 0 <ENTER>
Se alcanzado el fin del archivo.
45 73 74 65 20 74 65 78 74 6F 20 65 73 20 75 6E   Este texto es un
20 65 6A 65 6D 70 6C 6F 2C D A 71 75 65 20 73   ejemplo,..que s
65 72 FF AO 20 76 65 72 69 66 69 63 61 64 6F 20 65   er.verificado e
6E 20 65 60 20 73 69 67 75 69 65 6E 74 65 20 70   n el siguiente p

Introduzca sector, para salir valores negativos: 1 <ENTER>
Se alcanzado el fin del archivo.

Introduzca sector, para salir valores negativos: -1 <ENTER>›

11.14.3. Valor del Indicador de Posición del Archivo: *ftell()*

La función **ftell()** devuelve el valor actual del "indicador de posición del archivo" para el flujo binario especificado.

El prototipo de la función **ftell()** es: (SCHILDT, 1994)

long ftell (FILE *pt);

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**

La función **ftell()** devuelve -1L cuando se produce un error, en caso contrario devolvería:

- "***Para flujos binarios***", el valor del número de bytes desde el principio del archivo.

- "***Para flujos de texto***", un valor indefinido de acuerdo al compilador, debido a las posibles transformaciones de caracteres.

Si el flujo es incapaz de búsquedas aleatorias (por ejemplo si se trata de un terminal) el valor devuelto queda indefinido.

Por ejemplo, para determinar el valor actual del indicador de posición del flujo apuntado por **pt**, sería:

```
long i;
if ((i = ftell (pt)) == -1L)
```

```
printf ("Se ha producido un error en el archivo.\n");
```

La función **ftell()** se utiliza como argumento de la función **fseek()**.

11.15. ESCRITURA Y LECTURA DE CUALQUIER TIPO DE DATOS: fprintf() y fscanf()

Las funciones **fprintf()** y **fscanf()** se comportan exactamente como **printf()** y **scanf()** para imprimir y leer cualquier tipo de datos, excepto que operan sobre archivos de texto.

Los prototipos de las funciones **fprintf()** y **fscanf()** son: (SCHILDT, 1994)

```
int fprintf (FILE *pt, const char *cadena_contro1, ...);
int fscanf (FILE *pt, const char *cadena_control, ...);
```

donde:

- **pt**, es el "puntero a archivo" devuelto por **fopen()**.

Las funciones **fprintf()** y **fscanf()** dirigen sus operaciones de E/S al archivo asociado con el puntero a archivo **pt**.

Por ejemplo, si se desea escribir en la impresora se haría:

```
fprintf (stdprn, "Este texto se imprime en la impresora");
```

Cuando se escribe con la función **fprintf()**, se debe separar los datos con espacios en blanco, tabulados y cambio de línea, para diferenciar cada uno de esos datos. Esto es equivalente a como se diferencian los datos en una lectura desde teclado.

Aunque con **fprintf()** y **fscanf()** es más fácil escribir y leer datos ordenadamente en archivos de disco, no siempre son los más eficientes, debido a que los datos se escriben con formato ASCII como aparecerían en pantalla, en lugar de en binario, produciéndose una conversión de tipos para la lectura y la escritura. Por esto, si lo importante es la velocidad o el tamaño del archivo, se debe utilizar **fread()** y **fwrite()**.

Ejercicio

Realizar un programa para manejar una lista telefónica en un archivo de disco, esta lista contiene los nombres y números telefónicos, que deben ser introducidos desde teclado. Además, el programa debe buscar un número dado un nombre, y por último mostrar la lista telefónica completa. Usar para la E/S las funciones **fscanf()** y **fprintf()**.

```
/* PROG1110.C */
```

```
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "string.h"

void otro (void), buscar (void), listar (void);
char menu (void);

void main (void)
{
  for (;;) {
    switch (menu ()) {
      case 'A' : otro ();
                 break;
      case 'B' : buscar ();
                 break;
      case 'L' : listar ();
                 break;
      case 'T' : exit (0);
    }
  }
}

/* Mostrar el menú y obtener la opción. */

char menu (void)
{
  char ch;

  printf ("(A)ñadir\n");
  printf ("(B)uscar\n");
  printf ("(L)istar\n");
  printf ("(T)erminar\n");

  do{
    printf ("\nIngrese la opción...");
    ch  = toupper (getche ());
  } while (!strchr ("ABLT", ch));

  printf ("\n\n");
  return ch;
}
```

```
/* A¤adir un nombre y un número a la lista. */

void otro (void)
{
 FILE *pt;
 char nombre[31];
 unsigned long num;

 /* Abrir el archivo para añadir. */

 if ((pt = fopen ("list_tel.dat", "a")) == NULL) {
  printf ("No se puede abrir el archivo: LIST_TEL.DAT.\n");
  exit (1);
 }

 printf ("Introducir un solo nombre: ");
 fscanf (stdin, "%s", nombre);      /* Lee del teclado. */
 printf ("Introducir un número: ");
 fscanf (stdin, "%lu", &num);       /* Lee del teclado. */
 fscanf (stdin, "%*c");   /* Elimina ENTER del flujo de entrada. */

 fprintf (pt, "%s %lu\n", nombre, num); /* Escribe en el archivo por líneas. */

 fclose (pt);
}

/* Buscar un número dado un nombre. */

void buscar (void)
{
 FILE *pt;
 char nombre[31], refer[31];
 unsigned long num;

 /* Abrir el archivo para leer. */
 if ((pt = fopen ("list_tel.dat", "r")) == NULL) {
  printf ("No se puede abrir el archivo: LISTA_TE.DAT.\n");
  exit (1);
 }

 printf ("Introduzca el nombre a buscar: ");
 gets (refer);

 /* Buscar el número. */
```

```
  while (!feof (pt)) {
    fscanf (pt, "%s%lu%*c", nombre, &num);    /* Lee del archivo una línea. */
    if (!strcmp (nombre, refer)) {
      printf ("NOMBRE: %s, NUMERO: %lu\n", nombre, num);
      printf ("Digite una tecla para continuar...");
      getch ();
      break;
    }
  }

  fclose (pt);
}

/* Listar la lista completa. */

void listar (void)
{
  FILE *pt;
  char nombre[31];
  unsigned long num;

  /* Abrir el archivo para leer. */
  if ((pt = fopen ("list_tel.dat", "r")) == NULL) {
    printf ("No se puede abrir el archivo: LISTA_TE.DAT.\n");
    exit (1);
  }

  /* Imprimir la lista. */
  printf ("%-30s %6s\n","NOMBRE", "NUMERO");
  fscanf (pt, "%s%lu%*c", nombre, &num);    /* Lee del archivo una l¡nea. */
  while (!feof (pt)) {
    printf ("%-30s %-6lu\n", nombre, num);
    fscanf (pt, "%s%lu%*c", nombre, &num);  /* Lee del archivo una l¡nea. */
  }

  printf ("Digite una tecla para continuar...");
  getch ();

  fclose (pt);
}
```

La corrida del programa se debe realizar de tal manera que se ingresen nombres y números de teléfonos en la opción A, para luego realizar la búsqueda de un número

mediante su nombre en la opción B y escribir el listado telefónico completo en la opción L. La opción T finaliza el programa.

11.16. CONEXIÓN ENTRE LA CONSOLA Y LA E/S DE ARCHIVOS

El lenguaje C hace poca distinción entre la E/S por consola y la E/S en archivos, por lo que las funciones de E/S pueden ser utilizadas indistintamente.

11.16.1. Utilización de las Funciones de Archivos para Consola

Debido a que los flujos estándar son "punteros a archivos", el sistema E/S de archivos con buffer se puede utilizar para realizar operaciones de entrada y salida sobre la consola.

Las funciones de E/S por consola son versiones especiales de las funciones análogas para archivos. Entonces, se puede realizar E/S por consola utilizando cualquiera de las funciones de archivos del lenguaje C.

Por ejemplo, para imprimir el caracter 'c' en pantalla mediante la función **putc()** para archivos, se haría:

```
putc ('c', stdout);
```

NOTA: Se pueden utilizar los flujos estándar como "punteros a archivos" en cualquier función que use una variable de tipo FILE.

11.16.2. Utilización de las Funciones de Consola para Archivos

Normalmente los flujos estándar están asociados a la consola, pero pueden ser redirigidos por el sistema operativo hacia algún otro dispositivo, estos sistemas operativos son: UNIX, MAC OS, etc. A este proceso se le conoce como "Reenvío" o "Redirección". (SCHILDT, 1994)

Debido a que todas las funciones de E/S por consola operan en **stdin** y **stdout**, se puede realizar E/S sobre archivos utilizando las funciones de E/S por consola.

Por ejemplo, el siguiente programa lee desde teclado un texto hasta que sea fin de archivo, y muestra ese texto en pantalla:

```
#include "stdio.h"

void main ()
{
    char ch;
```

```
    while ((ch = getchar ()) != EOF)
    putchar (ch);
}
```

Suponer que este programa se llama PRUEBA. Luego de hacerlo ejecutable se debe realizar la redirección de la E/S, sea de **stdin**, de **stdout** o de ambos, para ser reasignado a un archivo.

1. ***Redirección de* stdout**

    Al ejecutar el programa PRUEBA en un entorno DOS u OS/2 se tendría que la entrada del texto es desde el teclado y la salida de PRUEBA será escrita, en un archivo llamado SALIDA, en lugar de serlo en la pantalla. La sentencia para el efecto sería:

    ```
    A:\> PRUEBA > SALIDA <ENTER>
    ```

2. ***Redirección de* stdin**

    Al ejecutar el programa PRUEBA, la entrada será desde el archivo ENTRADA y la salida será en la pantalla. La sentencia para el efecto sería:

    ```
    A:\> PRUEBA < ENTRADA <ENTER>
    ```

3. ***Redirección de* stdin *y* stdout**

    Al ejecutar el programa PRUEBA, la entrada será desde el archivo ENTRADA y la salida será escrita en un archivo llamado SALIDA. La sentencia para el efecto sería:

    ```
    A:\> PRUEBA < ENTRADA > SALIDA <ENTER>
    ```

Ejercicio

Desarrollar un programa para llevar una lista de correos. La información será almacenada en un archivo de estructuras del siguiente tipo:

```
typedef struct lista TIPO_LISTA;
    struct lista {
    char nombre[31];
    char calle[l5];
    char ciudad[15];
    char provincia[11];
    char codigo[5];
```

};

Realizar un menú como el siguiente:

a) Inicializar el archivo.
b) Ingresar la información desde el teclado y almacenarla en el archivo.
c) Consulta individual en base al nombre.
d) Consulta total de acuerdo a la siguiente cabecera:

