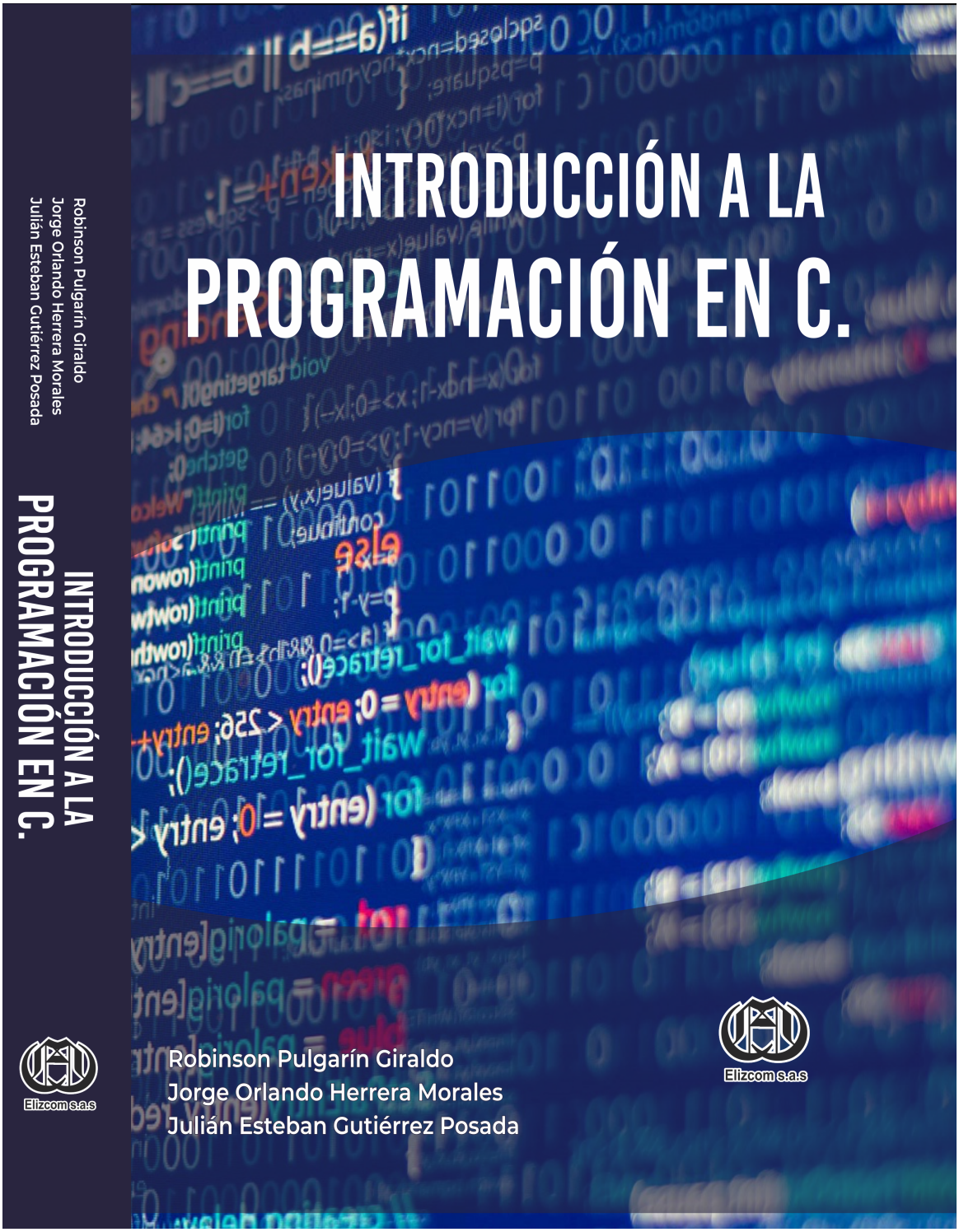


INTRODUCCIÓN A LA
PROGRAMACIÓN EN C.
Robinson Pulgarín Giraldo
Jorge Orlando Herrera Morales
Julián Esteban Gutiérrez Posada
Elizcom s.a.s

Introducción a la Programación en C

Robinson Pulgarín G., Jorge O. Herrera M., Julián E. Gutiérrez P.

Armenia - Quindío - Colombia
2018

Primera edición: Colombia, diciembre 2018

Introducción a la Programación en C
ISBN 978-958-8801-71-1

Editorial: ElIZCOM S.A.S
http://www.liber-book.com
ventas@elizcom.com
+57 311 334 97 48

Arte de los diagramas de flujo:
María Alejandra Martos Díaz.
Steven Sáenz Moscoso.

Editado en: LaTeX 2_ε

Diciembre de 2018

ÍNDICE GENERAL

PRESENTACIÓN

Esta obra pretende explorar la lógica de programación a través del Lenguaje C y los diagramas de flujo, con el propósito de ofrecer una alternativa diferente para el aprendizaje de estos temas a quienes se inician en el estudio de esta interesante área. En ella, los autores han expuesto sus conocimientos y han plasmado sus diversas experiencias obtenidas como profesores durante varios años en instituciones de educación superior en las carreras donde se hace imprescindible el aprendizaje de la lógica de programación y el Lenguaje C.

Sin duda alguna, el Lenguaje de Programación C, es uno de los lenguajes de programación de sistema que más se ha utilizado desde los inicios de la computación, no solo por el tiempo que tiene desde su creación, sino porque está disponible, básicamente, en todos los sistemas y arquitecturas de computadores existentes, ya sea a escala muy pequeña (sistemas embebidos o dedicados), hasta súper-computadoras de todo tipo; esto incluye, por supuesto, todos los sistemas operativos Unix y similares, así como Windows.

Algunas de las razones de su gran portabilidad, es que posee un número reducido de instrucciones y bibliotecas estándar, que facilitan su migración, además de ser un lenguaje muy eficiente que emplea pocos recursos computacionales; por esto, es uno de los principales lenguajes en las competencias de programación mundial y es utilizado con frecuencia en la computación de alto desempeño a nivel de clusters y supercomputación.

En este texto se exponen claramente los conceptos más importantes del Lenguaje, pero sin tratar de entrar en formalismos innecesarios que dificulten el aprendizaje. A su vez, se utilizan ejemplos de diferentes tipos para ilustrar cada uno de los tópicos estudiados en cada capítulo, desde los más sencillos hasta otros más complejos que guían paulatinamente al

lector en la forma como deben resolverse los problemas computacionales, promoviendo el desarrollo de las habilidades necesarias en la escritura de programas.

Para el desarrollo de los temas, la obra ha sido escrita en los siguientes siete capítulos:

Capítulo 1, Introducción: este capítulo introduce al lector en los conceptos generales sobre los lenguajes de programación, especialmente del Lenguaje C; seguidamente se abordan los conceptos como ambiente de trabajo, de datos y sus tipos, las variables y las constantes, los operadores y las expresiones, los tipos de algoritmos y cómo solucionar problemas a través del Lenguaje C.

Capítulo 2, Estructura secuencial: a través de este capítulo se hacen las explicaciones generales sobre cómo escribir los primeros programas, tanto en Lenguaje C como con diagramas de flujo y la forma adecuada de probarlos. El capítulo expone una buena cantidad de ejemplos de programas debidamente documentados.

Capítulo 3, Estructuras de decisión: en este apartado se exponen de forma clara las instrucciones necesarias para indicar a un algoritmo la forma de realizar un conjunto de tareas dependiendo de una condición. Se utilizan ejemplos documentados sobre las principales estructuras de decisión: simple, compuesta, anidada y múltiple.

Capítulo 4, Estructuras de repetición: se inicia definiendo los términos de contador, acumulador y bandera, utilizados durante todo el capítulo. Posteriormente se tratan cada una de las instrucciones repetitivas como son: el ciclo `while`, el ciclo `do-while` y el ciclo `for`. Todas estas estructuras son explicadas desde el punto de vista conceptual y práctico, con el fin de que el lector comprenda no solamente los conceptos sino que aprenda a utilizarlos en la solución de problemas computacionales.

Capítulo 5, Procedimientos y Funciones: se llevan a cabo las definiciones de lo que son procedimientos y funciones, su uso dentro del Lenguaje C y se expone una serie de ejemplos prácticos que ilustran la utilidad de este tipo de instrucciones.

Capítulo 6, Vectores y Matrices: en este capítulo se introduce al lector en el tema de los arreglos unidimensionales y bidimensionales con el propósito de que se comprendan los conceptos generales sobre estas estructuras de datos y se aprendan a utilizar cuando se deban resolver problemas algorítmicos que así lo requieran.

Capítulo 7, Archivos: en este último capítulo, se brindan las herramientas básicas para aprender a guardar datos en dispositivos de almacenamiento. Se exploran diferentes funciones para el manejo de archivos de texto y archivos binarios.

Anexo A, Instalación del ambiente: en este anexo, se presentan los conceptos generales de un ambiente de trabajo para programar en Lenguaje C. El ambiente incluye el uso del compilador de C - GCC/GNU y un entorno integrado de programación IDE, llamado Geany, en los sistemas operativos: Windows, MacOS, Linux. Además se recomienda una aplicación para tener un ambiente de trabajo en dispositivos móviles.

Organización de la obra.

Para abordar cada tema el libro fue organizado en capítulos, cada uno desarrolla los conceptos teóricos, llevados a la práctica con los suficientes ejemplos que ilustran su uso en la solución de problemas de tipo computacional.

Uno de los aspectos a destacar, es la forma como fueron abordadas las soluciones de los ejemplos propuestos. Para cada uno se hizo un análisis previo, tratando de tener un mayor dominio sobre la problemática, ya que es fundamental un total entendimiento para poder emprender la búsqueda de una adecuada solución. Cada ejemplo desarrollado fue cuidadosamente detallado desde su análisis hasta la concepción del programa en Lenguaje C y en la mayoría de los casos acompañados de su correspondiente diagrama de flujo; adicionalmente, se hace una descripción minuciosa de cada una de las instrucciones que fueron usadas. Esta metodología busca crear la buena práctica de realizar un análisis detallado del problema y entender cada uno de los pasos de la solución.

Adicionalmente, dentro de las diferentes páginas el lector encontrará el recuadro de Buenas prácticas y el recuadro de Aclaraciones:

Buena práctica:

En este recuadro se mostrarán recomendaciones que se espera sean seguidas por los lectores de esta obra, con el fin de que aprendan, desde sus inicios como desarrolladores, buenas prácticas para un mejor desempeño profesional.

Imagen tomada de [Pixabay.com, 2018a].

Aclaración:

Esta anotación, es usada para resaltar algunos conceptos o dar mayor claridad sobre aspectos que son fundamentales en el desarrollo de algoritmos.
Imagen tomada de[Pixabay.com, 2018b].

Otra característica importante, es que se aprovechó el formato digital de la obra, de tal forma que cada una de las referencias escritas tienen presente un link, permitiendo que con solo pulsar la referencia se puede dirigir al sitio correspondiente.

Por último, en cada capítulo, los autores han dejado una serie de ejercicios propuestos que van desde preguntas teóricas que pretenden afianzar en el lector los conceptos estudiados, hasta los enunciados de ejercicios prácticos para que el estudioso escriba los algoritmos que solucionen el problema propuesto utilizando la lógica y la programación en el Lenguaje C. Estos ejercicios están enmarcados como actividades y son reconocidos de la siguiente manera:

Actividad 0.1

Estos recuadros los encontrará a lo largo del libro; a medida que se van desarrollando los temas se plantean diferentes actividades para que el lector pueda afianzar sus conocimientos.
Imagen tomada de [Pixabay.com, 2018c].

Con todo el trabajo realizado desde la concepción de esta obra, pasando por su escritura y elaboración de la gran cantidad de programas que en ella se exponen, los autores esperan hacer una contribución en la difusión de la lógica de programación y la programación en Lenguaje C, así como orientar y facilitar el aprendizaje de estos temas a todos aquellos lectores apasionados de la informática y la computación. Es por eso, que para lograr este propósito el libro fue concebido bajo la **Licencia Creative Commons** permitiendo el uso y la distribución gratuita, siempre y cuando la obra se conserve igual y se haga el respectivo reconocimiento a sus creadores.

Así mismo, los autores desean expresar su agradecimiento a todos aquellos que directa o indirectamente han inspirado este libro, entre los cuales se encuentran, estudiantes quienes manifiestan la necesidad de un texto que facilite el aprendizaje, docentes que resaltan la importancia de contar con un libro de referencia y guía en el salón de clases y profesionales de diferentes áreas quienes desean contar con una referencia de consulta para resolver alguna duda o inquietud.

Finalmente, para facilitar el estudio de los ejemplos presentados en el libro, se cuenta con un sitio web oficial, de donde se pueden consultar y descargar todos los programas fuentes, así como también una copia de este documento:

`http://sara.uniquindio.edu.co/LenguajeC/`

Con el anterior código QR puede ir al sitio web oficial del libro, además en cada inicio de capítulo encontrará un código similar, que al ser escaneado o al dar clic sobre él, se dirigirá directamente a los programas específicos de cada tema.

CAPÍTULO 1

FUNDAMENTOS

Primero resuelve el problema.
Entonces, escribe el código

John Johnson

Objetivos del capítulo:

- Conocer los aspectos más generales de la programación.
- Utilizar operadores para construir expresiones básicas.
- Construir expresiones aritméticas que resuelvan problemas algorítmicos.
- Identificar, evaluar y construir operaciones aritméticas, relacionales y lógicas.
- Aplicar los pasos para la construcción de programas.

1.1. Algo de historia

1.1.1 Lenguajes de programación

Un lenguaje de programación, proporciona un conjunto de instrucciones que deben respetar ciertas reglas (*Sintaxis*) y que expresan un significado (*Semántica*). Por medio de estas instrucciones, es posible crear un programa que de la solución a un problema determinado.

Los programas, son una forma de escribir un algoritmo que pueda ser ejecutado por una máquina, ya sea un computador de escritorio, portátil o incluso un dispositivo móvil como un teléfono inteligente, entre otros dispositivos programables. Mediante un programa es posible controlar tanto elementos físicos (hardware) como lógicos (software) de la máquina.

Para que una máquina pueda ser programada, tal y como lo entendemos hoy en día, es necesario que ella posea al menos un procesador. Los procesadores son circuitos integrados que pueden ser programados mediante un lenguaje específico de cada uno, denominado lenguaje de máquina. El lenguaje de máquina es ideal para las máquinas, pero es difícil de comprender para las personas, ya que se escribe únicamente mediante números. Allí comienza la necesidad de facilitar la programación de las máquinas y comienza todo un desarrollo en la línea de los lenguajes de programación (Ver Figura 1.1).

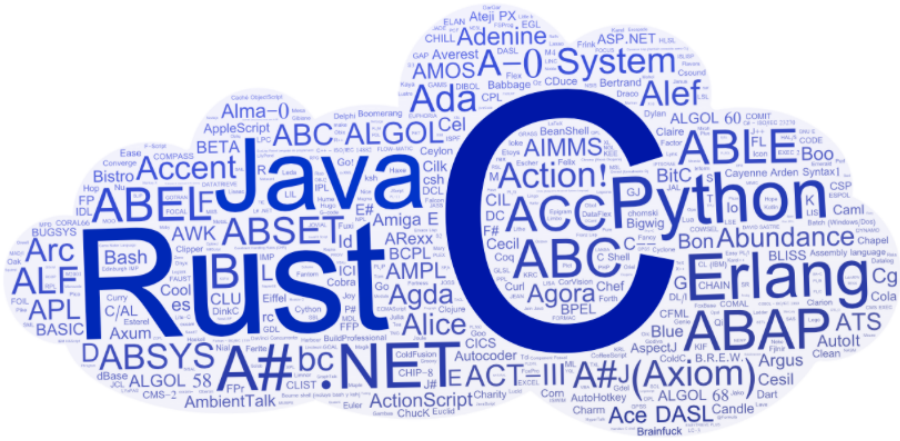


Figura 1.1: Algunos lenguajes de programación

Un lenguaje no numérico, pero que es traducido directamente al lenguaje de máquina, es el llamado Lenguaje Ensamblador. Por ser un lenguaje tan cercano al lenguaje de máquina se considera que es un lenguaje de bajo nivel. Sin embargo realizar todo tipo de tareas en Lenguaje Ensamblador, aunque posible, es dispendioso debido a sus propias características, cuyas instrucciones son poco expresivas, en comparación con otros lenguajes de más alto nivel, como el Lenguaje C; complicando la labor de programar tareas no triviales. Aunque al final, todos los lenguajes de programación deben ser traducidos al único lenguaje que el procesador comprende, el lenguaje de máquina, no obstante esta es una tarea oculta para las personas.

Por lo anterior, existen muchos lenguajes de programación, unos más fáciles de comprender y utilizar que otros; algunos con propósitos específicos y otros de propósito general. El Lenguaje C, es un lenguaje de sistema, considerado de propósito general y lo suficientemente cercano al procesador (sistema) para ser eficiente y poderoso, como para crear, desde sistemas operativos hasta aplicaciones generales de un usuario promedio.

Aunque existen otros lenguajes más fáciles de manejar que el Lenguaje C, estos lenguajes sacrifican control al programador o velocidad para ejecutar las aplicaciones, al agregar componentes entre el procesador y el programador que brindan esa facilidad de uso. Un buen programador, debe comprender este universo de lenguajes y usar a favor los beneficios de cada lenguaje según la necesidad actual en la solución de un problema. Los lenguajes son herramientas y como tal, deben ser utilizadas para resolver las tareas para las que fueron diseñadas.

El primer lenguaje de programación fue Plankalkül (plan de cálculo) propuesto por Konrad Zuse en 1946, no obstante permaneció en la teoría hasta el año 2000, año en el que fue implementado en la Universidad Libre de Berlín. Otro lenguaje importante es Fortran, creado por John Backus en 1953. En la actualidad, existen versiones modernas de este lenguaje, que es utilizado principalmente en el campo científico, ya que está especializado en cálculos matemáticos. De nuevo, es posible realizar estos cálculos en Lenguaje Ensamblador, o en Lenguaje C, pero la facilidad y eficiencia que ofrece Fortran para este tipo de tareas específicas no tiene comparación.

Todos los lenguajes de programación pueden ser clasificados en dos paradigmas de programación. El paradigma **Imperativo** y el **Declarativo**. Un lenguaje pertenece a uno u a otro dependiendo de las siguientes características: si el programador debe describir paso a paso un conjunto de instrucciones que deben ejecutarse para resolver un problema,

entonces se dice que pertenece al paradigma imperativo; si por el contrario, el programador solo describe el problema que quiere resolver, pero sin indicar el paso a paso, se dice que pertenece al paradigma declarativo. Los lenguajes declarativos posen mecanismos internos que son capaces de resolver un problema mediante la descripción del problema; un lenguaje de este paradigma es el lenguaje Prolog.

El Lenguaje C es un lenguaje de la familia de lenguajes imperativos. A continuación se presenta una breve historia del lenguaje.

1.1.2 Algo de historia del Lenguaje C

El lenguaje de programación C fue creado entre 1969 y 1973 por Dennis M. Ritchie en los Laboratorios Bell de AT&T y es una evolución de otro lenguaje de programación conocido como B.

Desde entonces, el Lenguaje C es ampliamente utilizado para el desarrollo de sistemas que requirieran mucha eficiencia. Ejemplos importantes a nivel de sistemas operativos son: Windows 10, Unix, Linux, entre miles de aplicaciones de propósito más general.

El Lenguaje C es considerado un lenguaje de nivel medio, en el sentido que tiene instrucciones que permite el acceso directo a la máquina y al mismo tiempo dispone de estructuras de alto nivel que facilitan la programación con respecto a otros lenguajes de bajo nivel.

C, es un lenguaje de programación compilado, esto quiere decir, que requiere que para poder ejecutar los programas escritos en este lenguaje se deba pasar por un proceso de conversión automático a un lenguaje binario. En este proceso de conversión, el sistema verifica que todo se encuentre correctamente escrito (análisis sintáctico) y que al mismo tiempo sea coherente (análisis semántico), solo después de ambos análisis y de un proceso de optimización, el compilador genera el código binario que es ejecutado por la computadora.

El Lenguaje C ha pasado por varias estandarizaciones que garantizan la portabilidad de los programas a otras plataformas y/o arquitecturas de computadores diferentes al sistema en donde fue desarrollado originalmente. El estándar más reciente es el ISO/IEC 9899:2011.

Algunas de sus características son:

- Lenguaje simple que extiende mediante biblioteca que son incorporadas mediante archivos en la cabecera de los programas.
- Posee un lenguaje que se procesa antes de la compilación, lo que permite definir entre otras cosas macro y constantes simbólicas.
- Permite el acceso a memoria directamente mediante el uso de punteros.
- Posee un número muy reducido de instrucciones (palabras reservadas).
- Es un lenguaje extremadamente eficiente y finalmente es un lenguaje estandarizado que facilita su portabilidad.

1.1.3 El ambiente de trabajo

Si el lector desea involucrarse activamente en su desarrollo personal y profesional, específicamente en lo relacionado con la programación con el Lenguaje C, es totalmente indispensable que instale un ambiente de trabajo (Ver Anexo A) que le permita experimentar los ejemplos del libro, además de tener un ambiente para crear sus propios proyectos.

Tenga presente que en este contexto, un ambiente de trabajo consta de dos elementos: el compilador de Lenguaje y un editor de texto, de preferencia, especializado para programar (IDE - Entorno Integrado de Desarrollo).

Entre los secretos para ser un gran programador, se encuentra el practicar mucho, el leer código bien escrito y en ponerse retos que vayan creciendo en complejidad. Tener retos demasiado simples con respecto al conocimiento actual, no produce un mayor impacto; el tener retos muy por encima del nivel de desarrollo, genera frustración; por supuesto, no deje de soñar y crezca motivado al superar verdaderos retos que le permitan crecer como persona y como profesional.

Durante el desarrollo de este capítulo, se explicarán los elementos fundamentales necesarios para la construcción de programas en Lenguaje C, entre estos elementos se encuentran: la declaración de variables, los diferentes tipos de operadores existentes en la programación y la construcción de expresiones.

Aclaración:

Un compilador es una aplicación que traduce de un lenguaje a otro, en este caso, del Lenguaje C a Lenguaje de máquina (código binario que es ejecutado por la computadora).

También es posible usar ciertos compiladores para generar código binario para máquinas y sistemas diferentes a la actual; en este caso, se habla de una "compilación cruzada". Por supuesto, el código generado no podrá ser ejecutado en la máquina actual, sino que deberá ser llevado a la máquina destino para su ejecución; no obstante, el proceso de compilación es útil para detectar y corregir los posibles errores.

También hay compiladores que generan código para una máquina virtual (ej: Java) y para poder ejecutar los programas, se requiere que dicha máquina virtual esté instalada en la máquina real que se está utilizando.

Finalmente, hay lenguajes que son interpretados por otro programa, como el caso de Python, en ese caso, es necesario disponer del intérprete instalado, para que pueda ejecutar el programa desarrollado, y de paso, detecte los posibles errores.

1.2. Dato

Un dato corresponde a una representación simbólica de una característica de un elemento, cosa u objeto. Esto quiere decir que un dato puede estar representado por una cifra, letra, palabra o conjunto de palabras que caracterizan el elemento o cosa que se está describiendo. Por ejemplo, "Cra 15 #8-26" y "Masculino" representarían la dirección de residencia y el género de una persona, entendiendo que la dirección y el género son las características de esa persona.

Es importante mencionar que un solo dato no representa información, sino que, la información está compuesta por todo un conjunto de datos. Los programas de computador procesan y almacenan datos con el propósito principal de producir información. Por esta razón, se hace necesario conocer los tipos de datos que maneja el Lenguaje C.

1.2.1 Tipos de datos

Un tipo de dato es una clasificación que permite tratar cada dato de la forma más adecuada, de acuerdo con lo que se necesita. El tipo de dato le indica al computador la cantidad de memoria a reservar para almacenar el dato, lo que permite administrar el uso de la memoria. Los tipos de datos manejados por Lenguaje C son:

Datos alfanuméricos

Datos compuestos por caracteres: letras del alfabeto, dígitos y caracteres especiales, incluyendo el espacio en blanco. Los datos alfanuméricos se dividen en teóricamente en: carácter y cadena.

- **carácter**: constituidos por los datos que solo tienen un carácter, que puede ser una letra cualquiera, un dígito del 0 al 9 o un carácter especial. Los valores de tipo carácter deben encerrarse entre comillas simples.

  Los siguientes son ejemplos de datos tipo carácter: 'a', 'R', '2', '#', '@'

  La palabra que se emplea para declarar una variable carácter es `char`. Para el Lenguaje C, un dato `char` también es considerado un dato entero y el valor es el código numérico que corresponde al carácter almacenado (Ver tabla de codigos ASCII[1]). Por ejemplo, el carácter 'A' tiene el código 65, el 'B' tiene el 66 y así sucesivamente. Para los computadores, los caracteres en minúsculas son diferentes a los mismos caracteres en mayúsculas, por tanto, el carácter 'a' tiene como código 97, 'b' es 98, entre todos los demás caracteres válidos.

  En Lenguaje C es posible realizar operaciones matemáticas con los caracteres, ya que internamente, se realiza con los código asociados a dichos caracteres y el resultado será el carácter que tenga asociado el código que se obtuvo. Por ejemplo: 'A' + 5 es equivalente a decir 65 + 5, lo que da como resultado 70; el carácter que tiene asociado el código 70 es 'F' y ese sería el resultado de dicha operación.
- **Cadena**: datos formados por un conjunto de letras del alfabeto, dígitos y caracteres especiales, incluyendo el espacio en blanco. Los datos de tipo cadena van entre comillas dobles.

[1] **ASCII** - *American Standard Code for Information Interchange* - https://es.wikipedia.org/wiki/ASCII

Algunos ejemplos de datos de tipo cadena: “Carrera 17 # 12 – 65”, “Perro”, “Adriana Ospina”, “7534523”, “018000-43433”.

Los dos últimos datos, aunque están compuestos por solo números, no pueden usarse para hacer operaciones aritméticas por ser de tipo cadena de caracteres.

En Lenguaje C no existe un tipo de dato cadena propiamente dicho, sino que, cuando se requieren almacenar datos de este tipo, deberán usarse arreglos de caracteres (`char`), tema que se estudiará más adelante.

Datos numéricos

Son datos compuestos por solo números y signos (positivo y negativo), es decir, dígitos del 0 al 9, que serán usados en operaciones aritméticas. Estos tipos de datos se subdividen en: Enteros y Reales.

- **Enteros**: datos compuestos por números sin punto decimal. Pueden ser números positivos, negativos o el cero.

  Algunos ejemplos son: 20, -5, 200, 1500000.

  En Lenguaje C, existen varios tipos de datos para declarar un número entero. La diferencia entre ellos es la cantidad de memoria del computador que requieren y por consiguiente, el rango de números que puede almacenar.

  Note del párrafo anterior que a diferencia de las matemáticas, en las computadoras, los números son finitos, ya que requieren de espacio en la memoria del computador para ser almacenados y su capacidad tiene un límite. Para comprender mejor esta afirmación vea la Tabla 1.1, en donde **Tipo** hace referencia a la palabra o palabras reservadas de Lenguaje C que se deben emplear para usar ese tipo de datos en particular; algunos ejemplos serán presentado un poco más adelante en este capítulo.

  Algunos de estos tamaños puede variar de un compilador a otro. Los valores presentados en la Tabla 1.1 son lo que emplea el compilador GCC/GNU que se está usando en el libro.

  Para la gran mayoría de los problemas con enteros, se suele usar `int`, que para el caso, es un número lo suficientemente grande.

Tipo	Tamaño(Bytes)	Rango válido
char	1	-128 a 127
signed **char**		
unsigned **char**		0 a 255
short	2	-32768 a 32767
signed **short**		
unsigned **short**		0 a 65535
int	4	-2.147'483.648 a +2.147'483.647
signed **int**		
unsigned **int**		0 a 4.294'967.295
long	8	-9'223.372'036.854'775.808 a +9'223.372'036.854'775.807
signed **long**		
unsigned **long**		0 a 18'446.744'073.709'551.615

Tabla 1.1: Tipos de datos enteros en Lenguaje C

- **Reales**: estos datos tienen componente decimal, pueden ser positivos, negativos o cero.

  Algunos ejemplos son: 1.75, 4.5, 1800000.00, -234.00.

  Es importante tener en cuenta que las unidades de mil no se deben separar con puntos o comas. El punto solo se usa para indicar la parte decimal. Al igual que sucede con los enteros, los datos reales tienen diferentes clasificaciones y se diferencian en la cantidad de memoria que requieren (tamaño) y por supuesto del rango de números que acepta. La palabra o palabras reservadas en Lenguaje C para emplear un tipo real se pueden ver en la columna **Tipo** de la Tabla 1.2. Aunque en la práctica, el tipo `float` suele ser más que suficiente para la mayoría de programas con un fin no científico.

Tipo	Tamaño(Bytes)	Rango válido
float	4	1.175494E-38 a 3.402823E+38
double	8	1.797693E-308 a 1.797693E+308
long **double**	16	3.362103E-4932 a 1.189731E+4932

Tabla 1.2: Tipos de datos reales en Lenguaje C

Datos lógicos o booleanos

Son datos que pueden tener solo uno de dos valores booleanos[2]: verdadero o falso. Estos valores son equivalentes a los dígitos del sistema binario: 1 corresponde a Verdadero y 0 a Falso [Mancilla et al., 2016]. Por ejemplo, el dato generado al preguntársele a una persona si es o no mayor de edad, cualquiera que sea la respuesta, corresponde a un dato lógico: verdadero o falso.

En el Lenguaje C no existe un tipo de dato directamente de tipo lógico, el lenguaje lo maneja como un número entero, por defecto 1 es considerado verdadero, mientras que 0 es falso; por lo anterior, si se desea definir manualmente dos elementos llamados, por ejemplo, `true` y `false`, con los valores enteros 1 y 0 respectivamente, se puede hacer como se indica a continuación:

```
#define true  1
#define false 0
```

Es de aclarar que, aunque el ejemplo se hace con los términos en inglés, es totalmente posible definir los valores en español.

Otra forma es incluir el archivo de cabecera (biblioteca) llamado `stdbool.h`. Esta biblioteca define exactamente estos valores como ya fue explicado mediante el empleo de `#define`. Pero además proporciona un tipo de dato nuevo llamado `bool`, el cual es el mismo tipo de dato `int` pero con otro nombre. Visto así, parece no ser muy útil, sin embargo, es importante en la medida que permite ganar legibilidad, tal y como será ejemplificado más adelante. Durante todo el libro y en los programas que hace uso de un tipo de dato lógico, se hace uso de la biblioteca `stdbool.h`.

1.3. Identificadores

Un identificador es el nombre que se le da a una variable, constante, función y/o procedimiento, el cual se utiliza para referirse a dicho elemento dentro de un programa.

La asignación adecuada de nombres obedece a una serie de reglas que facilitan su escritura. Vale la pena mencionar que, cada lenguaje de programación establece sus propias reglas y recomendaciones al

[2]Un valor booleano solo admite una de dos posibles respuestas que son mutuamente excluyentes: verdadero o falso.

nombrar los distintos elementos mencionados. Para el caso de este texto, se utilizarán las siguientes reglas o recomendaciones para escribir identificadores:

- Usar identificadores fáciles de recordar y relacionados con la tarea que realizan dentro del programa.
- Usar una letra como primer carácter del identificador.
- No usar caracteres especiales dentro de los identificadores como vocales tildadas, la letra ñ, o símbolos como: $, #, !, ?, entre otros.
- No dejar espacios en blanco dentro del nombre de un identificador.
- No usar palabras propias del lenguaje de programación, conocidas como "Palabras reservadas en Lenguaje C" (Ver Tabla 1.3).

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

Tabla 1.3: Palabras reservadas del Lenguaje C

- Si se desea, se pueden usar varias palabras como identificador, pero unidas. También se puede usar un guión bajo entre cada una de ellas.
- Evitar el uso de artículos y proposiciones, tales como: el, los, la, un, unos, a, para, de, entre otros.
- Los identificadores suelen tener reglas dependiendo del lenguaje, en general, se escriben en minúscula, cuando el identificador se componga de dos o más palabras, la primera letra a partir de la segunda deberá escribirse en mayúsculas.
- Si el identificador corresponde al nombre de una constante, este debe escribirse en mayúsculas.
- Hay que tener en cuenta que Lenguaje C es sensible al tamaño, lo que quiere decir que, un mismo identificador escrito en mayúsculas es distinto al mismo escrito en minúsculas.

1.4. Variables

La memoria principal de un dispositivo de procesamiento de datos o computador, es el lugar donde se almacenan los datos temporalmente.

La memoria se puede comparar con un conjunto de cuadros finitos (ej: n cuadros), y numerados desde cero (dirección de memoria) (Ver Figura 1.4). Cada cuadro tiene asociado un valor y una dirección. Algunas celdas pueden tener asociado un identificador para facilitar el acceso al valor y así evitar usar la dirección de memoria en todos los casos en los que se desee almacenar o recuperar un dato.

Identificador	**Valor en memoria**	**Dirección de la celda**
		0
a	$valor_a$	1
		2
		3
	⋮	
b	$valor_b$	$n-3$
		$n-2$
		$n-1$

Tabla 1.4: Representación gráfica de la memoria

Aclaración:

Una variable corresponde a una posición de memoria usada para almacenar un dato. Su valor puede cambiar en cualquier momento de la ejecución del programa, de allí su nombre.

Con las variables, se hace necesario tener en cuenta tres aspectos, a saber:

- Tipo de dato (depende del dato a almacenar)
- Nombre o identificador
- Valor o contenido

Importante recordar que Lenguaje C no tiene un tipo especial para las cadenas y los datos lógicos. Las cadenas deben ser trabajadas como un conjunto de `char` y los datos lógicos como elementos de tipo `int` o `char`, o emplear el tipo `bool` declarado en la biblioteca `stdbool.h`.

El nombre o identificador de la variable, da el nombre o identifica la posición de la memoria a la que se hace referencia.

Contenido, es el valor que almacena, por ejemplo, 20, “Gabriela”, 45579.35, etc.

Declaración de variables

Los programas en Lenguaje C, requieren de la declaración de variables para almacenar los datos. Declarar una variable significa separar un espacio de memoria, usando un nombre (identificador) y un tipo de dato.

La forma general para declarar variables es la siguiente:

```
Tipo variable;
```

Ejemplos:

```
char   categoria;
float  precioProducto;
float  salario;
int    edad;
char   estratoEconomico;
bool   esFumador;
```

Analizando las anteriores declaraciones, se puede observar y concluir lo siguiente:

- Se declararon 6 variables: `categoria`, `precioProducto`, `salario`, `edad`, `estratoEconomico` y `esFumador`.
- Con estas declaraciones se le indica al equipo de procesamiento de datos o computador que debe reservar 6 espacios en su memoria y a cada uno asignarle el respectivo nombre que se definió.
- Cada uno de estos identificadores representa una variable.
- En la asignación de nombres se tuvieron en cuenta las reglas vistas anteriormente al escribir los identificadores. Por ejemplo, no se usaron tildes en las variables `categoria` y `estratoEconomico`; en el identificador de esta última variable que está compuesta por dos palabras, la segunda inicia con letra mayúscula, y no se dejó espacio entre palabras.
- En cada una de las declaraciones de variables se está determinando el tipo de dato que almacenarán.
- En el Lenguaje C, el valor inicial de todas las variables no se conoce previamente y se denomina como “Información basura”, es decir,

toda variable declarada tiene un valor desconocido (arbitrario) y el diseñador del algoritmo debe asegurarse de asignarle un valor antes de intentar usar el contenido de la variable. Existen otros lenguajes de programación, como por ejemplo, Java que sí le asigna un valor por defecto a toda variable recién declarada dentro de los atributos de una clase.

Si al declarar variables se tienen varias del mismo tipo, pueden agruparse en una sola declaración, separándolas con comas. Ejemplo:

```
float  precioProducto, salario;
```

Actividad 1.1

Según lo trabajado previamente en el libro, realice los siguientes puntos:

1. Imagine que se desea declarar variables para almacenar datos de acuerdo con el enunciado de la primera columna de la Tabla 1.5 que aparece a continuación. Diligencie la segunda y tercera columna, declarando un identificador válido y adecuado para la variable, con su respectivo tipo de dato.

Dato a almacenar	**Variable**	
	Identificador	**Tipo de dato**
Placa de un vehículo		
Tamaño del motor en centímetros cúbicos		
Número de pasajeros		
Número de baños de una casa		
Área de la casa en metros		
Valor del alquiler		
Valor del descuento de un producto		
¿Encendió el computador?		

Tabla 1.5: Identificación de variables y su tipo

2. Para las variables que se declaran en la Tabla 1.6, diga si el identificador utilizado es correcto o incorrecto y justifique.

Variable	**¿Es válido?**	**Justifique**
Nombre medico		
@especialidad		
generoAspirante		
Valor a pagar		
#CEDULA		
tíituloLibro		
Años_de_Experiencia		
esCasado		

Tabla 1.6: Validez de variables

Almacenamiento de un dato en una variable

Esto puede hacerse de dos formas, a saber:

La primera consiste en leer el dato ingresado por el usuario; por ejemplo, suponga que se hace un programa que calcula la velocidad de un vehículo a partir de una distancia y un tiempo; el usuario deberá proporcionar la distancia y el tiempo respectivos. Más adelante en este capítulo, se indicará la forma de leer los datos.

La segunda forma implica el uso de una expresión de asignación. Una expresión de asignación, es el mecanismo por medio del cual una variable o una constante toma un valor. En Lenguaje C, una asignación se hace mediante el signo igual (=).

Su sintaxis o forma general es la siguiente:

```
variable = valor;
```

Lo anterior indica que `valor` será almacenado en `variable`. En este caso, `valor` puede ser una constante, otra variable o una expresión aritmética (cálculo).

De acuerdo con las variables declaradas anteriormente, se procederá a asignar valores en cada una de ellas:

```
categoria        = 'B';
precioProducto   = 45000.00;
salario          = 3000000.00;
edad             = 24;
estratoEconomico = '3';
esFumador        = true;
```

Si se asume por simplicidad que: cada variable ocupa una única celda, que todas las variables fueron creadas de forma consecutiva en la memoria a partir de la posición k arbitraria, y que el valor lógico `true` se almacena como 1; se puede asumir que la memoria queda como se presenta en la Tabla 1.7.

Identificador	**Valor en memoria**	**Dirección de la celda**
	⋮	
categoria	'B'	k
precioProducto	45000.00	$k+1$
salario	3000000.00	$k+2$
edad	24	$k+3$
estratoEconomico	'3'	$k+4$
esFumador	1	$k+5$
	⋮	

Tabla 1.7: Representación gráfica de la memoria - Variables

Aunque los supuestos no son del todo ciertos, sí son válidos para ilustrar el concepto de las variables en la memoria de forma simplificada. A partir de este momento, se ignoran las direcciones de memoria hasta el capítulo de funciones, en el cual se introduce el concepto de puntero. Un puntero no es más que una variable que almacena direcciones de memoria y por medio de él, es posible modificar otras variables o celdas de memoria (Ver capítulo sobre funciones para mayor información).

Los valores ubicados al lado derecho del igual (=), son asignados a cada una de las variables que están del lado izquierdo, lo que significa que, los espacios de memoria almacenarán temporalmente los valores.

Observe que, en el caso de `estratoEconomico` el valor asignado está entre comillas simples, debido a que es una variable de tipo `char`.

Como ya se mencionó, en la asignación pueden usarse valores provenientes de otras variables o de expresiones aritméticas. Ejemplo:

```
1  a = 5;
2  b = a;
3  c = 3 * b;
4  a = c - b;
```

- Línea 1: la variable `a` toma el valor de 5, que es un valor constante.
- Línea 2: la variable `b` toma una copia del valor contenido en `a`, o sea, tanto `a` como `b` contienen el valor de 5. Aqui la asignación se hace tomando el valor de otra variable. Cuando se hace referencia al nombre de la variable, realmente se está referenciando su contenido.
- Línea 3: la variable `c` toma el valor de 15, al multiplicar el valor de `b` que es 5, por 3. En este caso la asignación proviene de una expresión aritmética.
- Línea 4: la asignación almacena en la variable `a` el valor de 10, puesto que `c` tiene almacenado el valor de 15 y le es restado el valor de `b`, que en este caso es 5. Esta línea muestra una asignación realizada como resultado de una expresión aritmética. Así mismo se comprueba que una variable puede cambiar de valor; en la línea 1 la variable `a` tenía el valor de 5 y al finalizar, en la línea 4, esa misma variable termina con un valor de 10. Cada que una variable cambia su valor, el dato anterior que almacenaba, se pierde.

1.5. Constantes

A diferencia de una variable, una constante se refiere a un espacio de memoria que almacena un dato que permanece constante durante la ejecución de todo el programa. Las constantes, como las variables también se declaran.

La declaración de una constante tiene la siguiente forma general:

```
const Tipo IDENTIFICADOR = Valor;
```

- La palabra `const` es una palabra reservada en Lenguaje C que indica que se va a declarar una constante y que su valor no podrá modificarse durante la ejecución del programa.

- Tipo, se refiere al tipo de dato, por ejemplo a: `char`, `int`, `float`.
- IDENTIFICADOR, es el nombre que se le asigna a la constante. Se recomienda que se escriba en letras mayúsculas.
- `Valor`, es el valor que tendrá la constante y debe estar acorde con el tipo de dato que se le haya asignado a la constante.

Algunos ejemplos son:

```
const float  VALOR_PI = 3.1415926;
const float  DESCUENTO = 0.10;
const int    MAXIMO = 10;
const float  SALARIO_MINIMO = 781242.0;
const char   CATEGORIA_DEFECTO = 'B';
```

Otra forma de declarar constantes en Lenguaje C, consiste en la declaración simbólica con la siguiente expresión:

```
#define <identificador> <secuencia_de_caracteres>
```

Por ejemplo,

```
#define PI 3.141592
#define NUMERO_E 2.718281
```

Las constantes definidas de esta última forma se denominan "Constantes simbólicas". La diferencia entre una constante `const` y una declarada con `#define` es básicamente que para el compilador, la primera es una variable a la que no se le puede modificar su contenido (constante); mientras que la segunda son elementos que el compilador debe reemplazar por sus respectivos valores, antes de procesar el programa.

1.6. Operadores y expresiones

Los operadores básicos permiten realizar operaciones con números o con datos almacenados en las variables y constantes. En programación, se conocen 3 tipos de operadores: aritméticos, relacionales y lógicos.

De otro lado, una expresión es una instrucción que contiene operadores, variables, constantes y números y que generalmente produce un resultado, ya sea numérico o lógico.

Las expresiones deben escribirse en forma lineal (una sola línea), de acuerdo a la siguiente estructura:

```
Operando1 Operador Operando2
```

`Operando1` y `Operando2` son los datos o valores que intervienen en la expresión y, `operador` es el símbolo que especifica el tipo de operación a aplicar entre los operandos. Dentro de los operadores aritméticos existe una notación especial, que permiten escribir las expresiones de forma abreviada, más adelante se darán los respectivos ejemplos.

1.6.1 Operadores aritméticos

Se usan con el fin de hacer operaciones aritméticas entre datos de tipo entero, real e incluso carácter; al final el resultado es un número (así sea el código de un carácter). En Lenguaje C , se tienen los siguientes operadores aritméticos:

- + Suma
- - Resta
- * Multiplicación o producto
- / División real o entera
- % Módulo o Resto de la división entera

La suma, resta, multiplicación y división son operaciones aritméticas básicas. Para Lenguaje C, el resultado de una división depende de los operandos involucrados, si ambos operandos son enteros, el resultado será entero, de lo contrario, es decir, si hay un operando real, el resultado será real.

Por ejemplo, si se divide 7 en 2, el resultado será 3, ya que ambos valores son enteros.

Por el contrario, si uno de los dos operandos fuera de tipo real, por ejemplo, el 2.0, el resultado será 3.5.

En conclusión, si uno o ambos operandos son de tipo real, el resultado que se obtiene es del mismo tipo; si ambos operandos son enteros, la división genera un resultado entero:

$17.0/2.0 = 8.5$, ambos operandos reales, resultado real.

$13/2 = 6$, ambos operandos enteros, resultado entero (sin decimales).

Por su parte, el operador % (residuo, resto o módulo de la división entera), se trata de un operador de división entre datos de tipo entero,

que no obtiene el cociente, sino el residuo o resto de la división entera, de la siguiente forma:

$$\begin{array}{r|l} 1\ 5 & 2 \\ \hline 1 & 7 \end{array}$$

Para usarlo en un programa en Lenguaje C, se debe escribir así: 15 % 2, obteniendo 1 como resultado.

Aclaración:

Al utilizar / o %, el operando de la derecha, no debe tener el valor de 0, ya que genera un error, debido a que la división entre cero, matemáticamente, no está definida.

El operador % (Módulo o Resto de la división) solo se usa con datos de tipo `int`.

El operador / puede usarse con datos enteros o reales. Si los datos son enteros, el resultado es entero, si alguno de los datos es real, el resultado será del mismo tipo (real).

1.6.2 Operadores relacionales

Son operadores usados para escribir expresiones de comparación, también llamadas relacionales, que producen un resultado lógico o booleano, es decir, verdadero o falso. Recuerde que para el Lenguaje C, el valor verdadero es equivalente a 1, mientras que falso equivale a 0.

Los operadores relacionales en Lenguaje C son:

- < Menor que.
- > Mayor que.
- <= Menor o igual a.
- >= Mayor o igual a.
- ! = Diferente de.
- == Igual a.

Aclaración:

Un solo igual (=) realiza una asignación de valores en una variable o constante (valor inicial). El doble igual (==) permite realizar comparaciones entre dos operandos.

1.6.3 Operadores lógicos

Esta clase de operadores permiten escribir expresiones lógicas o booleanas que generan resultados de tipo lógico: verdadero o falso.

Los operadores lógicos son los que aparecen a continuación:

- `&&` Conjunción.
- `||` Disyunción.
- `!` Negación.

Para obtener el resultado de la aplicación de estos operadores, es indispensable conocer cómo funcionan las tablas de verdad de cada uno de ellos.

Operador `&&`, llamado **Conjunción**. Es un operador binario, esto significa que requiere de dos operandos para producir un resultado. El resultado es verdadero solo cuando ambos operandos son verdaderos (Ver Tabla 1.8).

*Operando*1	*Operando*2	*Operando*1 `&&` *Operando*2
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

Tabla 1.8: Tabla de verdad del operador `&&`

Operador `||`, llamado **Disyunción**. También es un operador binario. Su resultado será verdadero cuando al menos uno de los dos operandos tenga ese valor (Ver Tabla 1.9).

*Operando*1	*Operando*2	*Operando*1 `||` *Operando*2
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

Tabla 1.9: Tabla de verdad del operador `||`

Operador `!`, llamado **Negación**. Este es un operador unario, es decir, que requiere de un solo operando para generar su resultado. Si el operando es verdadero, cambia su estado a falso, y viceversa. En otras palabras, su trabajo consiste en cambiar el valor o estado lógico de su único operando (Ver Tabla 1.10).

*Operando*1	`!` *Operando*1
verdadero	falso
falso	verdadero

Tabla 1.10: Tabla de verdad del operador `!`

Aclaración:

Usando el operador `&&`, el resultado es verdadero solo si ambos operandos son verdaderos.

Usando el operador `||`, el resultado es verdadero si al menos uno de los operandos es verdadero.

El operador `!`, invierte el estado del operando, es decir, si el operando es verdadero, el resultado será falso y viceversa.

1.6.4 Expresiones aritméticas

Este tipo de expresiones contiene variables, constantes, números y operadores aritméticos, así como los paréntesis. Después de calcularse, la expresión genera un resultado de tipo numérico.

La escritura de una expresión aritmética en Lenguaje C sigue la siguiente forma general:

```
    Operando1 Operador Operando2
```

Aquí, `Operador`, es uno de los operadores aritméticos vistos en el numeral 1.6.1. A continuación, algunos ejemplos de expresiones aritméticas se pueden ver en la Tabla 1.11.

Ejemplo	Explicación	Resultado
3 + 5	Suma 3 con 5	8
2 - 8	Resta 8 de 2	-6
7 * 6	Multiplica 7 por 6	42
20.0 / 3.0	Divide 20.0 entre 3.0 (Real)	6.33
20 / 3	Divide 20 entre 3 (Entero)	6
20 % 3	Resto de 20 entre 3	2

Tabla 1.11: Ejemplos de expresiones aritméticas

Las expresiones aritméticas pueden involucrar también variables y/o constantes. Por ejemplo: Tabla 1.12.

Expresión	Explicación
$lado1 + lado2$	Se suman los valores almacenados en las variables *lado*1 y *lado*2.
`a - b`	Al valor de la variable `a` se le resta el valor de la variable `b`.
$base * altura$	Los valores almacenados en las variables *base* y *altura* son multiplicados entre sí.
$a \ / \ b$	El valor almacenado en la variable *a* es dividido entre el valor de la variable *b*. El resultado será entero, si ambas variables fueron declaradas de tipo `int`, en otro caso, será `float` (con decimales).
$25 \,\%\, divisor$	25 es dividido entre el valor de la variable *divisor*, el resultado será el resto de la división. La variable *divisor* deberá ser declarada de tipo `int`.

Tabla 1.12: Ejemplos de expresiones aritméticas con variables

Ahora es importante aclarar que, si una expresión aritmética contiene varios operadores, se requiere aplicar lo que se conoce como la precedencia o jerarquía de operadores, que especifica la prioridad en que deben realizarse las operaciones. La Tabla 1.13 muestra el orden de ejecución, que puede modificarse con el uso de paréntesis, que, en caso de usarse, pasarían a ser la máxima prioridad.

Orden	Operador
1	( )
2	*, /, %
3	+, -

Tabla 1.13: Prioridad de los operadores

Ejemplos:

```
10 + 3 * 5
```

Aquí hay dos operaciones: una suma y una multiplicación. Primero se hace el producto $*$, lo que genera la siguiente expresión:

```
 10 + 15
```

Posteriormente se realiza la suma, dando como resultado `25`.

Si en el anterior ejemplo, no se hubiese tenido en cuenta el orden de ejecución, de acuerdo a la prioridad de operación, sino que se hubiese ejecutado en el orden como está escrita la expresión, se obtendría el siguiente resultado, que obviamente es un **error**:

```
10 +  3 * 5
13 * 5
65
```

Si en una expresión se encuentran `2` o más operadores consecutivos de igual jerarquía o precedencia, las operaciones se van realizando de izquierda a derecha. Por ejemplo:

```
10 - 6 + 2
4 + 2
6
```

En la anterior expresión se encuentra una resta y una suma, ambas son de igual precedencia, por lo tanto, se ejecutan de izquierda a derecha, primero la resta y luego la suma.

Si en algún momento se hace necesario cambiar el orden de ejecución de algún operador, se deben utilizar paréntesis. Por ejemplo, si en la expresión anterior se quería realizar primero la suma y luego su resultado se restase con `10`, la expresión debería escribirse así:

```
10 - (6 + 2)
```

Esta expresión se calcularía:

```
10 - (6 + 2)
10 - 8
2
```

Analice ahora la siguiente expresión, asumiendo que `a = 17` y `b = 20`.

```
a % 3 + b
```

El valor de la variable `a` es dividido entre `3`, el residuo de esta división se suma con el dato contenido en la variable `b`. Entonces, 17 al dividirse entre 3 dará como residuo `2`, que se suma con el valor que hay en `b`, o sea `20`, dando un resultado final de `22`.

Ahora, suponga que la expresión se escribe de la siguiente forma:

```
a % (3 + b)
```

El resultado obtenido es distinto. Observe:

```
17 % (3 + 20)
17 % 23
17
```

El residuo es 17:

$$\begin{array}{r|l} 1\ 7 & 2\ 3 \\ \hline 1\ 7 & 0 \end{array}$$

Es importante mencionar que, si se tienen varios juegos de paréntesis en una expresión, estos se solucionan de izquierda a derecha, aclarando que, si hay paréntesis internos, estos se solucionan primero. Observe la expresión que aparece a continuación, en ella, se pueden identificar varios paréntesis, incluyendo el empleado para definir la operación potencia (`pow`), la cual será trabajada con mayor profundidad en la siguiente sección.

```
((4 + 8) * 3) / (pow(7, 2) % (2 + 1))
```

En esta expresión, lo primero será resolver los paréntesis antes de llevar a cabo la división que los separa; ya que hay dos juegos de paréntesis, se deben resolver de izquierda a derecha.

```
((4 + 8) * 3) / ( pow(7, 2) % (2 + 1))
```

Ahora, en el primer juego de paréntesis hay una suma y un producto; la suma está encerrada en paréntesis, lo que le dá la prioridad antes de llevar a cabo el producto:

```
((4 + 8) * 3) / (pow(7, 2) % (2 + 1))
(12 * 3) / (pow(7, 2) % (2 + 1))
```

Después de sumar ya se puede hacer la multiplicación dentro del paréntesis del lado izquierdo de la división, obteniendo el siguiente resultado:

```
36 / (pow(7, 2) % (2 + 1))
```

A continuación, se resuelve la expresión que está entre el paréntesis del lado derecho de la división. Al interior de este paréntesis hay una operación de potencia (`pow`), una división modular y una suma. De acuerdo con la jerarquía de los operadores aritméticos (Tabla 1.13) primero se resuelven los paréntesis, que para el caso es la potencia, luego la suma (dentro del paréntesis) y finalmente el módulo:

```
36 / (pow(7, 2) % (2 + 1))
```

La expresión queda así:

```
36 / (49 % (2 + 1))
```

En ella, lo siguiente a calcular será la suma. Al resolverla, la expresión se presenta así:

```
36 / (49 % 3)
```

Posteriorente, se resuelve la división modular:

```
36 / 1
```

Finalmente se calcula la división, lo que da como resultado `36`.

Para poder que un programa en Lenguaje C haga uso de funciones matemáticas como la potencia, debe incluir en su cabecera la biblioteca `math.c`.

1.6.5 Conversión de fórmulas aritméticas en notación algorítmica

Las fórmulas matemáticas deben escribirse adecuadamente para que el Lenguaje C las pueda interpretar correctamente y realizar los cálculos.

Generalmente, las fórmulas se encuentran en forma de notación matemática, como la siguiente:

$$a = \frac{2x + y}{x - y}$$

Sin embargo, para trabajarla en un programa en Lenguaje C, debe reescribirse en notación algorítmica (una sola línea), así:

```
a = (2 * x + y) / (x - y);
```

Esta transformación requiere tener en cuenta las siguientes reglas:

- **Primera regla**. Si la fórmula contiene una división y el dividendo y/o el divisor tiene una operación, esta operación se encierra entre paréntesis.

  Ejemplo:

  $x = \frac{a+1}{b}$, debe expresarse así: $x = (a + 1)/b$

- **Segunda regla**. Si la fórmula incluye raíces, estas se expresan como potencias fraccionarias, de la siguiente forma:

  $$\sqrt[n]{x} = \texttt{pow}(x, 1.0/n)$$

  Ejemplo:

  $\sqrt[3]{\sqrt[5]{x}}$,debe expresarse así: `pow( pow(` x, $(1.0/5.0)$), $(1.0/3.0)$)

Dentro de la biblioteca `math.h` existe una función llamada `sqrt` que permite calcular raíces cuadradas para números de tipo `double`; `sqrtf` números de tipo `float`; o `sqrtl` números de tipo `long double`. Algo análogo sucede con la función de potencia.

Ejemplo:

$\sqrt{a}$,debe expresarse así: `sqrt`(a) o si se prefiere `pow`(a, $1.0/2.0$)

1.6.6 Expresiones relacionales

Estas expresiones contienen operadores relacionales, además de variables, números y constantes y se usan para comparar el valor de sus operandos, arrojando un resultado de tipo lógico: verdadero o falso. Los operadores relacionales ya se analizaron en el numeral 1.6.2.

La Tabla 1.14 presenta ejemplos de expresiones relacionales.

Expresión	Resultado	Explicación
$5 < 9$	verdadero	Compara si 5 es menor que 9.
$2 > 8$	falso	Compara si 2 es mayor que 8.
$7 <= 7$	verdadero	Compara si 7 es menor o igual a 7.
$2 >= 9$	falso	Compara si 2 es mayor o igual a 9.
$4 \; != \; 3$	verdadero	Compara si 4 es diferente de 3.
$20 == 20$	verdadero	Compara si 20 es igual a 20.

Tabla 1.14: Ejemplo de expresiones relacionales

Es fundamental entender que, los operandos que hacen parte de una expresión relacional, pueden contener constantes, variables o la combinación de ambos. Ejemplos:

```
definitiva >= 3.0
```

En esta expresión se pregunta si el valor contenido en la variable `definitiva` es mayor o igual a `3.0`. La expresión entrega como resultado un verdadero o un falso, dependiendo de lo almacenado en `definitiva`.

Ahora, analice esta expresión:

```
(2 * a + b) > (3 * c - d)
```

Observe que en ambos lados del operador > hay expresiones aritméticas que involucran variables y números. Aquí, también se utiliza la prioridad en las operaciones. Recuerde que, el resultado de esta expresión será verdadero o falso, dependiendo de si lo calculado en el primer juego de paréntesis es mayor a lo calculado en el segundo.

1.6.7 Expresiones lógicas

Las expresiones lógicas permiten crear condiciones más complejas. Estas expresiones contienen operadores relacionales, números, variables que

forman expresiones que a su vez se conectan con otras expresiones por medio de operadores lógicos (estudiados en el numeral 1.6.3), generando resultados de tipo lógico, es decir, verdadero o falso.

A continuación, algunos ejemplos de este tipo de expresiones.

Ejemplo del operador `&&`: para realizar cualquier transacción en un cajero automático se debe poseer una tarjeta (débito o crédito) y tener la clave de dicha tarjeta. Si bien es posible hacer transacciones de otras formas, para ilustrar este operador se tomará solo este caso.

Observe en qué casos se podría hacer alguna transacción y en cuales no:

1. Si se tiene la tarjeta y su clave, se pueden hacer transacciones.
2. Si tiene la tarjeta, pero no la clave, no se pueden realizar transacciones.
3. Si no tiene la tarjeta y si tiene la clave, no se pueden hacer transacciones.
4. Si no posee la tarjeta, y tampoco tiene la clave, no se pueden realizar transacciones.

Antes que nada, recuerde que para usar el operador `&&`, debe tener la forma general de una expresión:

```
Operando1 Operador Operando2
```

Conforme al ejemplo anterior, el `operando1` estará representado por la `tarjeta de crédito` y el `operando2` será la `clave` de dicha tarjeta, el `operador` será la conjunción (`&&`).

Los resultados de falso o verdadero en cada operando estarán dados de la siguiente forma: si se tiene la `tarjeta` el valor será verdadero, en caso contrario será falso; igual sucede con la `clave`. (Ver Tabla 1.15).

tarjeta	**clave**	**tarjeta && clave**	**Explicación**
verdadero	verdadero	verdadero	Si hay transacción
verdadero	falso	falso	No hay transacción
falso	verdadero	falso	No hay transacción
falso	falso	falso	No hay transacción

Tabla 1.15: Tabla de verdad del operador **&&** - Ejemplo

En conclusión, solo se puede llevar a cabo la transacción si se cumplen las dos condiciones, tener la `tarjeta` y la `clave`, lo cual dará un resultado verdadero.

Ejemplo del operador `||`: en el próximo cumpleaños de una sobrinita, ella quiere que le regalen una mascota, dice que estaría feliz si recibe un gato o un perro, o ambos; pero que si no recibe la mascota sería infeliz en su día.

De acuerdo con lo anterior, se analiza en qué casos la sobrina estaría feliz y en cuáles no:

1. Si se le regala tanto el perro como el gato, estaría feliz.
2. Si se le regala solo el perro, estaría feliz.
3. Si recibe solo el gato, estaría feliz.
4. Pero si la niña no recibe ni el gato ni el perro, estaría infeliz todo el día.

La forma general de una expresión, usando el operador `||`, sería la siguiente:

```
Operando1 Operador Operando2
```

Basado en lo anterior, el `operando1` correspondería al `perro` y el `operando2` sería el `gato`, el operador ubicado en el medio sería la disyunción (`||`).

Los resultados de verdadero o falso en el operando1 dependen de si recibe el `perro` o no lo recibe; para el operando2, será verdadero o falso, dependiendo si recibe o no un `gato` (Ver Tabla 1.16).

En conclusión, cuando se use el operador `||`, la expresión se evaluará como verdadera con solo cumplir una de las condiciones.

perro	gato	perro || gato	Explicación
verdadero	verdadero	verdadero	La sobrina está feliz
verdadero	falso	verdadero	La sobrina está feliz
falso	verdadero	verdadero	La sobrina está feliz
falso	falso	falso	La sobrina no está feliz

Tabla 1.16: Tabla de verdad del operador || - Ejemplo

Ejemplo del operador !: este operador cambia el estado lógico de su único operando de verdadero a falso y viceversa.

```
!(4 < 12)
```

Esta expresión se debe leer: es falso que 4 sea menor que 12.

El resultado que arroja esta expresión es falso. La expresión relacional (4 < 12) arroja como resultado un verdadero, al ser negada con el operador !, cambia su estado de verdadero a falso.

1.6.8 Prioridad de operación

Las expresiones lógicas pueden contener expresiones relacionales. Cuando se están evaluando, al igual que con los operadores aritméticos, debe tenerse en cuenta la precedencia do operadores (La tabla 1.17 contiene el orden de ejecución de todos los operadores, incluyendo los paréntesis).

Orden	Operador
1	( ) incluyendo las funciones matemáticas
2	+, - (signo), ++, --, ! (Negación)
3	*, /, %
4	+, -
5	<, <=, >, >=
6	==, !=
7	&& (Conjunción)
8	|| (Disyunción)
9	=, +=, -=, *=, /=, ...(asignación)

Tabla 1.17: Prioridad completa de los operadores

Enseguida se ilustrará la aplicación de la tabla anterior con algunos ejemplos. Suponga la siguiente asignación de valores en las variables:

```
a = 5;
b = 10;
c = 15;
```

¿Cuál sería el resultado de la siguiente expresión lógica?

```
  a  <  b  &&   b <  c
  5  < 10  &&  10 <  15
verdadero && verdadero
      verdadero
```

Primero se evalúan las expresiones relacionales y luego el operador lógico (`&&`). Ambas expresiones relacionales son verdaderas, por lo tanto la expresión lógica es verdadera.

En el caso del ejemplo anterior no se usaron paréntesis, es una buena práctica usarlos para dar mayor claridad a las expresiones, aunque de acuerdo al orden de prioridad, no son requeridos. Esta expresión también puede expresarse así:

```
(a < b) && (b < c)
```

Enseguida se va a evaluar un segundo ejemplo:

```
(20 > 40 && 2 <= 10) || (32 < 50 && 20 <= 20)
```

En esta expresión se tienen dos juegos de paréntesis separados por el operador lógico `||`. Para que la expresión lógica sea verdadera, uno de los juegos de paréntesis deberá tener un resultado verdadero. Sin embargo, cada juego de paréntesis tiene en su interior un operador `&&` que implica que ambas expresiones dentro de cada paréntesis deben ser verdaderas para entregar un resultado verdadero. La evaluación, de acuerdo a la prioridad, es la siguiente:

Primero se evalúa el paréntesis ubicado al principio de la expresión:

```
(20 > 40 && 2 <= 10) || (32 < 50 && 20 <= 20)
```

Dentro de él, lo primero es evaluar los operadores relacionales:

```
(Falso && 2 <= 10) || (32 < 50 && 20 <= 20)
```

Ahora, sabiendo que la expresión lógica está conectada por el operador `&&`, no es necesario evaluar la expresión relacional `(2 <= 10)`, porque ya se sabe que el resultado será falso. No obstante, la explicación se hará para facilitar la comprensión:

```
(falso && verdadero) || (32 < 50 && 20 <= 20)
```

La primera expresión lógica arroja un resultado falso:

```
falso || (32 < 50 && 20 <= 20)
```

Ahora se evaluará la parte a la derecha del operador `||`.

```
falso || (32 < 50 && 20 <= 20)
falso || (verdadero && 20 <= 20)
falso || (verdadero && verdadero)
falso || verdadero
```

En conclusión, la expresión arroja un resultado verdadero.

Las expresiones lógicas también son muy útiles para definir intervalos:

```
(edad >= 18) && (edad <= 24)
```

Esta expresión lógica será verdadera, si el dato almacenado en la variable `edad` es mayor o igual a `18` y menor o igual a `24`. La expresión será falsa en caso contrario.

Entre los operadores existen algunos especiales: ++, −−, +=, −=, ⋆=, /=, entre otros. Todos ellos son formas simplificadas del uso de otros operadores, algunos ejemplos se puede ver en la Tabla 1.18.

Operador especial	**Equivalencia**
`i++;`	`i = i + 1;`
`++i;`	`i = i + 1;`
`i--;`	`i = i - 1;`
`--i;`	`i = i - 1;`
`i+=5;`	`i = i + 5;`
`i*=2;`	`i = i * 2;`
`i/=8;`	`i = i * 8;`
...	
`i-=5;`	`i = i - 5;`
`j= i++ * 5;`	`j = i * 5;`
	`i = i + 1;`
...	
`j= ++i * 5;`	`i = i + 1;`
	`j = i * 5;`

Tabla 1.18: Operadores especiales

Bien usados, estos operadores ayudan a escribir expresiones cortas, no obstante, si se abusa de ellas, pueden provocar expresiones difíciles de

comprender. Como una recomendación, invierta más tiempo en escribir las expresiones, para ganarlo, en el momento de comprenderlas; además, así disminuye la posibilidad de cometer errores que son difíciles de detectar.

Para ilustrar esto, analice el siguiente fragmento de código y determine el valor final de las variables allí utilizadas.

```
i = 2;
j = 4;
k = 8;
p = ++i * j-- + 2 * --k;
```

Después de ejecutar estas líneas, el resultado final sería:
`i = 3`, `j = 3`, `k = 7` y `p = 26`.

Un código equivalente se puede ver a continuación:

```
i = 2;
j = 4;
k = 8;

i = i + 1;
k = k - 1;
p = i * j + 2 * k;
j = j - 1;
```

Como se puede observar, la primera versión es más corta, pero requiere de más tiempo para comprender su funcionamiento; la segunda versión a pesar de ser más larga, es más fácil de comprender.

Actividad 1.2

1. Describa, de acuerdo a la precedencia de los operadores, en cuál orden se ejecutan las siguientes expresiones:

   a) `resultado = PI * pow(radio, 2);`

   b) `resultado = 2 * a + 3 * b - c;`

   c) `resultado = 2 * ( a + 3 ) * b - c;`

   d) `resultado = pow(a, 2) - b * pow(36, 1.0/2.0);`

   e) `resultado = ( a + b ) / ( 2 * c + 1 - a%3);`

2. Resuelva paso a paso las siguientes expresiones, teniendo en cuenta la prioridad de ejecución de los operadores.

   a) `3 * (3 + 4) * (5 - 2);`

   b) `pow(pow(3,(2 + 3)) - 1), (1.0/2.0));`

   c) `5.0/2.0 * 3 + pow(4, 3) * 2/(5.0 + 2.0) - 2;`

   d) `10 * (7 + 7)% (9 + 2)/10;`

   e) `3% 2 - (2 + 2) / 1 * (3 + 1) + 3;`

   f) `(5 + 8*2 - 3.0/5.0)/(4*1 - 2.0/3.0 + 8/2);`

   g) `(4 + 8)/(4 /(1 + 1)) - (3+2+1)/(pow(5,2)+4);`

3. Suponga que se requieren las variables: a, b, c, d de tipo entero.

   a) Declare estas variables.

   b) Asígnele un número arbitrario entre 10 y 20 a cada variable.

   c) Evalúe las siguientes expresiones, teniendo en cuenta los valores asignados:

   - `a = 3 * b + d%5;`
   - `d = (4 * a/2 - 3*c) / (4 + b%3);`
   - `c = 3 * pow(b,2) - 5 * c/3;`
   - `d = 2 * a + 3 * b + 4 * c/d;`
   - `a = 3 * a;`
   - `b = 7 + a;`
   - `c = b - a;`
   - `d = c + a - b;`

4. Se tienen las variables: a, b, c y d de tipo entero. A partir de las asignaciones que se dan enseguida, determine si las siguientes expresiones entregan como resultado un valor verdadero o falso:

```
a = 5;
b = 4;
c = 7;
d = 3;
```

   a) `3 * a <= 15`

   b) `c - d == a + 2`

   c) `(5 * a + 3 * b) >= (pow(c, d) - 35)`

   d) `(5% c + 3 * b/d) < (pow(c, d)/3)`

e) `(32%(4*c)+3*(b+d)) < pow(c,(1+a%2))`

f) `(5 * d + b != pow(4,d)) || (c + d >= a/b)`

g) `(c * a + 10 > b * d - 6) && ( (b + c)% 5 <a)`

h) `(c*(a + 10) > b*(8*d-6)) && ((b + c)% 5 < a)`

i) `((a+b+c)/d != 12) ||(4*c + 5 != 3 * a - d)`

j) `(c*a+7 > b*d-3) && (pow(b+c,1/2) < pow(a,3/2))`

k) `!(a == b)`

l) `!(a * b + c != d)`

5. Escriba en notación algorítmica las siguientes expresiones matemáticas:

a) $x = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

b) $x = \dfrac{3a + 4b}{5 + c} + \dfrac{8c - 10}{3a + b}$

c) $x = \dfrac{3a\sqrt{4b + 8}}{\frac{4b+7}{10+\sqrt{4c}}} + \sqrt[3]{8d}$

d) $x = \dfrac{5ab^7}{3b + a} + 2ac^{3/5}$

e) $x = \dfrac{3a + \frac{2b+4}{3c4^3}}{\frac{a+b+c}{\sqrt{a+c}+7}}$

1.7. Diagrama de flujo

Un diagrama de flujo (D.F.) es una forma de representar un algoritmo gráficamente. Un D.F. está compuesto por un conjunto de elementos que permiten representar acciones, decisiones o cálculos que unidos, solucionan un problema específico. Los diagramas de flujo ayudan en la tarea de codificar los programas en un lenguaje de programación como C.