ORD	NOMBRE	CALLE	CIUDAD	PROVINCIA	CODIGO

e) Modificar una estructura en base al nombre.
f) Salir.

```
/* PROG1111.C */

#include "stdio.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"

typedef enum boolean BOOLE;
enum boolean {false, true};

typedef struct lista TIPO_LISTA;
struct lista {
  char nombre[31];
  char calle[15];
  char ciudad[15];
  char provincia[11];
  char codigo[5];
};

char menu (void);
void inicializar (FILE *F), ingresar (FILE *F);
void consulta (FILE *F), consulta_total (FILE *F);
void modificar (FILE *F);
BOOLE buscar (FILE *F, char * nombre, TIPO_LISTA *datos);
FILE *abrir_archivo (char *nom_archivo, char *modo);
BOOLE continuar (void);
char opcion (char *opciones);
void mostrar_datos (TIPO_LISTA datos);
void pausa (void);
```

```
void main (void)
{
  FILE *F;

  for (;;) {
    switch (menu ()) {
      case 'a' : inicializar (F);
                 break;
      case 'b' : ingresar (F);
                 break;
      case 'c' : consulta (F);
                 break;
      case 'd' : consulta_total (F);
                 break;
      case 'e' : modificar (F);
                 break;
      case 'f' : exit (0);
    }
  }
}

/* Selecciona una opción del menú. */

char menu ()
{
  printf ("\n\n\n");
  printf ("(a) Inicializar archivo.\n");
  printf ("(b) Ingresar información.\n");
  printf ("(c) Consulta individual.\n");
  printf ("(d) Consulta total.\n");
  printf ("(e) Modificar una estructura.\n");
  printf ("(f) Terminar.\n");

  return tolower (opcion ("ABCDEF"));
}

/* Inicializar el archivo. */

void inicializar (FILE *F)
{
  printf ("\n\nCrea el archivo CORREO.\n");
  printf ("La información se pierde.\n");
  if (continuar ()) {
    F = abrir_archivo ("CORREO", "wb");
```

```
    fclose (F);
  }
}

/* Ingresa la información desde teclado y almacena en el archivo. */

void ingresar (FILE *F)
{
  TIPO_LISTA datos;

  printf ("\n\nIngreso de datos\n");
  F = abrir_archivo ("CORREO", "rb+");
  printf ("\nIngrese el nombre con <ENTER> termina: ");

  while (*gets (datos.nombre) != '\0') {
    if (!buscar (F, datos.nombre, &datos)) {
      printf ("Ingrese la calle    : ");
      gets (datos.calle);
      printf ("Ingrese la ciudad   : ");
      gets (datos.ciudad);
      printf ("Ingrese la provincia: ");
      gets (datos.provincia);
      printf ("Ingrese el código   : ");
      gets (datos.codigo);
      fseek (F, 0, SEEK_END);
      if (fwrite (&datos, sizeof (TIPO_LISTA), 1, F) != 1)
        printf ("ERROR de escritura en el archivo.\n");
    }
    else {
      printf ("\nNombre repetido\n\n");
      pausa ();
    }

    printf ("Ingrese el nombre con <ENTER> termina: ");
  }
  fclose (F);
}

/* Consulta de una estructura. */

void consulta (FILE *F)
{
  TIPO_LISTA datos;
  char nombre[31];
```

```
  printf ("\n\nConsulta de una estructura.\n\n");
  F = abrir_archivo ("CORREO", "rb");
  printf ("Ingrese el nombre: ");
  gets (nombre);
  if (buscar (F, nombre, &datos))
    mostrar_datos (datos);
  else
    printf ("\nNo se encuentra esa persona.\n");
  fclose (F);
  pausa ();
}

/* Realiza la consulta total de la lista de correos. */

void consulta_total (FILE *F)
{
  TIPO_LISTA datos;
  int ord = 0;

  printf ("\n\nConsulta total de la lista de correos.\n\n");
  F = abrir_archivo ("CORREO", "rb");
  printf ("OR NOMBRE                    CALLE        CIUDAD");
  puts ("        PROVINCIA  COD");

  if (fread (&datos, sizeof (TIPO_LISTA), 1, F) != 1)
    if (! feof (F))
      printf ("ERROR de lectura en el archivo.\n");

  while (! feof (F)) {
    printf ("%02d %-30s %-14s %-14s %-10s %-4s\n",
          ++ord, datos.nombre, datos.calle, datos.ciudad, datos.provincia,
          datos.codigo);

  if (ord % 18 == 0) {
      pausa ();

      printf ("OR NOMBRE                    CALLE        CIUDAD");
      puts ("       PROVINCIA  COD");
    }
    if (fread (&datos, sizeof (TIPO_LISTA), 1, F) != 1)
      if (! feof (F))
        printf ("ERROR de lectura en el archivo.\n");
  }
```

```
 fclose (F);
 pausa ();
}

/* Modifica una estructura. */

void modificar (FILE *F)
{
 TIPO_LISTA datos;
 char *nombre;

 printf ("\n\nModifica una estructura.\n\n");
 F = abrir_archivo ("CORREO", "rb+");
 printf ("Ingrese el nombre: ");
 gets (nombre);

 if (buscar (F, nombre, &datos))
   for (;;) {                     /* Para cambiar varios campos. */
     mostrar_datos (datos);
     printf ("\nModificar:\n");

     if (continuar ()) {
       switch (opcion ("NCIPD")) {
         case 'N' : printf ("Ingrese el nombre   : ");
                    gets (datos.nombre);
                    break;
         case 'C' : printf ("Ingrese la calle    : ");
                    gets (datos.calle);
                    break;
         case 'I' : printf ("Ingrese la ciudad   : ");
                    gets (datos.ciudad);
                    break;
         case 'P' : printf ("Ingrese la provincia: ");
                    gets (datos.provincia);
                    break;
         case 'D' : printf ("Ingrese el código   : ");
                    gets (datos.codigo);
       }
     }
     else {
       if (fseek (F, ftell (F) - sizeof (TIPO_LISTA), SEEK_SET))
         printf ("ERROR de posicionamiento del indicador de posición.\n");
       if (fwrite (&datos, sizeof (TIPO_LISTA), 1, F) != 1)
         printf ("ERROR de lectura en el archivo.\n");
```

```
        break;
      }
    }
  else {
    printf ("\nNo se encuentra esa persona.\n");
    pausa ();
  }
  fclose (F);
}

/* Busca una estructura, si le encuentra retorna true, si no false. */

BOOLE buscar (FILE *F, char *nombre, TIPO_LISTA *datos)
{
  TIPO_LISTA aux_datos;
  BOOLE esta = false;

  rewind (F);
  if (fread (&aux_datos, sizeof (TIPO_LISTA), 1, F) != 1)
    if (! feof (F))
      printf ("ERROR de lectura en el archivo.\n");
  while (! feof (F)) {
    if (!strcmp (nombre, aux_datos.nombre)) {
      *datos = aux_datos;
      esta = true;
      break;
    }
    if (fread (&aux_datos, sizeof (TIPO_LISTA), 1, F) != 1)
      if (!feof (F))
        printf ("ERROR de lectura en el archivo.\n");
  }
  return esta;
}

/* Abre un archivo con nombre nom_archivo y modo modo. */

FILE *abrir_archivo (char *nom_archivo, char *modo)
{
  FILE *F;

  if ((F = fopen (nom_archivo, modo)) == NULL) {
    printf ("No se puede abrir el archivo: CORREO.\n");
    exit (1);
  }
```

```
  return F;
}

/* Función para continuar (S/N). */

BOOLE continuar (void)
{
  char ch;

  printf ("Desea continuar (S/N)? . ");
  do {
    ch = toupper (getch ());
  } while (ch != 'S' && ch != 'N');
  printf ("%c\n", ch);
  return (ch == 'S') ? true : false;
}

/* Función para elegir la opción. */

char opcion (char *opciones)
{
  char s;

  printf ("Elija una opci¢n. ");
  do {
    s = toupper (getch ());
  } while (! strchr (opciones, s));
  printf ("%c\n",s);
  return s;
}

/* Imprime los datos de una estructura. */

void mostrar_datos (TIPO_LISTA datos)
{
  printf ("\n(N)ombre   : %s\n", datos.nombre);
  printf ("(C)alle    : %s\n", datos.calle);
  printf ("c(I)udad   : %s\n", datos.ciudad);
  printf ("(P)rovincia: %s\n", datos.provincia);
  printf ("C¢(D)igo   : %s\n", datos.codigo);
}

/* Realiza una pausa. */
```

```
void pausa (void)
{
  printf ("\nPulse una tecla para continuar. ");
  getch ();
}
```

PROBLEMAS PROPUESTOS

1) Realizar un programa que lea un texto caracter a caracter usando la función **getchar()**, convierta el texto de minúsculas a mayúsculas, y lo almacene en un archivo usando **putc()**. El archivo debe ser abierto solo para escritura y la función de biblioteca **toupper()** realiza la conversión de minúsculas a mayúsculas.

   El archivo a almacenar el texto se llama "**muestra.dat**", que es ingresado desde teclado en una variable cadena de caracteres.

   Luego el programa debe leer una línea de texto del archivo "**muestra.dat**", caracter a caracter, y mostrar el texto en la pantalla. Las funciones de biblioteca a utilizar son **getc()** y **putchar()** para leer y mostrar los datos. El archivo "**muestra.dat**" debe ser abierto solo para lectura, si el archivo "**muestra.dat**" no se puede abrir, se genera un mensaje de error.

2) Realizar un programa que utilice las funciones **fgetc()** y **fputc()**, para darle al usuario la opción de leer desde la entrada estándar y escribir a la salida estándar.

3) Realizar un programa que utilice las funciones **fgetc()** y **fputc()**, para darle al usuario la opción de leer de un archivo especificado y escribir a un archivo especificado. El usuario debe introducir desde teclado los nombres de los archivos requeridos en cadenas de caracteres.

4) Realizar un programa que lea caracteres desde teclado y los almacene en un archivo de disco, hasta que el usuario teclee fin de archivo (EOF). El nombre del archivo debe ser ingresando desde la línea de comandos. Luego el programa lee el archivo de disco para imprimir su contenido en la pantalla, controlando que se muestre 20 líneas por pantalla.

5) Realizar un programa para imprimir un archivo de texto, cuyo nombre es ingresado desde la línea de comandos. El texto se imprime en líneas de 80 caracteres, poniendo al final de cada línea un guión (-) considerando:

   - Si la línea es menor de 80 caracteres, completar con blancos.

   - Si la línea es mayor de 80 caracteres, cortar la línea y formar otra línea.

6) Realizar un programa que lea un archivo de texto, y que determine la frecuencia de cada palabra del texto. La palabra y su frecuencia deben almacenarse en una estructura, por lo tanto usar un arreglo dinámico de estructuras para almacenar todas las palabras con sus frecuencias. Por último, el programa debe imprimir la lista de las palabras ordenadas alfabéticamente de la siguiente forma:

ORD PALABRA FRECUENCIA

Ingresar desde la línea de comandos el nombre del archivo de texto, pero si no se detalla dicho nombre se debe ingresar desde teclado el nombre del archivo y el texto.

7) En un archivo de texto llamado "OFICIO" se tiene almacenado una carta de cotización de computadoras de la siguiente manera:

```
Quito, &.

Sr.
@.
I.
Presente.
```

Realizar un programa que ingrese: la fecha, el nombre de la institución o de la persona a la que se dirige la cotización y el cargo si trabaja en una institución; para reemplazar a los símbolos &, @ y I, respectivamente.

A continuación imprimir el archivo de texto con la nueva información, la impresión debe hacerse máximo en 60 líneas por página, por lo que si existe un número mayor imprimir en diferentes hojas.

NOTAS:
- La información del archivo "OFICIO" debe ser ingresada.
- El archivo no debe modificarse en el disco.
- El texto en el archivo está almacenado por líneas de 80 caracteres como máximo.
- En la búsqueda y reemplazo de los símbolos se debe usarse solo funciones de cadenas.

8) Escribir un programa que codifique y decodifique múltiples líneas de texto usando el procedimiento de codificación/decodificación que se muestra a continuación:

"*Para codificar una línea de texto*", proceder como sigue:

a) Convertir cada caracter, incluido espacios en blanco, en su equivalente ASCII.
b) Generar un entero positivo aleatorio, entre 0 y 127, con la funciones estándares del lenguaje C. Añadir este entero al ASCII equivalente a cada caracter. El mismo entero aleatorio será usado para la línea de texto completa.
c) Suponer que N1 representa el menor valor permisible en ASCII y N2 es el mayor valor permisible en ASCII. Si el número obtenido en el paso b) excede

de N2, se le resta N2 y se suma el resultado a N1. Así mismo el número codificado estará siempre entre N1 y N2 y representa siempre algún caracter ASCH.

"*Para decodificar una línea de texto*", el procedimiento es invertido, pero se debe asegurar que el entero es el mismo que se usa para codificar.

El programa almacena el texto codificado en un archivo de modo que pueda ser recuperado y decodificado en cualquier momento, con las siguientes características:

1. Introducir el texto desde teclado, codificarlo y almacenar en un archivo el texto codificado.
2. Recuperar el texto codificado y mostrarlo en su forma codificada.
3. Recuperar el texto codificado, decodificarlo y mostrarlo en su forma decodificada.
4. Finalizar.

Además, el programa debe generar enteros aleatorios para decodificar cada línea consecutiva: el primer entero aleatorio será usado para codificar la primera línea de texto, el segundo para codificar la segunda línea, y así sucesivamente. Incluir alguna forma de reproducir la misma secuencia de números enteros aleatorios, de modo que los mismos enteros aleatorios puedan usarse para decodificar el texto.

9) Hacer un programa que lea un archivo de personas, siendo la información para cada persona el nombre y la fecha de nacimiento. El archivo debe estar ordenado en forma alfabética, y una persona puede añadirse o eliminarse del archivo. Por último, hacer un reporte de acuerdo al siguiente ejemplo:

ORD	NOMBRE	FECHA DE NACIMIENTO
1	CORTEZ HUGO	8-FEBRERO-92
2	MARTINEZ CARLOS	4-ENERO-80

…

NOTA: Usar un arreglo de punteros para ordenar la información.

10) Construir un programa que convierta números romanos a números arábigos. La entrada se debe realizar desde un archivo de texto llamado "DATOS", como una secuencia de números romanos separados por uno o varios espacios en blanco o fin de línea, para cada uno de los cuales debe obtenerse el número arábigo correspondiente.

La siguiente tabla indica la relación entre los dos sistemas de números:

SÍMBOLOS ROMANOS	SÍMBOLOS ARÁBIGOS
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

NOTAS:
- La información del archivo "DATOS" debe ser ingresada.
- Realizar una función lógica que devuelva **true** si los datos en el archivo están correctos, caso contrario **false**. Si es **false** ya no debe seguir leyendo el archivo.

11) "***El generador de pig latin***" es una forma codificada de escribir y de hablar que suelen usar los niños ingleses como juego. Una palabra en "pig latin" se forma transponiendo el primer sonido, generalmente la primera letra, de una palabra original al final de la misma y añadiendo al final la letra 'a'. Tomar en cuenta marcas de puntuación, letras mayúsculas y sonidos de letras dobles. Por ejemplo, la palabra "perro" se convierte en "erropa".

    Escribir un programa que introduzca múltiples líneas de texto desde el teclado, para almacenar en un archivo el texto completo en inglés y en otro archivo el correspondiente "pig latin". Incluir en el programa la creación de un menú, que permita al usuario seleccionar cualquiera de las siguientes características:

    a) Introducir el texto convertido a "pig latin" y almacenarlo. (almacenar ambos el texto original y el "pig latin" como se indica arriba.)
    b) Leer un texto previamente introducido en un archivo y mostrarlo.
    c) Leer el "pig latin" equivalente al anterior y mostrarlo.
    d) Finalizar.

12) Realizar un programa que ingrese las materias que se dictan en un curso y las almacene en un archivo. Luego ingresar los nombres de los alumnos de acuerdo a cada materia que se vayan a matricular, considerando que no se repitan los alumnos, en este caso indicar con un mensaje "Alumno existente". E1 ingreso de los alumnos por materia debe hacerse en otro(s) archivo(s).

    El programa debe tener la cualidad de poder ingresar nuevos alumnos sin borrar o alterar los anteriores, y realizar una consulta por alumno indicando las materias que coge. Por último, hacer un reporte de los alumnos en orden alfabético de acuerdo a la materia que se hayan matriculado.

13) Realizar un programa que lea desde el teclado la siguiente información de una persona: el nombre de la persona, estado civil, nombre del cónyuge y número de hijos; considerando el siguiente cuadro que utilizará una **union** como estructura de datos:

ESTADO CIVIL	SE NECESITA ADICIONALMENTE
Soltero/a (S)	
Casado/a (C) o Unión Libre (U)	Nombre de cónyuge y número de hijos
Viudo/a (V) o Divorciado/a (D)	Número de hijos

Los datos ingresados de las personas se almacenan en un archivo, hasta que se digite 'N' cuando aparezca la pregunta:

¿Desea ingresar otra persona (S/N)?

Por último, imprimir un reporte como el siguiente:

ORD	NOMBRE	ESTADO CIVIL	NOMBRE CONYUGE	NUMERO DE HIJOS

14) Realizar un programa que acepte la siguiente información para cada equipo de la liga de beisbol o fútbol.

- Nombre del equipo.
- Número de victorias.
- Número de derrotas.

Para un equipo de beisbol, añadir la siguiente información:

- Número de bolas bateadas con éxito.
- Número de carreras.
- Número de errores.
- Número de juegos extras.

Para un equipo de fútbol, añadir la siguiente información:

- Número de empates.
- Número de goles.
- Número de puntos.
- Número de contra ataques.

Introducir desde teclado esta información para todos los equipos de liga, luego almacenarla en un archivo, donde cada componente del archivo debe ser una

estructura que contenga los datos de un equipo. Hacer uso de una unión para representar la información variable (beisbol o fútbol) que se incluye como parte de la estructura; la unión debe a su vez contener dos estructuras: una para los datos de beisbol y otra para el fútbol.

Realizar las siguientes operaciones:

a) Crear el archivo.
b) Añadir nuevos equipos.
c) Actualizar registros existentes.
d) Ordenar y escribir la lista de equipos en un arreglo dinámico de acuerdo al "número de victorias" con sus datos respectivos de beisbol o fútbol.

NOTAS:
- Introducir el nombre del archivo como argumento de la línea de comandos.
- La declaración de la estructura realizar con **typedef**.

15) Se tiene la siguiente estructura que corresponde a la información de una persona:

```
struct persona {
    char Primer_Nombre[15];
    char Ultimo_Nombre[15];
    char edad[2];
};
```

Escribir un programa que lleve a cabo cada uno de los siguientes literales:

a) Inicializar el archivo "**nom_edad.dat**" de tal forma que existan 100 estructuras con la información: **Ultìmo_Nombre** = "sin asignar", **Primer_Nombre** = "", y **edad** = "0".
b) Introducir **n** estructuras con los campos: apellido, nombre y edad, para escribirlos en el archivo.
c) Actualizar una estructura, si no existe información en la estructura, indicar al usuario "No información".
d) Borrar una estructura que tenga información mediante la reinicialización de dicha estructura en particular.

16) Una compañía de seguros ofrece tres tipos de pólizas: vida, automóvil y casa. Para los tres tipos de pólizas de seguros es necesario tener el nombre del asegurado, la dirección, la cantidad asegurada, el pago de la póliza mensual que es el 3% de la cantidad asegurada y un número de póliza. Considerar el siguiente cuadro para cada póliza:

POLIZA DE SEGUROS	SE NECESITA ADICIONALMENTE
Vida (V)	- Fecha de nacimiento del asegurado (año, mes y día). Debe ser una estructura de tres campos enteros. - Nombre del beneficiario.
Automóvil (A)	- Número de placa. Debe ser cadena de 8 caracteres - Provincia. - Modelo del auto. Debe ser cadena de 5 caracteres. - año del auto. Debe ser entero. - Alarma. Otra unión: - Falso: No hay descuento. - Verdadero: Un descuento del 10% del pago mensual de la póliza.
Casa (C)	- Año de construcción de la casa. - Seguridades existentes. Otra unión. - Ninguna, rejas o perro: No hay descuento - Alarma: Un descuento del 10% - Vigilante: Un descuento del 20% - Alarma y vigilante: Un descuento del 30%

Realizar un programa que lea la información para cada cliente y almacenarla en un archivo, hasta digitar 'N' cuando aparezca la pregunta:

¿Desea continuar S/N?

Luego el programa consulta todos los datos de un cliente de acuerdo al número de póliza, y por último hace un reporte como el siguiente:

ORD	NOMBRE	TIPO DE POLIZA	NUMERO POLIZA	CANTIDAD DESCUENTO ASEGURADA	PAGO DE LA POLIZA MENSUAL

NOTAS:
- Usar uniones como estructuras de datos.
- Validar el ingreso, en donde sea necesario.

17) Se necesita mantener un inventario de una ferretería que indique cuáles son las herramientas que se tiene, cuántas se tiene y el costo de cada una.

Escribir un programa que inicialice el archivo "**tools.dat**" a 100 registros vacíos, que permita introducir los datos correspondientes a cada herramienta, enlistar todas las herramientas, borrar un registro correspondiente a una herramienta que ya no se posea, y actualizar cualquier información dentro del archivo. El número de identificación de la herramienta deberá ser el número de registro.

Luego el programa debe realizar un reporte como el siguiente:

REGISTRO #	NOMBRE HERRAMIENTA	CANTIDAD	COSTO
3	Remachadora	7	57.98
17	Lima eléctrica	76	11.99
24	Martillo	21	11.00
39	Taladro	3	79.50

18) Escribir un programa que mantenga una lista de nombres, direcciones y números de teléfono en orden alfabético (por apellidos).

El programa debe incluir un menú, que permita al usuario seleccionar cualquiera de las siguientes características:

a) Añadir un nuevo registro.
b) Borrar un registro.
c) Modificar un registro existente.
d) Recuperar y mostrar un registro completo para un nombre dado.
e) Generar una lista completa de todos los nombres, direcciones y números de teléfono.
f) Finalizar.

Ordenar los registros cada vez que se añada un registro nuevo o si se borra un registro, de modo que todos los registros se mantengan siempre en orden alfabético.

19) Un archivo de datos "**datos.dat**" almacena la información de los alumnos de un curso, la información de cada alumno es una estructura de dos campos: un código lógico para indicar si aprueba o no el curso y el nombre del alumno.

Realizar un programa que lea la información desde el teclado, luego imprime un reporte en orden alfabético como el siguiente:

LISTA DE ALUMNOS

ORD	NOMBRE	APRUEBA
1	Acosta María	no
2	Enríquez Ana	si
…		

20) Realizar un programa que almacene en un archivo la información de una persona formada por los siguiente datos: nombre, sexo (M = mujer, H = hombre) y sueldo que gana. Almacenar las personas hasta que se digite 'N' cuando aparezca el mensaje:

¿Desea continuar (SN)?

Imprimir en forma ordenada alfabéticamente dos reportes; uno para los hombres y otro para las mujeres:

HOMBRES		
ORD	NOMBRE	SUELDO

.
.

MUJERES		
ORD	NOMBRE	SUELDO

.
.

Además buscar la persona que gane el mayor sueldo e indicar si .es hombre o mujer. Para esta búsqueda usar recursividad.

21) Escribir un programa que genere un archivo conteniendo una lista de países y sus correspondientes capitales. Colocar el nombre de cada país y su capital correspondiente en una estructura.

El programa debe tener el siguiente menú, que permita ejecutar una de las siguientes operaciones:

a) Consulta:
   1. Determinar la capital de un país especificado.
   2. Determinar el país cuya capital es especificada.
b) Añadir una nueva estructura.
c) Borrar una estructura.
d) Generar un listado de todos los países y sus capitales.
e) Terminar.

22) Crear un programa que contenga los registros de clientes cuya composición es como sigue:

```
struct fecha {
    int mes;
```

```
        int dia;
        int anio;
    };

    struct cuenta {
        char nombre[80];
        char calle[8O];
        char ciudad[80];
        int cuenta_num;
        char cuenta_tipo;
        float anterior_saldo;
        float nuevo_saldo;
        float pago;
        struct fecha ultimo_pago;
    } registro;
```

Ingresar la fecha actual para procesar los registros de clientes. Para cada cliente se leerán el nombre del cliente (nombre), la calle (calle), la ciudad (ciudad), el número de cuenta (cuenta_num), el saldo inicial (anterior_saldo). A continuación asignar el valor inicial 0 a los miembros de la estructura **nuevo_saldo** y pago, asignar el caracter 'C' a **cuenta_tipo** (que indica el estado actual) y la fecha actual se asigna a **uItimo_pago**. Cada registro de cliente almacenar en un archivo de solo escritura llamado "**registro.dat**".

El ingreso de clientes se realiza hasta que el nombre de un cliente comience con los caracteres "FIN", en mayúsculas y minúsculas.

Además, el programa debe tener un menú para leer los registros, añadir nuevos registros, borrar registros antiguos, y modificar registros existentes. Se debe mantener los registros en orden alfabético en un solo archivo.

23) Leer registros de libros que contienporl autor, título del libro y número de volúmenes, para almacenarlos en un archivo que debe estar ordenado alfabéticamente respecto al título del libro, si el archivo no existe crearlo caso contrario no perder la información. Además, los libros pueden añadirse o eliminarse del archivo de acuerdo al mensaje:

Pulse I para insertar; B para borrar o F para finalizar.

Al insertar un libro no puede haber 2 libros con el mismo título y al tratar de borrar un libro que no existe debe salir el mensaje:

No existe el libro: ________________________

Por último realizar un reporte como el siguiente:

TITULO DEL LIBRO	NOMBRE DEL AUTOR	VOLUMENES

24) Realizar un programa que lea desde teclado la información de un alumno para almacenarla en un archivo de datos llamado "**informa**". La información para cada alumno tiene la siguiente estructura:

```
#define MAXMAT 5
#define MAXNOT 4

typedel enum meses NOTAS_MESES;
enum meses {primer, segundo, tercer, cuarto};
typedel enum materias NOTAS_MATERIAS;
enum materias {Lenguaje_C, Quimica, Fisica, Algebra, Informatica};
typedel struct alumno DATOS;
struct alumno {
    char Nombre[31];
    int Codigo;
    float notas[MAXMA'l'][MAXNOT];
};
```

Determinar un reporte de los alumnos en orden alfabético de acuerdo a cada materia. Por ejemplo, para la materia **Lenguaje_C** se tendría:

MATERIA LENGUAJE_C							
ORD	NOMBRE	MES1	MES2	MES3	MES4	PROMEDIO	OBSERVACION

NOTAS:

- La información del archivo de datos "**informa**" debe ser ingresada con validación.
- En la columna "OBSERVACIÓN" indicar si el alumno aprueba o no la materia con "APR" y "REP", respectivamente. La materia es aprobada con un promedio de 14 o mayor sobre 20.

25) El rol de pagos de una empresa está constituido por los siguientes campos:

- Nombre del empleado.
- Sueldo básico.
- Aporte al Seguro Social (9.35 % del sueldo básico).
- Impuesto a la renta, de acuerdo al siguiente cuadro:

SI EL SUELDO AL AÑO ES:	IMPUESTO
<= 3.000	0%
> 3.000 y <= 5.000	10% del exceso a 3.000
> 5.000 y <= 8.000	20% del exceso a 6.000
> 8.000	30% del exceso a 8.000

- Sueldo a recibir, de acuerdo a la fórmula:

Recibe = Sueldo básico - Aporte al Seguro Social - Impuesto a la Renta

Hacer un programa que realice un menú como el siguiente:

a) Crear el archivo.
b) Almacenar en el archivo la información ingresada y actualizada de cada empleado.
c) Modificar un empleado usando el campo "nombre del empleado" para la búsqueda.
d) Hacer un reporte de acuerdo al siguiente cuadro:

EMPRESA "RPG"
ROL DE PAGOS

ORD	NOMBRE	SUELDO BASICO	APORTE SEGURO	IMPUESTO LARENTA	SUELDO "A RECIBIR"

NOTA: Para la parte (b) realizar la pregunta: "¿Desea continuar S/N? ".

26) En una empresa comercial se tiene varios departamentos de ventas, donde cada empleado tiene un registro de datos que contiene: El departamento (1 a 9), el número de vendedor (entero) y el monto de ventas (punto flotante). Se requiere diariamente ingresar los datos en un "archivo de datos" de todos los vendedores, para sacar al final del ingreso un reporte considerando lo siguiente:

a) Indicar por departamento los datos de los vendedores ordenados de acuerdo a su número de vendedor.
b) Cuando hay un cambio de página a mitad de un departamento, la primera línea de página siguiente deberá ser más o menos así:

	4 (continúa)	16141	9453

Por ejemplo:

	VENTAS TOTALES		
Número de departamento	1	Vendedor	Ventas
		500	1000
		501	1000
		Total de departamento:	2000
Número de departamento	2	Vendedor	Ventas
		8767	5000
		9325	16034
		9600	10007
		Total de departamento:	31042
		GRAN TOTAL:	201047

NOTA: Usar arreglo dinámico para el ordenamiento.

27) Los marcadores telefónicos estándar contienen los dígitos 0 al 9. Cada uno de los números del 2 al 9 tiene tres letras asociadas, como se indican en la tabla siguiente:

Dígito	Letras
2	ABC
3	DEF
4	GHI
5	JKL
6	MNO
7	PRS
8	TUV
9	WXY

Muchas personas encuentran difícil memorizar los números telefónicos, por lo que utilizan correspondencia existente entre dígitos y letras para desarrollar palabras de seis letras, que correspondan a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico es 1800-686-237 pudiera utilizar la correspondencia indicada en la tabla precedente, para desarrollar la palabra de seis letras "NUMBER". Es decir, cada palabra de seis letras corresponde exactamente a un número telefónico de seis dígitos.

Los negocios frecuentemente intentan obtener números telefónicos que sean fáciles de recordar para sus clientes. Si un negocio puede anunciar una sola palabra

para que la marquen sus clientes, entonces sin duda dicho negocio recibirá unas cuentas llamadas más.

Cada número telefónico de 6 dígitos corresponde a muchas palabras distintas de 6 letras. Desafortunadamente la mayor parte de ellas representan yuxtaposiciones irreconocibles de letras. Es posible sin embargo, que el propietario de un negocio tenga un nombre adecuado de acuerdo al negocio.

Escribir un programa "***generador de palabras de números telefónicos***", que dado un número de 6 dígitos, escriba a un archivo todas las palabras posibles de 6 letras que correspondan a dicho número. Existen 729 (3 a la sexta potencia) palabras posibles, que pueden estar formadas por una o dos palabras. Se debe evitar números telefónicos que contengan los dígitos 0 y 1.

28) Realizar un programa para usar como editor de texto orientado a líneas, con las siguientes posibilidades:
   a) Introducir varias líneas de texto y almacenarlas en un archivo.
   b) Listar el archivo.
   c) Recuperar y mostrar una línea particular, determinada por su número.
   d) Insertar **n** líneas.
   e) Borrar **n** líneas.
   f) Almacenar el nuevo texto y finalizar.

   Realizar cada una de las tareas respondiendo a una orden (línea de comandos) compuesta por el signo $, seguido por una orden de una letra, un entero sin signo opcional y un símbolo de nueva línea.