En las Tablas 1.19 y 1.20 se relacionan los gráficos usados para elaborar correctamente los diagramas de flujo.

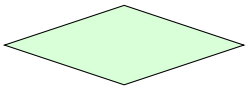
Símbolo	Nombre	Explicación
	**Terminal**	Ilustra el inicio y el final del algoritmo. Se rotula con la palabra Inicio o la palabra Final. En cada algoritmo solo puede haber un inicio y un final.
	**Entrada**	Se usa para la interacción con el entorno. por medio de este símbolo el algoritmo recibe datos. Indica lectura de datos.
	**Proceso**	Indica que el algoritmo realiza una acción.
	**Salida**	Permite la interacción con el entorno. Por medio de este símbolo se muestran resultados. Indica escritura.
	**Decisión**	Ilustra la toma de decisiones. Se rotula con una expresión relacional o lógica, dependiendo de su resultado se toma un camino de ejecución. Se usa en decisiones y condiciones de las instrucciones `while` y `do while`.

Tabla 1.19: Elementos en un diagrama de flujo - Parte 1

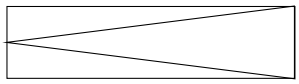
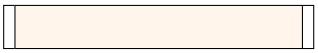
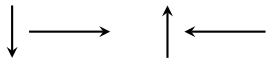
Símbolo	Nombre	Explicación
	**Decisión múltiple**	Se usa para determinar el paso que se ejecutará entre varios posibles a elegir.
	**Estructura for**	Se usa para establecer procesos repetitivos de la estructura `for`.
	**Procedimiento**	Permite dividir el algoritmo en módulos independientes, que realizan una tarea indispensable en el resultado del mismo [Pimiento, 2009].
	**Conector misma página**	Se usa para conectar dos partes del diagrama ubicados en una misma página. Siempre se utilizan en pares, uno de salida y otro de entrada.
	**Conector otra página**	Si el diagrama es extenso, este símbolo conecta dos partes del diagrama que están en páginas distintas. Igual que el anterior, se usa en pares.
	**Líneas de flujo**	Ilstran el orden y dirección en que las instrucciones se deben de ejecutar.

Tabla 1.20: Elementos en un diagrama de flujo - Parte 2

Para la construcción de diagramas de flujo, es importante tener en cuenta los siguientes aspectos:

1.7.1 Terminal

Al construir un diagrama solamente deben aparecer dos de estos símbolos, uno rotulado con la palabra `Inicio` y otro con la palabra `Final` (Ver Figuras 1.2 y 1.3).

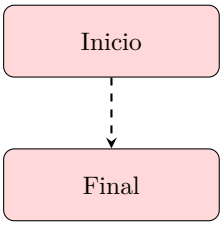


Figura 1.2: Terminal

Del símbolo `Inicio` solo puede salir una única línea de flujo. El símbolo `Final`, recibe una única línea de flujo.

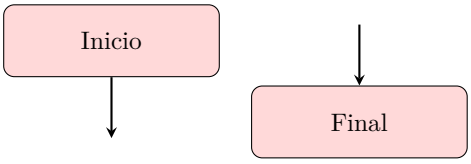


Figura 1.3: Elementos Terminal

Buena práctica:

El gráfico que representa el `Inicio` debe ser el primero en el diagrama, y el del `Final` debe ser la última parte del mismo.

1.7.2 Entrada

Este símbolo recibe y entrega una única línea de flujo (Ver Figura 1.4). Se rotula con el identificador de la variable que recibirá el valor que proporcione el usuario del algoritmo. La variable debe ser uno de los datos disponibles para la solución del problema.

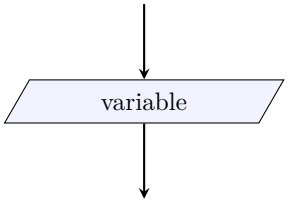


Figura 1.4: Entrada

1.7.3 Proceso

A este símbolo entra y sale una única línea de flujo (Ver Figura 1.5). Se rotula con la instrucción que se vaya a ejecutar, puede ser, por ejemplo, una instrucción de asignación.

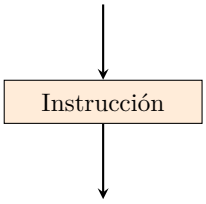


Figura 1.5: Proceso

1.7.4 Salida

A este símbolo entra y sale una única línea de flujo (Ver Figura 1.6). Se usa para mostrar resultados o mensajes. Dentro de él se puede escribir el nombre de una variable o una constante (Figura 1.6(a)), una operación matemática (Figura 1.6(b)), un mensaje con alguna variable o constante (Figura 1.6(c)) o la combinación de varios de estos elementos (Ver Figura 1.6(d)).

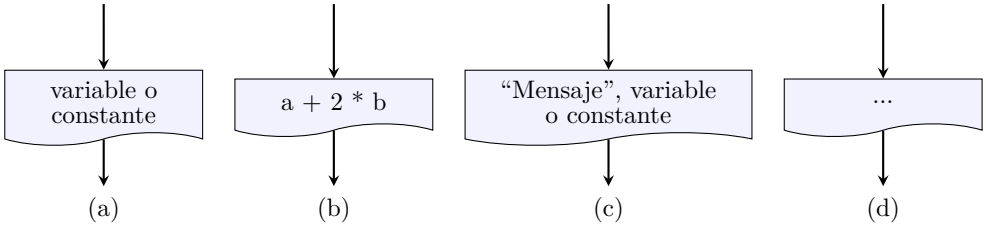


Figura 1.6: Salida

Es crucial tener en cuenta que los mensajes deben escribirse dentro de comillas dobles (“”) y que se mostrarán exactamente como fueron escritos. En el caso de la variable o constante, no llevan las comillas y se mostrará el valor que tengan almacenado. En las operaciones matemáticas, tampoco se usan las comillas y se mostrará el resultado de la operación.

Estos primeros conceptos serán desarrollados en este capítulo y en el Capítulo 2.

1.7.5 Decisión o bifurcación

A este rombo solo debe llegar una única línea de flujo y de él deben salir dos, una que indica el camino verdadero (cuando la condición se cumple) y otra que indica el camino falso, cuando la condición no se cumple. (Ver Figura 1.7).

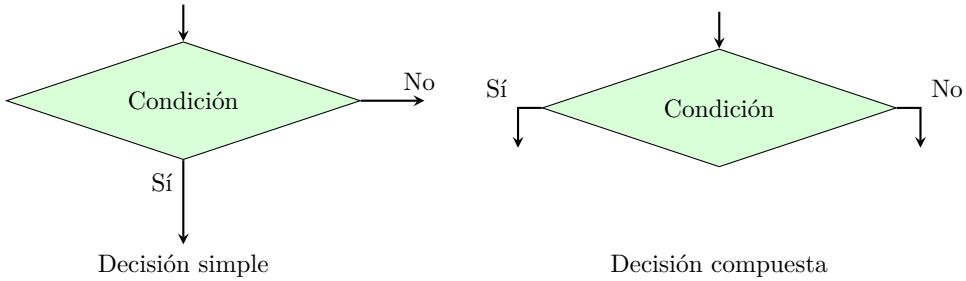


Figura 1.7: Decisiones

El símbolo se rotula con una condición que se representa mediante una expresión relacional o lógica, dependiendo de su resultado se elige uno de dos caminos. Las líneas de flujo que salen de él, deben rotularse con la palabra `Sí` para el camino a seguir cuando la condición es verdadera y con la palabra `No`, para el caso contrario.

Una decisión, representada por el símbolo que se está analizando, tiene varias configuraciones: **simple**, **compuesta** y **anidada**. El rombo también es utilizado para establecer la condición en los procesos repetitivos, que se tratarán más adelante.

- **Decisión simple**: este tipo de decisiones se dá cuando la condición arroja un resultado verdadero y por este flujo se ejecutan una o más instrucciones (asociadas al flujo rotulado con la palabra Si) (`Instrucción-V`) . Si el resultado es falso, no se lleva a cabo ninguna acción que dependa de la condición. Ver Figura 1.8.

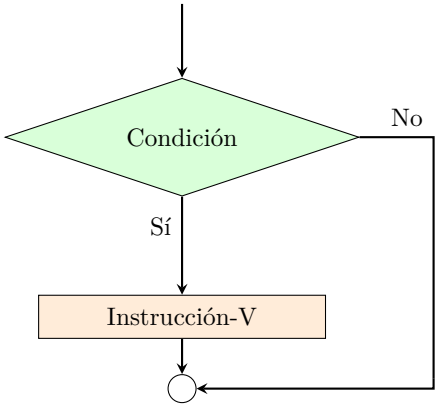


Figura 1.8: Decisión simple

- **Decisión compuesta**: si el resultado de la condición es verdadero se ejecutan una o más instrucciones asociadas al flujo rotulado con la palabra Si (`Instrucción-V`), en caso contrario se deben ejecutar una o más instrucciones asociadas al flujo rotulado con la palabra No (`Instrucción-F`). Ver Figura 1.9.
- **Decisión anidada**: estas decisiones se dan si es necesario tomar otras decisiones, dependiendo del resultado de una decisión anterior. Ver Figuras 1.10 y 1.11.

Aclaración:

Las decisiones anidadas pueden representarse de diversas maneras, la forma de hacerlo, dependerá del programador.

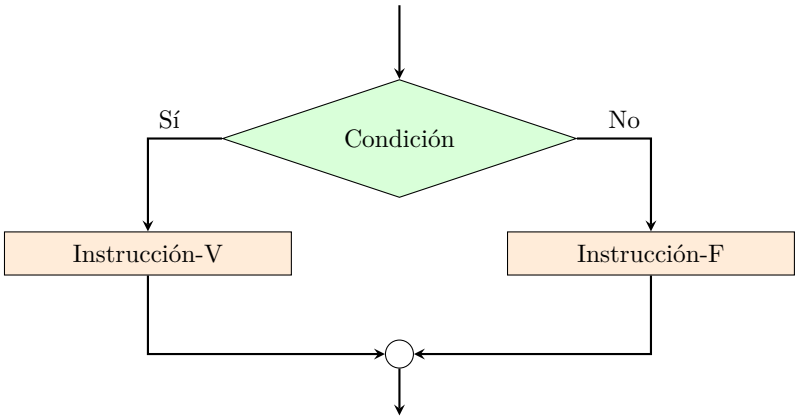


Figura 1.9: Decisión compuesta

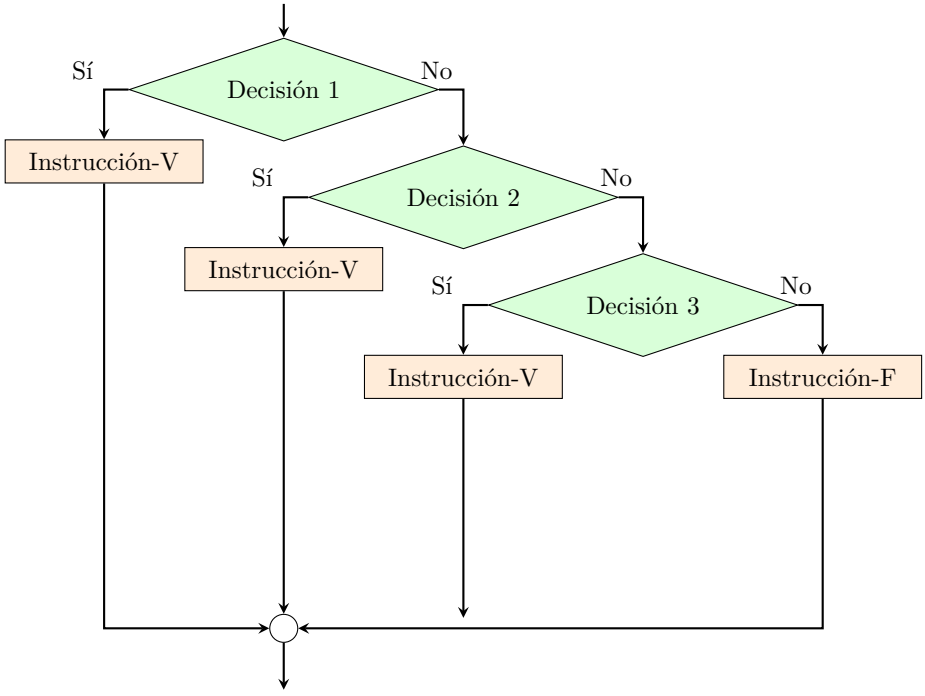


Figura 1.10: Decisión anidada - Ejemplo 1

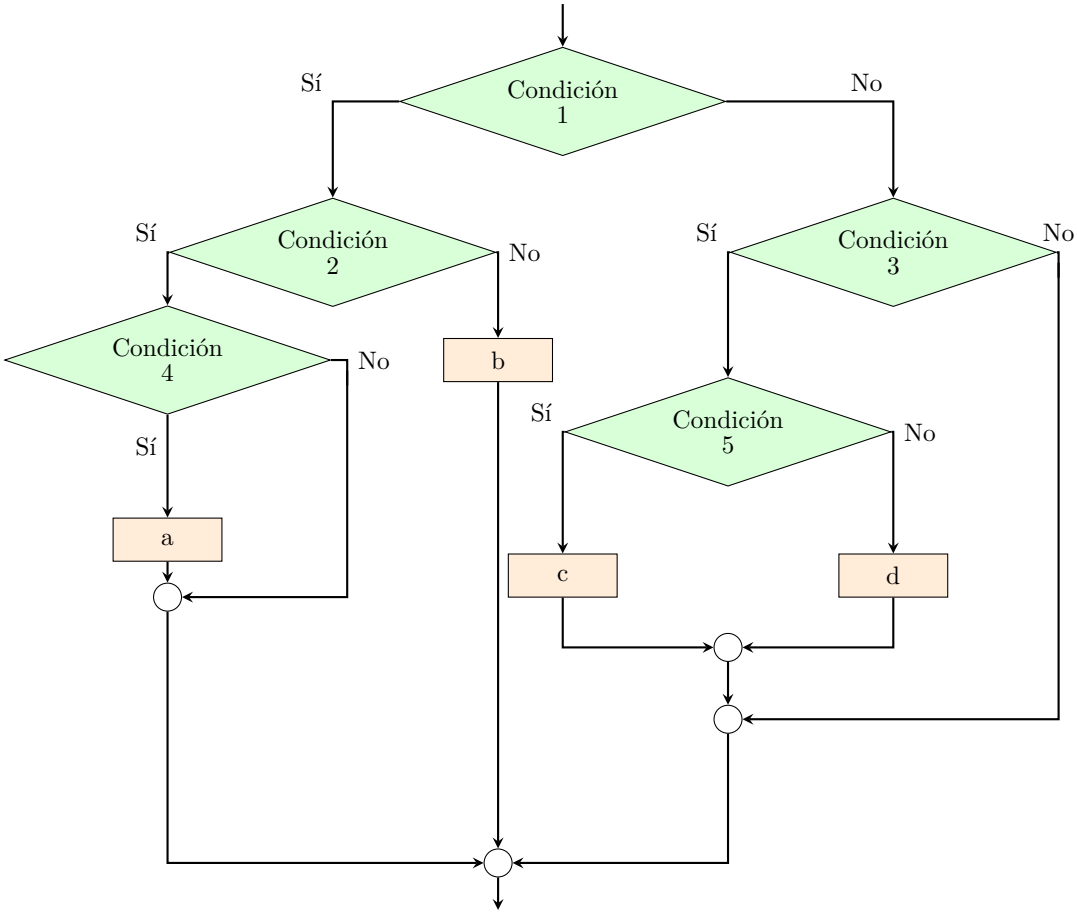


Figura 1.11: Decisión anidada - Ejemplo 2

Buena práctica:

Se recalca que, de todo símbolo de decisión solo debe salir una línea de flujo por la parte verdadera (`Si`) y otra por la parte falsa (`No`)

1.7.6 Selector o decisión múltiple

Con este símbolo se representa la elección de una alternativa entre varias posibles, dependiendo del valor de una variable o expresión que se usa para rotular el símbolo. La alternativa seleccionada puede tener una o varias instrucciones.La alternativa rotulada [En otro caso] indica que si el valor de la variable selector no es igual a ninguno de los valores relacionados (`valor 1`, `valor 2`,..., `valor n`) se ejecuta la instrucción por defecto. Ver Figura 1.12.

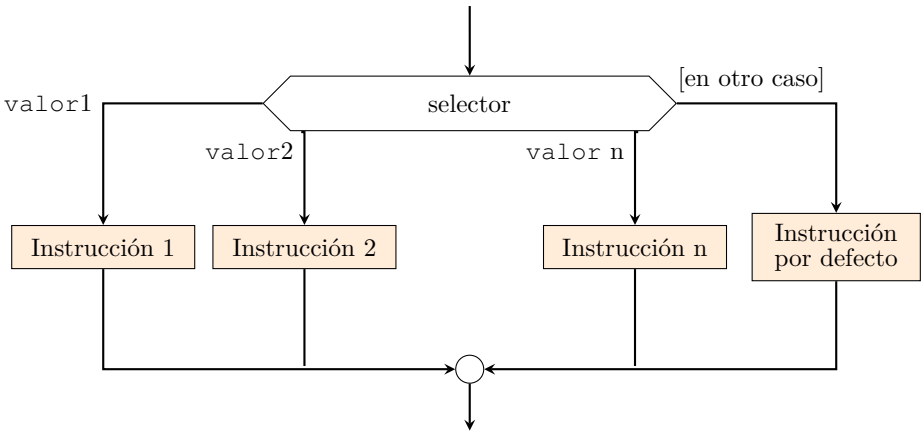


Figura 1.12: Decisión múltiple

Los conceptos de decisiones serán trabajados en el Capítulo 3.

1.7.7 Conector misma página

Como se mencionó antes, los símbolos de misma página conectan elementos de un diagrama que están separados pero en la misma página. Un conector recibe una o varias líneas de flujo y de otro conector sale una línea de flujo. Se deben rotular con una letra o un número, que se repetirá tanto en el conector de salida, como en el de entrada. Con el uso de los conectores se evita el cruce de líneas de flujo o dibujar líneas de flujo demasiado largas.

En un diagrama de flujo pueden haber varios conectores de salida con el mismo rótulo, pero solamente uno de entrada.

En la Figura 1.13 se observan los conectores de misma página.

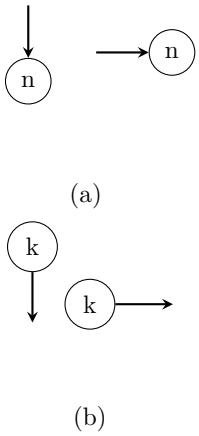


Figura 1.13: Conector misma página

Este símbolo también se puede usar como punto de concentración de varias líneas de flujo, en cuyo caso no se rotula y se usa uno solo. Pueden llegar varios flujos de entrada, pero solamente habrá un flujo de salida.

Conector otra página

Este símbolo posee las mismas características del conector anterior, pero se diferencia en que este se usa para hacer conexiones de partes del diagrama que están dibujadas en páginas distintas. Ver Figura 1.14.

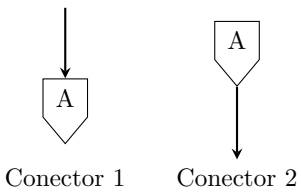


Figura 1.14: Conector otra página

El uso detallado de estos y los demás símbolos, se estudiará a lo largo de los siguientes capítulos.

Luego de estudiar los símbolos de un diagrama de flujo, es fundamental recordar que un algoritmo consta de 3 etapas: Entrada, Proceso y Salida. En un algoritmo expresado a través de un diagrama de flujo, estas etapas se observan como aparecen en la Figura 1.15.

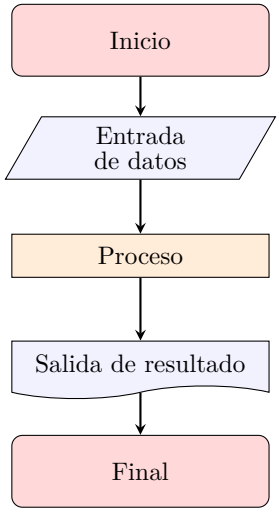


Figura 1.15: Diagrama de flujo (forma general)

Buenas prácticas:

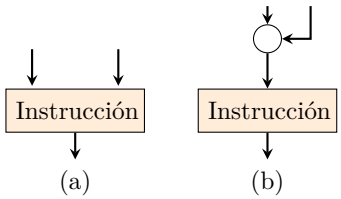
Es bueno seguir las siguientes reglas o recomendaciones al construir diagramas de flujo:
- Colocar un nombre al diagrama que identifique su función.
- En un diagrama de flujo no se declaran variables, ellas simplemente se usan.
- Construir el diagrama de arriba hacia abajo y de izquierda a derecha.
- Ubicar solo un símbolo de inicio y uno de final, ambos rotulados claramente.
- Dibujar solo líneas de flujo en rectas horizontales o verticales. No se deben usar líneas inclinadas.
- Toda línea de flujo debe estar conectada a uno de los símbolos o a otra línea de flujo, no pueden haber líneas sueltas, es decir, sin conexión.
- Evite pasar una línea sobre otra, en caso de que no sea posible, debe usar un conector a la misma página.

...

Buenas prácticas:

...

- Ningún símbolo puede recibir más de una línea, exceptuando el utilizado en la estructura repetitiva `for` y el `conector a la misma página` cuando se usa como punto de concentración para las líneas de flujo. El funcionamiento de la estructura `for` se estudiará de forma detallada en un capítulo posterior.
- Un proceso que recibe dos líneas de flujo es un error (Figura (a)); la forma correcta de diagramar este caso, es uniendo todos los flujos a una sola línea y que esta sea la que conecte al símbolo, como se puede observar al lado derecho de la gráfica (Figura (b)).



- Usar conectores solamente si es necesario.
- En caso de tener un diagrama que sea muy extenso, usar conectores, bien sea dentro de la misma página o a página diferente. Se recomienda que los conectores a la misma página se identifiquen con números y los conectores a página diferente usen letras, o viceversa.
- Los símbolos de decisión deben llevar las dos líneas de salida, especificando cual es la verdadera y cual es la falsa.

1.8. Lenguaje C

Un programa en Lenguaje C está compuesto por un conjunto de palabras en inglés (palabras reservadas) que representan una instrucción que es entendida de una manera específica por el programa en la solución de un problema (Ver Tabla 1.3).

Las palabras reservadas que se utilizan en el Lenguaje C se encuentran estandarizadas y cumplen con tareas específicas dentro de los programas y no pueden usarse para propósitos diferentes.

También existen las denominadas funciones del Lenguaje C, las cuales están agrupadas en biblioteca que se especifican al inicio de cada programa

(archivos de cabecera). Estas son funciones que ya están predefinidas para realizar diversas tareas, entre las más importantes se destacan: comprobación de tipos (`ctype.h`), las funciones matemáticas (`math.h`) que permiten realizar diversos cálculos, las funciones para el manejo de cadenas (`string.h`) que se usan para el tratamiento de datos de tipo alfanumérico y, las funciones de entrada y salida (`stdio.h`), entre otras.

Las Tablas 1.21, 1.22, 1.23, 1.24 y 1.25; presentan un resumen de algunas de estas funciones, sin entrar en mayores detalles; a medida que se requieran se explicará su uso. Sin embargo, se deja que el lector experimente y consulte sobre ellas para sus propios proyectos.

ctype.h	
Contiene varias funciones para comprobación de tipos y transformación de caracteres.	
`int isalnum ( int c );`	Verifica si c es alfanumérico.
`int isalpha ( int c );`	Verifica si c está en el alfabeto.
`int isblank ( int c );`	Verifica si c es un carácter de espacio en blanco.
`int iscntrl ( int c );`	Verifica si c es un carácter de control.
`int isdigit ( int c );`	Verifica si c es un dígito en base 10.
`int islower ( int c );`	Verifica si c está en minúsculas.
`int isprint ( int c );`	Verifica si c es imprimible.
`int ispunct ( int c );`	Verifica si c es un carácter de puntuación.
`int isupper ( int c );`	Verifica si c está en mayúsculas.
`int isxdigit ( int c );`	Verifica si c es un dígito en base 16.
`int tolower ( int c );`	Convierte c a minúsculas.
`int toupper ( int c );`	Convierte c a mayúsculas.
...	

Tabla 1.21: Funciones básicas de la biblioteca **`ctype.h`**

Note como en la biblioteca `ctype.h`, se usa el tipo de dato `int` en lugar de `char` o `bool`. Recuerde que para el Lenguaje C, todo carácter o dato lógico, es en realidad un número entero.

Sin duda alguna, existe un número mayor de bibliotecas y funciones disponibles en el Lenguaje C, más aún si se instalan bibliotecas adicionales. En el transcurso del desarrollo del libro se explicarán y se utilizarán algunas de ellas, pero se motiva al lector a investigar y experimentar con estas y otras funciones del lenguaje en sus proyectos personales; incluso, puede indagar desde ahora, como llegar a crear sus propias bibliotecas que le faciliten la creación de nuevos proyectos; tema que será presentado en el Capítulo 5: Funciones.

math.h

Contiene funciones para cálculos matemáticos

`double` **`abs`** `( double x );`	Valor absoluto de `x`.
`double` **`acosh`** `( double x );`	Cos-hiperbólico inv de `x`.
`double` **`asinh`** `( double x );`	Sen-hiperbólico inv de `x`.
`double` **`atanh`** `( double x );`	Tan-hiperbólica inv de `x`.
`double` **`acos`** `( double x );`	Coseno hiperbólico de `x`.
`double` **`asin`** `( double x );`	Seno hiperbólico de `x`.
`double` **`atan`** `( double x );`	Tangente hiperbólica de `x`.
`double` **`ceil`** `( double x );`	Entero más pequeño >= `x`.
`double` **`cos`** `( double x );`	Coseno de `x`.
`double` **`cosh`** `( double x );`	Coseno hiperbólico de `x`.
`double` **`exp`** `( double x );`	Exponencial de `x`.
`double` **`floor`** `( double x );`	Entero más grande <= `x`.
`double` **`fmod`** `( double x, double y );`	Resto real de `x` con `y`.
`double` **`log`** `( double x );`	Logaritmo en base e de `x`.
`double` **`log10`** `( double x );`	Logaritmo en base 10 de `x`.
`double` **`pow`** `( double x, double y );`	Potencia de `x` a la `y` .
`double` **`round`** `( double x );`	Redondea `x`.
`double` **`sin`** `( double x );`	Seno de `x`.
`double` **`sinh`** `( double x );`	Seno hiperbólico de `x`.
`double` **`sqrt`** `( double x );`	Raíz cuadrada de `x`.
`double` **`tan`** `( double x );`	Tangente de `x`.
`double` **`tanh`** `( double x );`	Tangente hiperbólico de `x`.
`double` **`trunc`** `( double x );`	Trunca `x`.
...	
`bool` **`isfinite`** `( double x );`	Determina si `x` es finito.
`bool` **`isinf`** `( double x );`	Determina si `x` es infinito.
`bool` **`isnan`** `( double x );`	Determina si `x` es NaN
...	

`M_E`	Constante e
`M_PI`	Constante π
...	

Tabla 1.22: Funciones básicas de la bibliotecas **`math.h`**

stdio.h

Contiene varias funciones para entrada/salida

```
int     remove    (const char* filename);
FILE*   tmpfile   (void);
int     fclose    (FILE* stream);
int     fflush    (FILE* stream);
FILE*   fopen     (const char* filename, const char* mode);
int     fprintf   (FILE* stream, const char* format, ...);
int     fscanf    (FILE* stream, const char* format, ...);
int     printf    (const char* format, ...);
int     scanf     (const char* format, ...);
int     sprintf   (char* s, const char* format, ...);
int     sscanf    (const char* s, const char* format, ...);
int     fgetc     (FILE* stream);
char*   fgets     (char* s, int n, FILE* stream);
int     fputc     (int c, FILE* stream);
int     fputs     (const char* s, FILE* stream);
int     getc      (FILE* stream);
int     getchar   (void);
char*   gets      (char* s);
int     putc      (int c, FILE* stream);
int     putchar   (int c);
int     puts      (const char* s);
size_t  fread     (void* ptr, size_t size,
                   size_t nmemb, FILE* stream);
size_t  fwrite    (const void* ptr, size_t size,
                   size_t nmemb, FILE* stream);
int     fgetpos   (FILE* stream, fpos_t* pos);
int     fseek     (FILE* stream, long offset, int whence);
int     fsetpos   (FILE* stream, const fpos_t* pos);
long    ftell     (FILE* stream);
void    rewind    (FILE* stream);
void    clearerr  (FILE* stream);
int     feof      (FILE* stream);
int     ferror    (FILE* stream);
...
```

Tabla 1.23: Funciones básicas de la biblioteca **stdio.h**

stdlib.h

Contiene varias funciones generales (biblioteca estándar)

```
double        atof    ( const char* nptr );
int           atoi    ( const char* nptr );
long          atol    ( const char* nptr );
int           rand    ( void );
void          srand   ( unsigned int seed );
void          free    ( void* ptr );
void*         malloc  ( size_t size );
void          exit    ( int status );
char*         getenv  ( const char* name );
int           system  ( const char* string );
...
```

EXIT_SUCCESS	Constante que indica "*Terminación normal de la aplicación*"
EXIT_FAILURE	Constante que indica "*Terminación anormal de la aplicación*"
...	

Tabla 1.24: Funciones básicas de la biblioteca **stdlib.h**

string.h

Contiene varias funciones para el manejo de cadenas.

```
char*    strcpy    ( char* s1, const char* s2 );
char*    strcat    ( char* s1, const char* s2 );
int      strcmp    ( const char* s1, const char* s2 );
char*    strchr    ( char* s, int c );
char*    strstr    ( char* s1, const char* s2 );
char*    strtok    ( char* s1, const char* s2 );
char*    strerror  ( int errnum );
size_t   strlen    ( const char* s );
...
```

Tabla 1.25: Funciones básicas de la biblioteca **string.h**

Aclaración:

Las funciones también son palabras reservadas y no pueden ser utilizadas como identificadores para declarar variables, constantes, procedimientos, funciones de usuario e incluso los algoritmos.

1.8.1 Comentarios

Es otro de los componentes de un programa; se utilizan de manera opcional para documentarlo. No hacen parte de su lógica, por lo cual no afectará su ejecución.

La documentación es una buena práctica que permite que otra persona o el mismo autor tenga claridad sobre algunos aspectos que se diseñaron dentro del programa. Esto permite que a futuro los ajustes o modificaciones a que haya lugar, sean mucho más sencillos de realizar o por lo menos más entendibles.

Los comentarios pueden ser de dos formas.

1. De una sola línea. Para ello se usan dos barras (`//`).

   ```
   // Esto es un comentario.
   ```

   Toda la línea, a partir de estos caracteres, no se tendrá en cuenta como parte de la lógica del programa.

2. De varias líneas. En este caso se usa la pareja de caracteres `/*` para abrir y la pareja `*/` para cerrar.

   ```
   /* Este es un comentario de varias líneas. Todo lo que
      se escriba entre estas parejas de caracteres será
      ignorado.
   */
   ```

1.8.2 Forma general de un programa en Lenguaje C

Ahora que ya se conoce todo lo que puede contener un programa en Lenguaje C, el siguiente paso es determinar cómo se combinan o conjugan las palabras reservadas con las variables, constantes, expresiones

y comentarios. Para ello observe la siguiente forma general:

Un programa en Lenguaje C tiene una estructura como la que se presenta a continuación:

```
/*
    Programa:    ......
    Descripción: ......
    Autor(es):   ......
    Fecha:       ......
 */

#include <...> // Inclusión de bibliotecas
// Declaración de constantes simbólicas (#define)
// Declaración de consantes (const)

/*
   Comentario sobre la función
 */
int main()
{
   /*
       Instrucciones de la función principal
    */

  return valorDeSalida;
}

/*
 Otras funciones del programa (Ver Capítulo 5)
*/
```

Antes de analizar esta forma general, recuerde que los comentarios son opcionales, se escribieron para hacer más didáctica la explicación.

Se escriben en orden lógico, todas las instrucciones que van a ser ejecutadas por el algoritmo. Cada Instrucción debe ser escrita mediante una de las palabras reservadas que se estudiaron con anterioridad o puede ser una expresión de asignación o una expresión aritmética escrita en notación algorítmica.

Un ejemplo de un programa en Lenguaje C para imprimir un mensaje de bienvenida se puede observar a continuación:

```c
/*
    Programa:    bienvenida.c
    Descripción: Imprime un saludo en la pantalla.
    Autor(es):   Robinson, Orlando, Julian
    Fecha:       Mayo 2018
 */


#include <stdio.h>

/*
   Función principal del programa
 */
int main()
{
   printf ( "Bienvenido a la programación.\n" );

   return 0;
}
```

Se puede apreciar que en la estructura general y en el primer ejemplo, se emplearon las primera seis líneas para documentar el programa (Programa, descripción, autor(es), fecha). Aunque esta parte no es obligatoria, es importante que el lector documente sus programas. Por comodidad para el lector y sabiendo que cada ejemplo del libro tiene una explicación, estos comentarios iniciales fueron omitidos de todos los ejemplos, no obstante, se motiva al lector a crearlos en sus propios programas.

A continuación de la documentación inicial, se realiza la inclusión de todas los archivos de cabecera (bibliotecas) que se requieren, para el ejemplo, se requiere `stdio.h`, biblioteca estándar de entrada y salida del lenguaje. Ella es fundamental para poder emplear, por ejemplo, la función `printf`, entre otras funciones (Ver Tabla 1.23).

Las partes básicas de todo programa son: la entrada de los datos disponibles o conocidos, con los cuales se hará el proceso para hallar un resultado, para finalmente mostrar el resultado al usuario (Entrada-Proceso-Salida), tal y como será explicado en la parte final de este capítulo.

Al terminar el programa, se utiliza la instrucción `return` para indicar el valor de salida. Para este caso, el valor de `0` indica que el programa terminó normalmente, mientras que un valor diferente de 0 se emplea para indicar que el programa terminó de manera anormal, debido principalmente a error al ejecutar alguna de las instrucciones. En el último capítulo se estudian algunos casos.

Si se emplea el archivo de cabecera llamado stdlib.h, es posible retornar una de las dos constantes (EXIT_SUCCESS y EXIT_FAILURE), que allí están definidas, en lugar de usar directamente los valores de 0 o 1 respectivamente. Esto produce el mismo resultado, pero sin duda aumenta la legibilidad, especialmente, para aquellas personas que desconocen el significado de los valores 0 y 1.

1.8.3 Entrada de datos

Además de presentar información al usuario, es importante poder ingresar información al programa (entrada de los datos disponibles); para lograr esta tarea, se pueden utilizar diversas funciones, entre ellas se tienen: scanf, fgets, getLine, entre otras. En el libro, se emplearán principalmente las dos primeras.