   - En la orden recuperar c) debe ir seguida por un entero sin signo para indicar la línea que debe recuperar.
   - En las órdenes de inserción d) y borrado e) pueden ir seguidas por un entero sin signo opcional, si se quiere insertar o borrar varias líneas consecutivas.
   - Cada orden debe aparecer en una línea para distinguir las órdenes del texto.

   Se recomienda las siguientes órdenes:
   $E Introducir texto.
   $L Listar el bloque de texto completo.
   $Fk Recuperar la línea de número **k**.
   $In Insertar **n** líneas después de la línea número **k**.
   $Dn Borrar **n** líneas después de la línea número **k**.
   $S Salvar el bloque de texto editado y terminar.

29) Escribir un programa que codifique y decodifique el contenido de un archivo de texto mediante el reemplazo de cada caracter con su complemento a uno. Tomar en cuenta que el complemento a uno del complemento a uno de un caracter es el

caracter original, por lo tanto se puede usar para codificar el texto original como para decodificar el texto codificado.

El programa debe incluir las siguientes características:

a) Introducir el contenido de un archivo ordinario de texto desde el teclado.
b) Almacenar el archivo de texto actual en su estado presente (codificado o decodificado).
c) Recuperar el texto que ha sido almacenado (codificado o decodificado).
d) Codificar o decodificar el archivo de texto actual (cambiar el estado actual obteniendo el complemento a uno de cada uno de los caracteres).
e) Mostrar el archivo de texto en su estado presente (codificado o decodificado).

Generar un menú que permita al usuario seleccionar cualquiera de estas características según desee.

También puede ser codificado y decodificado el contenido del archivo de texto, realizando el codificado y decodificado con la "OR exclusiva" a nivel de bits o una operación de "enmascaramiento" en vez de la operación de "complemento a uno". Si se elige esta alternativa, permitir al usuario introducir una clave (una máscara que será el segundo operando en la operación "OR exclusiva"). Como la operación "OR exclusiva" produce una operación de inversión, se puede usar para codificar el texto original o para decodificar el texto codificado. Se debe usar la misma clave para codificar y decodificar.

30) Escribir un programa que utilice el operador **sizeof** para determinar los tamaños en bytes de varios tipos de datos en su sistema de computación. Escribir los resultados al archivo "**data_tam.dat**" de tal forma que más tarde se puedan imprimir los resultados. El formato para los resultados del archivo deberá ser:

DATA TYPE	SIZE
char	1
unsigned char	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
unsigned long int	4
float	4
double	8
long double	16

NOTA: Los tamaños de tipo en su computadora pudieran no ser iguales a los de arriba listados.

Capítulo 12

EL PREPROCESADOR

Se llaman "directivas de preprocesador", a las sentencias dirigidas al compilador en el código fuente de un programa en lenguaje C, y no son realmente parte del lenguaje C, pero amplían el ámbito de entorno de programación.

El preprocesador de lenguaje C, definido por el estándar ANSI, contiene las siguientes "directivas de compilación": (SCHILDT, 1994)

#define
#error
#include
#if
#endif
#else
#elif
#ifdef
#ifndef
#undef
#line
#pragma

Todas las directivas del preprocesador empiezan con el símbolo **#**. Además, cada directiva de preprocesador debe estar en su propia línea de declaración, y no hay punto y coma al final de la sentencia.

12.1. DIRECTIVA: #*define*

La directiva **#defìne**, define una macro que consiste en un "*identificador*" y una "*secuencia de caracteres*"; el identificador será sustituido por la secuencia de caracteres cada vez que se encuentre en el archivo fuente, en el momento de compilar el programa.

La forma general de la directiva **#define** es: (SCHILDT, 1994)

#define nombre_macro secuencia_caracteres

donde:

- **nombre_macro**, es el identificador.

- **secuencia_caracteres**, es cualquier secuencia de caracteres.

Puede haber cualquier número de espacios en blanco entre el "*identificador*" y la "*secuencia de caracteres*", pero una vez que empieza la secuencia de caracteres, solo acaba con un salto de línea.

Por ejemplo, si se desea usar VERDADERO para el valor 1 y FALSO para el valor 0, se haría:

```
#define VERDADERO 1
#define FALSO 0
```

Esto hace que el compilador sustituya por 1 o 0, cada vez que encuentre en el archivo fuente VERDADERO o FALSO, respectivamente. Por ejemplo, la siguiente sentencia:

```
printf ("%d %d %d", FALSO, VERDADERO, VERDADERO + 1);
```

imprime en pantalla:

0 1 2

El "*identificador*" definido con la directiva **#define**, también se llama "***nombre de macro***". Una vez que se ha definido un "nombre de macro" se puede usar como parte de la definición de otros nombres de macro. Por ejemplo:

```
#define UNO 1
#define DOS UNO+UNO   /* Esta macro sería 1 + 1. */
#define TRES UNO+DOS /* Esta macro sería 1 + UNO + UNO. */
```

La sustitución de macro. Es simplemente el proceso de reemplazar un "*identificador*" por su "*secuencia de caracteres*" asociada. Por ejemplo, para definir un mensaje de error estándar, se haría:

```
#define MS_ERROR "Error estándar en la entrada\n"
printf (MS_ERROR);
```

En donde el compilador sustituye la *secuencia de caracteres* "Error estándar en la entrada\n" cuando encuentra el "*identificador*" MS_ERROR.

Para el compilador la sentencia **printf()** será realmente de la siguiente forma:

```
printf ("Error estándar en la entrada\n");
```

Si el "*identificador*" aparece dentro de una cadena de caracteres, no se llevan a cabo sustituciones. Por ejemplo, a continuación no se realiza la sustitución de la macro CADENA:

```
#define CADENA "Este es un ejemplo"
printf ("CADENA");
```

Si la secuencia de caracteres es más larga que una línea, se puede continuar en la siguiente línea poniendo una barra invertida (\) al final de la línea. Por ejemplo:

```
#define CADENA_LARGA "Esta es una cadena muy larga \
que se usa como ejemplo"
```

Se acostumbra a usar letras mayúsculas en la declaración de los "nombres de macros" para diferenciarlos de las variables. Además, es mejor poner todas las definiciones de la directiva **#define** al principio del archivo o en un archivo separado incluido, para poder usarlas en todo el programa.

Usos de la sustitución de macros

1. "*Definición de constantes*". Es mejor definir un "nombre de macro" y utilizar ese nombre siempre que se necesite, de esta forma es más descriptivo un nombre que un valor, y cuando se desea modificar el valor, si es preciso, solo se requiere un cambio en un único sitio, que es en la declaración de la macro. Los valores más comunes a ser definidos como macros son: las constantes matemáticas, los valores que se repiten a menudo y el tamaño del arreglo estático, como se muestra a continuación:

```
#define MAX 100
#define FIN "Ingrese <FIN> para finalizar\n"
#define TOTAL 50

float arreglo[MAX];
```

2. "*Macro con argumentos*". El "nombre de macro" puede tener argumentos. Estos argumentos asociados con la macro se reemplazan por los argumentos reales del programa, cada vez que se encuentra el "nombre de macro". Por ejemplo, para calcular el valor absoluto de un número, se haría:

```
#incIude "stdio.h"

#define ABS(a) (a) < 0 ? -(a) : (a)

void main (void)
{
    printf ("VaIor absoluto de -1 es: %d", ABS (-1));
}
```

Cuando este programa se compila, el argumento **a** de la definición de macro se sustituye por el valor -1, es decir,

ABS (-1) se sustituirá por (-1) < 0 ? -(-1) : (-1)

imprimiéndose:

Valor absoluto de -1 es: 1

Los paréntesis que rodean al argumento de la macro **a**, aseguran que se lleve a cabo adecuadamente la sustitución en todos los casos. Si son eliminados los paréntesis sobre **a**, se llega a los dos casos de error siguientes:

1) Error de sintaxis.
   Sustituyendo ABS (-1) en la macro:

   -1 ? 0 : --1

   Se tiene una operación de decremento errada a una constante

2) Error de cálculo.
   Sustituyendo ABS (10-20) en la macro:

   10-20 < 0 ? -10 - 20 : 10 - 20

   Se llega a un resultado equivocado de -30.

En el uso de "sustituciones de macros" en lugar de "funciones reales", se tiene la siguiente ventaja: incrementa la velocidad de ejecución porque no se gasta tiempo en llamar a la función, puesto que el código ya está presente. Sin embargo, hay que pagar un precio por el aumento de velocidad: el tamaño del programa aumenta debido a la incorporación redundante de código.

Ejercicio

Realizar un programa que contenga una macro de varias líneas, que representa a una sentencia compuesta. Esta sentencia consiste de lazos **for** anidados, para imprimir un triángulo con asteriscos de un número de líneas determinado por el valor **n**, ingresado por teclado. Por ejemplo, si **n=3** el triángulo impreso sería:

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

```
/* PROG1201.C */

#include "stdio.h"

#define LAZOS(n)  for (lineas = 1; lineas <= n; lineas ++) {        \
                    for (cont = 1; cont <= n - lineas; cont ++)     \
                      putchar (' ');                                \
                    for (cont = 1; cont <= 2 * lineas - 1; cont ++) \
                      putchar ('*');                                \
                    printf ("\n");                                  \
                  }
void main ()
{
  int cont, lineas, n;

  printf ("Número de líneas = ");
  scanf ("%d", &n);
  printf ("\n");
  LAZOS (n)
}
```

Una salida del programa sería:

Número de líneas = 3 <ENTER

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

12.2. DIRECTIVA: #*error*

La directiva **#error** obliga a parar la compilación del programa. Se usa principalmente en la depuración de programas.

La forma general de la directiva **#error** es: (SCHILDT, 1994)

#error mensaje_error

donde:

- **mensaje_error**, es el mensaje de error que no va entre comillas.

Cuando se encuentra la directiva **#error**, se muestra el mensaje de error, junto con otra información definida por el compilador, como se muestra a continuación:

Fatal: *file line Error directive: mensaje_error*

12.3. DIRECTIVA: *#include*

La directiva **#include** hace que el compilador incluya otro archivo fuente en el archivo que contiene esta directiva, en el sitio donde está declarada.

El nombre del archivo fuente a incluir debe estar entre "comillas" o entre los caracteres < y >. Por ejemplo:

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

- Si el nombre del archivo está entre los caracteres < y >, el archivo se busca por defecto en el directorio en el que se hayan definido las inclusiones. A menudo esto significa que se busca en algún directorio especial dispuesto para archivos incluidos **(#include**), que se direcciona en el momento de instalar el compilador o cuando el programador lo establezca.

- Si el nombre del archivo está entre "*comillas*" y sin "*path*", se busca primero en el directorio actual de trabajo; si no se encuentra ahí el archivo, se repite la búsqueda como si estuviera entre los caracteres < y >.

Si no se desea la inclusión por defecto, se especifica el "*path*" donde se encuentra el archivo. Por ejemplo, si el archivo PRUEBA.C se encuentra en la raíz de la unidad A, se escribiría así:

```
#include "C:\PRUEBA.C"
```

Se debe tener en cuenta que no se puede incluir más de una vez un archivo en otro, porque habría duplicidad de código.

Anidamiento de inclusiones. Se tiene cuando los archivos incluidos pueden contener otras directivas **#include**.

El número de niveles de anidamiento permitido depende del compilador. Sin embargo, el estándar ANSI estipula que al menos se deben permitir ocho niveles de anidamiento. Por ejemplo, a continuación el programa almacenado en PRINCIPA.C incluye el archivo UNO.C, que éste a su vez incluye el archivo DOS.C:

```
/* Archivo del programa. */
/* PRINCIPA.C */

void main ()
```

```
{
    #include "uno.c"
    /* Sentencias. */
}
```

```
/* Archivo incluido uno. */
/* UNO.C */

printf ("Este es el primer archivo incluido.\n");

#include "dos.c"
```

```
/* Archivo incluido dos */
/* DOS.C */

printf ("Este es el segundo archivo incluido.\n");
```

12.4. DIRECTIVAS DE COMPILACIÓN CONDICIONAL

Hay varias directivas que permiten compilar selectivamente partes del código fuente de un programa. Este proceso se llama "***compilación condicional***".

12.4.1. Directivas: *#if* y *#endif*

Si la expresión constante que sigue a la directiva **#if** es verdadera, se compila el código que hay entre el **#if** y el **#endif**, caso contrario se ignora ese código. Es decir, la directiva **#endif** marca el final de un bloque **#if**.

La forma general de la directiva **#if** es: (SCHILDT, 1994)

```
#if expresión_constante
    /* Secuencia de sentencias. */
#endif
```

La expresión que sigue a **#if** se evalúa en tiempo de compilación. Por tanto, solo debe contener constantes y macros que se hayan definido anteriormente; no se puede usar variables.

Por ejemplo, el siguiente programa es demostrativo para explicar el uso de la directiva **#if**:

```
/* PROG1202.C */

#include "stdio.h"

#define MAX 100

void main (void)
{
  #if MAX > 99
    printf ("Compilado para valores mayores de 99.\n");
  #endif
}
```

Como MAX es mayor que 99, la salida de este programa será:

Compilado para valores mayores de 99.

12.4.2. Directiva: #*else*

Funciona de una manera muy parecida al **else** que forma parte del lenguaje C; establece dos alternativas de compilación tomando en cuenta el caso en que el **#if** falle.

La directiva **#else** se usa para marcar el final de bloque **#if** y el principio del bloque **#else**. Esto es así porque solo puede haber un **#endif** asociado a un **#if**.

Por ejemplo, el programa del literal anterior se puede expandir como se muestra a continuación:

```
/* PROG1203.C */

#include "stdio.h"

#define MAX 10

void main (void)
{
  #if MAX > 99
    printf ("Compilado para valores mayores de 99.\n");
  #else
    printf ("Compilado para valores pequeños.\n");
  #endif
}
```

La salida de este programa será:

Compilado para valores pequeños.

Debido a que MAX es menor que 99, el bloque **#if** no se compila. En su lugar se compila la alternativa **#else**.

12.4.3. Directiva: #*elif*

Quiere decir "**else if**" establece un anidamiento del tipo **if-else-if** para opciones de compilación múltiples. Esta directiva se debe usar como parte de la directiva **#if**.

La directiva **#elif** va seguida de una expresión constante. Si la expresión es verdadera, ese bloque de código se compila y no se comprueba ninguna expresión **#elif** más. En cualquier otro caso, se comprueba la siguiente condición del anidamiento.

La forma general de la directiva **#elif** es: (SCHILDT, 1994)

```
#if expresión
    /* Secuencia de sentencias. */
#elif expresión1
    /* Secuencia de sentencias. */
#elif expresión2
    /* Secuencia de sentencias. */

/* Otras altemativas. */
#eIif expresiónN
    /* Secuencia de sentencias. */
#else
    /* Secuencia de sentencias. */
#endif
```

Por ejemplo, la macro PAIS_ACTIVO se usa para definir un símbolo de moneda aplicando las directivas de compilación condicional, como se muestra a continuación:

```
#define USA 0
#define GB 1
#define ESP 2
#define PAIS_ACTlVO ESP

#if PAIS_ACTIVO == USA
    char moneda[] = "dólar"
#if PAIS_ACTIVO == GB
    char moneda[] = "libra"
#eIse
```

```
    char moneda[] = "peseta"
#endif
```

Las directivas **#if** y **#elif** se pueden anidar con **#endif**, **#else** o **#elif** asociadas al **#if** o al **#elif** más cercano. Por ejemplo, como se muestra a continuación:

```
#if MAX > 100
#if VERSION_SERIE
    int puerto = 198;
#eIif VERSION_PAR
    int puerto = 200;
#eIse
    printf ("No hay puerto.\n");
#endif
#eIse
    char bufier_saI[100];
#endif
```

12.4.4. Directivas: #*ifdef* e *ifndef*

Otro método de compilación condicional utiliza las directivas **#ifdef** e **#ifndef** que quieren decir "*si se ha definido*" y "*si no se ha definido*" una macro, respectivamente.

La forma general de la directiva **#ifdef** es:

```
#ifdef nombre_macro
    /* Secuencia de sentencias. */
#endif
```

Si se ha definido previamente el "***nombre_macro***" en una sentencia **#define**, se compila la "***Secuencia de sentencias***".

La forma general de la directiva **#ifndef** es: (SCHILDT, 1994)

```
#ifndef nombre_macro
    /* Secuencia de sentencias. */
#endif
```

Si no se ha definido previamente el "***nombre_macro***" mediante una sentencia **#define**, se compila la "***Secuencia de sentencias***".

Ambas directivas **#ifdef** e **#ifndef**, pueden usar una directiva **#else**, pero no una directiva **#elìf**, porque solo deben tener dos alternativas de existencia o no de una macro.

Por ejemplo, a continuación se presenta un programa para indicar cómo se utilizan las directivas **#ifdef** e **#ifndef**:

```
/* PROG1204.C */

#include "stdio.h"

#define JUAN 10

void main (void)
{
  #ifdef JUAN
    printf ("Hola Juan.\n");
  #else
    printf ("Hola, c¢mo est s?.\n");
  #endif
  #ifndef RAFAEL
    printf ("RAFAEL no definido.\n");
  #endif
}
```

La salida de este programa será:

Hola Juan.
RAFAEL no definido.

Se puede anidar **#ifdef** e **#ifndef** hasta un determinado nivel, de la misma forma que se anidan las directivas **#if**.

También se puede usar la directiva **#if** para ver si se ha definido una macro, utilizando el operador **defined**, el cual no existe en el estándar ANSI. El operador **defined** tiene como argumento el nombre de una macro, para verificar si ésta ya ha sido definida. Su forma general es:

```
#if defined (nombre_macro)
    /* Secuencia de sentencias. */
#endif
```

donde:

- **nombre_macro**, es el nombre de la macro que se comprueba.

Esta forma de **#if** es específica para Turbo C y C++.

Un archivo de cabecera puede contener directivas **#include**, lo que puede dar lugar a incluir en un programa más de una copia de otro archivo. Para evitar este problema cada archivo de cabecera a incluir puede tener la siguiente forma:

```
#if !defined (_NOMBRE_H)
    #define _NOMBRE_H
    /* Contenido del archivo: "nombre.h" */
#endif
```

o también:

```
#ifndef (_NOMBRE_H)
    #define _NOMBRE_H
    /* Contenido del archivo: "nombre.h" */
#endif
```

donde:

- _NOMBRE_H, representa la macro del archivo "***nombre.h***".

Cuando un archivo de cabecera es incluido por primera vez, _NOMBRE_H no está definido, el preprocesador **#define** la define y a continuación procesa el archivo "***nombre.h***". Si posteriormente se trata de incluir el mismo archivo de cabecera, _NOMBRE_H ya está definido, siendo falsa la condición de la directriz **#if**, ignorándose el contenido hasta **#endif**.

12.5. DIRECTIVA: *#undef*

La directiva **#undef** elimina la definición anterior del nombre de macro que la siga.

La forma general de la directiva **#undef** es: (SCHILDT, 1994)

```
#undef nombre_macro
```

Por ejemplo, las macros FILA Y COLUMNA están definidas hasta que se encuentran las sentencias **#undef**, como se muestra a continuación:

```
#define FILA 100
#define COLUMNA 100
char arreglo [FILA][COLUMNA];

#undef FILA
#undef COLUMNA
```

El propósito principal de **#undef** es asignar los nombres de macro solo a aquellas secciones de código que se las necesite.

12.6. DIRECTIVA: #line

La directiva **#line** cambia los contenidos de las macros predefinidas _LINE_ y _FILE_.

La macro _LINE_ contiene el número de línea de la línea que se está compilando actualmente.

La macro _FILE_ es una cadena que contiene el nombre del archivo fuente que se está compilando.

La forma general de directiva **#line** es: (SCHILDT, 1994)

#line numero "nombre_archivo"

donde:

- **numero**, es cualquier entero positivo que se convierte en el nuevo valor de _LINE_.

- **nombre_archivo**, es opcional, y es cualquier identificador válido de archivo que se está compilando y que se convierte en el nuevo valor de _FILE_.

La directiva **#line** se usa principalmente para depuración y para aplicaciones especiales.

Por ejemplo, el siguiente programa especifica que la cuenta de líneas empieza en 100:

```
/* PROG1205.C */

#include "stdio.h"

#line 100                /* Reinicializa el contador de líneas. */
                         /* Linea 100. */
void main (void)         /* Línea 101. */
{                        /* Linea 102. */
  printf ("%d\n", __LINE__);   /* Línea 103. */
}
```

La salida de este programa será:

103

porque en la cuarta línea del programa la sentencia **#line 100**, define el nuevo número de línea a 100.

12.7. DIRECTIVA: #*pragma*

La directiva **#pragma** es una directiva que permite que se den varias instrucciones al compilador.

Esta directiva depende del compilador, por lo que los detalles y las opciones deben ser consultados en el manual del compilador que se está usando.

La forma general de la directiva **#pragma** es: (SCHILDT, 1994)

#pragma nombre

donde:

- **nombre**, es el nombre de la **#pragma** que se desea.

Para Turbo C se define las tres siguientes sentencias **#pragma**:

inline
saveregs
warn

Para Turbo C++ se añade las siguientes cuatro sentencias **#pragma**:

argsused
exit
startup
option

12.7.1. Directiva *inline*

Indica a Turbo C que el programa contiene código ensamblador incluido.

La forma general de la directiva **inline** es:

#pragma inline

12.7.2. Directiva *saveregs*

Previene que una función declarada como "***huge***" (modelo de memoria grande, utilizada durante el proceso de compilación), altere el valor de cualquier registro.

Esta directiva debe preceder inmediatamente a la función que va a afectar.

12.7.3. Directiva *warn*

Hace que Turbo C pase por alto las opciones de mensajes de advertencia.

La forma general de la directiva **warn** es: (SCHILDT, 1994)

#pragma warn ajuste

donde:

- **ajuste**, es una de las distintas opciones de mensajes de advertencia que dependen del compilador.

12.7.4. Directiva *argsused*

Debe preceder a una función, se usa para prevenir mensajes de advertencia, cuando no se utiliza en el cuerpo de la función un argumento de la función a la que precede **#pragma**.

12.7.5. Directiva *exit*

Especifica una o más funciones que se llamarán al terminar el programa.

La forma general de la directiva **exit** es: (SCHILDT, 1994)

#pragma exit nombre_función prioridad

donde:

- prioridad, es un valor entre 64 y 255 (los valores entre 0 y 63 son reservados por el compilador).

La prioridad determina el orden en que se llamarán las funciones; si no se da ninguna prioridad, implícitamente se toma 100.

12.7.6. Directiva *startup*

Especifica una o más funciones que se llamarán al comenzar la ejecución del programa.

La forma general de la directiva **startup** es:

#pragma startup nombre_función prioridad

donde:

- **prioridad**, es un valor entre 64 y 255, igual que en la directiva **exit**, la prioridad determina el orden en que se llamarán las funciones.

Se debe proporcionar un prototipo de función para todas las funciones **exit** y **startup**, antes de la sentencia **#pragma**, y se definen como sigue:

void func (void);

Por ejemplo, para definir una función **startup** denominada **comienzo()** y una función **exit** denominada **final()**, se hace así:

```
/* PROG1206.C */

#include "stdio.h"

void comienzo (void);
void final (void);

#pragma startup comienzo 64
#pragma exit final 65

void main ()
{
 printf ("En main.\n");
}

void comienzo (void)
{
 printf ("En comienzo.\n");
}

void final (void)
{
 printf ("En final.\n");
}
```

La salida de este programa será:

En comienzo.
En main.
En final.

12.7.7. Directiva *option*

Permite especificar opciones de línea de comandos en el programa, en lugar de hacerlo en la propia línea de comandos.

La forma general de la directiva **option** es: (SCHILDT, 1994)

```
#pragma option lista_opciones
```

Por ejemplo, la siguiente sentencia hace que el programa que la contiene se compile para el modelo grande:

```
#pragma option –ml
```

Las siguientes opciones no pueden ser usadas con la directiva option:

```
-B -c -d -D -e
-I* -L -l* -M -s -U
```

NOTA: Las opciones marcadas con el caracter asterisco (*), corresponden a la **i** mayúscula y a la **l** minúscula, respectivamente.

Para algunas opciones la directiva **option** debe preceder a todas las declaraciones, incluyendo a los prototipos de las funciones. Por esta razón se debe hacer que sea la primera sentencia del programa.

12.8. LOS OPERADORES DEL PREPROCESADOR # Y

El lenguaje C del ANSI proporciona dos operadores de preprocesamiento: **#** y **##**. Estos operadores se utilizan cuando se usa una definición de macro con **#define**. (SCHILDT, 1994)

1. ***El operador #.***

Cambia el argumento de la macro en una cadena entre comillas.

Por ejemplo, en el siguiente programa:

```
#incIude "stdio.h"
#define CADENA(s) # s
void main (void)
{
    printf (CADENA (Me gusta C));
```

```
}
```

el preprocesador de lenguaje C cambia la sentencia:

```
printf (CADENA (Me gusta C));
```

a

```
printf ("Me gusta C");
```

NOTAS:

- Los caracteres blancos consecutivos dentro del argumento actual, serán reemplazados por un solo espacio en blanco.

- Cualquier caracter especial como: ', " y \ será reemplazado por su correspondiente secuencia de escape: \', \" y \\.

- La cadena resultante será automáticamente concatenada con cualquier cadena adyacente.

Por ejemplo, en el siguiente programa:

```
#define MUESTRA(texto) printf (#texto "\n")

void main ()
{
    MUESTRA (Por favor "atiende"!);
}
```

el cuerpo de la función **main()** será:

```
printf ("Por favor \"atiende\"!\n");
```

2. ***El operador ##.***

Concatena dos elementos.

Por ejemplo, en el siguiente programa:

```
#incIude "stdio.h"

#define CONCAT(a,b) a ## b

void main (void)
```

```
{
    int xy = 10;

    printf ("%d", CONCAT (x. y));
}
```

el preprocesador de lenguaje C transforma la sentencia de función **printf()**:

```
printf ("%d", concat (x, y));
```

en

```
printf ("%d", xy);
```

Un ejemplo utilizando los dos operadores anteriores, podría ser:

```
#incIude "stdio.h"

#define MUESTHA(i) printf ("x" #i "=%f\n", x ## i)

void main ()
{
    float x3 = 3.4;
    MUESTRA (3);
}
```

El resultado será:

```
printf ("x3=%f\n", x3);
```

Estos operadores existen principalmente para permitir al preprocesador manejar ciertos casos especiales.

12.9. NOMBRES DE MACROS PREDEFINIDAS

El estándar ANSI especifica cinco nombres de macros predefinidas:

```
__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
```

Dependiendo de los compiladores también puede proporcionar otras macros predefinidas. Por ejemplo, para Turbo C y Turbo C++, existen más nombres de MACROS.

- Las macros __LINE__ y __FILE__ se han comentado en la sección de **#line**.
- La macro __DATE__ contiene una cadena de la forma *mes/día/año*. Esta cadena representa la fecha de traducción del código fuente a código objeto.
- La hora de la traducción del código fuente a código objeto está contenida como una cadena en __TIME__. La forma de esta cadena es *horas:minutos:segundos*.
- La macro __STDC__ contiene la constante decimal 1, esto significa que el compilador se ajusta al estándar ANSI. Si la macro contiene cualquier otro número, es que el compilador varía del estándar ANSI.

Por ejemplo, en el siguiente programa se utiliza los nombres de macros predefinidas:

```
/* PROG1207.C */

#include "stdio.h"

void main ()
{
  printf ("%s, %d, %s, %s\n", __FILE__, __LINE__, __DATE__, __TIME__);
}
```

La salida del programa será:

A:\> CAPITULO.12\PROG1207.C, 14, oct 20 2010, 18:31:42

PROBLEMAS PROPUESTOS

1) Escribir un programa que defina una macro con un argumento, para calcular el volumen de una esfera. El programa deberá calcular el volumen para esferas de radios de 1 a 10, e imprimir los resultados en formato tabular. La fórmula del volumen de una esfera es:

   $V = (4/3)\pi x^3$

   donde: $\pi = 3.14159$.