La función scanf requiere que se indique: qué tipo de datos se desea leer y/o que información se desea ignorar[3], además de requerir la dirección de memoria en donde se almacenará la información ingresada por el usuario, un ejemplo se puede apreciar a continuación:

```
1 #include <stdio.h>
2
3 int main()
4 {
5    int a;
6    printf ( "Ingrese un valor entero: " );
7    scanf  ( "%d", &a );
8    printf ( "El  %d es el valor ingresado", a);
9
10   return 0;
11 }
```

Al ejecutar el programa

```
Ingrese un valor entero: 60520
El 60520 es el valor ingresado
```

Observe como al lado del nombre de la variable (a), en la función scanf, se emplea el carácter &, el cual indica que se desea enviar la dirección de la memoria asociada a la variable a. En oposición al uso de la misma variable en la función printf en donde no se requiere dicho carácter, porque lo que se desea imprimir, es el valor de la variable y no su dirección. Si por el

[3]Se estudiará más adelante en el libro, en los ejemplos de lectura de cadenas.

contrario, se desea imprimir la dirección de memoria asociada a la variable (a), entonces, sí se debe emplear el carácter &.

Por otro lado, la función scanf como la función printf, emplean una cadena especial para indicar que el dato que será ingresado por el usuario, o que será impreso respectivamente es de tipo entero (Ver Tabla 1.26 para los otros tipos de datos). Así, por ejemplo, si la variable a fuera de tipo char, habría que cambiar "%d" por "%c".

Tipo	Formatos válidos
char signed **char** unsigned **char**	%c
short signed **short**	%hi
unsigned **short**	%hu
int signed **int**	%d, %i, %X
unsigned **int**	%u
long signed **long**	%li
unsigned **long**	%lu
float	%f, %g o %e
double	%lf, %lg o %le
long **double**	%Lf, %Lg, o %Le
Cadena	%s
Dirección de memoria	%p

Tabla 1.26: Tipos de datos en Lenguaje C

Entre el carácter de porcentaje % y la letra o letras siguientes, es posible escribir un número que limita la cantidad de caracteres individuales a ingresar, o la cantidad mínima de espacio reservado para mostrar el valor[4]; para datos reales es posible indicar hasta la cantidad de números decimales a imprimir.

Para ilustrar mejor el concepto, suponga que: se desea un programa que solicite al usuario un número máximo de 4 caracteres (como un año), entonces se debe usar "%4d" en la función scanf.

Los caracteres adicionales no serán almacenados en la variable indicada; de ser requerido, podrán ser almacenados posteriormente en otra variable.

[4]Incluso definiendo la alineación, empleando o no, el signo menos.

Si al mismo tiempo, se desea imprimir el valor de la variable en un espacio mínimo de 10 caracteres en la pantalla y con alineación derecha o izquierda; entonces se debe emplear `" %10d"` o `" %-10d"` respectivamente.

Finalmente, se aprovecha el ejemplo para ilustrar el uso del carácter especial '\n', el cual indica que lo siguiente a ser impreso, debe ser presentado al comienzo de la siguiente línea de la pantalla.

```
1  #include <stdio.h>
2
3  int main()
4  {
5     int a;
6     printf ( "Ingrese un valor entero: " );
7     scanf  ( "%4d", &a );
8     printf ( "El  %10d es el valor ingresado\n", a);
9     printf ( "El  %-10d es el valor ingresado",  a);
10
11    return 0;
12 }
```

Al ejecutar el programa:

```
  Ingrese un valor entero: 60529
  El       6052 es el valor ingresado
  El 6052       es el valor ingresado
```

Incluso, existen otras combinaciones especiales que se pueden usar, por citar solo otro ejemplo:

```
1  #include <stdio.h>
2
3  int main()
4  {
5     int   n1;
6     float n2;
7     n1 = 2019;
8     n2 = 31.051973;
9     printf ( "El  %010d es el valor de n1\n",  n1);
10    printf ( "El  %10.3f es el valor de n1\n", n2);
11
12    return 0;
13 }
```

Al ejecutar el programa:

```
  El 0000002019 es el valor de n1
  El  32341.053 es el valor de n1
```

Aquí se le indica al Lenguaje C, que debe reservar mínimo 10 espacios para imprimir el número `n1` y completar con ceros los espacios no utilizados. En el segundo caso, se indica que para el número `float` (`n2`) se deben emplear mínimo 10 espacios, de los cuales 3 serán para los decimales y uno más para el punto. Para el caso se emplea `%f` para notación tradicional para un número real; también es posible usar `%g` para imprimir una versión con menos decimales; o si se desea, usar `%e` para emplear notación científica.

Así como la función `printf` tiene sus particularidades, la función `scanf` también las tiene. Algunas de estas formas especiales de lectura, serán presentadas en la sección relacionada con la lectura de cadenas de caracteres.

Entrada de cadenas

Existen diversas formas de leer una cadena en el Lenguaje C. Entre ellas se encuentran las funciones: `scanf`, `gets`, `fgets`. Suponga por ejemplo que se desea preguntar por el nombre de un producto, entonces:

- **Usando la función `scanf`:**

```
scanf( "%s", nombreProducto );
```

  Esta versión, toma los espacios como separador entre cadenas y no como un carácter que hace parte de la cadena, como sería entre un nombre y un apellido. Además no garantiza que el usuario intente almacenar más caracteres que la longitud máxima de la cadena, provocando en algunas ocasiones, la destrucción del contenido de otras variables que se definieron cerca de la cadena utilizada. Cuando se inserta una cadena con un espacio, la función toma la parte antes del espacio y la asigna a la variable; dejando la parte restante para la próxima instrucción de lectura, provocando comportamientos "extraños"con respecto al comportamiento esperado.

- **Uso especial de la función `scanf`:**

```
scanf( "%[^\n]%*c", nombreProducto );
```

  Elimina la restricción del espacio en blanco, pero continúa con los demás comportamientos antes mencionados. Se considera un uso especial, por la cadena de control ($\%*c$) usada para `scanf`. Para el ejemplo: `"%[^\n]"` indica que se acepten todos los caracteres hasta

presionar la tecla `Enter` (sin incluirlo), representado por (`\n`); la segunda parte `"%*c"` elimina un carácter de la memoria que aún no se haya leído, en este caso el `Enter` que el usuario presionó y que fue ignorado.

- **Usando la función `gets`:**

  Aunque algunos compiladores lo soportan, ella es obsoleta desde 1999[5]y fue eliminada oficialmente en 2011[6].

```
gets( nombreProducto );
```

  Como alternativa al uso de la función `gets` se cuenta con la función `fgets`.

  En este libro se recomienda el uso de la función `fgets`. En el Capítulo 7, se estudiará otro uso de la función para leer el contenido de un archivo, por ahora se usará para leer datos del teclado (`stdin`).

- **Usando la función `fgets`:**

```
fgets ( nombreProducto, 30, stdin );
```

  Soporta el ingreso de espacios en blanco dentro de la cadena, garantiza que a la variable se le asigne correctamente la cantidad de caracteres válidos a la cadena, incluyendo los caracteres especiales de salto de línea (`'\n'`) y el de finalización de cadena (`'\0'`). Si el usuario intenta escribir más caracteres de los permitidos, la función deja los caracteres sobrantes para la siguiente vez que realice una lectura.

- Es otra forma de usar la función en donde es el Lenguaje C el que determina el tamaño de la cadena mediante la instrucción `sizeof`, en lugar de dejar esta tarea al programador. Ejemplo:

```
fgets( nombreProducto sizeof(nombreProducto), stdin );
```

  Esta función, aunque más segura que las anteriores por garantizar que solo se almacenan en la memoria un número máximo de caracteres, tiene algunas características que provocan ciertos comportamientos no del todo deseables, obligando muchas veces a realizar cierta acciones adicionales para que opere como se supone debería funcionar.

  Dos de estas características que provocan estos comportamientos son:

[5]ISO/IEC 9899:1999/Cor.3:2007(E)
[6]ISO/IEC 9899:2011

- La función incluye dentro de la cadena leída, el carácter especial de salto de línea (`'\n'`). Obligando, en algunas ocasiones, a eliminar dicho carácter de forma manual, ya que no siempre se requiere el uso de dicho salto de línea. Una forma para eliminar este carácter es usando la función `srttok` de la biblioteca `string.h`.
- La función lee del teclado hasta que el usuario presione la tecla `Enter` (`'\n'`). Hasta aquí todo parece deseable, el problema, es que algunas instrucciones como `scanf` normalmente dejan este carácter, en la memoria (*buffer*) del teclado, cuando se le pide leer por ejemplo solo un número entero. Para resolver este problema, se emplea un caso particular del uso especial de la función `scanf` anteriormente presentada.

Un pequeño ejemplo para ilustrar lo que sucede:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int  edad;
6      char nombre[30];
7
8      printf ( "Ingrese la edad: " );
9      scanf ( "%d", &edad );
10
11     printf ( "Ingrese el nombre: " );
12     fgets ( nombre, sizeof ( nombre ), stdin );
13
14     printf ( "Nombre \"%s\"",      nombre );
15     printf ( " y tiene %d años",  edad   );
16
17     return 0;
18 }
```

Al ejecutar el programa

```
Ingrese la edad: 45
Ingrese el nombre: Nombre "
" y tiene 45 años
```

Al ejecutar el programa se observa que al ingresar la edad 45, el Lenguaje C toma únicamente la parte numérica y deja pendiente por leer el carácter correspondiente a la tecla `Enter` (`'\n'`) para la siguiente instrucción de lectura.

Al pasar a la instrucción de `fgets`, ella lee el carácter pendiente en el *buffer* del teclado, el cual corresponde al `Enter` (`'\n'`) y como

esta función termina al detectar dicho carácter, la ejecución continúa a la siguiente instrucción, impidiendo que el usuario ingrese el dato correspondiente a esta lectura.

Una segunda versión sería:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int  edad;
6      char nombre[30];
7
8      printf ( "Ingrese la edad: " );
9      scanf ( "%d%*c", &edad );
10
11     printf ( "Ingrese el nombre: " );
12     fgets ( nombre, sizeof ( nombre ), stdin );
13
14     printf ( "Nombre \"%s\"", nombre );
15     printf ( " y tiene %d años", edad );
16
17     return 0;
18 }
```

Al ejecutar el programa

```
Ingrese la edad: 45
Ingrese el nombre: Julian Esteban
Nombre "Julian Esteban
" y tiene 45 años
```

Ahora, al indicarle a la función `scanf` que lea un entero (`%d`) e ignore el carácter siguiente (`%*c`), que para el caso es el salto de línea, la ejecución continúa sin problemas. Sin embargo, como la función `fgets` almacena el carácter *Enter* dentro del texto que contiene el nombre, al imprimir dicho nombre, se produce un salto de línea en la salida, salto, que en este momento es indeseable debido a que se desea imprimir en nombre encerrado entre comillas, de allí el uso de (`\"`).

La solución final para resolver ambos problemas es usar, además, la función `strtok` de la biblioteca `string.h`, tal y como se muestra a continuación:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
```

```
5  {
6      int  edad;
7      char nombre[30];
8
9      printf ( "Ingrese la edad: " );
10     scanf ( "%d%*c", &edad );
11
12     printf ( "Ingrese el nombre: " );
13     fgets ( nombre, sizeof ( nombre ), stdin );
14     strtok ( nombre, "\n" );
15
16     printf ( "Nombre \"%s\"",   nombre );
17     printf ( " y tiene %d años", edad  );
18
19     return 0;
20 }
```

Al ejecutar el programa

```
Ingrese la edad: 45
Ingrese el nombre: Julian Esteban
Nombre "Julian Esteban" y tiene 45 años
```

Son precisamente todas estas acciones adicionales, ajenas a la lógica de la solución del problema, las que en muchos casos dificultan la construcción de un código limpio y elegante.

Actividad 1.3

Experimente con el siguiente código, usando las diferentes formas de leer una cadena, emplee entradas simples y con espacios; tome nota de lo que sucede y contraste los resultados con la teoría.

```
1  #include <stdio.h>
2
3  int main ()
4  {
5      char a[ 5 ], b[ 5 ];
6
7      printf ( "Ingrese a: " );
8      fgets ( a, sizeof( a ), stdin );
9
10     printf ( "Ingrese b: " );
11     fgets ( b, sizeof( b ), stdin );
12
```

```
13      printf ( "\nCadenas son (%s) y (%s)\n", a, b );
14
15      return 0;
16  }
```

En este momento el lector ya conoce varias de las herramientas que le permiten diseñar una solución algorítmica para resolver un problema. En los siguientes párrafos de este capítulo, se explicará cómo hacer el análisis de esos problemas y cómo implementar los programas en Lenguaje C que los solucionen.

1.9. Cómo solucionar un problema por computador

Con el propósito de desarrollar programas que resuelvan adecuadamente los problemas o necesidades de los usuarios que los requieren, se hace necesario aplicar una serie de pasos que facilitan el proceso. Esos pasos son los siguientes: [Jiménez et al., 2016] y [Corona and Ancona, 2011]:

1. Definición del problema.
2. Análisis del problema.
3. Diseño del algoritmo.
4. Codificación (implementación).
5. Compilación y ejecución.
6. Verificación y depuración.
7. Mantenimiento.
8. Documentación.

Aunque en este libro se utilizan todos estos pasos, se trabajarán principalmente los cinco primeros de los planteados en la lista anterior.

Definición del problema. Se debe tener claridad sobre el problema a solucionar; esto se logra mediante una definición concreta del enunciado que se plantea para cada uno de los problemas que se quieren solucionar; en el desempeño profesional, se debe trabajar con los usuarios para describir

perfectamente sus necesidades y cuáles de ellas se podrían implementar mediante el uso de un programa de computador.

En ocasiones, el programador solo no puede comprender en su totalidad el problema que se le plantea, lo que lo obliga a trabajar con otros profesionales en la definición del problema. Esto fomenta el trabajo interdisciplinario, siendo este una de las características del desarrollo de software.

Análisis del problema. Analizar significa estudiar un problema descomponiéndolo en las partes que lo constituyen. Al dividir el problema, es más sencillo comprender cada parte. El análisis del problema se fundamenta entonces en encontrar la respuesta a cuatro preguntas:

- ¿Cuáles resultados se esperan?
- ¿Cuáles son los datos disponibles?
- ¿Qué procesos hay que realizar?
- ¿Qué variables se deben utilizar?

Los **resultados que se esperan**, conforman las salidas del programa y son el resultado del proceso que realice. En general, cualquier programa debe entregar un resultado. A pesar de que en la ejecución del programa, la salida es el último paso, dentro del análisis que se está haciendo será lo primero a realizar, puesto que con esta información se podrá conocer qué hay que hacer para llegar a la resolución del problema implementando un programa.

Los **datos disponibles**, corresponde al conjunto de datos que se tiene o que se posee y que se va a procesar para encontrar la solución del problema. Los datos disponibles, casi siempre representan los datos de entrada al programa, aunque es importante aclarar que habrán ocasiones en las que los programas no tienen entrada de datos y los datos disponibles son suministrados por alguna instrucción dentro del proceso a realizar.

En cuanto a los **procesos**, son todas las acciones a los que se someten los datos disponibles para encontrar la solución del problema. Aquí, se realizan los cálculos matemáticos en el orden lógico que solucione el problema, se aplican las decisiones y las repeticiones necesarias para el correcto procesamiento de los datos.

Por último y, basados en todo lo anterior, se identifican las **variables requeridas** que almacenarán los valores de los datos disponibles, de los

resultados esperados y de las fórmulas matemáticas que se tengan que hacer. Al establecer cada variable, también es fundamental identificar el tipo de dato que se requiera, de acuerdo con los valores de los datos con los que se trabajará.

Diseño. La etapa del diseño se lleva a cabo luego de terminado el análisis y consiste en la creación del algoritmo que soluciona el problema, para posteriormente llevarlo a un lenguaje de programación, para este caso, Lenguaje C.

Para diseñar el algoritmo, el enunciado del problema debe ser lo suficientemente claro y analizado detalladamente por el diseñador. Algunos enunciados no informan idóneamente lo que se requiere; es allí donde el diseñador debe usar sus habilidades y donde radica la importancia de aplicar correctamente los pasos vistos hasta este punto.

Codificación o Implementación. Posterior al diseño del algoritmo, este se escribe mediante el uso de un **entorno de desarrollo**, que facilita la escritura en el lenguaje de programación deseado, para esta caso, en Lenguaje C, respetando las normas y reglas que el lenguaje exige. Después de escribir el algoritmo, se tiene el programa que se compila y puede ser ejecutado.

Como ejemplo de lo explicado hasta aquí, se analizará el problema que se describe a continuación:

El profesor de Física Mecánica, desea que cada uno de sus estudiantes puedan calcular, mediante un algoritmo, la velocidad con que se desplazan de su casa a la Universidad. Todos los estudiantes deben medir la distancia que recorren y tomar el tiempo que invierten en él.

En este enunciado hay que tener en cuenta lo siguiente:

- No se proporcionó la fórmula para el cálculo de la velocidad. Esto significa que la fórmula tendrá que consultarse de alguna manera.
- El enunciado no aclara qué información o dato debe mostrarse, pero puede deducirse que se requiere mostrar la velocidad calculada. En algunas oportunidades, se llevan a cabo cálculos intermedios que no se tienen que mostrar.
- Algunos datos no representan información importante para hallar la solución, mientras que otros son fundamentales, como en este caso la distancia y el tiempo. El mencionar al profesor de Física Mecánica, no es significativo para la solución.

Una vez analizado todo lo anterior, la aplicación de los pasos para solucionar el problema planteado en el ejemplo, se lleva a cabo de la siguiente forma:

Resultado esperado: la velocidad.

Como ya se tiene el resultado esperado, se pasa a determinar los datos conocidos o disponibles con los cuales poder calcular la velocidad.

Datos conocidos: distancia y tiempo invertidos en el recorrido, teniendo en cuenta las respectivas unidades.

Proceso: consiste en el cálculo de la velocidad, a partir de la distancia y el tiempo. Para hacer este cálculo se necesita de una fórmula, que no se especificó en el enunciado y que debe ser consultada para poder plantear la solución. Obtener la fórmula hace parte del trabajo de levantamiento de información que se hace cuando se quiere resolver un problema.

Suponiendo que ya se tienen la distancia (`x`) y el tiempo (`t`), la fórmula que permite encontrar la velocidad (`v`) es la siguiente: $v = \frac{x}{t}$, que al escribirla como una expresión algorítmica se visualiza así: $v = x/t$

No olvide que las fórmulas en su notación matemática, deben ser convertidas en su notación algorítmica.

El siguiente paso consiste en especificar las variables que se emplearán en el algoritmo.

Variables requeridas: las variables se definirán conforme a la fórmula matemática:

- `v`: velocidad.
- `x`: distancia.
- `t`: tiempo.

Como los valores que se van a almacenar pueden ser reales, se utilizará este tipo dato, que en Lenguaje C se declara como `float`.

A continuación, se usará este ejemplo para ilustrar mediante la Figura 1.16 las 3 etapas que tiene un algoritmo. Esto, con el objetivo de dar mayor claridad a lo visto hasta aqui.

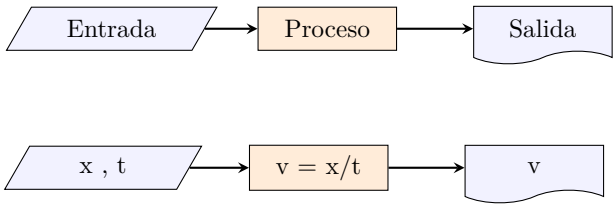


Figura 1.16: Las tres etapas de un algoritmo

Ya se ha realizado el análisis del problema; ahora se continúa con el **diseño del algoritmo**; para ello se utilizarán las 2 formas más comunes de representación.

Diagrama de flujo

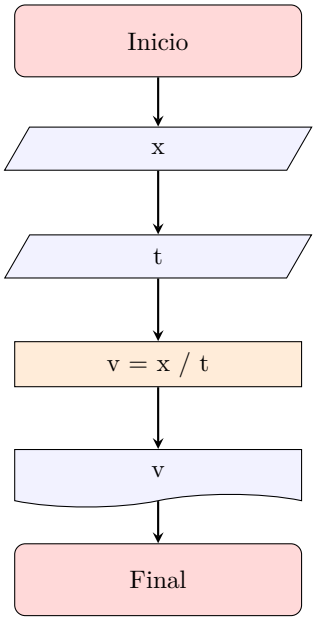


Figura 1.17: Diagrama de flujo Velocidad

A continuación, se presentan las versiones completas en Lenguaje C.

Programa en C - Versión 1

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float v, x, t;
6
7     scanf( "%f", &x );
8     scanf( "%f", &t );
9
10    v = x / t;
11
12    printf ( "%.2f", v );
13 }
```

A diferencia de la solución que se presentó a través el diagrama de flujo, en esta versión en Lenguaje C, se hizo la declaración de variables (línea 5), con esto se le especifica al algoritmo que habrán 3 posiciones de memoria de tipo `float`, llamadas `v,x` y `t`, en las cuales se almacenarán los datos que requieren para solucionar el problema planteado.

La anterior versión en Lenguaej C, puede ser mejorada, incluyendo algunos mensajes con la instrucción `printf`, que serán de ayuda para el usuario del programa. Con ellos se solicitará la distancia y el tiempo, de igual forma la salida se acompañará de un mensaje que proporcione más información.

Programa en C - Versión 2

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float v, x, t;
6
7     printf( "Ingrese la distancia: " );
8     scanf( "%f", &x );
9
10    printf( "Ingrese el tiempo: " );
11    scanf( "%f", &t );
12
13    v = x / t;
14
15    printf ( "La velocidad es de %.2f", v );
16 }
```

Al ejecutar este programa, suponiendo que la distancia es de 300 metros

y el tiempo empleado es de 15 minutos, se observará lo siguiente:

```
Ingrese la distancia: 300
Ingrese el tiempo: 15
La velocidad es de 20.00
```

En los siguientes capítulos, se profundizará en el desarrollo de programas.

Aclaración:

En la versión en Lenguaje C, se hace la declaración de variables, en los diagramas de flujo no se requiere.

La impresión de mensajes también se puede hacer dentro de los diagramas de flujo.

Actividad 1.4

1. Preguntas sobre los conceptos vistos

   a) Defina con sus propias palabras los siguientes términos, vistos en este capítulo: Variable, Constante, Dato, Tipo de dato, Operador, Expresión.

   b) Dado que existen diferentes tipos de datos, operadores y expresiones, haga una clasificación para: Tipos de datos, Operadores, Expresiones.

2. Algoritmos a resolver mediante diagramas de flujo.

   a) Para los siguientes enunciados realice el análisis y el diseño del algoritmo que permita realizar las acciones que a continuación se listan:

   - Adquirir un libro a través de una librería virtual.
   - Descargar un vídeo de YouTube.
   - Calcular cuánto dinero se gastará el día de mañana.
   - Invitar a un amigo a desayunar en la cafetería.
   - Desplazarse desde su casa a un centro comercial.

b) Si el lector es un estudiante universitario, también realice los siguientes ejercicios, los cuales le permitirán familiarizarse con los procesos de su institución educativa. Tenga en cuenta que, si no los conoce, deberá realizar las consultas necesarias:

- Solicitar la homologación de un espacio académico.
- Realizar un proceso de validación de un espacio académico.
- Cancelar materias y semestre.
- Realizar un préstamo de un libro en la biblioteca.
- Realizar una consulta bibliográfica en las bases de datos de la biblioteca.
- Solicitar una cita en el Centro Médico de Bienestar Institucional.

CAPÍTULO 2

ESTRUCTURA SECUENCIAL

El optimismo es un riesgo laboral de la programación; el feedback es el tratamiento.

Kent Beck

Objetivos del capítulo:

- Analizar problemas en los que se puedan aplicar estructuras de programación secuencial para su solución.
- Diseñar diagramas de flujo para resolver problemas de programación.
- Construir programas en Lenguaje C utilizando la estructura de programación secuencial.
- Realizar pruebas de escritorio para verificar el funcionamiento de los programas construidos.

En el capítulo pasado se analizaron los aspectos básicos de la programación en Lenguaje C. Durante este capítulo se escribirán los primeros programas, pequeños y sencillos, pero que permitirán explicar la estructura de programación secuencial; estos programas se escribirán en Lenguaje C y luego tendrán su representación a través de diagramas de flujo. La escritura de estos programas en Lenguaje C, requiere el uso de las palabras reservadas vistas anteriormente, así como la declaración de variables, constantes, y el uso de operadores y expresiones ya expuesto en el capítulo anterior.

Un programa secuencial es un programa en que las instrucciones se escriben unas después de otras secuencialmente y no consta de decisiones y estructuras cíclicas ni funciones que hagan que se salten o repitan líneas de código durante su ejecución.

Todo programa secuencial consta de tres secciones:

- La primera sección es la de "las entradas" (o datos disponibles), en la que se solicitan e ingresan al programa los datos conocidos y requeridos en la solución del problema.
- La segunda parte, conocida como "procesos" o cálculos, es la que implementa las operaciones que llevarán a encontrar los resultados que el programa está buscando.
- La última sección corresponde a las denominadas "salidas" (o resultados esperados), que muestran al usuario los resultados obtenidos en los procesos.

2.1. Estructura básica de un programa secuencial

Con el propósito de tener una manera formal para expresar los programas, tanto en Lenguaje C como en diagramas de flujo, se usará la notación descrita en el punto "Forma general de un programa en Lenguaje C" del capítulo anterior, en el que se describieron los aspectos básicos para escribir los programas en este lenguaje.

De la misma manera, en el capítulo anterior, en el apartado "Diagramas de flujo" se analizó la forma general a utilizar para desarrollar estos diagramas.

En las siguientes páginas, se describen algunos ejemplos que ilustran la programación secuencial siguiendo esta estructura: primero el enunciado del ejercicio, luego un análisis del problema, después la solución en Lenguaje C..

Aclaración:

Durante el ingreso de datos, es posible que el usuario ingrese datos erróneos, por ejemplo, una edad negativa o números donde deben ir solo letras. Por medio de la misma programación se puede validar el ingreso de los datos, sin embargo, durante este capítulo se dará por sentado que el usuario ingresa los datos adecuadamente. Los procesos de validación se abordarán en capítulos posteriores.

.:Ejemplo 2.1. *Imagine que en una organización requieren de un programa que calcule el salario a pagarle a un empleado. El empleado trabaja n horas y su salario es esa cantidad de horas multiplicadas por el valor de la hora asignado a dicho empleado.*

Análisis del problema:

- **Resultados esperados:** el programa deberá determinar el salario del empleado.
- **Datos disponibles:** se deben conocer el número o cantidad de horas laboradas por el empleado y el valor al que le pagan cada hora.
- **Proceso:** el salario se calcula multiplicando el número de horas por el valor de la hora del empleado.
- **Variables requeridas:**
  - `numeroHoras:` guarda el número de horas trabajadas por el empleado.
  - `valorHora:` corresponde al valor que recibe el empleado por cada hora de trabajo.
  - `salarioPagar:` es el salario que recibirá el empleado.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.1 y se escribe el Programa 2.1.

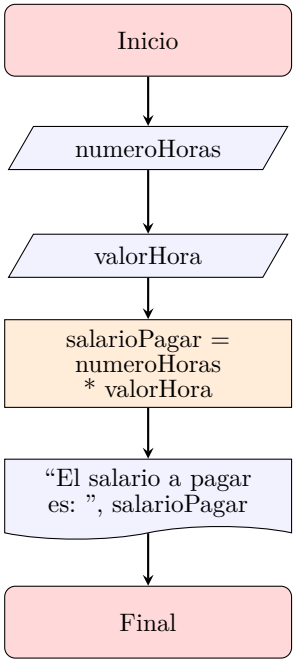


Figura 2.1: Diagrama de flujo del programa `SalarioEmpleado`

Programa 2.1: SalarioEmpleado

```c
#include <stdio.h>

int main()
{
    int numeroHoras;
    float valorHora, salarioPagar;

    printf( "Ingrese el número de horas laboradas: " );
    scanf( "%d", &numeroHoras );

    printf( "Ingrese el valor de la hora: " );
    scanf( "%f", &valorHora );

    salarioPagar = numeroHoras * valorHora;

    printf("El salario a pagar es: %.2f\n", salarioPagar );

    return 0;
}
```

Al ejecutar el programa:

Primera ejecución:

```
Ingrese el número de horas laboradas: 10
Ingrese el valor de la hora: 45000
El salario a pagar es: 450000.00
```

Segunda ejecución:

```
Ingrese el número de horas laboradas: 48
Ingrese el valor de la hora: 22500
El salario a pagar es: 1080000.00
```

Aclaración:

Para todos los ejemplos del libro existe una sección denominada "Explicación del programa", cuyo propósito es ayudar a comprender el programa propuesto en Lenguaje C.

Con el fin de facilitar la explicación, se seleccionan algunas partes del programa (las de mayor complejidad o no explicadas previamente) y se aclaran línea por línea. No obstante, en los programas que se presentan mas adelante, ya no se harán las explicaciones que se hayan realizado en ejercicios anteriores.

Explicación del programa:

La primera parte del programa consiste en incluir la biblioteca que se necesita (archivos de cabecera), en esta caso, la biblioteca `<stdio.h>`.

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

A continuación se encuentra la función `main` y, en ella se declaran las variables indicando tanto su nombre como su tipo de dato.

```
3   int main()
4   {
5       int numeroHoras;
6       float valorHora, salarioPagar;
```

Luego, se usa la instrucción `printf` dos veces para solicitar el ingreso del número de horas laboradas y el valor de la hora.

En estas instrucciones aparece también la sentencia `scanf`, con la cual se capturan los datos ingresados por el usuario y se llevan a la dirección

de memoria a la que apuntan las variables, tal y como fue explicado en el capítulo anterior.

```
8     printf( "Ingrese el número de horas laboradas: " );
9     scanf( "%d", &numeroHoras );
10
11    printf( "Ingrese el valor de la hora: " );
12    scanf( "%f", &valorHora );
```

Luego, se realiza el cálculo del salario a pagar con el producto del número de horas laboradas por el valor de la hora.

```
14  salarioPagar = numeroHoras * valorHora;
```

Posteriormente, se muestra el salario a pagar, usando para ello un `printf` que incluye un salto de línea al final (`\n`).

```
16    printf("El salario a pagar es: %.2f\n", salarioPagar );
```

Finalmente, se termina el programa indicando al sistema operativo que este finalizó sin errores; para ello, solo basta con retornar el número cero.

```
18    return 0;
```

.:Ejemplo 2.2. *Escriba un programa en Lenguaje C que, con el valor del radio de un círculo obtenga su área y su perímetro.*

Tenga en cuenta que la fórmula para calcular el área de un círculo es $PI * radio^2$*, y la que calcula el perímetro es* $2 * PI * radio$*.*

Análisis del problema:

- **Resultados esperados:** el programa debe calcular y mostrar el área y el perímetro del círculo.
- **Datos disponibles:** se requiere el valor del radio del círculo al que se le desean calcular área y perímetro. Otro dato disponible para este ejercicio es el valor de PI, que corresponde a 3.141592 aproximadamente y que es constante para cualquier círculo.
- **Proceso:** consiste en aplicar las fórmulas para calcular el área y perímetro escritas correctamente como expresiones en Lenguaje C.
- **Variables requeridas:**
  - `radio`: almacenará el valor del radio del círculo.
  - `area`: almacenará el cálculo del área del círculo.

- `perimetro`: almacenará el cálculo del perímetro del círculo.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.2 y se escribe el Programa 2.2.

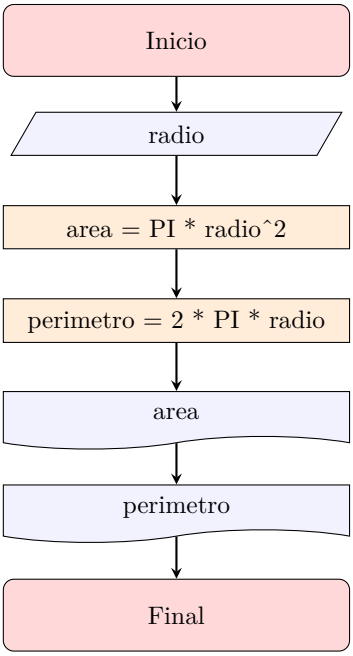


Figura 2.2: Diagrama de flujo del programa `Circulo`

Programa 2.2: Circulo

```
1  #include <stdio.h>
2
3  int main()
4  {
5      const float PI = 3.141592;
6      float radio, area, perimetro;
7
8      printf( "Ingrese el radio del círculo: " );
9      scanf( "%f", &radio );
10
11     area = PI * radio * radio;
12     perimetro = 2 * PI * radio;
13
14     printf( "El área del círculo es: %.2f\n", area );
15     printf( "El perímetro es: %.2f\n", perimetro );
16
17     return 0;
18 }
```

Aclaración:

Observe que, en el diagrama de flujo de este ejercicio (Ver Figura 2.2), las instrucciones de salida no muestran los mensajes exactamente como están escritos en el programa en Lenguaje C, sino que muestran solo los nombres de las variables; en otros ejercicios del libro se observará que ambos textos coinciden (el programa en Lenguaje C y diagrama de flujo). El tener o no los mensajes no afecta en últimas la estructura de la solución, así que por simplicidad, en algunos casos se omiten.

Al ejecutar el programa:

```
Ingrese el radio del círculo: 4
El área del círculo es: 50.27
El perímetro del círculo es: 25.13
```

Explicación del programa:

Luego de incluir la biblioteca que se requiere, se lleva a cabo la declaración de las variables y la constante PI (ya dentro de la función main), que almacenarán el `radio` del circulo ingresado por el usuario, los cálculos del `area` y el `perimetro`. Todas las variables se declararon reales o `float`, porque almacenarán números, incluso con decimales.

```
5     const float PI = 3.141592;
6     float radio, area, perimetro;
```

Con la instrucción `printf` se muestra el mensaje solicitando el `radio`, mientras que con `scanf` se captura el dato y se asigna en la respectiva variable `radio`. Note que en la instrucción `scanf` aparece "f", lo que le indica al compilador que debe recibir un dato de tipo `float`; además aparece `&radio`, lo que indica que el dato ingresado se almacena en la dirección de memoria asociada a la variable `radio`.

```
8     printf( "\n Ingrese el radio del círculo: " );
9     scanf( "%f", &radio );
```

Posteriormente, se calculan área y perímetro a partir de las fórmulas matemáticas que involucran la constante `PI`.

```
11    area = PI * radio * radio;
12    perimetro = 2 * PI * radio;
```

Finalmente, mediante la instrucción `printf` se muestran los resultados encontrados.