2) Escribir un programa que produzca la siguiente salida:

   *La suma de **x** e **y** es: 13.*

   El programa deberá definir la macro SUM con dos argumentos, **x** e **y**, y utilizar SUM para producir dicha salida.

3) Escribir un programa que tenga una macro multilínea llamada INTERES, que evalúe la fórmula de interés compuesto:

   $F = P(1 + i)^n$

   donde:
   - **F**, es el valor futuro.
   - **P**, es el capital inicial (principal).
   - **i**, es la tasa de interés anual expresado en tanto por uno.

   Evaluar **i** en una línea de la macro y **F** en otra línea separada. La macro tiene que tener los siguientes argumentos **P**, **i** y **n**.

4) Escribir un programa que utilice la macro MINIMUM2, para determinar el más pequeño de dos valores numéricos.

   El programa debe definir la macro MINIMUN3 para determinar el más pequeño de **n** valores numéricos, la macro MINIMUN3 deberá utilizar la macro MINIMUMZ para determinar el valor más pequeño. Introducir los valores desde el teclado.

5) Escribir un programa que tenga una macro llamada MAXIMO que utilice el operador condicional (**?:**), para determinar el valor máximo de dos cantidades enteras **a** y **b**. La macro debe tener los argumentos **a** y **b**.

6) Escribir un programa que utilice la macro PRINT para imprimir una cadena.

7) Escribir un programa que utilice la macro PRINT_ARREGLO, para imprimir un arreglo de enteros. La macro deberá recibir como argumentos al arreglo y el número de elementos del arreglo.

8) Escribir un programa que utilice la macro SUMA_ARREGLO, para sumar los valores de un arreglo numérico. La macro deberá recibir como argumentos el arreglo y el número de elementos del arreglo.

9) El factorial de un entero positivo se define como:

   n! = 1 * 2 * ... * n

   Realizar un programa para calcula el factorial mediante una macro multilínea.

10) Escribir un programa para generar una tabla de valores de la siguiente ecuación:

   $y = 2e^{-0.1*}$ sen 0.5t

   donde: **t**, varía entre 0 y **n**.

   Usar una macro para evaluar la ecuación, los valores de incremento **t** y el límite **n** deben ser ingresados desde la línea de comandos.

11) Definir las siguientes macros:

   - Para calcular el área de un círculo en función de su radio que se exprese como un argumento. Usar una constante predefinida PI en el cálculo.

   - Para calcular la circunferencia de un círculo en función de su radio que se exprese como argumento. Usar una constante predefinida PI en el cálculo.

   Realizar un programa para ingresar el número de datos para el arreglo dinámico que va a almacenar los radios de tipo **float**. Luego ingresar el radio acompañado de la unidad de medida: **mm**, **cm**, **m**, **pulgadas** y **pies**. Por último imprimir una tabla como la siguiente:

RADIO (cm)	AREA (cm)	CIRCULO (cm)

12) Escribir un programa que realice las siguientes directivas de preprocesamiento:

   a) Definir la constante simbólica YES para que tenga el valor 1.
   b) Definir la constante simbólica NO para que tenga el valor 0.

c) Incluir el archivo de cabecera "***common.h***". El archivo de cabecera se encuentra en el directorio **c:\tc\bin**.
d) Reenumerar las líneas restantes del archivo, empezando con el número de línea 1000.
e) Si está definida la constante simbólica TRUE, eliminar su definición, y volver a definir como 1. No utilizar **#ifdef**.
f) Si está definida la constante simbólica TRUE, eliminar su definición, y volver a definir como 1. Utilizar **#ifdef**.
g) Si la constante simbólica TRUE no es igual a 0, definir la constante simbólica FALSE como 0. De lo contrario definir FALSE como 1.
h) Definir la macro VOLUMEN, que calcule el volumen de la esfera. La macro tiene un argumento.

13) Realizar un programa para escribir una o más directivas del preprocesador, para cada uno de los siguientes enunciados.

a) Si la constante simbólica LOGICA ha sido definida, definir las constantes simbólicas CIERTO y FALSO tales que sus valores sean 1 y 0, respectivamente, y negar las dos definiciones de las constantes simbólicas SI y NO.
b) Si INDICADOR tiene valor 0, definir la constante simbólica COLOR con valor 1. En otro caso, si el valor de INDICADOR es menor que 3, definir COLOR con valor 2; si el valor de INDICADOR es mayor o igual que 3 definir COLOR con valor 3.
c) Si la constante simbólica TAMANO tiene el mismo valor que la constante AMPLIO, definir la constante simbólica ANCHO con el valor 500; en otro caso definir ANCHO con valor de 100.
d) Usar el operador # para definir una macro llamada **ERROR(texto)** que escribirá texto como una cadena.
e) Usar el operador concatenador ## para definir una macro llamada **ERROR(i)** que escribirá el valor de la variable cadena **errori**, por ejemplo **error1**.

14) Indicar exactamente lo que imprime el siguiente programa:

```
#incIude "stdio.h"

#define GRANDE 3
#undef GRANDE
#define GRANDE 5
#define ENORME 5

#define tituIo(s) #s"\n"
#define CUAD(x) (x)*(x)
#define PH(x) printf ("x %d.\n", x)
```

```
void main ()
{
    Int x=4, z;

    #ifdef GRANDE
        printf ("Ejercicio de aplicación.\n");
    #endif

    #if ENORME != GRANDE
        printf("GRANDE igual a ENORME.\n");
    #else
        printf("--------------------------\n");
    #endif

    printf (titulo (Imprimir exactamente:));
    z = CUAD (x);
    PR (z);
    z = CUAD (2);
    PR (z);
    PR (CUAD (x));
    PR (CUAD (++x));
    PR (CUAD (x+2));
    PR (CUAD (++x));
    PR (100 / CUAD (2));
}
```

Apéndice A

Código ASCII para el PC

ASCII (American Standard Code for Information Interchange)

Código norteamericano estándar para el intercambio de información

(Página de Códigos de Estados Unidos 437 (CP-437), Personal Computer de IBM)

Dec	Hex	Símbolo ASCII	Código de control	Tecla Ctrl	Dec	Hex	Símbolo ASCII	Código de control	Tecla Ctrl
0	00		NUL	^@	16	10	►	DLE	^P
1	01	☺	SOH	^A	17	11	◄	DC1	^Q
2	02	☻	STX	^B	18	12	↕	DC2	^R
3	03	♥	ETX	^C	19	13	‼	DC3	^S
4	04	♦	EOT	^D	20	14	¶	DC4	^T
5	05	♣	ENQ	^E	21	15	§	NAK	^U
6	06	♠	ACK	^F	22	16	▬	SYN	^V
7	07	•	BEL	^G	23	17	↨	ETB	^W
8	08	◘	BS	^H	24	18	↑	CAN	^X
9	09	○	HT	^I	25	19	↓	EM	^Y
10	0A	◙	LF	^J	26	1A	→	SUB	^Z
11	0B	♂	VT	^K	27	1B	←	ESC	^[
12	0C	♀	FF	^L	28	1C	∟	FS	^\
13	0D	♪	CR	^M	29	1D	↔	GS	^]
14	0E	♫	SO	^N	30	1E	▲	RS	^^
15	0F	○	SI	^O	31	1F	▼	US	^_

Dec	Hex	Símbolo ASCII	Dec	Hex	Símbolo ASCII
32	20		76	4C	L
33	21	!	77	4D	M
34	22	"	78	4E	N
35	23	#	79	4F	O
36	24	$	80	50	P
37	25	%	81	51	Q
38	26	&	82	52	R
39	27	'	83	53	S
40	28	(	84	54	T
41	29	)	85	55	U
42	2A	*	86	56	V
43	2B	+	87	57	W
44	2C	,	88	58	X
45	2D	-	89	59	Y
46	2E	.	90	5A	Z
47	2F	/	91	5B	[
48	30	0	92	5C	\
49	31	1	93	5D	]
50	32	2	94	5E	^
51	33	3	95	5F	_
52	34	4	96	60	`
53	35	5	97	61	a
54	36	6	98	62	b
55	37	7	99	63	c
56	38	8	100	64	d
57	39	9	101	65	e
58	3A	:	102	66	f
59	3B	;	103	67	g
60	3C	<	104	68	h
61	3D	=	105	69	i
62	3E	>	106	6A	j
63	3F	?	107	6B	k
64	40	@	108	6C	l
65	41	A	109	6D	m
66	42	B	110	6E	n
67	43	C	111	6F	o
68	44	D	112	70	p
69	45	E	113	71	q
70	46	F	114	72	r
71	47	G	115	73	s
72	48	H	116	74	t

Dec	Hex	Símbolo ASCII	Dec	Hex	Símbolo ASCII
73	49	I	117	75	u
74	4A	J	118	76	v
75	4B	K	119	77	w
120	78	x	164	A4	ñ
121	79	y	165	A5	Ñ
122	7A	z	166	A6	ª
123	7B	{	167	A7	º
124	7C	|	168	A8	¿
125	7D	}	169	A9	⌐
126	7E	~	170	AA	¬
127	7F	⌂	171	AB	½
128	80	Ç	172	AC	¼
129	81	ü	173	AD	¡
130	82	é	174	AE	«
131	83	â	175	AF	»
132	84	ä	176	B0	░
133	85	à	177	B1	▒
134	86	å	178	B2	▓
135	87	ç	179	B3	│
136	88	ê	180	B4	┤
137	89	ë	181	B5	╡
138	8A	è	182	B6	╢
139	8B	ï	183	B7	╖
140	8C	î	184	B8	╕
141	8D	ì	185	B9	╣
142	8E	Ä	186	BA	║
143	8F	Å	187	BB	╗
144	90	É	188	BC	╝
145	91	æ	189	BD	╜
146	92	Æ	190	BE	╛
147	93	ô	191	BF	┐
148	94	ö	192	C0	└
149	95	ò	193	C1	┴
150	96	û	194	C2	┬
151	97	ù	195	C3	├
152	98	ÿ	196	C4	─
153	99	Ö	197	C5	┼
154	9A	Ü	198	C6	╞
155	9B	¢	199	C7	╟
156	9C	£	200	C8	╚
157	9D	¥	201	C9	╔

Dec	Hex	Símbolo ASCII	Dec	Hex	Símbolo ASCII
158	9E	Pt	202	CA	╩
159	9F	ƒ	203	CB	╦
160	A0	á	204	CC	╠
161	A1	í	205	CD	═
162	A2	ó	206	CE	╬
163	A3	ú	207	CF	╧
208	D0	╨	232	E8	φ
209	D1	╤	233	E9	θ
210	D2	╥	234	EA	Ω
211	D3	╙	235	EB	δ
212	D4	╘	236	EC	∞
213	D5	╒	237	ED	∅
214	D6	╓	238	EE	∈
215	D7	╫	239	EF	∩
216	D8	╪	240	F0	≡
217	D9	┘	241	F1	±
218	DA	┌	242	F2	≥
219	DB	█	243	F3	≤
220	DC	▄	244	F4	⌠
221	DD	▌	245	F5	⌡
222	DE	▐	246	F6	÷
223	DF	▀	247	F7	≈
224	E0	α	248	F8	°
225	E1	ß	249	F9	•
226	E2	Γ	250	FA	·
227	E3	π	251	FB	√
228	E4	Σ	252	FC	η
229	E5	σ	253	FD	^{2}
230	E6	µ	254	FE	■
231	E7	τ	255	FF	

Apéndice B

Prototipos de las Funciones de Biblioteca del Lenguaje C más utilizados
(Compilador Turbo C++ 3.0 de Borland, Versión de Evelio Granizo)

Prototipos de las Funciones de Biblioteca del lenguaje C más utilizadas

PROTOTIPO DE LA FUNCIÓN	PROPÓSITO	ARCHIVO DE CABECERA	VALOR RETORNADO
int abs(int n);	Calcula el valor absoluto del argumento entero *int*	<stdlib.h> o <math.h>	La función **abs()** retorna el valor absoluto del argumento entero ***int*** n.
double acos(double x);	Calcula el arco coseno	<math.h>, opcional <errno.h›	La función **acos()** retorna el arco coseno de x en el rango de 0 a π radianes. El parámetro x se especifica en radianes en el rango de -1 a 1, y si x es menor que -1 o mayor que 1, **acos()** retorna un valor indefinido.
double asin(double x);	Calcula el arco seno	<math.h>	La función **asin()** retorna el arco seno de x en el rango de -π/2 a π/2 radianes. El parámetro x se especifica en radianes en el rango de -1 a 1, y si x es menor que -1 o mayor que 1, **asin()** retorna un valor indefinido.
double atan(double x);	Calcula el arco tangente	<math.h>	La función **atan()** retorna el arco tangente de x en el rango de -π/2 a π/2 radianes. Si x es 0, **atan()** retorna 0. El parámetro x se especifica en radianes en el rango de -1 a 1, y si x es menor -1 o mayor que 1, **atan()** retorna un valor indefinido.

doubk atan2(double y, double x);	Calcula el arco tangente de **y/x**	<math.h>	La función **atan2()** retorna el arco tangente de *y/x* en el rango de -π a π radianes, usando los signos de ambos parámetros para determinar el cuadrante del valor retornado. Si x es 0, **atan2()** retorna 0, y si ambos parámetros de **atan2()** son 0, la función retorna 0. El parámetro x se especifica en radianes en el rango de -1 a 1, y si x es menor que -1 o mayor que 1, **atan2()** retorna un valor indefinido.
double atof(const char *string);	Conviene una cadena a tipo **double**	<math.h> y <stdlib.h>	La función **atof()** retorna el valor ***doublé*** producido por la conversión de la cadena *string* que está formado por dígitos decimales. El valor de retorno es 0.0, si la cadena no puede ser convertida a un valor ***double***. El valor de retorno es indefinido en caso de overflow.
int atoi(const char *string);	Conviene una cadena a tipo **int**	<stdlib.h>	La función **atoi()** retorna el valor de **int** producido por la conversión de la cadena *string* que está formada por dígitos decimales. El valor de retorno es 0, si la cadena no puede ser convertida a un valor ***int***. El valor de retorno es indefinido en caso de overflow.
long atol(const char *string);	Conviene una cadena a tipo **long**	<stdlib.h>	La función **atol()** retorna el valor ***long*** producido por la conversión de la cadena de *string* que está formada por dígitos decimales. El valor de retorno es 0L, si la cadena no

			puede ser convertida a un valor ***long.*** El valor de retorno es indefinido en caso de overflow.
void *calloc(size_t num, site_t size);	Reserva un espacio de memoria con elemento inicializados a 0	<stdlib.h> y <malloc.h>	La función **calloc()** retorna un puntero al primer byte del espacio reservado, pero si no hay suficiente espacio para satisfacer la petición, devuelve un puntero NULL. La cantidad de memoria reservada viene determinada por *num*size* (*size* está en bytes), es decir, se obtiene un arreglo de *num* elementos de tamaño *size* bytes. Para obtener un puntero de diferente tipo a ***void***, se debe usar una conversión de tipo con *casting* sobre el retorno de valor.
double ceil(double x);	Redonda por exceso al siguiente entero mayor	<math.h>	La función **ceil()** retorna el menor entero mayor o igual que *x*, donde *x* está representado como valor ***double.***
double cos (double x);	Calcula el coseno	<math.h>	La función **cos()** retorna el coseno de *x* en el rango de -1 a 1. El argumento *x* debe especificarse en radianes.
double cosh(double x);	Calcula el coseno hiperbólico	<math.h>	La función **cosh()** retorna el coseno hiperbólico de *x*. El argumento *x* debe especificarse en radianes.
double difftime(time_t timer1, time_t timer0);	Encuentra la diferencia entre dos tiempos	<time.h>	La función **difftime()** retorna el tiempo transcurrido en segundos desde *timer0* a *timer1*. El valor retornado por la función es un número punto

			flotante ***double***.
void exit(int status);	Provoca la terminación inmediata del programa	<process.h> o <stdlib.h>	Ninguno. El valor *status* será 0 para indicar que la terminación del programa es normal. Los valores distintos de 0 indicarán un error definido en la implementación. La llamada **exit()** pasa a disco y cierra todos los archivos abiertos, escribe todo lo que quede en los flujos de salida, y llama a todas las funciones de terminación registradas mediante **atexit().**
double exp(double x);	Calcula la exponencial en base **e**	<math.h>	La función **exp()** retorna el logaritmo natural de **e** elevado a la potencia del argumento en punto flotante *x*.
double fabs(double x);	Calcula el valor absoluto del argumento **double**	<math.h>	La función **fasb()** retorna el valor absoluto del argumento ***double*** *x*.
int fclose(FILE *stream);	Cierra el archivo asociado con **stream**	<stdio.h>	La unción **fclose()** retorna 0 si el flujo apuntado por *stream* se ha cerrado correctamente, caso contrario devuelve el valor EOF para indicar un error. Después de la llamada a **fclose()** el *stream* se desliga del archivo y se libera cualquier buffer que estuviera asignado.
im feof(FILE *stream);	Comprueba si se alcanzado el final de archivo	<stdio.h>	La función **feof()** retorna un valor diferente de 0 (Verdadero) si el indicador de posición del archivo se encuentra al final del mismo, caso contrario devuelve 0.

int ferror(FILE *stream);	Comprueba el error del archivo del **stream** dado	<stdio.h>	La función **ferror()** retorna 0 si no se ha producido un error y un valor distinto de 0 si se ha producido un error. Esta función comprueba un error de escritura o lectura sobre un archivo asociado con *stream*. Los indicadores de error asociados a*stream* permanecen activos hasta que se cierre el archivo o se llama a **rewind()** o a **clearerr().**
int fflush(FILE *stream);	Vacía el contenido de un flujo especificado en el argumento	<stdio.h>	La función **fflush()** retorna 0 si se ha escrito correctamente el contenido del buffer de salida en el archivo especificado por *stream,* o cuando el contenido del buffer de entrada se vacía. En ambos casos el archivo permanece abierto. Un valor de retorno de EOF indica una situación de error.
int fgetc(FILE * stream);	Lee un caracter desde el flujo **stream**	<stdio.h>	La función **fgetc()** retorna como ***int*** el siguiente carácter leído del flujo de entrada, e incrementa el indicador de posición del archivo. Un valor de retorno de EOF indica que la función **fgetc()** ha alcanzado el final de archivo o que se ha encontrado un error.
char *fgets(char string, int n, FILE *stream);	Obtiene una cadena desde el flujo **stream**	<stdio.h>	La función **fgets()** retorna una cadena leída desde el flujo *stream*, o un valor de retorno NULL que indica un error o una condición de fin de archivo. La función **fgets()** lee caracteres del

			flujo *stream* y los almacena en la cadena *string*, añadiendo el carácter nulo al final del último carácter leído, hasta que se reciba un carácter de salto de línea o un EOF, o hasta que se llega al límite n-1. Si se lee el carácter nueva línea '\n' es incluido en la cadena *string*.
double floor(double x);	Calcula el truncamiento fraccionario de un valor	<math.h>	La función **floor()** retorna el mayor entero que no es mayor que x, donde x está representado como valor **double.**
double fmod(double x. double y);	Calcula el residuo en punto flotante	<math.h>	La función **fmod()** retorna el residuo en punto flotante *x/y.*
FILE *fopen(const char *filename, const char *mode);	Abre un archivo	<stdio.h>	La función **fopen()** retorna un puntero al achivo abierto cuyo nombre se especifica por la cadena *filename.* Un puntero NULL indica un error. El tipo de las operaciones permitidas en el archivo están definidas por la cadena *mode.*
int fprintf(FILE *stream, const char *format [, argument]…);	Imprime datos formateados en un flujo	<stdio.h>	La función **fprintf()** retorna el número de caracteres escritos, y si se produce un error retorna un valor negativo. La función **fprintf()** escribe en el flujo apuntado por *stream* los valores de los argumentos que componen *[, argument]...* según se especifica en *format.*
int fpuc(int c, FILE * stream);	Escribe un caracter desde un flujo	<stdio.h>	La función **fputc()** retorna el caracter escrito en el flujo, y si se produce un error retorna EOF.

			La función **fputc()** escribe el caracter c en el flujo especificado *stream* a partir de la posición actual del archivo y después incrementa el indicador de posición del archivo.
int fputs(const char *string, FILE *stream);	Escribe una cadena desde un flujo	<stdio.h>	La función **fputs()** retorna un valor no negativo (el último caracter escrito) si se tiene éxito, y EOF si se produce un error. La función **fputs()** escribe el contenido de la cadena de caracteres apuntada por *string* en el flujo especificado *stream*. El caracter nulo de terminación no se escribe.
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);	Lee datos desde un flujo	<stdio.h>	La función **fread()** retorna el número de argumentos leídos, el cual puede ser menor que *count* si un error ha ocurrido o si se ha alcanzado el fin de archivo. La función **fread()** lee *count* elementos, cada uno de *size* bytes de longitud, del flujo apuntado por *stream* y los almacena en la porción de memoria apuntada por *buffer*. El indicador de posición del archivo se incrementa en el número total de bytes leídos.
void free(void *memblock);	Libera memoria reservada previamente por **malloc()**	<stdlib.h> y <malloc.h>	Ninguno. La función **free()** devuelve al sistema la memoria apuntada por *memblock* que previamente fue asignada por una función de asignación dinámica. Una vez que la memoria ha sido liberada,

			puede ser reutilizada posteriormente. Nunca debe llamarse a **free()** con un argumento inválido (NULL). porque se destruirá la lista de memoria libre.
int fscanf(FILE *stream, const char *format [, argument]…);	Lee datos formateados desde un flujo	<stdio.h>	La función **fscanf()** retorna el número de argumentos a los que realmente se han asignados valores, este número no incluye los argumentos ignorados. El valor retornado EOF establece que se desea leer luego del final del archivo. La función **fscanf()** lee la información del flujo especificado por *stream*, en las direcciones de memoria dadas por los punteros de las variables que componen los argumentos *[, argument]…* según se especifica en *format*.
int fseek(FILE *stream, long offset, int origen);	Mueve el puntero de archivo a una posición especificada	<stdio.h>	La función **fseek()** retorna 0 si se ha ejecutado correctamente, de lo contrario retorna un valor distinto de 0. La función **fseek()** mueve el indicador de posición de archivo asociado *stream* a una nueva posición *origen* con un desplazamiento *offset*.
long ftell(FILE *stream);	Obtiene la posición actual de un puntero de archivo	<stdio.h>, opcional <errno.h>	La función **ftell()** retorna el valor actual del indicador de posición del archivo del flujo especificado por *stream*, este valor es el número de bytes desde el principio del archivo. Esta función retorna un valor -1L cuando se produce un error,

			y un valor indefinido si en el flujo *stream* especificado no se puede realizar búsquedas aleatorias.
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);	Escribe datos en un flujo	<stdio.h>	La función **fwrite()** retorna el número de argumentos escritos, el cual puede ser más pequeño que *count* si un error ha ocurrido. La función **fwrite()** escribe *count* elementos, cada uno de *size* bytes de longitud, el flujo apuntado por *stream* desde la posición del archivo se incrementa en el número total de bytes escritos.
int getc(FILE *stream);	Obtiene un caracter desde un flujo	<stdio.h>	La función **getc()** retorna el caracter leído del flujo especificado *stream* desde la posición actual, e incrementa el indicador de posición del archivo al siguiente caracter. Para indicar un error de lectura o una condición de fin de archivo la función **getc()** retorna EOF.
~~int getch(void);	Obtiene un caracter desde la consola sin eco	<conio.h>	La función **getch()** retorna el caracter leído. Esta función lee un simple caracter desde la consola sin eco.
~~int getche(void);	Obtiene un caracter desde la consola con eco	<conio.h>	La función **getche()** retorna el caracter leído. Esta función lee un simple carácter desde la consola con eco del carácter leído.
int getchar(void);	Obtiene un caracter desde la consola (	<stdio.h>	La función **getchar()** retorna el caracter leído desde la consola.

	stdin)		Para indicar un error de lectura o una condición de fin de archivo la función **getchar()** retorna EOF.
char *gets(char*bufer);	Obtiene una línea de caracteres (cadena) del flujos **stdin**	<stdio.h>	La función **gets()** retorna el argumento *buffer* si no hay error, y un puntero nulo en caso de error o de una condición de fin de archivo. La función **gets()** lee caracteres de **stdin** y los almacena en la dirección del argumento *buffer*. Los caracteres son leídos hasta que se reciba un caracter salto de línea o una marco EOF. El caracter salto de línea no forma parte de la cadena, porque se transforma en un caracter nulo para terminar la cadena.
int isalnum(int c);	Comprueba si su argumento es un caracter alfanumérico	<ctype.h>	La función **isalnum()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter alfanumérico (una letra del alfabeto inglés o un dígito decimal). Si el caracter no es alfanumérico la función devuelve 0.
int isalpha(int c);	Comprueba si su argumento es un caracter alfabético	<ctype.h>	La función **isalpha()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter alfabético (letra del alfabeto inglés). Si el caracter no es alfabético la función devuelve 0.
int issacii(int c);	Comprueba si su argumento es un caracter ASCIl	<ctype.h>	La función **isascii()** devuelve un valor distinto de 0 (verdadero)

			si su argumento es un caracter ASCII (caracter en el rango 0 a 0x7F). Si el caracter no es ASCII la función devuelve 0.
int iscntrl(int c);	Comprueba si su argumento es un caracter de control	<ctype.h>	La función **iscntrl()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter de control (caracter en el rango 0 a 0x1F o igual a 0x7F). Si el caracter no es de control la función devuelve 0.
int isdigit(int c);	Comprueba si su argumento es un caracter de un dígito decimal	<ctype.h>	La función **isdigit()** devuelve un valor distinto de 0 (verdadero) si su argumento es un dígito decimal (caracter en el rango 0 a 9). Si el caracter no es un digito decimal la función devuelve 0.
im isgraph(int c);	Comprueba si su argumento es un caracter imprimible	<ctype.h>	La función **isgraph()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter imprimible distinto de espacio (caracter en el rango 0x21 a 0x7E). Si el caracter no es imprimible la función devuelve 0.
int islower(int c);	Comprueba si su argumento es un caracter letra minúscula	<ctype.h>	La función **islower()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter letra minúscula. Si el caracter no es letra minúscula la función devuelve 0.
int isprint(int c);	Comprueba si su argumento es un caracter ASCIl imprimible	<ctype.h>	La función **isprint()** devuelve un valor distinto de 0 (verdadero) si su argumento es un carácter ASCII imprimible (carácter en el rango 0x20 a 0x7E), incluyendo el espacio. Si el caracter no es