```
14      printf( "El área del círculo es: %.2f\n", area );
15      printf( "El perímetro es: %.2f\n", perimetro );
```

Aclaración:

Si el lector lo desea, puede incluir una nueva biblioteca llamada `math.h` y usar la constante propia del Lenguaje C llamada `M_PI`, la cual posee una mayor precisión que el valor de PI utilizado en este programa (3.141592). Ver Programa 2.3.

Programa 2.3: Circulo2

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      float radio, area, perimetro;
7
8      printf( "Ingrese el radio del círculo: " );
9      scanf( "%f", &radio );
10
11     area = M_PI * radio * radio;
12     perimetro = 2 * M_PI * radio;
13
14     printf( "El área del círculo es: %.2f\n", area );
15     printf( "El perímetro es: %.2f\n", perimetro );
16
17     return 0;
18 }
```

.:Ejemplo 2.3. *Escriba un programa que, para un número cualquiera ingresado por el usuario calcule y muestre los resultados de las funciones matemáticas principales de Lenguaje C.*

Con este ejercicio se ilustra la manera en que trabajan algunas de las funciones matemáticas del Lenguaje y que están disponibles en la biblioteca <math.h>.

Análisis del problema:

- **Resultados esperados:** al ejecutar el programa, se mostrarán el seno, el coseno, la tangente, la raíz cuadrada, el logaritmo natural y el logaritmo en base 10 de un número ingresado.
- **Datos disponibles:** el número que se usará para llamar las funciones.
- **Proceso:** llamar las funciones matemáticas, enviándoles como argumento el número para que puedan hacer el cálculo; este cálculo se almacena en variables que se mostrarán al final del programa.
- **Variables requeridas:**
  - `numero`: valor ingresado por el usuario.
  - `seno`: resultado obtenido con la función `sin` (seno) sobre el número ingresado.
  - `coseno`: resultado obtenido con la función `cos` (coseno) sobre el número ingresado.
  - `tangente`: resultado obtenido con la función `tan` (tangente) sobre el número ingresado.
  - `raiz`: resultado obtenido con la función `sqrt` (raíz cuadrada) sobre el número ingresado.
  - `logaritmoNatural`: resultado obtenido con la función `log` (logaritmo natural) sobre el número ingresado.
  - `logaritmo10`: resultado obtenido con la función `log10` (logaritmo en base 10) sobre el número ingresado.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.3 y se escribe el Programa 2.4.

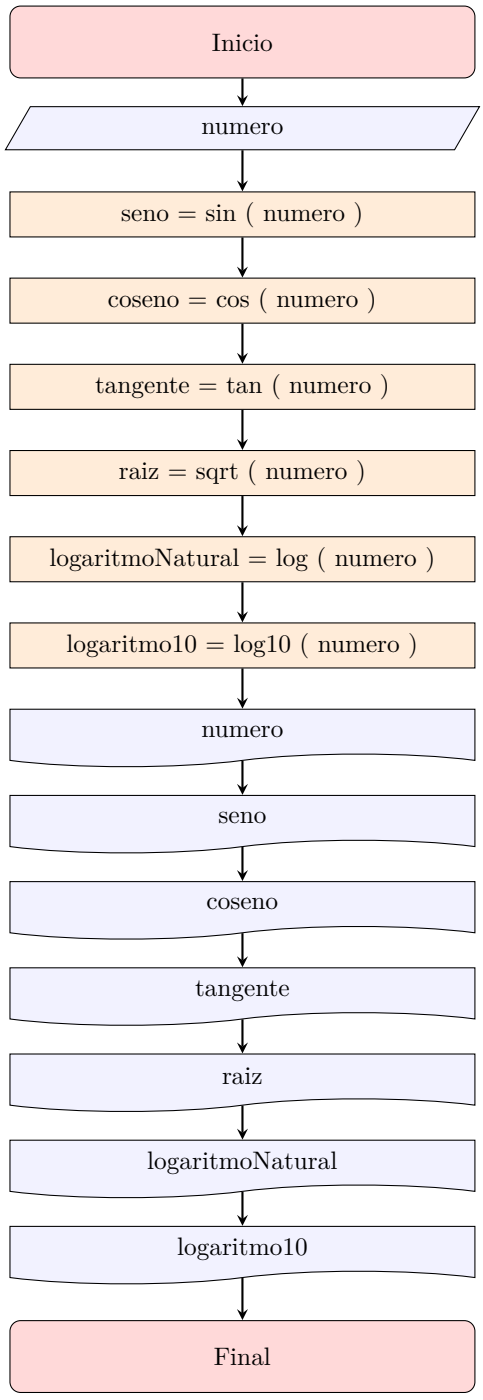


Figura 2.3: Diagrama de flujo del programa `UsoFunciones`

Programa 2.4: UsoFunciones

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

int main()
{
    float  numero, seno,   coseno, tangente,
           raiz,   logNat, loga10;

    printf( "Ingrese un número: " );
    scanf( "%f", &numero );

    seno     = sin  ( numero );
    coseno   = cos  ( numero );
    tangente = tan  ( numero );
    raiz     = sqrt ( numero );
    logNat   = log  ( numero );
    loga10   = log10( numero );

    printf( "Del número : %.2f\n", numero );
    printf( "Su seno es : %.2f\n", seno );
    printf( "Su coseno es : %.2f\n", coseno );
    printf( "Su tangente es : %.2f\n", tangente );
    printf( "Su raíz es : %.2f\n", raiz );
    printf( "Su logaritmo natural es : %.2f\n", logNat );
    printf( "Su logaritmo en base 10 es : %.2f\n", loga10);

    return 0;
}
```

Al ejecutar el programa:

```
Ingrese un número: 25
Del número: 25
Su seno es: -0.13
Su coseno es: 0.99
Su tangente es: -0.13
Su raíz cuadrada es: 5.00
Su logaritmo natural es: 3.22
Su logaritmo en base 10 es: 1.40
```

Explicación del programa:

El programa inicia con el llamado a las bibliotecas y declarando las variables que se necesitan.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6     float  numero, seno,   coseno, tangente,
7            raiz,   logNat, loga10;
```

Después, se le solicita al usuario con `printf` el número con el que se harán los cálculos y se captura en la variable `numero` utilizando `scanf`.

```
9     printf( "Ingrese un número: " );
10    scanf( "%f", &numero );
```

Luego, se van llamando las funciones matemáticas para obtener los resultados deseados, los cuales se van almacenando en las variables respectivas.

```
12    seno     = sin  ( numero );
13    coseno   = cos  ( numero );
14    tangente = tan  ( numero );
15    raiz     = sqrt ( numero );
16    logNat   = log  ( numero );
17    loga10   = log10( numero );
```

Finalmente, se muestran los resultados obtenidos con las instrucciones `printf`.

```
19    printf( "Del número : %.2f\n", numero );
20    printf( "Su seno es : %.2f\n", seno );
21    printf( "Su coseno es : %.2f\n", coseno );
22    printf( "Su tangente es : %.2f\n", tangente );
23    printf( "Su raíz es : %.2f\n", raiz );
24    printf( "Su logaritmo natural es : %.2f\n", logNat );
25    printf( "Su logaritmo en base 10 es : %.2f\n", loga10);
```

.:Ejemplo 2.4. *En los países de habla inglesa, es común medir la estatura de las personas en pies y pulgadas; sin embargo, en muchos otros países, la estatura se mide en metros. Construya un programa que permita ingresar los pies y pulgadas que mide una persona y convertir esta medida a metros.*

Análisis del problema:

- **Resultados esperados:** la estatura de la persona en metros.
- **Datos disponibles:** la cantidad de pies y de pulgadas que mide la persona.

- **Proceso:** luego de tener los pies y las pulgadas, se multiplican las primeras por 30.48 y las segundas por 2.54. Con esto, se pasan esas medidas a centímetros; luego se suman los centímetros obtenidos y, por último se convierte la suma a metros dividiendo entre 100.
- **Variables requeridas:**
  - `numPies`: número o cantidad de pies que mide la persona.
  - `numPulgadas`: número de pulgadas que mide la persona, adicionales al número de pies.
  - `numMetros`: metros que mide la persona.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.4 y se escribe el Programa 2.5.

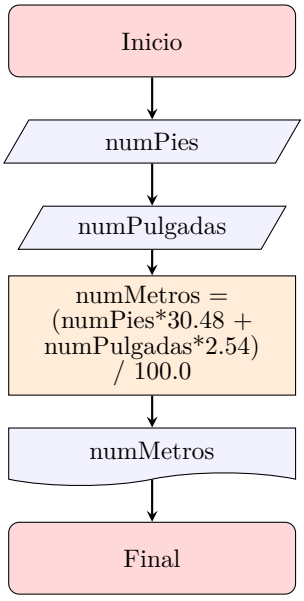


Figura 2.4: Diagrama de flujo del programa `EstaturaPersona`

Programa 2.5: EstaturaPersona

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int   numPies, numPulgadas;
6    float numMetros;
7
8    printf("Ingrese los Pies que mide la persona: ");
9    scanf("%i", &numPies);
10
11   printf("Pulgadas adicionales que mide la persona: ");
12   scanf("%i", &numPulgadas);
13
14   numMetros = (numPies*30.48 + numPulgadas*2.54) / 100.0;
15
16   printf("Una persona que mide %i Pies y %i Pulgadas, mide
         %.2f Metros", numPies, numPulgadas, numMetros);
17
18   return 0;
19 }
```

Al ejecutar el programa:

```
Ingrese los Pies que mide la persona: 5
Pulgadas adicionales que mide la persona: 11
Una persona que mide 5 Pies y 11 Pulgadas, mide 1.80 Metros
```

Explicación del programa:

Después de hacer el llamado a las bibliotecas, se declaran las variables necesarias, dos enteras y una real:

```
6    int   numPies, numPulgadas;
7    float numMetros;
```

Enseguida, se solicitan los datos y se almacenan en las variables que representan los Pies y las Pulgadas, haciendo uso de `printf` y `scanf`

```
9    printf("\n Ingrese los Pies que mide la persona: ");
10   scanf("%i", &numPies);
11
12   printf("\n Pulgadas adicionales que mide la persona: ");
13   scanf("%i", &numPulgadas);
```

Note que en la instrucción `scanf` se usa el carácter de control **i**, que indica que el dato a capturar es de tipo `int`.

Lo que sigue en el programa es la realización del cálculo que permite

obtener la estatura en metros. Esto se lleva a cabo convirtiendo tanto los pies y las pulgadas a centímetros; para ello se multiplican por su equivalente (30.48 y 2.54 respectivamente), se suman y, por último se divide el valor de la suma en 100.0; esto dará la medida en metros.

```
15    numMetros = (numPies*30.48 + numPulgadas*2.54) / 100.0;
```

El resultado de este cálculo se almacena en la variable `numMetros`. Finalmente, se muestra un mensaje que indica a cuánto equivale la estatura en metros de una persona ingresada inicialmente en pies y pulgadas, usando para ello la instrucción `printf`. Observe que en la instrucción `printf` se encuentra un `.2f` que se usa para mostrar solo 2 decimales del valor almacenado en la variable `numMetros`.

```
17     printf("\n Una persona que mide %i Pies y %i Pulgadas,
          mide %.2f Metros", numPies, numPulgadas, numMetros);
```

.:Ejemplo 2.5. *Escriba un programa usando Lenguaje C que determine el valor de los intereses obtenidos por un monto de dinero invertido en un Certificado de Depósito a Término (CDT) en una entidad financiera durante un periodo de tiempo (en días). Esto se obtiene aplicando la siguiente fórmula:*

$$valorIntereses = (cantidad * porcentajeInteres * periodo)/360$$

El programa también debe hallar el valor total a retirar por el cliente que invirtió en el CDT al final del periodo, teniendo en cuenta que existe un descuento del 7% sobre los intereses ganados por concepto de impuesto de retención en la fuente.

Análisis del problema:

- **Resultados esperados:** se deben encontrar y mostrar: el valor de los intereses ganados por el CDT en el periodo ingresado, el valor del descuento por concepto del impuesto mencionado y el valor total a retirar por el cliente.
- **Datos disponibles:** el monto de dinero depositado en el CDT, el periodo de tiempo en días y el porcentaje de interés a aplicar.
- **Proceso:** después de que el usuario ingrese los datos requeridos, se procede a calcular el valor de los intereses generados con la fórmula mencionada en el enunciado, luego se obtiene el valor del impuesto que es del 7% sobre los intereses calculados. Finalmente se suma la cantidad ingresada con el valor de los intereses y se resta el valor de los impuestos.

- **Variables requeridas:**
  - `cantidad`: monto de dinero en el CDT
  - `periodo`: tiempo al que se coloca el CDT.
  - `porcentajeInteres`: tasa de interés del CDT.
  - `valorIntereses`: cantidad de dinero que genera el monto ingresado en un principio.
  - `valorImpuesto`: cantidad de dinero a pagar por impuestos.
  - `netoPagar`: valor total obtenido por el cliente.

Aclaración:

Observe la importancia de tener una clara comprensión del problema; es por ello que se debe hacer un análisis que facilite la comprensión del ejercicio y consultar aquellos aspectos que aún no están entendidos en su totalidad. Por esta razón, se explica brevemente a continuación lo que es un CDT.

Un Certificado de Depósito a Término, mejor conocido como CDT es un producto ofrecido por las entidades financieras para ahorrar de una manera distinta a las cuentas de ahorros. Un CDT se abre con un monto inicial, por un plazo de tiempo específico y gana unos intereses durante ese plazo.

Al finalizar el plazo, el cliente recibe su dinero más el valor de los intereses ganados y, debe pagar un impuesto sobre los intereses generados. Generalmente, el porcentaje de interés es mayor entre mayor sea el plazo, lo que hace que los CDTs produzcan un mayor rendimiento que otros productos financieros.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.5 y se escribe el Programa 2.6.

Programa 2.6: CdtBancario

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float  cantidad, porcentajeInteres, valorIntereses,
6             valorImpuesto, netoPagar;
7      int    periodo;
8
9      printf( "Ingrese la cantidad de dinero: " );
10     scanf( "%f", &cantidad );
11
12     printf( "Ingrese el periodo en días: " );
13     scanf( "%d", &periodo );
14
15     printf( "Ingrese el porcentaje de interés: " );
16     scanf( "%f", &porcentajeInteres );
17
18     valorIntereses = ( cantidad * porcentajeInteres / 100.0
           * periodo ) / 360.0;
19     valorImpuesto  = valorIntereses * 0.07;
20     netoPagar      = cantidad+valorIntereses-valorImpuesto;
21
22     printf( "Intereses ganados: %.2f\n", valorIntereses );
23     printf( "Valor del impuesto: %.2f\n", valorImpuesto );
24     printf( "Total a pagar: %.2f\n", netoPagar );
25
26     return 0;
27 }
```

Al ejecutar el programa:

```
Ingrese la cantidad de dinero: 1000000
Ingrese el periodo en días: 360
Ingrese el porcentaje de interés: 6
Intereses ganados: 60000.00
Valor del impuesto: 4200.00
Total a pagar:  1055800.00
```

Explicación del programa:

Posterior a la declaración de variables:

```
5      float  cantidad, porcentajeInteres, valorIntereses,
6             valorImpuesto, netoPagar;
7      int    periodo;
```

Se le pide al usuario que ingrese los datos requeridos con la instrucción `printf`: cantidad o monto de dinero, porcentaje[1] de interés y periodo.

Como con la instrucción `scanf`, se utiliza `%f` cuando se ingresan variables reales como la cantidad de dinero y el porcentaje de interés; y se emplea `%i` cuando la variable a ingresar es entera, como es el caso de la variable `periodo`.

```
9     printf( "Ingrese la cantidad de dinero: " );
10    scanf( "%f", &cantidad );
11
12    printf( "Ingrese el periodo en días: " );
13    scanf( "%d", &periodo );
14
15    printf( "Ingrese el porcentaje de interés: " );
16    scanf( "%f", &porcentajeInteres );
```

Con los datos ingresados, se calcula el valor de los intereses ganados multiplicando la cantidad por el porcentaje (dividido entre 100.0) y por el periodo y se divide todo esto en 360.0, ya que es la cantidad de días que tiene el año comercial.

Al valor de los intereses calculados con la instrucción anterior, se le aplica el 7 % o 0.07 para encontrar el valor a pagar por concepto de impuesto de retención en la fuente.

```
18    valorIntereses = ( cantidad * porcentajeInteres / 100.0
          * periodo ) / 360.0;
19    valorImpuesto  = valorIntereses * 0.07;
20    netoPagar      = cantidad+valorIntereses-valorImpuesto;
```

Después se suman la cantidad de dinero con el valor de los intereses y se resta el valor del impuesto; con esto se obtiene el total o neto a pagar. Las instrucciones finales utilizan `printf` para mostrar al usuario los resultados esperados. Observe como se formatea la salida con dos decimales usando `%.2f`.

```
22    printf( "Intereses ganados: %.2f\n", valorIntereses );
23    printf( "Valor del impuesto: %.2f\n", valorImpuesto );
24    printf( "Total a pagar: %.2f\n", netoPagar );
```

[1]Observe que en la ejecución, el porcentaje se ingresa como un número entre 1 y 100 y que en la fórmula es convertido a un número entre 0 y 1 dividiéndolo por 100.

.:Ejemplo 2.6. *Utilice Lenguaje C y un diagrama de flujo para crear un programa que recibe un número entero de 3 cifras (Por ejemplo, 927 o 483) y luego calcula el valor de la suma de las 3 cifras, 18 para el primer ejemplo y 15 para el segundo ejemplo.*

Análisis del problema:

- **Resultados esperados:** la suma de los tres dígitos que componen el número ingresado.
- **Datos disponibles:** un número cualquiera de 3 dígitos.
- **Proceso:** el programa va a separar los 3 dígitos que componen el número y luego va a sumar los mismos. Cada dígito se obtiene a través de una división, entera o modular. para obtener el primer dígito se divide el número en 100, con esto se encuentra el primer dígito, pues el resultado es la parte entera de la división.

  El segundo dígito se obtiene haciendo una división entera en 10 del número de tres cifras para después, dividir este resultado parcial en 10 mediante el operador módulo. Por último, el tercer dígito se obtiene dividiendo modularmente el número de tres cifras en 10, ya que esto dá como resultado el resto de la división, es decir, el último dígito del número.
- **Variables requeridas:**
  - `numero`: almacena el número de tres cifras ingresado.
  - `digito1`: primera cifra del número ingresado.
  - `digito2`: segunda cifra del número ingresado.
  - `digito3`: tercera cifra del número ingresado.
  - `suma`: almacena la suma de los tres dígitos.

Aclaración:

Aunque en este programa no existe una validación que obligue al usuario a ingresar estrictamente un número de tres cifras, se dá por sentado que el usuario lo ingresa de esa forma.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.5 y se escribe el Programa 2.7.

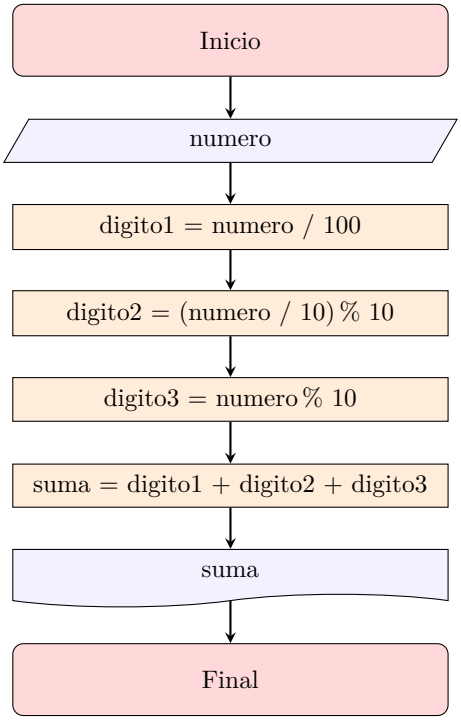


Figura 2.5: Diagrama de flujo del programa `SumaDigitosNumero`

Programa 2.7: SumaDigitosNumero

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int numero, digito1, digito2, digito3, suma;
6
7      printf( "Ingrese un número entero de tres digitos: " );
8      scanf( "%d", &numero );
9
10     digito1 = numero / 100;
11     digito2 = (numero / 10)  % 10;
12     digito3 = numero  % 10;
13
14     suma = digito1 + digito2 + digito3;
15
16     printf( "Suma de digitos de  %d es  %d\n", numero, suma);
17
18     return 0;
19 }
```

Al ejecutar el programa:

```
Ingrese un número entero de tres dígitos: 927
Suma de digitos de 927 es 18
```

Explicación del programa:

La primera línea escrita en el programa luego de la función `main`, corresponde a la declaración de variables:

```
5     int numero, digito1, digito2, digito3, suma;
```

Después, se solicita el único dato disponible (`numero`).

```
7     printf( "Ingrese un número entero de tres digitos: " );
8     scanf( "%d", &numero );
```

Luego se procede a realizar los cálculos que determinan los resultados esperados:

```
10   digito1 = numero / 100;
11   digito2 = (numero / 10) % 10;
12   digito3 = numero % 10;
13   suma    = digito1 + digito2 + digito3;
```

La expresión `digito1 = numero / 100`, divide el número en 100 y obtiene la parte entera correspondiente al primer dígito, que guarda en digito1. Por otro lado, `digito2 = (numero / 10) % 10`, encuentra el segundo dígito así: el numero al ser dividido entre 10 entrega como resultado los dos primeros dígitos que después, al dividir en `% 10` obtiene como resultado su residuo, es decir, el segundo dígito del número. La expresión `digito3 = numero % 10`, dá como resultado el residuo de esta división, esto es, el tercer dígito. A continuación, se suman los tres dígitos y se guarda la suma en la variable `suma` que al final, se muestra con la instrucción `printf`.

.:Ejemplo 2.7. *Mensualmente, un empleado debe hacer un aporte a seguridad social de un 4 % para salud y un 4 % para pensión sobre su salario base. Escriba un programa en C, con su respectivo diagrama de flujo, que calcule el valor del aporte que debe hacer un empleado a salud y a pensión sobre su salario base, que encuentre el total del descuento por estos conceptos y que determine el salario neto a pagar con el descuento realizado.*

Análisis del problema:

- **Resultados esperados:** aporte por salud, aporte por pensión y salario neto a pagar al empleado.
- **Datos disponibles:** se debe tener disponible el salario base del empleado.
- **Proceso:** se calcula el valor del aporte a salud y pensión, cada uno de ellos corresponde al 4 % sobre el salario base. Se suman los dos resultados anteriores para obtener el total de los descuentos. Finalmente, al salario base se le resta dicho total, lo que permite encontrar el salario neto a pagar.
- **Variables requeridas:**
  - `salarioBase`: salario base del empleado.
  - `aporteSalud`: almacena el aporte por concepto de salud.
  - `aportePension`: almacena el aporte por concepto de pensión.
  - `descuento`: almacena el total de los descuentos.
  - `salarioNeto`: valor del salario incluyendo los descuentos.

Aclaración:

La seguridad social es un aporte que se paga entre empleadores y empleados. Estos últimos aportan un 4 % de su salario a salud y otro tanto a pensión. Cada mes, los aportes se le descuentan al empleado de su salario y se hacen los pagos a las EPSs y fondos de pensiones; esto significa que el salario neto que recibe un empleado tiene esos descuentos. Por ejemplo, si un empleado recibe como salario base 1000000 (un millón de pesos), el descuento por salud corresponde a 40000, lo mismo que el descuento por pensión; así, el salario neto que recibiría el empleado sería de 920000 (novecientos veinte mil pesos).

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.6 y se escribe el Programa 2.8.

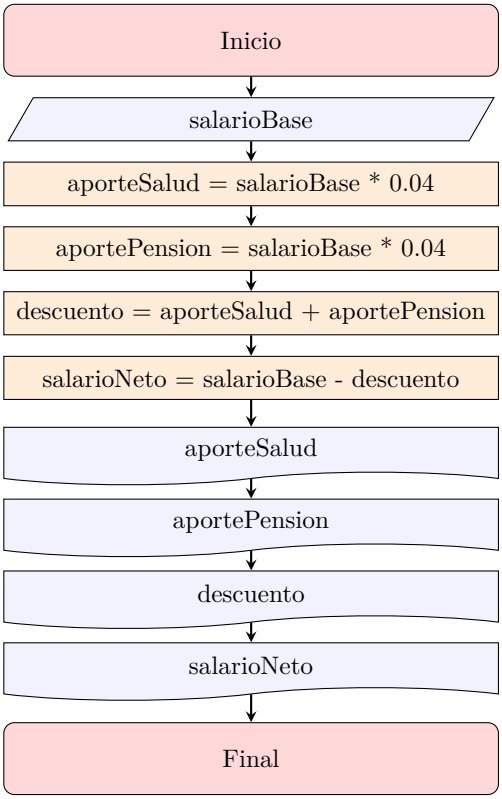


Figura 2.6: Diagrama de flujo del programa `SeguridadSocial`

Programa 2.8: SeguridadSocial

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float  salarioBase, aporteSalud, aportePension,
6             descuento, salarioNeto;
7
8      printf( "Ingrese el salario base del empleado: " );
9      scanf( "%f", &salarioBase );
10
11     aporteSalud   = salarioBase * 0.04;
12     aportePension = salarioBase * 0.04;
13     descuento     = aporteSalud + aportePension;
14     salarioNeto   = salarioBase - descuento;
15
16     printf( "El aporte a salud es %.2f\n", aporteSalud );
17     printf( "El aporte a pensión es %.2f\n", aportePension);
18     printf( "El descuento es de %.2f\n", descuento );
19     printf( "El salario a pagar es %.2f\n", salarioNeto );
```

```
20
21      return 0;
22  }
```

Al ejecutar el programa:

```
Ingrese el salario base del empleado 1000000
El aporte a salud es 40000.00
El aporte a pensión es 40000.00
El descuento es 80000.00
El salario a pagar es 920000.00
```

Explicación del programa:

Al inicio, se declaran las variables necesarias para almacenar los datos a ingresar y a calcular. Como todos ellos representan dinero, se declaran las variables de tipo **float**.

```
5       float  salarioBase, aporteSalud, aportePension,
6              descuento, salarioNeto;
```

A continuación, se pide al usuario el salario base que gana el empleado.

```
8       printf( "Ingrese el salario base del empleado: " );
9       scanf( "%f", &salarioBase );
```

Se calculan los aportes a salud y pensión, que son del 4 % cada uno; observe que se usa el valor de 0.04. Cada descuento se calcula en una línea separada, una para el aporte a salud y otra para el aporte a pensión.

```
11      aporteSalud   = salarioBase * 0.04;
12      aportePension = salarioBase * 0.04;
```

Posteriormente, se totaliza el descuento sumando los aportes.

```
13      descuento = aporteSalud + aportePension;
```

Enseguida, el salario neto se calcula restando el descuento al salario base.

```
14      salarioNeto = salarioBase - descuento;
```

Finalmente, se imprimen los datos calculados.

```
16      printf( "El aporte a salud es %.2f\n", aporteSalud );
17      printf( "El aporte a pensión es %.2f\n", aportePension);
18      printf( "El descuento es de %.2f\n", descuento );
19      printf( "El salario a pagar es %.2f\n", salarioNeto );
```

.:Ejemplo 2.8. *Construya un programa en Lenguaje C que, al ingresarle un número de días cualquiera, determine cuántos minutos y cuántos segundos tiene esa cantidad de días.*

Para resolver el problema, es importante conocer los minutos que tiene un día, así como los segundos contenidos en cada día, esto se logra multiplicando el número de días que se ingresan por el número de horas que tiene un día (24) y a su vez, por la cantidad de minutos que tiene una hora (60) y por los segundos que tiene un minuto (60).

Análisis del problema:

- **Resultados esperados:** la cantidad de minutos que tiene el número de días ingresados y el número de segundos.
- **Datos disponibles:** el número de días que van a ser convertidos en minutos y segundos.
- **Proceso:** calcular la cantidad de minutos que tienen los días ingresados multiplicando los días por 24 horas que tiene cada día y por 60 minutos que tiene cada hora. Para obtener la cantidad de segundos, se multiplican los minutos obtenidos por 60, que es el número de segundos que tiene cada minuto.
- **Variables requeridas:**
  - `numeroDias`: cantidad de días.
  - `numeroMinutos`: cantidad de minutos.
  - `numeroSegundos`: cantidad de segundos.

Aclaración:

Existen muchos problemas que exigen realizar una conversión, por ejemplo, convertir un peso que está en libras a otro en kilogramos, una temperatura en grados kelvin a centígrados, etc.

Para resolver un problema de conversión, es imprescindible conocer el equivalente de una medida con respecto a la otra, de esta forma, se puede construir una expresión que realice la conversión.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.7 y se escribe el Programa 2.9.

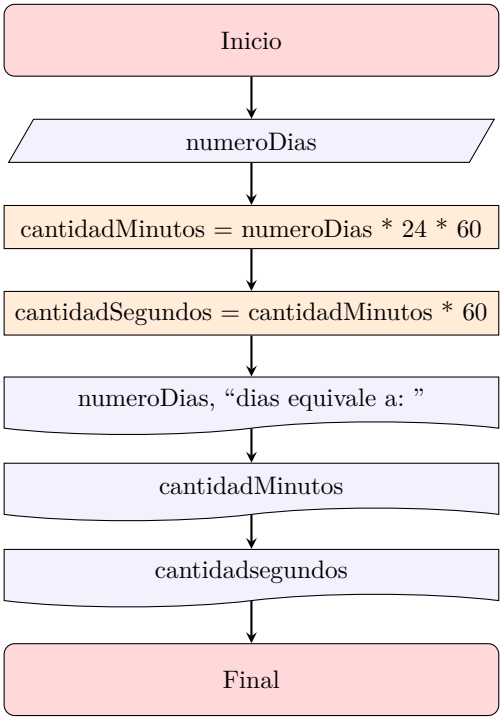


Figura 2.7: Diagrama de flujo del programa `ConversionDias`

Programa 2.9: ConversionDias

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int numeroDias, cantidadMinutos, cantidadSegundos;
6
7      printf( "Ingrese el número de días: " );
8      scanf(  "%d", &numeroDias );
9
10     cantidadMinutos  = numeroDias * 24 * 60;
11     cantidadSegundos = cantidadMinutos * 60;
12
13     printf( "%d días equivalen a:\n", numeroDias );
14     printf( "%d Minutos\n", cantidadMinutos );
15     printf( "%d Segundos\n", cantidadSegundos );
16
17     return 0;
18 }
```

Al ejecutar el programa:

```
Ingrese el número de días: 2
2 Días equivalen a:
2880 Minutos
172800 Segundos
```

Explicación del programa:

Como en todos los programas, se empieza declarando las variables requeridas.

```
5     int numeroDias, cantidadMinutos, cantidadSegundos;
```

Enseguida, se le pide al usuario el dato, es decir, el número de días.

```
7     printf( "Ingrese el número de días: " );
8     scanf(  "%d", &numeroDias );
```

Ya con el número de días, se lleva a cabo el primer cálculo que consiste en encontrar el equivalente en minutos a los días ingresados; esto se logra multiplicando los días por 24 (cada día tiene 24 horas) y multiplicando nuevamente por 60 (cada hora tiene 60 minutos). La cantidad de minutos que tienen los días ingresados, se encuentra multiplicando la cantidad de minutos ya obtenidos en el cálculo anterior por 60 (un minuto consta de 60 segundos).

```
10    cantidadMinutos  = numeroDias * 24 * 60;
11    cantidadSegundos = cantidadMinutos * 60;
```

Finalmente, se muestran los resultados al usuario con la instrucción `printf`.

```
13    printf( "%d días equivalen a:\n", numeroDias );
14    printf( "%d Minutos\n", cantidadMinutos );
15    printf( "%d Segundos\n", cantidadSegundos );
```

.:Ejemplo 2.9. *Escriba un programa en Lenguaje C que encuentre las dos soluciones reales de una ecuación algebraica de segundo grado utilizando la fórmula general.*

Por ahora, es necesario suponer que el ejercicio planteado no tiene soluciones imaginarias y que el usuario ingresa un valor para `a` *diferente de cero.*

Fórmula general:

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

Aclaración:

En el Álgebra que se estudia en la secundaria se tratan las ecuaciones de segundo grado. Este tipo de ecuaciones se resuelven de diversas formas, entre ellas, con el uso de la fórmula general que se acaba de presentar. La fórmula exige conocer los valores de `a`, `b` y `c` y genera dos respuestas x_1 y x_2, una cuando se toma el signo + en la fórmula, antes del radical y la otra al tomar el signo menos.
Las ecuaciones de segundo grado, presentan la siguiente forma general, de donde se obtienen los valores de `a`, `b` y `c`:

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

Análisis del problema:

- **Resultados esperados:** las dos respuestas a la ecuación: `x1` y `x2`, suponiendo que son reales[2].
- **Datos disponibles:** los valores de `a`, `b` y `c` deben ser conocidos por el usuario.
- **Proceso:** después de ingresar los datos o valores de `a`, `b` y `c`, se aplica la fórmula general escrita como expresión algorítmica de Lenguaje C, una para encontrar `X1` (usando el signo + antes del radical) y otra para encontrar `X2` (usando el signo – antes del radical). Tenga en cuenta que, la fórmula implica elevar al cuadrado la variable `b` y obtener la raíz cuadrada de una parte de la fórmula; esto requiere el uso de funciones del lenguaje presentes en la biblioteca `<math.h>`.
- **Variables requeridas:**
  - `a`: valor del primer coeficiente de la ecuación.
  - `b`: valor del segundo coeficiente de la ecuación.
  - `c`: valor del tercer coeficiente de la ecuación.
  - `x1`: primera solución de la ecuación.
  - `x2`: segunda solución de la ecuación.

[2]Las ecuaciones de segundo grado también pueden dar un resultado con valores imaginarios, pero estos escapan al análisis del ejercicio que se plantea, ya que para poder obtenerlos sería necesario utilizar otras estructuras de programación que se verán más adelante en el libro.

De acuerdo al análisis planteado, se propone el Programa 2.10.

Programa 2.10: FuncionCuadratica

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      float a, b, c, x1, x2;
7
8      printf( "Ingrese el primer coeficiente: " );
9      scanf( "%f", &a );
10
11     printf( "Ingrese el segundo coeficiente: " );
12     scanf( "%f", &b );
13
14     printf( "Ingrese el tercer coeficiente: " );
15     scanf( "%f", &c );
16
17     x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
18     x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
19
20     printf( "Primera solución : %.2f\n", x1 );
21     printf( "Segunda solución : %.2f\n", x2 );
22
23     return 0;
24 }
```

Al ejecutar el programa:

```
Ingrese el primer coeficiente: 3
Ingrese el segundo coeficiente: 8
Ingrese el tercer coeficiente: 4
Primera solución -0.67
Segunda solución -2.0
```

Explicación del programa:

Se declaran las variables requeridas y, a continuación, se le solicita al usuario el ingreso de los valores de `a`, `b` y `c`. Nuevamente, se utilizan las instrucciones `printf()` y `scanf()`.

```
6      float a, b, c, x1, x2;
7
8      printf( "Ingrese el primer coeficiente: " );
9      scanf( "%f", &a );
10
11     printf( "Ingrese el segundo coeficiente: " );
12     scanf( "%f", &b );
```

```
13
14      printf( "Ingrese el tercer coeficiente: " );
15      scanf( "%f", &c );
```

Luego, se convierte la fórmula algebraica a expresión algorítmica reconocida por Lenguaje C, usando allí la función matemática `sqrt()` que calcula la raíz cuadrada. Esta fórmula se aplica dos veces, en dos líneas distintas, una obtiene el valor de `x1` y otra haya el valor de `x2`.

```
17      x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
18      x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
```

Después de encontrar los resultados, se muestran con la instrucción `printf`.

```
20      printf( "Primera solución : %.2f\n", x1 );
21      printf( "Segunda solución : %.2f\n", x2 );
```

.:Ejemplo 2.10. *Un maestro de construcción se ha especializado en la instalación de pisos en cerámica para casas y apartamentos. Cada vez que va a realizar una obra, el maestro requiere saber la cantidad de cajas de cerámica que debe comprar, así como su costo. El maestro toma medidas y obtiene el número de metros cuadrados que debe cubrir; sabe que una caja cubre un área de* $2.26\ m^2$ *y conoce su costo. Escriba un programa que, conociendo el número de metros cuadrados de la obra y el costo de la caja de cerámica le permita saber al maestro la cantidad de cajas a comprar y el costo total de las mismas.*

Aclaración:

Con este ejercicio se pretende ilustrar la forma en que un programa puede ser útil en diversas profesiones y labores. Así como, en este caso, el programa ayuda a determinar cantidades de materia prima y sus costos, también se podrían crear programas para muchas otras profesiones y tareas que le faciliten a los usuarios la realización de sus trabajos.