			ASCII imprimible la función devuelve 0.
int ispunct(int c);	Comprueba si su argumento es un caracter de puntuación o un espacio	<ctype.h>	La función **ispunct()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter de puntuación o un espacio. Si el carácter no es de puntuación o un espacio la función devuelve 0.
int isspace(int c);	Comprueba si su argumento es un caracter de espacio	<ctype.h>	La función **isspace()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter de espacio (espacio en blanco, salto de carro, tabulador, horizontal, tabulador vertical, avance de página o salto de línea). Si el caracter no es de espacio la función devuelve 0.
int issupper(int c);	Comprueba si su argumento es un caracter letra mayúscula	<ctype.h>	La función **isupper()** devuelve un valor distinto de 0 (verdadero) si su argumento es un carácter letra mayúscula. Si el caracter no es letra mayúscula la función devuelve 0.
int isxdigit(int c);	Comprueba si su argumento es un caracter dígito hexadecimal	<ctype.h>	La función **isxdigit()** devuelve un valor distinto de 0 (verdadero) si su argumento es un caracter dígito hexadecimal (caracter en los rangos: 'A' a 'Z', 'a' a 'z' y '0' a '9'). Si el caracter no es un dígito hexadecimal la función devuelve 0.
long labs(long n);	Calcula el valor absoluto de un entero largo (**long**	<stdlib.h> y <math.h›	La función **labs()** retorno el valor absoluto del argumento

	int)		***longint n.***
double log(double x);	Calcula el logaritmo natural	<math.h>	La función **log()** retorna el logaritmo natural del argumento *x* si no hay error. Se produce error de dominio si *x* es negativo, retornando la función un valor indefinido; y un error de rango si el argumento es 0, retornando la función un valor infinito.
double log10(double x);	Calcula el logaritmo en base 10	<math.h>	La función **log10()** retorna el logaritmo en base 10 del argumento *x* si no hay error. Se produce error de dominio si *x* es negativo, retornando la función un valor indefinido; y un error de rango si el argumento es 0, retornando la función un valor infinito.
void *malloc(sizze_t size);	Reserva un espacio de memoria	<stdlib.h> y <malloc.h>	La función **malloc()** retorna un puntero ***void*** al primer byte de una región de memoria de tamaño *size*, o NULL si no existe suficiente memoria libre requerida. Para retornar un puntero de diferente tipo de ***void***, se usa una conversión de tipo con ***casting*** sobre el valor.
double pow(double x, double y);	Calcula **x** elevado al exponente **y**	<math.h>	La función **pow()** retorna la base x elevada al exponente y. Un desbordamiento produce un error de rango.
int printf(const char *format [, argument]…);	Imprime una salida formateada en el flujo de salida estándar **stdout**	<stdlib.h>	La función **printf()** retorna el número de caracteres escritos o un valor negativo si un error ocurre. Esta función escribe en

			el flujo *stdout* los valores de los argumentos que componen *[, argument]...* según se especifica en *format*.
int putc(int c, FILE *stream);	Escribe un caracter en un flujo dado	<stdio.h>	La función **putc()** retorna el caracter escrito *c* en el flujo *stream* si no se ha producido un error. Para indicar un error o condición de fin de archivo, **putc()** retorna EOF.
int putchar(int c);	Ecribe un caracter en la consola (**stdout**)	<stdio.h>	La función **putchar()** retorna el caracter escrito *c* en el flujo *stdout* si no se ha producido un error. Para indicar un error o condición de fin de archivo, **putchar()** retorna EOF.
int puts(const char *string);	Escribe una cadena en la consola (**stdout**)	<stdio.h>	La función **puts()** retorna un valor diferente de 0 (salto de línea), si no se ha producido un error al escribir la cadena apuntada por *string* en el flujo estándar *stdout*. El caracter nulo de terminación se transformará en caracter de salto de línea.
int rand(void);	Genera un número pseudoaleatorio	<stdlib.h>	La función **rand()** retorna un número pseudoaleatorio entre 0 y RAND_MAX
~~ int random(int num);	Genera un número aleatorio	<stdlib.h>	La función **random()** retorna un número aleatorio entre 0 y *num - 1*.
~~ void randomize(void);	Inicializa el generador de números aleatorios	<stdlib.h>	Ninguno. La función **randomize()** inicializa el generador de números aleatorios a un valor aleatorio. Para esto se utiliza la función

			time().
Int remove(const char *path);	Borra el archivo específico por **path**	<stdlib.h> o <io.h>	La función **remove()** retorna 0 si el archivo se ha borrado correctamente, pero si se ha producido un error retorna -1.
int rename (const char *oldname, const char *newname);	Renombra un archivo o un directorio	<stdio.h>	La función **rename()** retorna 0 si tiene éxito en renombrar el archivo o directorio especificado por *oldmame a newname,* y un valor no nulo si se ha producido un error
void rewind(FILE *stream);	Reposiciona el puntero de archivo al principio del archivo	<stdio.h>	Ninguno. La función **rewind()** reposiciona el puntero de archivo al principio del archivo asociado con el flujo *stream*. También inicializa los indicadores de error y de final de archivo asociado con el flujo *stream.* Una llamada a **rewind()** es similar ***a (void) fseek(stream, 0L, SEEK_SET)***;
int scanf(const char, *format [,argument]...);	Lee datos formateados desde el flujo de entrada estándar **stdin**	<stdio.h>	La función **scanf()** retorna el número de argumentos a los que realmente se han asignados valores, este número no incluye los argumentos ignorados. El valor retornado 0 establece que no se han asignado valores a ningún argumento. Y el valor EOF indica que se ha intentado leer después del final del archivo. La función **scanf()** lee la información de cualquier tipo de datos del flujo *stdin* y automáticamente son transformados en el formato

			interno adecuado, en las direcciones de memoria dadas por los punteros de las variables que componen los argumentos [, argument]… según se especifica en *format*.
double sin(double x);	Calcula el seno	<math.h>	La función **sin()** retorna el seno del argumento x. El valor de x se especifica en radianes,
double sinh(double x);	Calcula el seno hiperbólico	<math.h>	La función **sinh()** retorna el seno hiperbólico del argumento x. El valor de x se especifica en radianes.
double sqrt(double x);	Calcula la raíz cuadrada	<math.h>	La función **sqrt()** retorna la raíz cuadrada del argumento x. Si x es negativo se produce un error de dominio y **sqrt()** retorna un valor indefinido.
void srand(unsigned int seed);	Establece un punto inicial de la secuencia que genera **rand()**	<stdlib.h>	La función **srand()** establece el punto inicial para generar una serie de números pseudoaleatorios. Para reinicializar la generación, se usa 1 como argumento y cualquier otro valor para inicializar una generación randomica.
char *strcat(char *strDestination, const char *strSource);	Añade una cadena el final de otra	<string.h>	La función **strcat()** retorna la cadena *strDestination*. La función **strcat()** concatena una copia de *strSource* en *strDestination* y añade al final de *strDestination* un carácter nulo, eliminando el carácter nulo anterior. La cadena *strSource* no

			se modifica en esta operación.
char *strchr(const char *string, int c);	Encuentra un caracter en una cadena	<string.h>	La función **strchr()**retorna un puntero a la primera ocurrencia del carácter c en la cadena apuntada por *string,* si no lo encuentra retorna un puntero nulo.
int strcmp(const char *string1, const char *string2);	Compara dos cadenas	<string.h>	La función **strcmp()** retorna un entero del resultado de la comparación lexicográfica de las cadenas *string1* y *string2,* dicho valor puede ser: menor a 0 si *string1* es menor que *string2,* igual a 0 si *string1* es igual a *string2* y mayor a 0 si *string1* es mayor que *string2.*
int strcopy(const char *strDestination, const char *strSource);	Copia el contenido de una cadena en otra	<string.h>	La función **strcpy()** retorna un puntero a la cadena *strDestination.* La función **strcpy()** copia el contenido de *strSource* a *strDestination*; el argumento de *strDestination* debe ser un puntero a cadena.
int strlen(const char *string);	Obtiene la longitud de una cadena	<string.h>	La función **strlen()** retorna el número de caracteres de la cadena apuntada por *string.* El caracter nulo no se contabiliza.
int strstr(const char *string, const char *strCharSet);	Encuentra una subcadena dentro de otra	<string.h>	La función **strstr()** retorna un puntero a la primera ocurrencia en la cadena apuntada por *string* de la cadena apuntada por *strCharSet,* sin tener en cuenta el caracter nulo de *strCharSet,* devolviendo un puntero nulo si no la encuentra.
int strtok(const char *strToken, const char *strDelimit);	Encuentra la próxima señal o marca de una cadena	<string.h>	La función **strtok()** retorna un puntero a la siguiente palabra de la cadena apuntada por

			strToken y un puntero nulo cuando no hay ninguna palabra que retornar. Los caracteres que constituyen la cadena apuntada por *strCharDelimit* son los delimitadores que identifican la palabra. La primera vez que se llama **strtoken()** se utiliza realmente *strToken* en la llamada, las llamadas posteriores utilizan un puntero nulo como primer argumento. De este modo la cadena completa se puede reducir a sus palabras. Además, se modifica la cadena apuntada por *strToken,* porque cada vez que se encuentra una palabra se pone un caracter nulo donde estaba el delimitador.
int system (const char *command);	Ejecuta un commando	<process.h> o <stdlib.h>	La función **system()** retorna un valor diferente de 0 (estado de salida de ese comando) si el comando interpretado es encontrado o es NULL, caso contrario retorna 0. La fundón **system()** pasa como una orden al DOS la cadena (comando) apuntada por *command.*
double tan(double x);	Calcula la tangente	<math.h>	La función **tan()** retorna la tangente del argumento *x*. El valor de x se especifica en radianes.
double tanh(double x);	Calcula la tangente hiperbólica	<math.h>	La función **tanh()** retorna la tangente hiperbólica del argumento x. El valor de x se

			especifica en radianes.
time_t time(time_t *timer);	Obtiene el tiempo del sistema	<time.h>	La función **time()** retorna la hora actual del calendario del sistema. La función **time()** puede llamarse con un puntero nulo o con un puntero a una variable de tipo *time_t*, si se utiliza este último, el argumento es asignado a la hora del calendario.
int_toascii(int c);	Convierte un entero en caracteres	<ctype.h>	La función **toascii()** retorna el resultado de convertir el argumento entero c a caracter, es decir, limpia el byte más significativo y retorna el resultado como caracter.
int tolower(int c);	Convierte letras mayúsculas en letras minúsculas	<stdlib.h> y <ctype.h>	La función **tolower()** retorna el equivalente en minúscula del argumento c representado en letra mayúscula, caso contrario devuelve c sin modificar.
int toupper(int c);	Convierte letras minúsculas en letras mayúsculas	<stdlib.h> y <ctype.h>	La función **toupper()** retorna el equivalente en mayúscula del argumento c representado en letra minúscula, caso contrario devuelve c sin modificar.
type va_arg(va_list arg_ptr, type);	Devuelve el argumento actual	<stdio.h> y <stdarg.h>	La función **va_arg()** retorna el argumento actual. Ver tema 6.11.1.
void va_end(va_list arg_ptr);	Restaura correctamente la pila	<stdio.h> y <stdarg.h>	Ninguno. Ver tema 6.11.1.
void va_start(va_list arg_ptr);	inicializa el puntero **arg_ptr**	<stdio.h> y <stdarg.h>	Ninguno. Ver tema 6.11.1.
int vprintf(const char *format, va_list argptr);	Escribe en un flujo mediante un	<stdio.h› y <stdarg.h›	La fundón **vprintf()** retorna el número de caracteres escritos,

puntero a una lista de argumentos

no incluye el caracter nulo de terminación, o retorna un valor negativo si un error de salida ocurre. La función **vprintf()** retorna un puntero a una lista de argumentos (de tipo *va_list*), entonces da un formato y escribe los datos en *stdout*.

NOTAS:

- Las funciones macadas con ~~~ no son compatibles con ANSI, pero son compatibles con Win 95 y Win NT
- Las funciones no marcadas son compatibles con ANSI, Win 95 y Win NT

Apéndice C

Índice de Programas

DIRECTORIO	ARCHIVOS	PÁGINA
CAPITULO.3	prog0301.c	57
	prog0302.c	61
	prog0303.c	68
	prog0304.c	69
CAPITULO.4	prog0401.c	93
	prog0402.c	95
	prog0403.c	98
	prog0404.c	99
	prog0405.c	100
	prog0406.c	101
	prog0407.c	106
	prog0408.c	108
CAPITULO.5	prog0501.c	132
	prog0502.c	133
	prog0503.c	135
	prog0504.c	137
	prog0505.c	139
CAPITULO.6	prog0601.c	154
	prog0602.c	156
	prog0603.c	157
	prog0604.c	158
	prog0605.c	160
	prog0606.c	162
	prog0607.c	163
	prog0608.c	170
	prog0609.c	173
	prog0610.c	176
	prog0611.c	177
CAPITULO.7	prog0701.c	202
	prog0702.c	204
	prog0703.c	206

	prog0704.c	209
CAPITULO.8	prog0801.c	231
	prog0802.c	234
	prog0803.c	235
	prog0804.c	237
	prog0805.c	239
	prog0806.c	244
	prog0807.c	247
	prog0808.c	249
	prog0809.c	252
	prog0810.c	255
	prog0811.c	257
	prog0812.c	259
	prog0813.c	266
	prog0814.c	267
CAPITULO.9	prog0901.c	301
	prog0902.c	307
	prog0903.c	308
	prog0904.c	311
	prog0905.c	312
	prog0906.c	319
	prog0907.c	321
	prog0908.c	322
CAPITULO. 10	prog1001.c	345
	prog1002.c	352
	prog1003.c	354
	prog1004.c	356
	prog1005.c	357
	prog1006.c	358
	prog1007.c	360
	prog1008.c	369
	prog1009.c	373
	prog1010.c	378
	prog1011.c	384
CAPITULO.11	prog1101.c	411
	prog1102.c	412
	prog1103.c	414
	prog1104.c	416
	prog1105.c	417

	prog1106.c	419
	prog1107.c	423
	prog1108.c	426
	prog1109.c	427
	prog1110.c	431
	prog1111.c	436
CAPITULO.12	prog1201.c	465
	prog1202.c	468
	prog1203.c	468
	prog1204.c	471
	prog1205.c	474
	prog1206.c	476
	prog1207.c	480

Referencias Bibliográficas

1. SCHILDT Herbert, ***TURBO C/C++ 3.1 Manual de referencia***, Primera Edición en Español, Editorial McGraw Hill/ Interamericana de España, España, 1994.

2. DEITEL, H. M., DEITEL, P. J., ***Cómo programar en C/C++***, Segunda Edición en Español, Editorial Prentice Hall Hispanoamericana, México, 1995.

3. GOTTFRIED, Byron, ***Programación en C***, Primera Edición en Español, Editorial McGraw Hill /Interamericana de España, España, 1991.

4. WAIT E, Mitchell, PRATA Stephen, MARTIN Donald, ***Programación en C***, Primera Edición en Español, Editorial Anaya Multimedia, España, 1984.

5. BECERRA, César; ***Lenguaje C***, Primera Edición, Editorial Arfo Colombia, 1991.

6. KERNIGHAN, Brian, RITCHIE, Dennis, ***El lenguaje de programación C***, Primera Edición en Español, Editorial Prentice Hall Hispanoamericana, ` México, 1985.

7. BORLAND, ***Turbo C++ 3.0***, Estados Unidos de América (USA), 1992.

Publicaciones Científicas