Análisis del problema:

- **Resultados esperados:** la cantidad de cajas de cerámica que se deben comprar y el valor total de las mismas.

- **Datos disponibles:** el número de metros cuadrados que posee el inmueble (casa o apartamento) al que se le va a instalar el piso y el valor de cada caja de cerámica seleccionada para hacer la instalación.

- **Proceso:** a partir del número de metros cuadrados se realiza una división entre `2.26`, que equivale a los metros cuadrados que cubre una caja; con esto se obtiene el número de cajas a comprar. Luego, conociendo la cantidad de cajas, este número se multiplica por el valor de cada caja, encontrando así el costo total de la cerámica.

  Generalmente en los almacenes de materiales, las cajas de cerámica se venden completas y no por fracciones. Por lo tanto, es importante resaltar que, para este ejercicio cuando se refiere a la cantidad de las cajas se está haciendo un cálculo exacto de esa cantidad, es decir, el resultado se va a obtener con parte decimal.

- **Variables requeridas:**

  - `tamanioPiso`: cantidad de metros cuadrados del piso.
  - `valorCaja`: costo de cada caja de cerámica.
  - `cantidadCajas`: cantidad de cajas requeridas.
  - `valorTotal`: valor total de las cajas de cerámica.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 2.8 y se escribe el Programa 2.11.

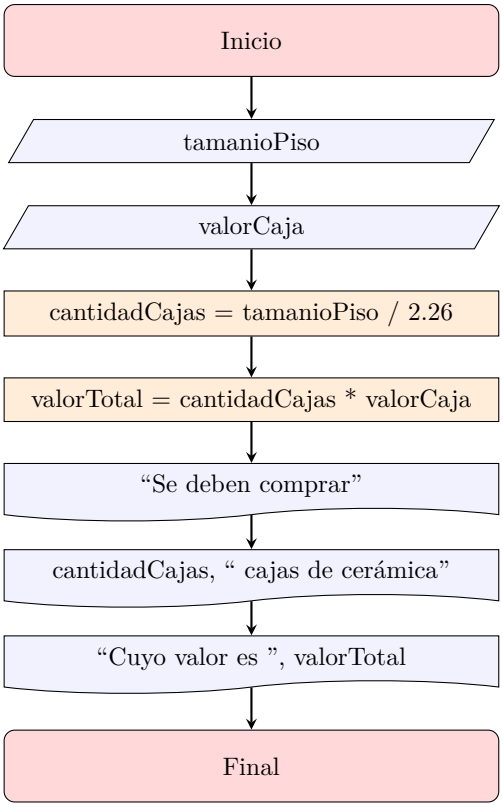


Figura 2.8: Diagrama de flujo del programa `CeramicaPiso`

Programa 2.11: CeramicaPiso

```
1  #include <stdio.h>
2
3  int main()
4  {
5     float tamanioPiso, valorCaja, cantidadCajas, valorTotal;
6
7     printf( "Ingrese la cantidad de metros cuadrados: " );
8     scanf( "%f", &tamanioPiso);
9
10    printf( "Ingrese el valor de la caja de cerámica: " );
11    scanf( "%f", &valorCaja );
12
13    cantidadCajas = tamanioPiso / 2.26;
14    valorTotal    = cantidadCajas * valorCaja;
15
16    printf( "Se deben comprar:\n" );
17    printf( "\%.2f  de cajas de cerámica\n", cantidadCajas );
18    printf( "Cuyo valor es  %.2f\n", valorTotal );
19
20    return 0;
21 }
```

Al ejecutar el programa:

```
Ingrese la cantidad de metros cuadrados: 84
Ingrese el valor de la caja de cerámica: 20000
Se deben comprar
37.17 de cajas de cerámica
Cuyo valor es 743362.83
```

Explicación del programa:

La primera parte del código es la declaración de las variables a utilizar:

```
5     float tamanioPiso, valorCaja, cantidadCajas, valorTotal;
```

A continuación, se pide al usuario que ingrese los datos requeridos que son: la cantidad de metros cuadrados a instalar y el valor de una caja de cerámica.

```
7     printf( "Ingrese la cantidad de metros cuadrados: " );
8     scanf( "%f", &tamanioPiso);
9
10    printf( "Ingrese el valor de la caja de cerámica: " );
11    scanf( "%f", &valorCaja );
```

Posteriormente, con las expresiones aritméticas que siguen en el código se encuentra el número de cajas de cerámica necesarias, esto se logra dividiendo los metros cuadrados a instalar entre 2.26 que corresponde a la cantidad de metros cuadrados que trae cada caja; la otra expresión determina el costo de todas las cajas a comprar, multiplicando la cantidad de cajas que se acaban de obtener por el valor de cada una de ellas.

```
13    cantidadCajas = tamanioPiso / 2.26;
14    valorTotal    = cantidadCajas * valorCaja;
```

Por último, se muestran los resultados obtenidos por medio de la instrucción `printf`.

```
16    printf( "Se deben comprar:\n" );
17    printf( "\%.2f  de cajas de cerámica\n", cantidadCajas );
18    printf( "Cuyo valor es %.2f\n", valorTotal );
```

2.2. Pruebas de escritorio

Una prueba de escritorio es una herramienta que se usa en la programación de computadores, con el propósito de verificar el buen funcionamiento de un programa. La prueba de escritorio se debe diseñar adecuadamente para que encuentre las fallas, si las hay, que a simple vista no se han encontrado. Después de aplicar una prueba, si se encuentra una inconsistencia en los resultados esperados, se tendrán que analizar las líneas de código escritas en el programa para determinar cuál o cuáles de ellas son las causantes del mal funcionamiento del mismo.

Una prueba de escritorio se diseña de la siguiente manera:

- Se escogen unos datos de entrada para ejecutar el programa.
- Se ejecuta cada línea de código haciendo lo que ella indique. Este proceso puede ser hecho manualmente o de forma automatizada. Para el alcance del libro, será realizada únicamente de forma manual.
- Al finalizar la ejecución, se comparan los resultados que arrojó el programa con los resultados esperados y se analiza si fueron los mismos o no.
- Si los resultados arrojados por el programa no corresponden con los resultados esperados, se recorre el programa para encontrar la/las

líneas de código que generan la falla, de lo contrario, puede concluirse que el programa funciona adecuadamente, al menos, lo hace para los datos de prueba.

Aclaración:

Una estrategia para diseñar una prueba de escritorio consiste en construir una tabla (denominada **tabla de verificación**). Esta tabla contiene las variables, desde las que almacenan los datos ingresados por el usuario, los cálculos y por último, las instrucciones que muestran los resultados esperados. Si el programa contiene estructuras como condiciones, estas deben ubicarse también en la tabla en el orden en que van apareciendo en el programa.
Se recomienda ejecutar el programa varias veces con diferentes datos de entrada con el fín de analizar su comportamiento en cada ejecución.

2.2.1 Ejemplos

En la Tabla 2.1 se encuentran los datos con los que se hará la prueba de escritorio para el programa del Ejemplo 2.1, nombrado como `SalarioEmpleado` que calcula el salario a apagar a un empleado al que le pagan de acuerdo al número de horas laboradas y al valor de la hora.

Ejecución	`numeroHoras`	`valorHora`	`salarioAPagar`	`Respuesta`
1	10	30000	300000	300000
2	40	25000	1000000	1000000
3	20	27000	540000	540000

Tabla 2.1: Prueba de escritorio para el programa del Ejemplo 2.1

Y en la Tabla 2.2 están los datos para la prueba de escritorio del programa 2.6 denominado `CDTBancario`.

Por limitaciones de espacio, las variables que aparecen en la Tabla 2.2 fueron renombradas de la siguiente forma:

- `cantidad` por `cant`
- `porcentajeInteres` por `pInter`

- `valorIntereses` por `vInter`
- `valorImpuesto` por `vImpuesto`
- `netoPagar` por `nPagar`

Ej.	cant	periodo	pInter	vInter	vImpuesto	nPagar
1	1000000	360	6	60000	4200	1055800
2	500000	180	5	12500	875	511625
3	2000000	90	4	20000	1400	2018600

Tabla 2.2: Prueba de escritorio para el programa del Ejemplo 2.5

Actividad 2.1

Escriba programas en Lenguaje C y diseñe los diagramas de flujo para los problemas que se enuncian a continuación. Posteriormente, haga la prueba de escritorio con la tabla de verificación para determinar la validez del código escrito:

1. El profesor de la asignatura de Introducción a la Lógica de Programación, va a evaluar a sus estudiantes mediante 4 parciales, cada uno con un peso sobre la nota definitiva del 25 %. Construya un programa que permita calcular la nota definitiva de un estudiante.

2. Con base en el ejercicio anterior, suponga que cada nota tendrá un porcentaje, como aparece en la tabla 2.3

Nota Parcial	Porcentaje
Primera	15 %
Segunda	15 %
Tercera	30 %
Cuarta	40 %

Tabla 2.3: Porcentaje de notas para el Ejercicio Propuesto 2

Escriba un programa que calcule la nota definitiva a partir de los porcentajes dados.

3. Se tiene el valor del lado de un cubo. Haga un programa que calcule el área total de las seis caras del cubo, su perímetro y su volumen. El área de una cara se calcula multiplicando dos de sus lados. El perímetro de un cubo corresponde a la suma de todos sus lados y su volumen se encuentra elevando al cubo el valor de un lado conocido.

4. Elabore un programa que encuentre el valor de la liquidación de un contrato de un empleado que laboró n días a un salario x. La liquidación consta de prima, cesantías, intereses a las cesantías y vacaciones. Estos valores se calculan de la siguiente manera:

   - **Prima**: `(salario * diasLaborados) / 360`
   - **Cesantías**: `(salario * diasLaborados) / 360`
   - **Intereses cesantías**: `cesantías * (12 % / diasLaborados)`
   - **Vacaciones**: `(salario * diasLaborados) / 720`

5. Escriba un programa que reciba el nombre, el costo de un producto y la cantidad comprada por un cliente y que determine el valor a pagar, incluyendo el valor del impuesto al valor agregado IVA, que es del 16 % sobre el valor de la compra. El programa debe mostrar el subtotal, el valor del IVA y el total a pagar.

6. Escriba un programa que reciba un número entero positivo (`n`) y calcule la suma de los primeros `n` números enteros. Use la fórmula que aparece a continuación:

$$suma = \frac{n * (n + 1)}{2}$$

7. Construya un programa que permita conocer el precio de venta de un bien inmueble basado en la cantidad de metros cuadrados que tiene y el valor del metro cuadrado. El programa debe calcular también el valor de la comisión por la venta que corresponde al 2.5 % del valor de venta del inmueble.

8. Investigue las fórmulas necesarias para convertir un ángulo expresado en grados, minutos y segundos a su equivalente en radianes. Construya un programa en Lenguaje C que le permita calcular los radianes.

9. Un artesano requiere determinar el precio de venta de un producto que acaba de fabricar. El precio se asigna así: el artesano conoce el valor de la materia prima que usó, le establece un precio a la mano de obra y, por último, determina una utilidad del 30 %. Diseñe un programa que reciba el nombre de un producto, el valor de las materias primas usadas en su elaboración y el valor de la mano de obra con que se construyó y que determine el precio de venta del producto.

10. Construya un programa que determine la altura de un edificio del que cae un objeto que tarda `n` segundos en tocar el piso, así como la velocidad con la que el objeto llega al suelo. Tenga en cuenta las fórmulas que aparecen a continuación:

$$altura = velocidadInicial * tiempo + \frac{1}{2} * gravedad * tiempo^2$$

$$velocidadFinal = velocidadInicial^2 + 2 * gravedad * altura$$

Recuerde que en la tierra, la gravedad es una constante de valor `9.8` m/s^2.

CAPÍTULO 3

ESTRUCTURAS DE DECISIÓN

Los ordenadores son inútiles.
Solo pueden darte respuestas.

Pablo Picasso

Objetivos del capítulo:

- Construir programas en Lenguaje C usando condiciones simples, compuestas, anidadas y múltiples.
- Reconocer y utilizar los árboles de decisión en el análisis de problemas.
- Elaborar diagramas de flujo que representen los tipos de decisiones.

Una herramienta fundamental en la escritura de programas corresponde a las estructuras de decisión. Estas estructuras permiten que un programa ejecute, ya sea una o varias instrucciones en lugar de otra u otras, dependiendo de cómo se evalúe una expresión lógica/relacional que contiene la estructura. La expresión lógica/relacional puede evaluarse como verdadera o falsa, lo que le indicará al programa qué camino seguir.

Existen diversas clases de estructuras de decisión. A continuación, se empieza el estudio de las mismas.

3.1. Decisiones simples y compuestas

Esta estructura contiene al principio la palabra reservada `if` seguida de una condición dentro de paréntesis; enseguida se escriben las instrucciones, encerradas entre llaves, que se ejecutarán si la condición se evalúa cómo verdadera; después, se encuentra la palabra `else` que indica el comienzo de las instrucciones, que también van entre llaves y que ejecutará el programa si la condición se evalúa como falsa. En el segmento de código que se presenta a continuación, se muestra la forma general de esta estructura:

```
1 if( condición )
2 {
3   InstrucciónV-1;
4   InstrucciónV-2;
5   ...
6   InstrucciónV-n;
7 }
8 else
9 {
10  InstrucciónF-1;
11  InstrucciónF-2;
12  ...
13  InstrucciónF-m;
14 }
```

En el fragmento de código anterior, `Instrucción-V` representa las instrucciones que se llevarán a cabo cuando la expresión lógica/relacional (conjunto de condiciones) se evalúe como verdadera. Por su parte, el conjunto de instrucciones `Instrucción-F` representa las acciones a realizar en la parte falsa de la decisión, en otras palabras, las instrucciones que el programa realiza cuando la condición o conjunto de condiciones dan como resultado un valor falso.

Durante la escritura de programas, puede pasar que no sea necesario ejecutar instrucciones en el caso de que la condición se evalúe como falsa, lo que indica que la parte del `else` no se tendrá que escribir, lo que genera un bloque de código como el que se presenta en el segmento de programa que se presenta a continuación. Este tipo de estructuras se denominan "**Decisión simple**" y el diagrama de flujo que las representa se puede observar en la página 57.

```
1 if( condición )
2 {
3   InstrucciónV-1;
4   InstrucciónV-2;
5   ...
6   InstrucciónV-n;
7 }
```

Árboles de Decisión

Los árboles de decisión son herramientas gráficas que permiten ilustrar una o más decisiones relacionadas, así como las instrucciones que se ejecutan en cada alternativa de la decisión. Lo anterior indica lo que se hace cuando la respuesta a la decisión es verdadera (**sí**) o falsa (**no**).

Se recomienda el uso de los árboles de decisión para el análisis de los problemas que implican el uso de decisiones, incluyendo los problemas de programación de computadores.

Con el fín de dar claridad a estos conceptos, imagine que usted necesita trasladarse rápidamente a cierta parte de la ciudad, entonces usted analiza si: "*tengo suficiente dinero, abordo un taxi; sino, me voy en bus*". Los árboles de decisión (Ver Figura 3.1) son herramientas útiles para representar las estructuras de decisión de forma gráfica y fácil de comprender.

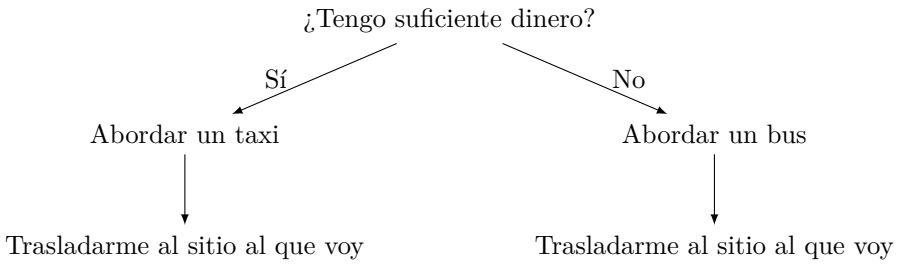


Figura 3.1: Árbol de decisión para trasladarme a un sitio

Aclaración:

En la Figura 3.1 se puede concluir que, si tengo suficiente dinero, me puedo ir en taxi y llego rápido al sitio al que debo ir; si no tengo el dinero suficiente, abordo un bus para trasladarme al sitio, pero en ningún caso me transporto simultáneamente en taxi y en bus, es decir, solo se lleva a cabo una de las alternativas.

Otra forma de representar las estructuras de decisión es a través del uso de los diagramas de flujo (Ver Figura 3.2). Con este tipo de diagramas se puede representar todo un algoritmo, incluyendo las estructuras de decisión que se están estudiando, mediante la utilización de una figura en forma de rombo.

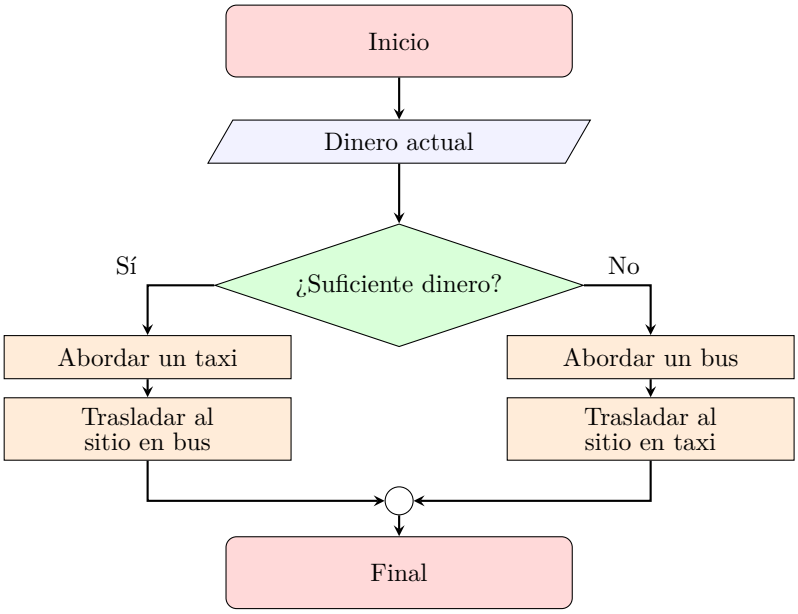


Figura 3.2: TrasladoCiudad - Versión 1

Como se ha mencionado antes, puede ocurrir que una decisión solo requiera de la parte verdadera de la decisión; esto se ilustra a través del Ejemplo 3.1.

.:Ejemplo 3.1. *Escriba un programa en Lenguaje C, que reciba el nombre de un producto, su valor y la cantidad comprada de ese producto por un cliente; el programa debe determinar el valor a pagar, teniendo en cuenta que si el valor de la compra es superior a 100000.00, el cliente recibe un descuento del 5 por ciento. La solución debe mostrar el valor de la compra, el valor del descuento y el valor a pagar, que incluye el descuento.*

Análisis del problema:

- **Resultados esperados:** el programa debe mostrar el valor de la compra, el valor del descuento y el valor neto a pagar por el cliente.
- **Datos disponibles:** el nombre del producto, su valor unitario y la cantidad comprada por el cliente.
- **Proceso:** se obtiene el valor de la compra multiplicando la cantidad comprada por el valor del producto. Mediante un `if` se determina si el valor de la compra es mayor a 100000.00; si esto ocurre, se calcula el descuento del 5 por ciento. Al final, se muestran los resultados. La Figura 3.3) muestra la estructura de decisión:

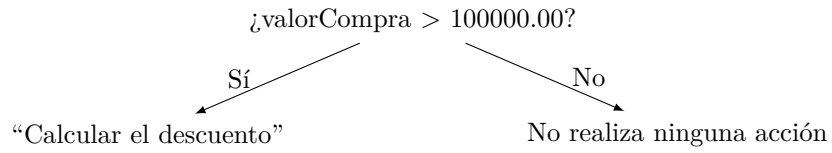


Figura 3.3: Árbol de decisión del Ejemplo 3.1

- **Variables requeridas:**
  - `nombreProducto`: almacena el nombre del producto.
  - `valorUnitario`: valor unitario del producto.
  - `cantidad`: cantidad de productos comprados por el cliente.
  - `valorCompra`: valor de la compra sin descuento.
  - `desto`: valor del descuento.
  - `valorPagar`: valor a pagar por el cliente con descuento incluido.

Coniforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.4 y se escribe el Programa 3.1.

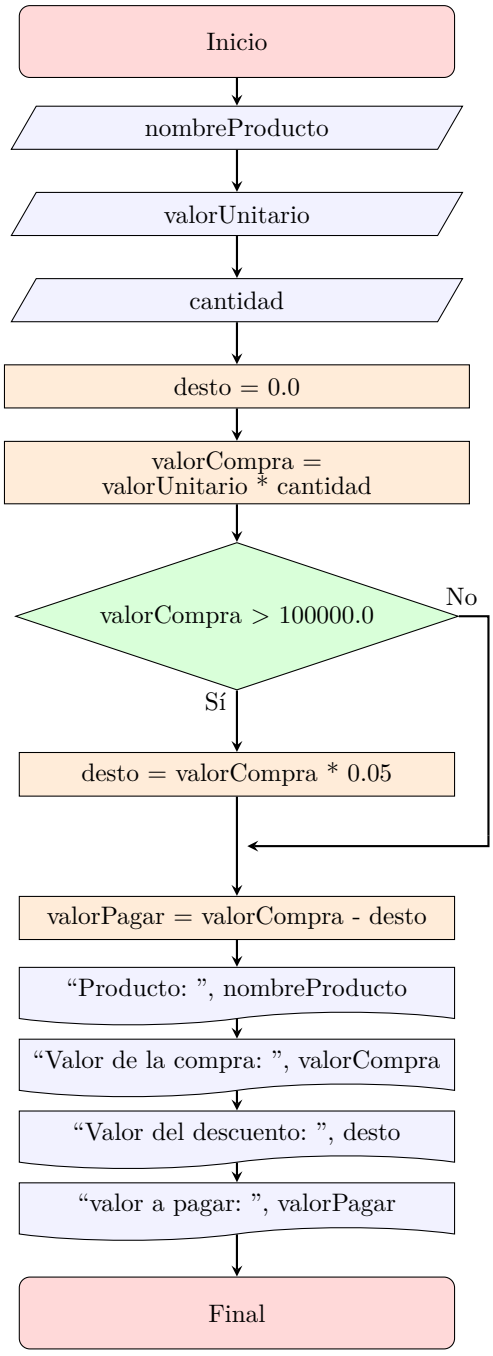


Figura 3.4: Diagrama de flujo del Programa `Producto`

Programa 3.1: Producto

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char  nombreProducto[ 30 ];
6      float valorUnitario, valorCompra, valorPagar, desto;
7      int   cantidad;
8
9      printf( "Ingrese el nombre del producto: " );
10     scanf( "%s", nombreProducto );
11
12     printf( "Ingrese el valor del producto: " );
13     scanf( "%f", &valorUnitario );
14
15     printf( "Ingrese la cantidad comprada: " );
16     scanf( "%i", &cantidad );
17
18     desto = 0.0;
19
20     valorCompra = valorUnitario * cantidad;
21
22     if ( valorCompra > 100000.0 )
23     {
24        desto = valorCompra * 0.05;
25     }
26
27     valorPagar = valorCompra - desto;
28
29     printf("Producto: %s \n", nombreProducto);
30     printf("Valor de la compra: %.2f \n", valorCompra);
31     printf("Valor del descuento: %.2f \n", desto);
32     printf("Valor a pagar: %.2f \n", valorPagar);
33
34     return 0;
35 }
```

Al ejecutar el programa se obtiene:

Primera ejecución

```
Ingrese el nombre del producto: Camisa
Ingrese el valor del producto: 20000.00
Ingrese la cantidad comprada: 6
Producto: Camisa
Valor de la compra: 120000.00
Valor del descuento: 6000.00
Valor a pagar: 114000.00
```

Segunda ejecución

```
Ingrese el nombre del producto: Camisa
Ingrese el valor del producto: 20000.00
Ingrese la cantidad comprada: 4
Producto: Camisa
Valor de la compra: 80000.00
Valor del descuento: 0.00
Valor a pagar: 80000.00
```

Explicación del programa:

Posterior al encabezado del programa en el que se llaman las bibliotecas necesarias, se encuentra la función `main` en la que se declaran inicialmente, las variables que el programa necesita.

A continuación, se utilizan las instrucciones `printf` y `scanf` para mostrar los mensajes que solicitan los datos y capturarlos. Al capturar el nombre del producto con `scanf`, la variable `nombreProducto` no requiere del carácter **&** por ser un tipo especial de variable, mientras que todas las demás sí.

```
9     printf( "Ingrese el nombre del producto: " );
10    scanf( "%s", nombreProducto );
11
12    printf( "Ingrese el valor del producto: " );
13    scanf( "%lf", &valorUnitario );
14
15    printf( "Ingrese la cantidad comprada: " );
16    scanf( "%i", &cantidad );
```

La variable `nombreProducto` es un arreglo[1] de caracteres de longitud 30; sin embargo, este tipo particular de arreglos, se denomina "cadena de caracteres" (cadenas) o en inglés `String`. Las cadenas, son ampliamente utilizadas en la programación y son necesarias para, por ejemplo, almacenar un nombre propio, una dirección, un código alfanumérico, entre otros. La longitud de una cadena, restringe la cantidad máxima de caracteres que pueden ser almacenados, para el caso, el nombre del producto no puede ser mayor a 30 caracteres (incluyendo el carácter obligatorio de fin de cadena que se simboliza como `'\0'`).

Por ahora y mientras se llega al Capítulo 6, las cadenas no serán tratadas como arreglos, sino, simplemente como una cadena de caracteres de un tamaño determinado, como si se tratara de un tipo de dato estándar

[1]El tema de arreglos será tratado en mayor detalle en el Capítulo 6.

(primitivo) en Lenguaje C, tal y como lo son: `int`, `long`, `float`, ... Recuerde que en Lenguaje C, el dato de tipo cadena no existe.

En Lenguaje C las instrucciones `scanf` y `printf` utlilizan el carácter de control `"%s"` para la captura e impresión de cadenas, vea las líneas 10 y 29; aunque también hay otras formas especiales para leer una cadena, las cuales se analizarán más adelante.

Después, en el programa se calcula el valor de la compra; con este valor y mediante la instrucción `if`, se determina si lo comprado por el cliente amerita descuento. Si es así, dicho descuento se calcula dentro del `if`.

```
18      desto = 0.00;
19
20      valorCompra = valorUnitario * cantidad;
21
22      if ( valorCompra > 100000.0 )
23      {
24          desto = valorCompra * 0.05;
25      }
```

En el caso de que el valor de la compra no supere los 100000.0, el programa no calcula descuento alguno. Observe que en la línea 18 se inicializa la variable `desto` en cero.

Por último, se calcula el valor a pagar restando el descuento obtenido del valor de la compra; finalmente se muestran los resultados obtenidos.

```
27 valorPagar = valorCompra - desto;
28
29 printf("Producto: %s \n", nombreProducto);
30 printf("Valor de la compra: %.2f \n", valorCompra);
31 printf("Valor del descuento: %.2f \n", desto);
32 printf("Valor a pagar: %.2f \n", valorPagar);
```

.:Ejemplo 3.2. *Diseñe un programa en Lenguaje C que reciba la nota definitiva de un estudiante en el rango 0.0 a 5.0, obtenida en una materia.*

El programa debe determinar si el estudiante: "`Aprobó el curso`" o "`Reprobó el curso`" informándolo a través de un mensaje. El curso se aprueba con una nota mayor o igual a 3.0.

Análisis del problema:

- **Resultados esperados:** un mensaje que indique si el estudiante aprobó o reprobó el curso.

- **Datos disponibles:** la nota del estudiante.
- **Proceso:** luego del ingreso de la nota definitiva, se usa la estructura `if` para determinar si el estudiante aprobó o reprobó el curso y mostrar el mensaje respectivo. (Ver el árbol de decisión, Figura 3.5).

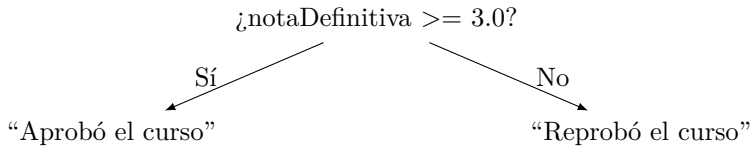


Figura 3.5: Árbol de decisión del Ejemplo 3.2

- **Variables requeridas:**
  - `notaDefinitiva`: guarda la nota del estudiante, ingresada por el usuario.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.6 y se escribe el Programa 3.2.

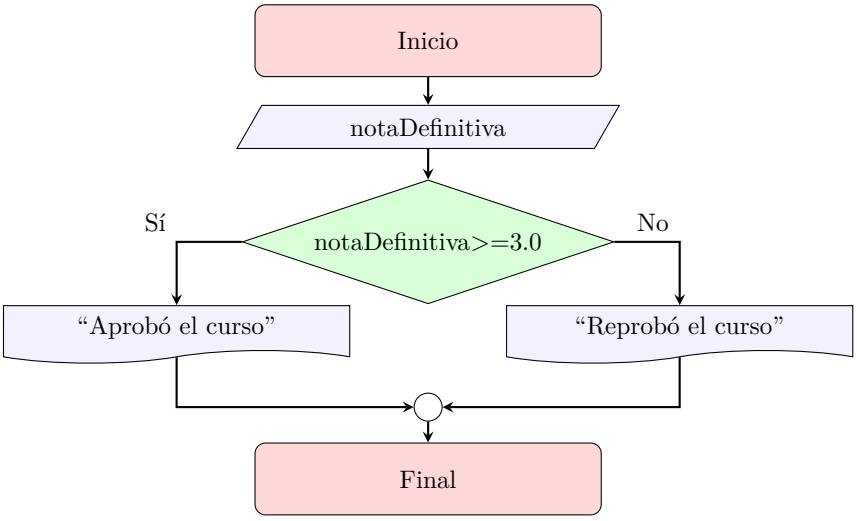


Figura 3.6: Diagrama de flujo del Programa `Definitiva`

Programa 3.2: Definitiva

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float notaDefinitiva;
6
7      printf( "Ingrese la nota definitiva: " );
8      scanf( "%f", &notaDefinitiva );
9
10     printf ( "Su definitiva es: %.2f ", notaDefinitiva );
11
12     if ( notaDefinitiva >= 3.0 )
13     {
14        printf( "Aprobó el curso" );
15     }
16     else
17     {
18        printf( "Reprobó el curso" );
19     }
20
21     return 0;
22 }
```

Explicación del programa:

Primera ejecución

```
Ingrese la nota definitiva: 2.5
Su definitiva es: 2.50 Reprobó el curso
```

Segunda ejecución

```
Ingrese la nota definitiva: 4.3
Su definitiva es: 4.30 Aprobó el curso
```

Explicación del programa:

Este ejercicio contiene los elementos ya trabajados en los ejemplos anteriores, excepto la estructura de decisión; por eso, la explicación se centra solo en esta estructura; en ella puede observarse que se tienen tanto la parte verdadera como la falsa, lo cual indica que la solución emplea una estructura de decisión compuesta.

```
12     if ( notaDefinitiva >= 3.0 )
13     {
14        printf( "Aprobó el curso" );
15     }
```

```
16      else
17      {
18          printf( "Reprobó el curso" );
19      }
```

Luego de que el programa evalúe la expresión `notaDefinitiva >= 3.0`, se determinará si se muestra el mensaje "Aprobó el curso" o "Reprobó el curso". Es importante entender que, en ningún caso el programa mostrará los dos mensajes, ya que la lógica indica que una nota o es mayor o igual a 3.0 o, sencillamente es menor.

.:Ejemplo 3.3. *Escriba un programa al que se le ingresa tanto el nombre como la edad de una persona y el programa muestra si ella es "Mayor de edad" o "Menor de edad". Una persona es mayor de edad si tiene 18 años o más y menor de edad si ocurre lo contrario.*

Análisis del problema:

- **Resultados esperados:** el mensaje que indica si la persona de cuyos datos se ingresaron es "*Mayor de edad*" o "*Menor de edad*" según sea el caso.
- **Datos disponibles:** el nombre del individuo y su edad.
- **Proceso:** luego de ingresar los datos, el programa determina qué mensaje debe mostrar: "*Mayor de edad*" o "*Menor de edad*" de acuerdo con la condición que se debe escribir en la estructura `if` a utilizar. (Ver el árbol de decisión, Figura 3.7).

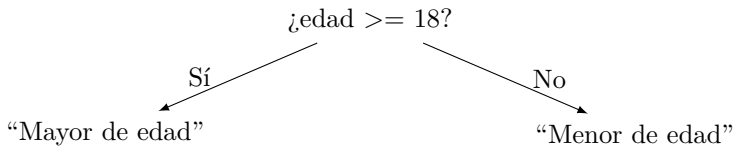


Figura 3.7: Árbol de decisión del Ejemplo 3.3

- **Variables requeridas:**
  - `nombre`: almacena el nombre de la persona.
  - `edad`: almacena la edad de la persona.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.8 y se escribe el Programa 3.3.

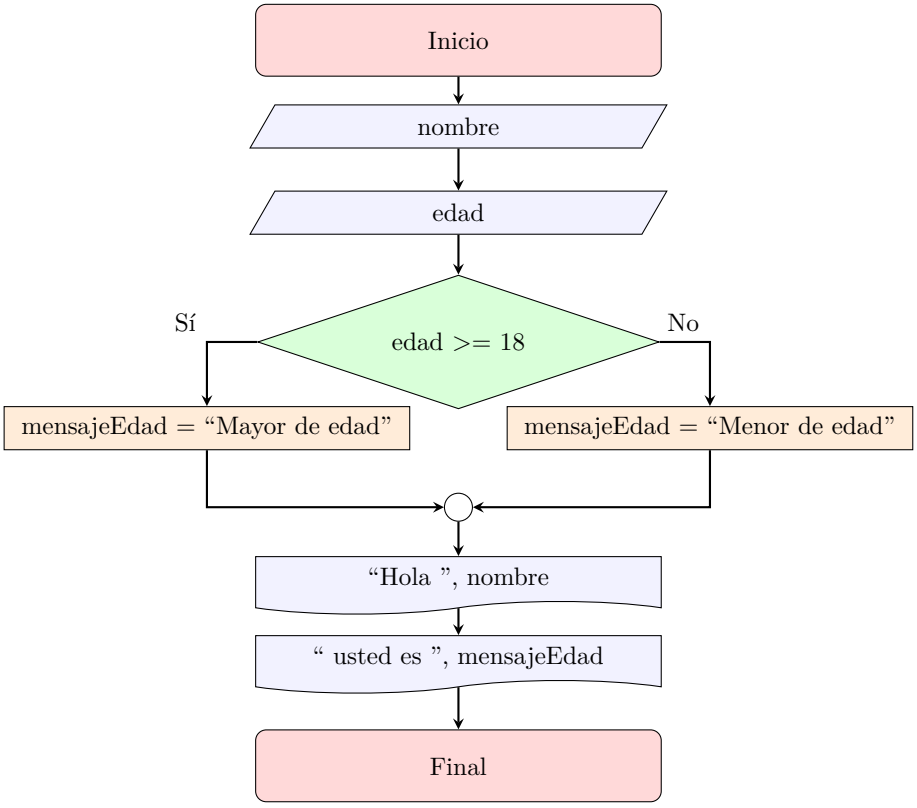


Figura 3.8: Diagrama de flujo del Programa `MayoriaEdad`

Programa 3.3: MayoriaEdad

```
1  #include <stdio.h>
2
3  int main()
4  {
5
6      char nombre[ 20 ];
7      char mensajeEdad[ 20 ];
8      int  edad;
9
10     printf( "Ingrese su nombre: " );
11     fgets( nombre, sizeof( nombre ), stdin );
12
13     printf( "Ingrese su edad: " );
14     scanf( "%i", &edad );
15
16     if ( edad >= 18 )
17     {
18       sprintf ( mensajeEdad, "mayor de edad" );
19     }
```

```
20      else
21      {
22        sprintf ( mensajeEdad, "menor de edad" );
23      }
24
25      printf ( "Hola %s", nombre );
26      printf ( "Usted es %s", mensajeEdad );
27
28      return 0;
29  }
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese su nombre: Juan Pablo
Ingrese su edad: 13
Hola Juan Pablo
Usted es menor de edad
```

Segunda ejecución

```
Ingrese su nombre: Carmen Cilia
Ingrese su edad: 56
Hola Carmen Cilia
Usted es mayor de edad
```

Explicación del programa:

El Programa 3.3 presenta una forma de implementar la solución a este problema. Como es usual, en la primera parte se declaran las variables a utilizar:

```
5       char nombre[ 20 ];
6       char mensajeEdad[ 20 ];
7       int  edad;
```

Posteriormente, se le solicita al usuario que ingrese los datos necesarios, en este caso, el nombre y la edad de la persona. Observe que se utiliza la instrucción `fgets` para capturar el nombre; como alternativa al `scanf` para leer una cadena. Tenga en cuenta que la función `fgets` introduce dentro de la misma cadena el salto de línea '\n', equivalente al `Enter` presionado al ingresar el nombre.

```
9       printf( "Ingrese su nombre: " );
10      fgets( nombre, sizeof( nombre ), stdin );
11
12      printf( "Ingrese su edad: " );
13      scanf( "%i", &edad );
```

Luego, se utiliza el resultado de la evaluación de la expresión relacional (`edad >= 18`), para determinar el mensaje a imprimir, usando una estructura de decisión compuesta.

En las líneas 17 y 21 se usa la instrucción `sprintf`. Esta instrucción trabaja con dos parámetros: el primero es una variable de tipo arreglo de caracteres y el segundo es un mensaje que quedará almacenado en el arreglo de caracteres. Para este ejercicio, se puede observar que los mensajes `"mayor de edad"` y `"menor de edad"` se almacenan en la variable **mensajeEdad**.

```
15      if ( edad >= 18 )
16      {
17        sprintf ( mensajeEdad, "mayor de edad" );
18      }
19      else
20      {
21        sprintf ( mensajeEdad, "menor de edad" );
22      }
```

Finalmente, el programa imprime el mensaje adecuado incluyendo el nombre de la persona.

```
22      printf ( "Hola %s ", nombre );
23      printf ( "Usted es %s", mensajeEdad );
```

Observe que este programa usa dos instrucciones para imprimir el mensaje correspondiente, al final del mismo. No obstante, otra manera de resolver el problema consiste en imprimir el mensaje dentro de la estructura de decisión, de la siguiente forma:

```
if( edad >= 18 )
{
  printf( "Hola %s usted es mayor de edad", nombre );
}
else
{
  printf( "Hola %s usted es menor de edad", nombre );
}
```

Tenga presente que la variable `nombre` contiene el carácter de salto de línea introducido al presionar la tecla `Enter` y por eso el mensaje “usted es ...” se imprime en la línea siguiente.

Una tercera alternativa, sería imprimir la primera parte del mensaje antes de la decisión (ya que este mensaje siempre debe salir) y posteriormente, mostrar el mensaje complementario con la estructura

de decisión. En este caso, tampoco sería necesario utilizar la variable **mensajeEdad**. Recuerde que un problema puede solucionarse adecuadamente de diversas formas.

```
printf( "Hola %s usted es ", nombre );

if( edad >= 18 )
{
  printf( "mayor de edad" );
}
else
{
  printf( "menor de edad" );
}
```

También es posible escribir la estructura de decisión invirtiendo la condición, o sea, `edad <18`; esto obligaría a colocar los mensajes en orden inverso, resolviendo igualmente el problema de forma adecuada.

```
printf( "Hola %s  usted es ", nombre);
if( edad < 18 )
{
  printf( "menor de edad" );
}
else
{
  printf( "mayor de edad" );
}
```

Aclaración:

Si el programador desea eliminar el salto de línea ('\n') de la cadena, puede incluir dentro de los archivos de cabecera, la biblioteca `string.h` y emplear la función `strtok` para eliminar ese carácter u otro que se indique.

La función `strtok` es mucho más interesante que simplemente eliminar un carácter, pero es útil para ello. Se motiva al lector a buscar otros ejemplos de su uso una vez haya estudiado el Capítulo 6 sobre arreglos.

.:Ejemplo 3.4. *Construya un programa en Lenguaje C que permita determinar el mayor entre dos números enteros diferentes ingresados por el usuario.*

Análisis del problema:

- **Resultados esperados:** el número mayor entre los dos números ingresados por el usuario.
- **Datos disponibles:** los dos números enteros.
- **Proceso:** luego de que el usuario ingrese los dos números, se determina el mayor de ellos usando una estructura de decisión y, finalmente se imprime el mayor. (Ver el árbol de decisión, Figura 3.9).

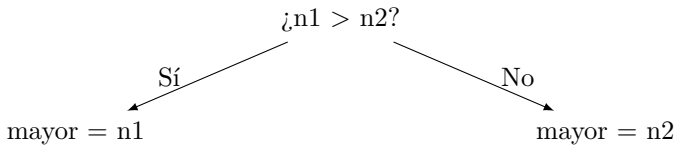


Figura 3.9: Árbol de decisión del Ejemplo 3.4

- **Variables requeridas:**
  - `n1`: primer número ingresado por el usuario.
  - `n2`: segundo número ingresado por el usuario.
  - `mayor`: mayor entre los dos números ingresados.

.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.10 y se escribe el Programa 3.4.

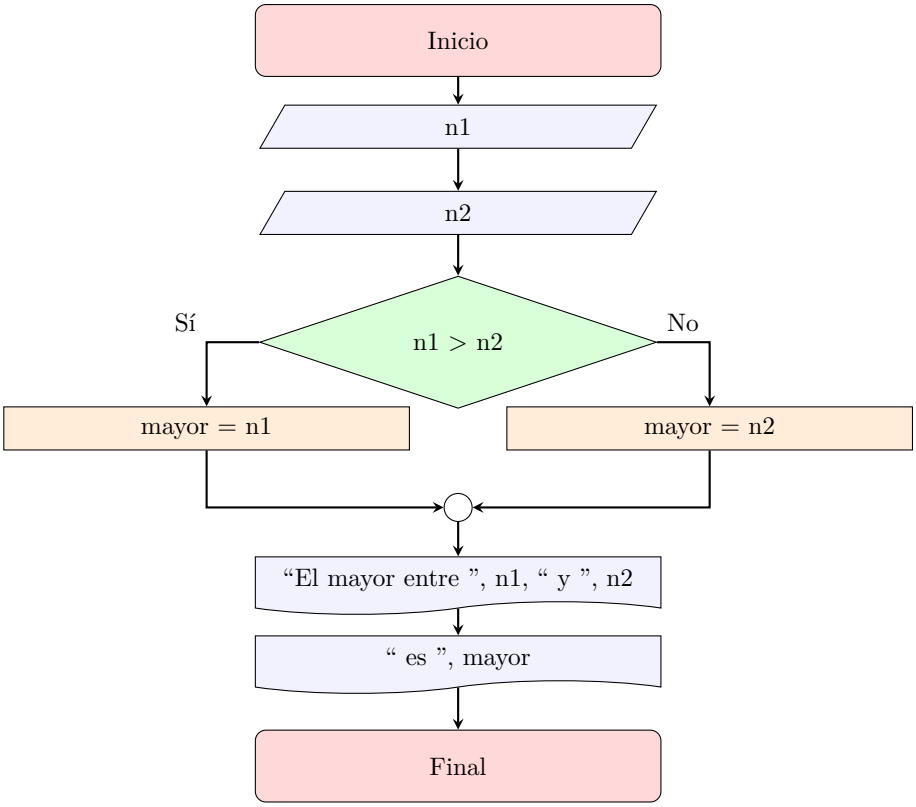


Figura 3.10: Diagrama de flujo del Programa `MayorNumero`

Programa 3.4: MayorNumero

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n1, n2, mayor;
6
7      printf( "Ingrese el primer número: ");
8      scanf( "%i", &n1 );
9
10     printf( "Ingrese el segundo número: ");
11     scanf( "%i", &n2 );
12
13     if ( n1 > n2 )
14     {
15       mayor = n1;
16     }
17     else
18     {
19       mayor = n2;
20     }
21
22     printf( "El mayor entre %i y %i es: %i", n1, n2, mayor);
23
24     return 0;
25 }
```

Explicación del programa:

Primera ejecución

```
Ingrese el primer número: 4
Ingrese el segundo número: 56
El mayor entre 4 y 56 es 56
```

Segunda ejecución

```
Ingrese el primer número: 78
Ingrese el segundo número: 12
El mayor entre 78 y 12 es 78
```

Explicación del programa:

Como en todo programa, se escribe el encabezado que contiene la biblioteca `stdio.h` que se está utilizando en estos ejercicios y se declaran inicialmente las variables que se necesitan:

```
5      int n1, n2, mayor;
```

A continuación, se solicita el ingreso de los dos números enteros que se van a comparar y se almacenan en las variables respectivas. Para ello se usan las instrucciones `printf` y `scanf`:

```
7     printf( "Ingrese el primer número: ");
8     scanf( "%i", &n1 );
9
10    printf( "Ingrese el segundo número: ");
11    scanf( "%i", &n2 );
```

Luego, el programa procede a determinar cuál es el mayor entre `n1` y `n2`, almacenándolo en la variable `mayor`. Para lograr esto, se hace uso de una estructura de decisión que contiene una condición.

```
13    if ( n1 > n2 )
14    {
15      mayor = n1;
16    }
17    else
18    {
19      mayor = n2;
20    }
```

El programa está diseñado para decidir con dos números de diferente valor, si el valor ingresado por el usuario para ambas variables es igual, el programa mostraría a `n2` como mayor valor; no obstante, este sería un resultado tan válido como decir que el mayor es el valor contenido en la variable `n1`.

Finalmente, el programa muestra la información solicitada, incluyendo los números ingresados inicialmente por el usuario.

```
22    printf( "El mayor entre %i y %i es: %i", n1, n2, mayor);
```

.:Ejemplo 3.5. *Construya un programa que indique si un número ingresado por el usuario es par o impar. Un número es par si es divisible por 2 e impar en caso contrario.*

Análisis del problema:

- **Resultados esperados:** un mensaje que diga si el número ingresado es par o impar.
- **Datos disponibles:** un número entero cualquiera.

- **Proceso:** Después de capturar el número ingresado por el usuario, se determina si es par o no por medio de una estructura de decisión que utiliza una condición en la aque se divide modularmente el número ingresado entre 2. (Ver el árbol de decisión, Figura 3.11).

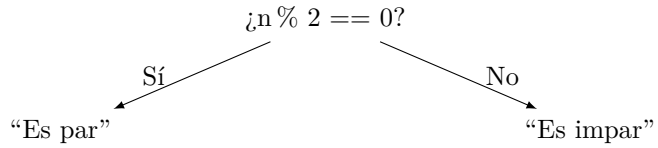


Figura 3.11: Árbol de decisión del Ejemplo 3.5

- **Variables requeridas:**
  - `n`: número entero que el usuario ingresa.

.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.12 y se escribe el Programa 3.5.

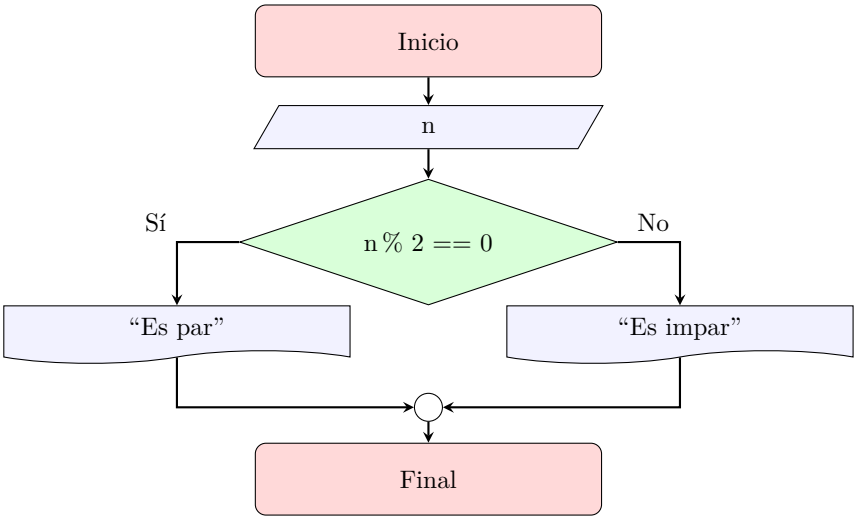


Figura 3.12: Diagrama de flujo del Programa `ParImpar`

Programa 3.5: ParImpar

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6
7      printf( "Ingrese un número: " );
8      scanf( "%i", &n );
9
10     if ( n % 2 == 0 )
11     {
12       printf( "Es par" );
13     }
14     else
15     {
16       printf( "Es impar" );
17     }
18
19     return 0;
20 }
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese un número: 42
Es par
```

Segunda ejecución

```
Ingrese un número: 73
Es impar
```

Explicación del programa:

Luego de escribir el encabezado del programa y la función `main`, se declara la variable y se procede a solicitar al usuario los datos disponibles, en este caso (`n`).

```
7      printf( "Ingrese un número: " );
8      scanf( "%i", &n );
```

Posteriormente, se determina si el número ingresado es par o impar a través de una estructura de decisión cuya condición contiene una operación para encontrar el resto de la división entera entre 2. Si este resto es cero, el número es par, en caso contrario, se concluye que el número es impar.

```
10      if ( n % 2 == 0 )
11      {
12        printf( "Es par" );
13      }
14      else
15      {
16        printf( "Es impar" );
17      }
```

Analice que la condición de la estructura `if` puede escribirse de forma distinta, por ejemplo, si el resultado de la división modular del número entre dos es diferente de cero. En este caso, sería necesario invertir los mensajes de salida.

```
10      if ( n % 2 != 0 )
11      {
12        printf( "Es impar" );
13      }
14      else
15      {
16        printf( "Es par" );
17      }
```

.:Ejemplo 3.6. *Escriba un programa en Lenguaje C que permita saber si un número real (x) se ubica dentro del rango abierto-cerrado (3.5, 7.8].*

Aclaración:

Tenga en cuenta que un límite se considera abierto o cerrado, si el valor de límite está o no incluido también. En el ejemplo que se está tratando, el valor 3.5 no hace parte del rango, pues hace parte del intervalo abierto; esto se identifica fácilmente gracias al paréntesis '('; por otro lado, el valor de 7.8 sí hace parte del rango ya que el intervalo es cerrado, lo que se concluye por haber un corchete ']'. Es importante aclarar que un intervalo puede ser abierto o cerrado en cualquiera de los extremos del mismo.

Análisis del problema:

- **Resultados esperados:** un mensaje que muestra si un número real (x) está dentro del rango abierto-cerrado (3.5, 7.8].
- **Datos disponibles:** un número real que ingresa el usuario.

- **Proceso:** luego del ingreso del valor x, se determina si dicho valor se ubica dentro del rango (Figura 3.13) o, si por el contrario, está por fuera del mismo. (Figura 3.14).

  Este ejemplo se ilustra con en el árbol de la Figura 3.13, no obstante, también puede ilustrarse mediante la Figura 3.14.

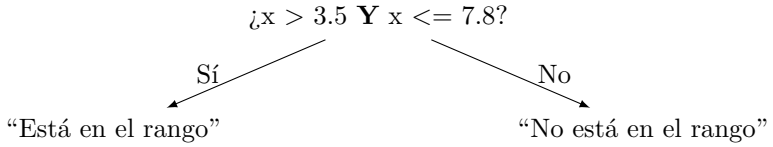


Figura 3.13: Árbol de decisión del Ejemplo 3.6 - Solución 1

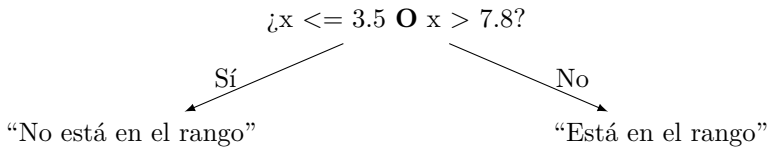


Figura 3.14: Árbol de decisión del Ejemplo 3.6 - Solución 2

Aclaración:

En lógica, hay una ley de operadores lógicos[a] que explica que la negación de una expresión lógica con el operador lógico **Y**, equivale a negar cada operador relacional[b] y cambiar el operador lógico por **O**. La ley también puede aplicarse para lo contrario.

Por ejemplo:
No (x > 3.5 **Y** x <= 7.8) equivale a (x <= 3.5 **O** x > 7.8)
No (x <= 3.5 **O** x > 7.8) equivale a (x > 3.5 **Y** x <= 7.8)

[a] Ley de De Morgan
[b] Lo contrario de: > es <=, < es >=, == es ! = y ! = es ==

- **Variables requeridas:**

  - x: almacena el valor real que el usuario ingresa.

.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.15 y se escribe el Programa 3.6.

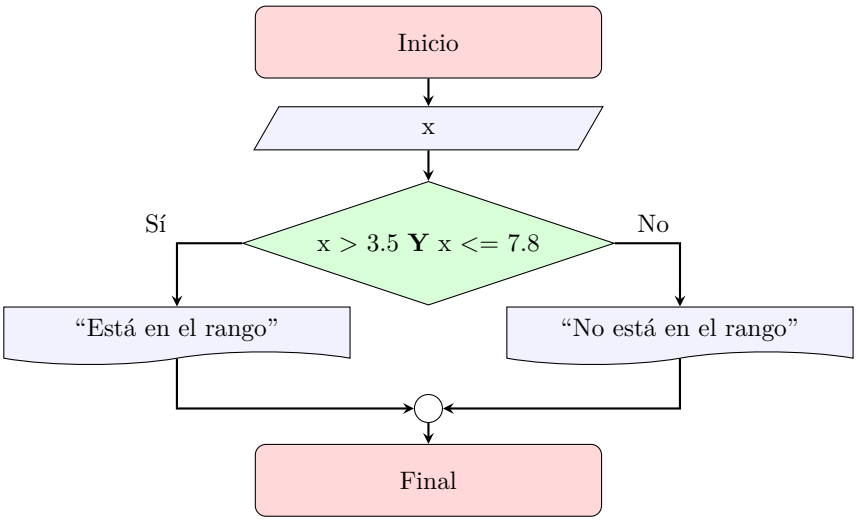


Figura 3.15: Diagrama de flujo del Programa `Rango`

Programa 3.6: Rango

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x;
6
7      printf( "Ingrese un número: " );
8      scanf( "%f", &x );
9
10     if ( x > 3.5 && x <= 7.8 )
11     {
12       printf( "El número %.2f está en el rango", x );
13     }
14     else
15     {
16       printf( "El número %.2f no está en el rango", x );
17     }
18
19     return 0;
20 }
```

Explicación del programa:

Primera ejecución

```
Ingrese un número: 9.3
El número 9.30 no está en el rango
```

Segunda ejecución

```
Ingrese un número: 2.8
El número 2.80 no está en el rango
```

Tercera ejecución

```
Ingrese un número: 4.1
El número 4.10 está en el rango
```

Explicación del programa:

Se escribe, inicialmente, el encabezado del programa y se inicia la función `main`. Posteriormente, se declara la variable `x` y se solicita al usuario el ingreso de su valor. A continuación, se procede a determinar mediante un `if` si el número ingresado se encuentra o no dentro del rango, para mostrar el respectivo mensaje, a través de la instrucción `printf`.

```
10      if ( x > 3.5 && x <= 7.8 )
11      {
12        printf( "El número %.2f está en el rango", x );
13      }
14      else
15      {
16        printf( "El número %.2f no está en el rango", x );
17      }
```

Como se ilustró antes con los árboles de decisión, la condición del **`if`** puede escribirse de forma negada.

```
10      if ( !(x > 3.5 && x <= 7.8 ) )
11      {
12        printf( "El número %.2f no está en el rango", x );
13      }
14      else
15      {
16        printf( "El número %.2f está en el rango", x );
17      }
```

Y, como se explicó en la aclaración, mediante la ley de De Morgan, la condición puede escribirse de la siguiente manera:

```
10      if( x <= 3.5 || x > 7.8 )
11      {
12        printf( "El número %.2f no está en el rango", x );
13      }
14      else
15      {
16        printf( "El número %.2f está en el rango", x );
17      }
```

.:Ejemplo 3.7. *Construya un programa en Lenguaje C que permita saber si un número real (x) se ubica dentro de uno de los rangos que aparecen a continuación: (3.5, 7.8], [9.3, 4.5) y [23.4, 45.3].*

Análisis del problema:

- **Resultados esperados:** un mensaje que muestre si un número real (x) está o no dentro de uno de los siguientes rangos: (3.5, 7.8], [9.3, 4.5) y [23.4, 45.3].

- **Datos disponibles:** un número real ingresado por el usuario.

- **Proceso:** luego del ingreso del valor de x, se determina, empleando un `if` si dicho valor está o no dentro de alguno de los rangos (Figura 3.16).

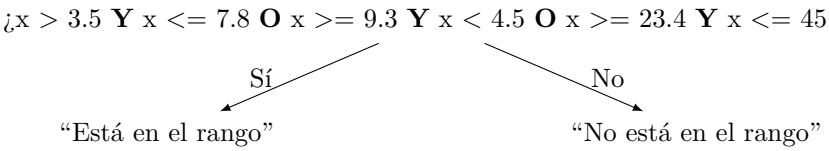


Figura 3.16: Árbol de decisión del Ejemplo 3.7

- **Variables requeridas:**

  - x: almacena el número real que el usuario ingresa.

De acuerdo al análisis planteado, se propone el Programa 3.7.

Programa 3.7: Rangos

```
#include <stdio.h>

int main()
{
    float x;

    printf( "Ingrese un número: " );
    scanf( "%f", &x );

   if ( ( x >   3.5 && x <=  7.8 ) ||
        ( x >=  9.3 && x <   4.5 ) ||
        ( x >= 23.4 && x <= 45.3 ) )
    {
      printf( "\nEl número %.2f está en el rango ", x );
    }
    else
    {
      printf( "\nEl número %.2f no está en el rango ", x );
    }
    return 0;
}
```

Explicación del programa:

Como en todos los programas anteriores, se escribe el encabezado y la función `main`, declarando las variables necesarias y solicitando los datos al usuario. Posteriormente, viene la estructura de decisión `if`, que permite determinar si el número se ubica o no dentro de alguno de los rangos mencionados en la descripción del problema.

Como se observa, la expresión lógica de la decisión no necesita paréntesis para agrupar las expresiones relacionales, ya que el operador `&&` posee mayor precedencia que el operador `||`. Sin embargo, para facilitar la comprensión, se recomienda adicionar los paréntesis.

```
   if ( ( x >   3.5 && x <=  7.8 ) ||
        ( x >=  9.3 && x <   4.5 ) ||
        ( x >= 23.4 && x <= 45.3 ) )
    {
      printf( "\n el número %.2f está en el rango ", x );
    }
    else
    {
      printf( "\n el número %.2f no está en el rango ", x );
    }
```

El diagrama de flujo del Ejemplo 3.7 ilustra la implementación de este algoritmo. Este diagrama es similar al de la Figura 3.15; la única diferencia radica en que se incrementa la expresión lógica que forma la condición, según lo explicado en los párrafos anteriores.

Aclaración:

El Lenguaje C, posee un operador especial que permite simplificar cierto tipo de decisiones compuestas que están relacionadas con la asignación de un valor a una variable, dependiendo del valor de verdad de una expresión. Por ejemplo, asignar el valor de `15` a una variable llamada `b`, solo si el valor ingresado por el usuario es igual a `2`; o asignar `31` en otro caso. Una primera versión, que intencionalmente resumen la decisión empleada a una única línea, se puede observar a continuación:

```c
#include <stdio.h>

int main ()
{
    int a, b;

    printf ( "Ingrese un valor: " );
    scanf ( "%d", &a );

    if ( a == 2 )  {  b = 15; }  else  { b = 31; }

    printf ( "%d", b );
}
```

Es posible simplificar la decisión empleando el operador interrogación (`?`), tal y como se presenta a continuación:

```c
#include <stdio.h>

int main ()
{
    int a, b;

    printf ( "Ingrese un valor: " );
    scanf ( "%d", &a );

    b = (a == 2)? 15 : 31;

    printf ( "%d", b );
}
```

Actividad 3.1

Para los siguientes enunciados, escriba un programa en Lenguaje C y construya el diagrama de flujo correspondiente.

1. Escriba un programa que determine el valor del descuento de un artículo el cual es del 5% solo si el artículo tiene un costo superior al $150.000.

2. El tanque de agua de una casa debe estar siempre entre 250 y 450 litros. Construya un programa que indique si la llave del tanque de agua debe ser abierta o si por el contrario, se debe cerrar.

3. Escriba un programa que, dado un número entero entre 0 y 20 diga si es o no un número primo.

   Recuerde que los números primos menores o iguales a 20 son: $2, 3, 5, 7, 11, 13, 17, 19$.

4. Haga un programa que muestre si un estudiante aprobó o reprobó un curso después de presentar los cinco parciales del mismo (Notas entre 0.0 y 5.0). La definitiva se obtiene con una media aritmética y se aprueba el curso con una definitiva superior a 3.5

5. Cree un programa que permita conocer si una ecuación cuadrática tiene o no solución. En caso de tener solución, encontrarla.

   Recuerde que una ecuación cuadrática se define como:

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

   Y tiene solución si el valor del discriminante (que corresponde al cálculo interno de la raíz cuadrada $b^2 - 4ac$) es mayor o igual a cero y el valor de a es diferente de cero.

6. Escriba un programa que informe si un número entero x, ingresado por el usuario, está por dentro o por fuera del intervalo cerrado-cerrado $[minimoValor,\ maximoValor]$ también ingresados por el usuario.

   Por ejemplo: suponga los valores mínimo y máximo 3 y 7 respectivamente, el valor 5 está dentro, mientras que el valor 8 está por fuera del intervalo.

7. Haga un programa que diga si un número entero x se encuentra por dentro o por fuera de tres intervalos abierto-abierto cuyo rangos que no se interceptan entre sí y sus límites son ingresados por el usuario.

3.2. Decisiones anidadas

Las estructuras de decisión pueden utilizarse dentro de un programa las veces que sean necesarias, incluso dentro de otra estructura de decisión. Con el propósito de explicar estas estructuras de decisión, como ejemplo, suponga que usted está planeando las actividades para realizar el día domingo dependiendo del clima que haga ese día; por tanto, usted analiza lo siguiente:

Si `¿hace calor?` **Entonces**
- Vestir ropa deportiva
- Ir al parque

Sino
- Vestir informalmente
- Ir al cine
- **Si** `¿tengo compañía?` **Entonces**
  - Usar el carro

  Sino
  - Usar transporte público

En el diagrama de flujo de la Figura 3.17 se puede observar de manera ilustrativa que la decisión de qué hacer si: "*¿tengo compañía?*" depende del clima que se presente, concretamente de si no está haciendo calor. En otras palabras, si decide ir al parque, no tiene que considerar el hecho de tener o no compañía para determinar el tipo de transporte a usar.

Ahora, suponga que requiere decidir el tipo de transporte a usar, independiente del lugar a visitar; esto implica que la segunda decisión no debe depender de la primera, es decir, las decisiones serían independientes y simples y no una anidada, como se observa a continuación y como se visualiza en la Figura 3.18.

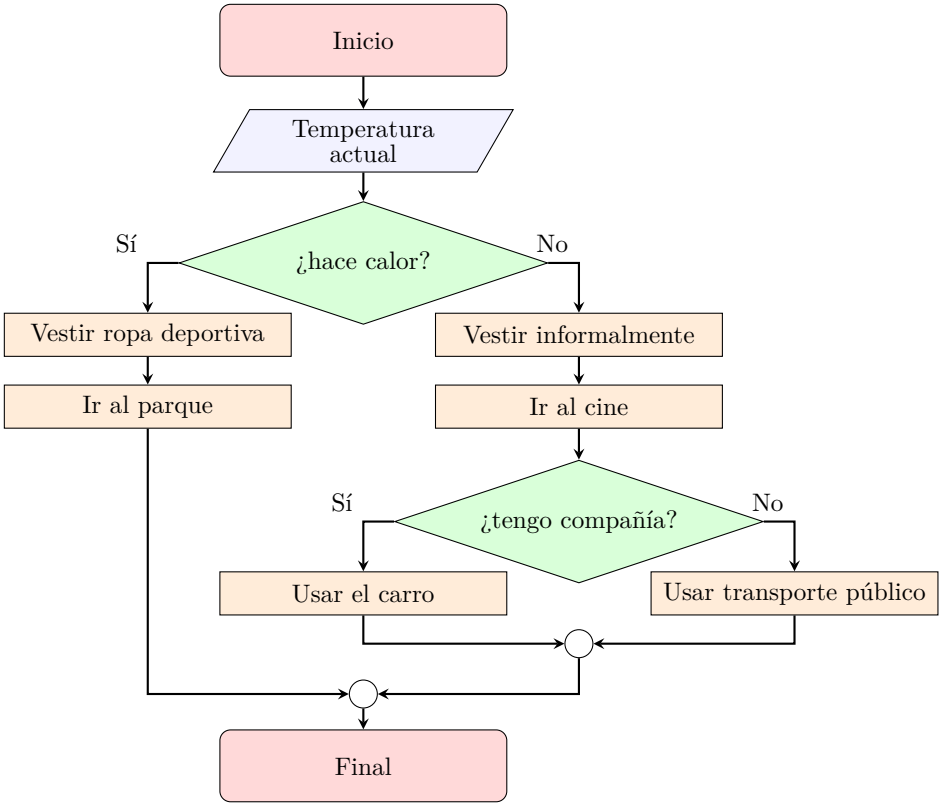


Figura 3.17: Planeando el domingo - Versión 1

Si ¿hace calor? **Entonces**
- Vestir ropa deportiva
- Ir al parque

Sino
- Vestir informalmente
- Ir al cine

Si ¿tengo compañía? **Entonces**
- Usar el carro

Sino
- Usar transporte público

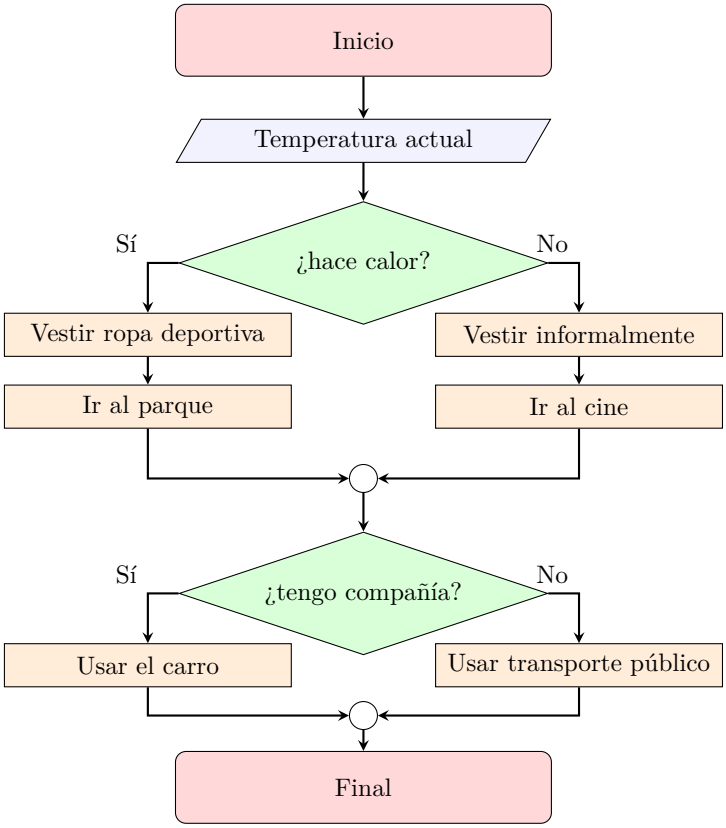


Figura 3.18: Planeando el domingo - Versión 2

Aclaración:

El uso de las decisiones simples o anidadas depende de las características de cada problema que se está resolviendo, sin embargo, cuando se tiene una variable que toma múltiples valores y, con cada valor que tome se pueden llevar a cabo diversas acciones, es importante considerar el uso de una estructura de decisión anidada. Existe una estructura de decisión anidada, cuando dentro de la parte verdadera o falsa de una decisión, hay otra estructura de decisión.

.:Ejemplo 3.8. *Construya un programa en Lenguaje C que imprima un mensaje de acuerdo con el carácter ingresado por el usuario, sin importar si se ingresó en mayúscula o minúscula, según la Tabla 3.1.*

Carácter	Mensaje a imprimir
'a'	"Android"
'i'	"iOS"
otro	"Opción inválida"

Tabla 3.1: Opciones para el Ejemplo 3.8

Análisis del problema:

- **Resultados esperados:** un mensaje de acuerdo con el carácter ingresado por el usuario conforme a la Tabla 3.1.
- **Datos disponibles:** el carácter ingresado por el usuario.
- **Proceso:** después de que el usuario ingrese el carácter, se determina el mensaje a mostrar, como lo indica el árbol de decisión de la Figura 3.19.

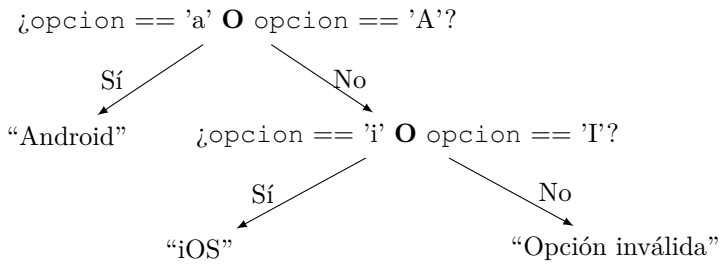


Figura 3.19: Árbol de decisión del Ejemplo 3.8

- **Variables requeridas:**
  - `opcion`: almacena el carácter ingresado por el usuario.

.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.20 y se escribe el Programa 3.8. En este diagrama se omitió el operador lógico **O** en las condiciones, para la opción en mayúsculas.

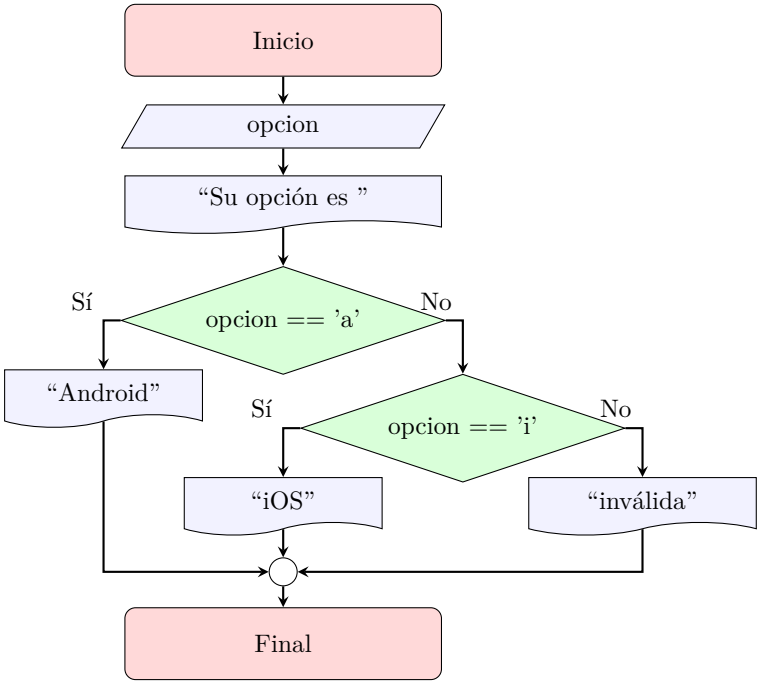


Figura 3.20: Diagrama de flujo `SistemaOperativo`

Programa 3.8: SistemaOperativo

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char opcion;
6
7      printf( "Ingrese un carácter: " );
8      scanf( "%c", &opcion );
9
10     printf( "Su opción es ");
11
12     if ( opcion == 'a' || opcion == 'A' )
13     {
14       printf( "Android" );
15     }
16     else
17     {
18       if ( opcion == 'i' || opcion == 'I' )
19       {
20         printf( "iOS " );
21       }
22       else
23       {
24         printf( "inválida" );
```

```
25        }
26      }
27
28      return 0;
29  }
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese un carácter: i
Su opción es iOS
```

Segunda ejecución

```
Ingrese un carácter: A
Su opción es Android
```

Tercera ejecución

```
Ingrese un carácter: x
Su opción es inválida
```

Explicación del programa:

Posterior al encabezado del programa y a la declaración de la función `main`, se declara la variable que se requiere y se solicita al usuario el ingreso del carácter.

```
1       #include <stdio.h>
2
3       int main()
4       {
5         char opcion;
6
7         printf( "Ingrese un carácter: " );
8         scanf( "%c", &opcion );
```

Luego, mediante una estructura de decisión se determina si el carácter ingresado es la letra ('a' || 'A') para mostrar un mensaje "Android"; si el carácter ingresado es otro, se usa otra decisión para determinar si corresponde a la letra ('i' || 'I') y se muestra "iOS"; si el carácter ingresado es otro, se imprime "inválida".

```
12      if ( opcion == 'a' || opcion == 'A' )
13      {
14        printf( "Android" );
15      }
16      else
17      {
18        if ( opcion == 'i' || opcion == 'I' )
19        {
20          printf( "iOS " );
21        }
22        else
23        {
24          printf( "inválida" );
25        }
26      }
```

.:Ejemplo 3.9. *Escriba un programa en Lenguaje C que muestre un mensaje de acuerdo con la nota definitiva (entre 0.0 y 5.0) obtenida por un estudiante en una asignatura, conforme a la Tabla 3.2.*

Nota	Mensaje a imprimir
< 3.0	"Insuficiente"
<= 3.5	"Aceptable"
<= 4.0	"Sobresaliente"
<= 5.0	"Excelente"

Tabla 3.2: Opciones para el Ejemplo 3.9

Análisis del problema:

- **Resultados esperados:** un mensaje relacionado con la nota definitiva según la Tabla 3.2.
- **Datos disponibles:** la nota definitiva obtenida por el estudiante.
- **Proceso:** se pide al usuario que ingrese la nota definitiva del estudiante, luego se determina el mensaje a mostrar, según el árbol de decisión de la Figura 3.21.

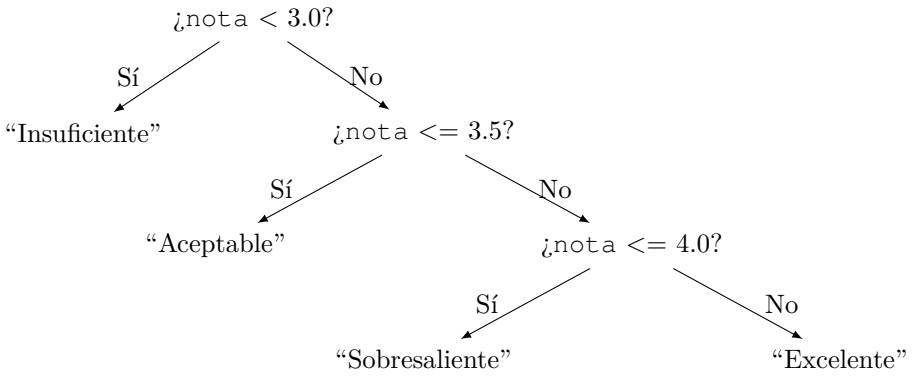


Figura 3.21: Árbol de decisión del Ejemplo 3.8

Se observa que, en el árbol de decisión, no se requiere preguntar si la nota es menor o igual 5.0, ya que con las otras condiciones, todas las posibilidades están cubiertas, y la única posibilidad sería una nota excelente, esto es, una nota definitiva mayor a 4.0 y menor o igual a 5.0.

Cuando en una estructura de decisión anidada, como la que se representa en la anterior figura, no se hace la última pregunta y se asume una acción a realizar, se dice que esta acción se toma por descarte, es decir, si ninguna de las anteriores decisiones arrojó un resultado verdadero, entonces la opción a elegir es la que queda.

- **Variables requeridas:**
  - `nota`: almacena la nota definitiva del estudiante ingresada por el usuario.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 3.22 y se escribe el Programa 3.9.

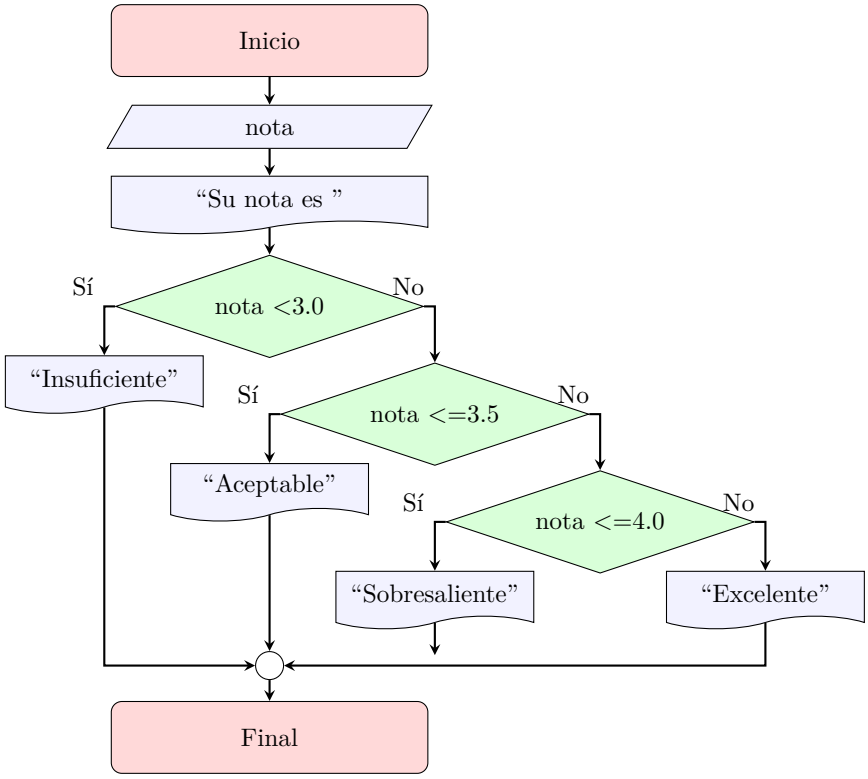


Figura 3.22: Diagrama de flujo del Programa `MensajeNota`

Programa 3.9: MensajeNota

```
#include <stdio.h>

int main()
{
    float nota;

    printf( "Ingrese la nota definitiva: " );
    scanf( "%f", &nota );

    printf( "Su nota es " );
    if ( nota < 3.0 )
    {
      printf( "Insuficiente" );
    }
    else
    {
      if ( nota <= 3.5 )
      {
        printf( "Aceptable " );
      }
      else
      {
        if ( nota <= 4.0 )
        {
           printf( "Sobresaliente" );
        }
        else
        {
           printf( "Excelente" );
        }
      }
    }

    return 0;
}
```

Explicación del programa:

Primera ejecución

```
Ingrese la nota definitiva: 4.7
Su nota es Excelente
```

Segunda ejecución

```
Ingrese la nota definitiva: 3.8
Su nota es Sobresaliente
```

Tercera ejecución

```
Ingrese la nota definitiva: 3.2
Su nota es Aceptable
```

Cuarta ejecución

```
Ingrese la nota definitiva: 1.3
Su nota es Insuficiente
```

Explicación del programa:

Lo primero que se hace es la declaración y lectura de la única variable requerida (líneas de la 7 y 8).

A continuación, se implementa el árbol de decisión de la Figura 3.21 mediante las estructuras de decisión anidadas:

```
11 if ( nota < 3.0 )
12     {
13       printf( "Insuficiente" );
14     }
15     else
16     {
17       if ( nota <= 3.5 )
18       {
19         printf( "Aceptable " );
20       }
21       else
22       {
23         if ( nota <= 4.0 )
24         {
25            printf( "Sobresaliente" );
26         }
27         else
28         {
29            printf( "Excelente" );
30         }
31       }
32     }
```

En este programa, es importante aclarar que la nota debe ser ingresada por el usuario dentro del rango de 0.0 a 5.0. Si la nota es menor a cero, el programa imprime "`Insuficiente`" y "`Excelente`" para notas mayores a 4.0., incluyendo también las superiores a 5.0. En programación, es posible validar el ingreso de los datos para que estén en los rangos correctos; esto se abordará mas adelante en el libro.

.:Ejemplo 3.10. *Escriba un programa que determine el número mayor entre cuatro números enteros ingresados por el usuario.*

Análisis del problema:

- **Resultados esperados:** mostrar el número mayor entre cuatro números que el usuario haya ingresado.
- **Datos disponibles:** los cuatro números (n_1, n_2, n_3 y n_4).
- **Proceso:** Se solicita al usuario el ingreso de cada uno de los cuatro números. Posteriormente y, con el uso de decisiones (Ver árbol de decisión de las Figura 3.23 o 3.24) se encuentra el mayor de ellos y se informa al usuario.

  - Este programa puede resolverse mediante la implementación de diferentes estructuras de decisión, con condiciones sencillas o con complejas que involucran operadores lógicos. La primera versión de este ejercicio contiene estructuras `if` anidadas sin operadores lógicos, como se observa en la Figura 3.23.

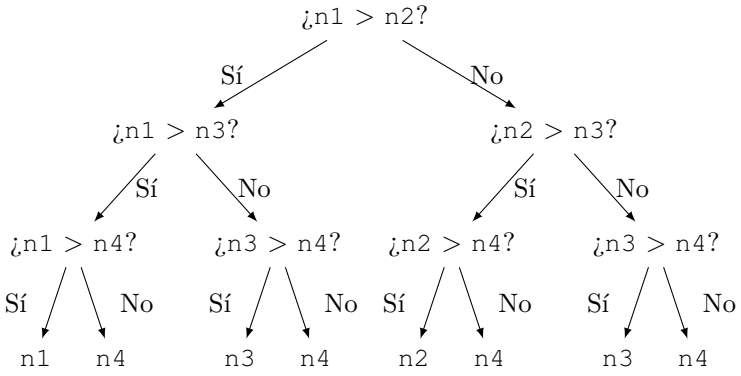


Figura 3.23: Árbol de decisión del Ejemplo 3.10 - Versión 1

  - En la segunda versión de este árbol, se utiliza el operador lógico **Y** en las condiciones que conforman la estructura `if` (ver Figura 3.24).

Como puede observarse, el árbol de la Figura 3.24 es más pequeño que el de la Figura 3.23, sin embargo, desde el punto de vista algorítmico, en algunas ocasiones realiza más comparaciones que el primero[2]; por ejemplo con los valores: $n1 = 4$, $n2 = 3$, $n3 = 2$ y $n4 = 10$, se hacen

[2]Siempre hace tres comparaciones para obtener el resultado.

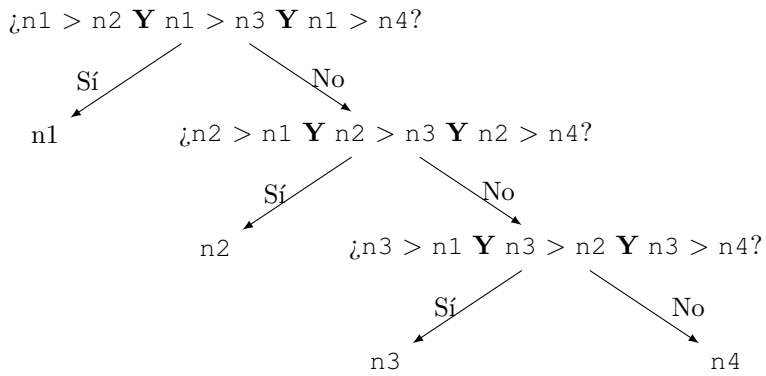


Figura 3.24: Árbol de decisión del Ejemplo 3.10 - Versión 2

cuatro comparaciones, en el mejor de los casos, lo cual lo hace en algunas ocaciones, más ineficiente.

Aclaración:

La mayoría de los lenguajes de programación buscan hacer optimizaciones para ganar velocidad cuando deben evaluar las expresiones lógicas, entre ellas están:

- Si una expresión lógica contiene el operador `&&` y alguna de las comparaciones es falsa, se ignoran las demás, pues ya se sabe que toda la expresión se valuará como falsa.
- Si una expresión lógica contiene el operador `||` y alguna de las comparaciones es verdadera, se ignoran las demás, pues ya se sabe que toda la expresión sera verdadera.

- **Variables requeridas:**
  - `n1`: almacena el primer número.
  - `n2`: almacena el segundo número.
  - `n3`: almacena el tercer número.
  - `n4`: almacena el cuarto número.
  - `mayor`: almacena el mayor valor de los cuatro ingresados.

Basado en el análisis planteado, se propone el Programa 3.10.

Programa 3.10: Mayor4 - Versión 1

```
#include <stdio.h>

int main()
{
    int n1, n2, n3, n4, mayor;

    printf( "Ingrese el primer número: " );
    scanf( "%i", &n1 );

    printf("Ingrese el segundo número: " );
    scanf( "%i", &n2 );

    printf( "Ingrese el tercer número: " );
    scanf( "%i", &n3 );

    printf( "Ingrese el cuarto número: " );
    scanf( "%i", &n4 );


    if ( n1 > n2 )
    {
      if ( n1 > n3 )
      {
        if ( n1 > n4 )
        {
            mayor = n1;
        }
        else
        {
            mayor = n4;
        }
      }
      else
      {
        if ( n3 > n4 )
        {
          mayor = n3;
        }
        else
        {
          mayor = n4;
        }
      }
    }
    else
    {
      if ( n2 > n3 )
      {
```

```
49          if ( n2 > n4 )
50          {
51              mayor = n2;
52          }
53          else
54          {
55              mayor = n4;
56          }
57        }
58        else
59        {
60          if ( n3 > n4 )
61          {
62            mayor = n3;
63          }
64          else
65          {
66            mayor = n4;
67          }
68        }
69      }
70
71      printf( "El número mayor es %i ", mayor );
72
73      return 0;
74  }
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese el primer número: 27
Ingrese el segundo número: 11
Ingrese el tercer número: 343
Ingrese el cuarto número: 20
El mayor de todos es 343
```

Segunda ejecución

```
Ingrese el primer número: 31
Ingrese el segundo número: 5
Ingrese el tercer número: 73
Ingrese el cuarto número: 19
El mayor de todos es 73
```

Explicación del programa:

Como en todos los programas que se han escrito, inicialmente se declaran

las variables requeridas, adicionlamente se solicitan los cuatro números a comparar.

```
5       int n1, n2, n3, n4, mayor;
6
7       printf( "Ingrese el primer número: " );
8       scanf( "%i", &n1 );
9
10      printf("Ingrese el segundo número: " );
11      scanf( "%i", &n2 );
12
13      printf( "Ingrese el tercer número: " );
14      scanf( "%i", &n3 );
15
16      printf( "Ingrese el cuarto número: " );
17      scanf( "%i", &n4 );
```

A continuación, por medio de las estructuras de decisión anidadas, se determina el número mayor que será impreso al final del programa.

Como se mencionó anteriormente, una alternativa a las estructuras de decisión anidadas con las que se resolvió el ejercicio, consiste en reemplazar todas las líneas de código donde se encuentran estas estructuras (líneas de la 20 a la 69) por la siguiente porción de código, lo que permite obtener el mismo resultado.

```
if ( n1 > n2 && n1 > n3 && n1 > n4 )
{
    mayor = n1;
}
else
{
    if ( n2 > n1 && n2 > n3 && n2 > n4 )
    {
        mayor = n2;
    }
    else
    {
        if ( n3 > n1 && n3 > n2 && n3 > n4 )
        {
            mayor = n3;
        }
        else
        {
            mayor = n4;
        }
    }
}
```

La lógica de programación otorga la facultad de encontrar otras alternativas para resolver este problema. Enseguida se propone una tercera versión que, va descartando aquellos números que no son mayores a medida que se van realizando las comparaciones, lo que va reduciendo el número de comparaciones en las siguientes líneas de código.

```
if ( n1 > n2 && n1 > n3 && n1 > n4 )
{
    mayor = n1;
}
else
{
    if ( n2 > n3 && n2 > n4 )
    {
        mayor = n2;
    }
    else
    {
        if ( n3 > n4 )
        {
            mayor = n3;
        }
        else
        {
            mayor = n4;
        }
    }
}
```

Aclaración:

El ejemplo de obtener el mayor de cuatro números muestra claramente que existen diferentes alternativas para resolver un problema mediante un algoritmo. Está en el programador determinar cuál sería la "mejor" forma de escribir un programa o algoritmo de manera que sea lo más simple y eficiente posible.

En este ejercicio de encontrar el mayor de cuatro números se puede observar que, a mayor cantidad de variables y de condiciones, mayor será la complejidad del programa, lo que puede ocasionar dificultad para su legibilidad y posiblemente, reduzca su eficiencia. Es por esto que se propone otra alternativa algorítmica que soluciona el problema utilizando solo condiciones simples, de la siguiente forma:

```
mayor = n1;
if ( n2 > mayor )
{
    mayor = n2;
}

if ( n3 > mayor )
{
    mayor = n3;
}

if ( n4 > mayor )
{
    mayor = n4;
}
```

Como puede verse, la solución que se obtiene a través de estas líneas de código es muy sencilla y se puede ampliar a cualquier cantidad de variables sin hacer más complejo el algoritmo.

.:Ejemplo 3.11. *La empresa de servicios públicos, aguas y alcantarillado de la ciudad necesita un programa que liquide las facturas de sus clientes mensualmente. El cobro de las facturas es como sigue: se cobra el cargo fijo, el valor de la cantidad de agua consumida (la cantidad de metros cúbicos consumidos durante el mes) y el valor de la recolección de basuras y alcantarillado. Estos cobros dependen del estrato socioeconómico al que pertenece el predio, conforme a la tabla que aparece a continuación:*

Estrato	Cargo Fijo	Metro^3 consumido	Basuras y alcantarillado
1	*$2500*	*$2200*	*$5500*
2	*$2800*	*$2350*	*$6200*
3	*$3000*	*$2600*	*$7400*
4	*$3300*	*$3400*	*$8600*
5	*$3700*	*$3900*	*$9700*
6	*$4400*	*$4800*	*$11000*

Escriba un programa en Lenguaje C que, al ingresar el estrato socioeconómico del predio y la cantidad de metros cúbicos de agua consumidos, calcule y muestre el valor a pagar por el servicio.

Análisis del problema:

- **Resultados esperados:** el valor a pagar por el servicio de agua, incluyendo y mostrando el valor del cargo fijo, el valor del consumo,

el del servicio de recolección de basura y alcantarillado y el valor total a pagar por el cliente.

- **Datos disponibles:** el estrato socioeconómico del predio y la cantidad de metros cúbicos consumidos por el cliente.
- **Proceso:** luego del ingreso de los datos, basado en el estrato socioeconómico, se calculan los valores a pagar, a través de una estructura de decisión anidada que encuentre los valores del cargo fijo, del consumo (que se obtiene multiplicando la cantidad de metros consumidos por el valor de cada metro dependiendo del estrato) y, además, calcule el valor de la recolección de basuras y alcantarillado. Al tener todos los valores anteriores, se suman para hallar el valor total a pagar. Al finalizar, se muestran al usuario todos los resultados encontrados.
- **Variables requeridas:**
  - `estrato`: corresponde al estrato socioeconómico del predio.
  - `cantidad`: cantidad de metros cúbicos consumidos por el cliente.
  - `cargoFijo`: valor del cargo fijo.
  - `valorConsumo`: valor del consumo.
  - `valorRecoleccion`: valor de la recolección de las basuras y alcantarillado.
  - `totalPago`: valor total a pagar por el servicio.

De acuerdo al análisis realizado, se propone el Programa 3.11.

Programa 3.11: FacturaAgua

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char  estrato;
6      float cantidad, cargoFijo, consumo,
7            valorRecoleccion, totalPago;
8
9      printf( "Ingrese estrato socioeconómico del predio: " );
10     scanf( "%c", &estrato );
11
12     printf( "Cantidad de metros consumidos: " );
13     scanf( "%f", &cantidad );
14
15     if ( estrato == '1' )
```

```
16      {
17          cargoFijo = 2500;
18          consumo = cantidad * 2200;
19          valorRecoleccion = 5500;
20      }
21      else
22      {
23          if ( estrato == '2' )
24          {
25              cargoFijo = 2800;
26              consumo = cantidad * 2350;
27              valorRecoleccion = 6200;
28          }
29          else
30          {
31              if ( estrato == '3' )
32              {
33                  cargoFijo = 3000;
34                  consumo = cantidad * 2600;
35                  valorRecoleccion = 7400;
36              }
37              else
38              {
39                 if ( estrato == '4' )
40                  {
41                      cargoFijo = 3300;
42                      consumo = cantidad * 3400;
43                      valorRecoleccion = 8600;
44                  }
45                   else
46                   {
47                      if ( estrato == '5' )
48                      {
49                          cargoFijo = 3700;
50                          consumo = cantidad * 3900;
51                          valorRecoleccion = 9700;
52                      }
53                      else
54                      {
55                          cargoFijo = 4400;
56                          consumo = cantidad * 4800;
57                          valorRecoleccion = 11000;
58                      }
59                   }
60              }
61          }
62      }
63      totalPago = cargoFijo + consumo + valorRecoleccion;
64
```

```
65      printf( "Valor del cargo fijo: %.2f\n", cargoFijo );
66      printf( "Valor del consumo: %.2f\n", consumo );
67      printf( "Valor recolección: %.2f\n", valorRecoleccion );
68      printf( "Total a pagar: %.2f\n", totalPago );
69
70      return 0;
71  }
```

Explicación del programa:

Como en todos los programas anteriores, se escribe el encabezado y luego la función `main`. Inmediatamente, se declaran las variables con sus tipos de datos y se solicitan los datos conocidos, en este caso `estrato` y `cantidad`.

```
5       char  estrato;
6       float cantidad, cargoFijo, consumo,
7             valorRecoleccion, totalPago;
8
9       printf( "Ingrese estrato socioeconómico del predio: " );
10      scanf( "%c", &estrato );
11
12      printf( "Cantidad de metros consumidos: " );
13      scanf( "%f", &cantidad );
```

A continuación, el programa usa una estructura `if` anidada que va encontrando los cálculos adecuados a realizar según el estrato socioeconómico ingresado. El primer `if`, indaga si el `estrato` ingresado es '1', si esto llega a ocurrir, se hacen los cálculos correspondientes a ese estrato. Si el `estrato` no es '1', dentro del `else` del primer `if`, se ubica una segunda estructura `if` que indaga si el `estrato` es '2', de llegar a ocurrir esto, se hacen los cálculos correspondientes a este segundo estrato pero, si el `estrato` no es '2', por el `else` del segundo `if`, se tiene un tercer `if` que pregunta si el `estrato` es '3'. De esta misma manera, el programa posteriormente indaga los estratos '4' y '5' para hacer los cálculos adecuados. Observe que, al finalizar el último `if`, en el `else`, se llevan a cabo los cálculos del estrato '6'; esto se escribe de esta forma, ya que si el programa no ingresó a ninguna de las partes verdaderas de los `if` anteriores, es por que el estrato no es ni '1', '2', '3', '4', '5', por lo cuál solo queda por descarte, el estrato '6'.

```
15  if ( estrato == '1' )
16  {
17      cargoFijo = 2500;
18      consumo = cantidad * 2200;
19      valorRecoleccion = 5500;
```

```
20  }
21  else
22  {
23     if ( estrato == '2' )
24     {
25        cargoFijo = 2800;
26        consumo = cantidad * 2350;
27        valorRecoleccion = 6200;
28     }
29     else
30     {
31         if ( estrato == '3' )
32         {
33             cargoFijo = 3000;
34             consumo = cantidad * 2600;
35             valorRecoleccion = 7400;
36         }
37         else
38         {
39              if ( estrato == '4' )
40              {
41                  cargoFijo = 3300;
42                  consumo = cantidad * 3400;
43                  valorRecoleccion = 8600;
44              }
45              else
46              {
47                  if ( estrato == '5' )
48                  {
49                      cargoFijo = 3700;
50                      consumo = cantidad * 3900;
51                      valorRecoleccion = 9700;
52                  }
53                  else
54                  {
55                      cargoFijo = 4400;
56                      consumo = cantidad * 4800;
57                      valorRecoleccion = 11000;
58                  }
59              }
60         }
61     }
62  }
```

En las anteriores líneas de código la indentación y el órden juegan un papel importante, ya que facilitan la comprensión del programa. Observe también como cada `if` desarrolla su parte verdadera como falsa cerrando al final, las llaves que quedan abiertas. Esto muestra la correcta utilización de las estructuras de decisión.

Actividad 3.2

Para los siguientes enunciados, escriba un programa en Lenguaje C y diseñe el diagrama de flujo correspondiente.

1. Escriba un programa que indique el valor del descuento de un artículo dependiendo de su tipo:

Tipo	**Porcentaje de descuento**
Textil	0 %
Electrodoméstico	3.7 %
Elementos de cocina	4.2 %
Video juego	7.8 %

Tabla 3.3: Tabla de descuento por tipo de artículo

2. Construya un programa que muestre el valor del descuento de un artículo dependiendo de su valor:

Rango de valores	**Porcentaje de descuento**
$0.0 hasta $100.000	0 %
Más de $100.000 hasta $225.000	1.5 %
Más de $225.000 hasta $375.000	3.8 %
Más de $375.000	10.3 %

Tabla 3.4: Tabla de descuento según el rango de valores

3. En un taller de mantenimiento de computadores se requiere saber el costo del mantenimiento a cobrar a los clientes. Este costo se puede observar en la tabla 3.5

Sistema operativo	Tipo de mantenimiento
Windows	`basico = 30000, intermedio = 40000, avanzado = 50000`
Linux	`basico = 45000, intermedio = 55000, avanzado = 65000`
IOS	`basico = 42000, intermedio = 52000, avanzado = 62000`
Android	`basico = 47000, intermedio = 57000, avanzado = 67000`

Tabla 3.5: Tabla de valores de mantenimiento según S.O.

El tipo de mantenimiento depende que las acciones a realizar con el equipo y la complejidad de las mismas. Haga un programa que, dado el sistema operativo del equipo y el tipo de mantenimiento a realizar, permita conocer el costo.

3.3. Decisiones múltiples

Este tipo de decisiones son una forma abreviada y "más simple" de escribir programas que requieren de estructuras de decisión anidadas. También conocida como "estructura selectiva", corresponde a una estructura alternativa para escribir un tipo especial de decisión anidada, la que se usa para seleccionar un opción entre un conjunto de posibilidades; por tanto, toda decisión múltiple puede escribirse como una decisión anidada, pero no toda decisión anidada se puede escribir como una decisión múltiple.

La sintaxis para escribir esta estructura múltiple se presenta a continuación, en la porción de código en Lenguaje C.

```
1 switch( selector )
2 {
3   case valor1: InstruccionA1-1;
4                InstruccionA1-2;
5                ...
6                InstruccionA1-n1;
7                break;
```

```
8
9    case valor2: InstruccionA2-1;
10                InstruccionA2-2;
11                ...
12                InstruccionA2-n2;
13                break;
14                ...
15
16   case valork: InstruccionAk-1;
17                InstruccionAk-2;
18                ...
19                InstruccionAk-nk;
20                break;
21
22   default:     InstruccionOtroCaso-1;
23                InstruccionOtroCaso-2;
24                ...
25                InstruccionOtroCaso-n;
26                break;
27 }
```

En esta porción de código puede notarse que existen un conjunto `k` de valores (`valor1` hasta la `valork`), que corresponden a los posibles valores que tomaría la variable `selector`.

Por cada valor, se encuentra un conjunto independiente de instrucciones que está delimitado desde la instrucción (`case valor:`) hasta la palabra reservada `break` que cierra cada opción. Todas las instrucciones delimitadas se ejecutan cuando el contenido de la variable denominada `selector` coincide con el valor en cuestión.

Aclaración:

Observe que la estructura selectiva tiene una llave para abrir la estructura y una para cerrarla.

Es posible omitir la instrucción `break`. Cuando esto ocurre el programa continúa la ejecución de las instrucciones del siguiente valor y así sucesivamente hasta encontrar un `break` o bien, hasta encontrar la llave que cierra la decisión múltiple. Esta característica permite crear ciertos comportamientos deseables en un algoritmo.

Si el contenido de la variable `selector` no incide con ninguno de los valores descritos con la palabra `case`, se ejecutan las instrucciones

de la sección denominada `default`, misma que se escribe al final de la estructura.

En Lenguaje C, el tipo de dato de la variable `selector` está limitada solamente a: `int` y `char`. Recuerde utilizar comillas simples si la variable `selector` fué declarada como `char`. Ejemplo:

```
switch( categoria )
{
  case 'A': InstruccionA1-1;
            InstruccionA1-2;
            ...
            InstruccionA1-n1;
            break;

  case 'B': InstruccionA2-1;
            ...
            InstruccionA2-n2;
            break;
            ...

 default:   InstruccionOtroCaso-1;
            ...
            InstruccionOtroCaso-n;
            break;
}
```

La estructura de decisión anidada que es equivalente a la estructura de decisión múltiple o selectiva es la siguiente:

```
if( selector == valor1 )
{
  InstruccionA1-1;
  InstruccionA1-2;
  ...
  InstruccionA1-n1;
}
else
{
  if( selector == valor2 )
  {
    InstruccionA2-1;
    InstruccionA2-2;
    ...
    InstruccionA2-n2;
  }
  else
  {
    ...

```

```
21      if( selector == valork )
22      {
23        InstruccionAk-1;
24        InstruccionAk-2;
25        ...
26        InstruccionAk-nk;
27      }
28      else
29      {
30        InstruccionOtroCaso-1;
31        InstruccionOtroCaso-2;
32        ...
33        InstruccionOtroCaso-n;
34      }
35    }
36  }
```

Por su parte, la forma general de la estructura selectiva en un diagrama de flujo, se ve en la Figura 3.25.

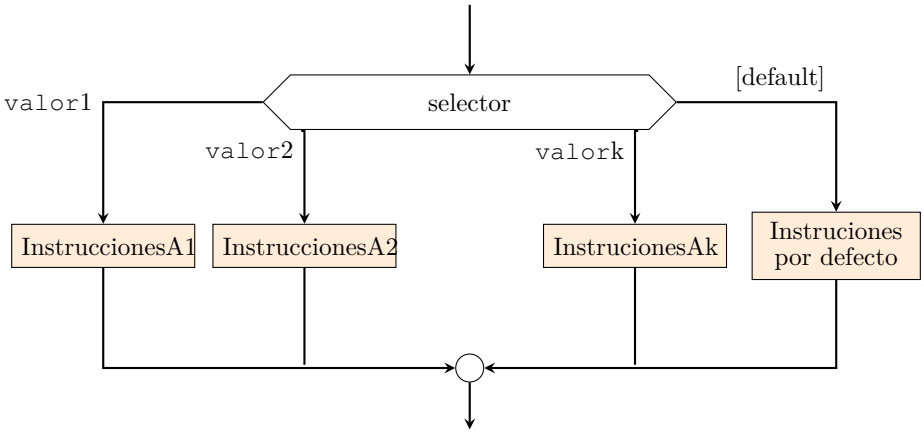


Figura 3.25: Decisiones múltiples - Estructura General

Tenga en cuenta que el diagrama de flujo de la decisión múltiple (Figura 3.25) ilustra un algoritmo en el que todas las instrucciones `case` cierran con su respectivo `break`. Si un (`break`) fuera omitido, por ejemplo en `valor2`, el diagrama general se modificaría al presentado en la Figura 3.26.

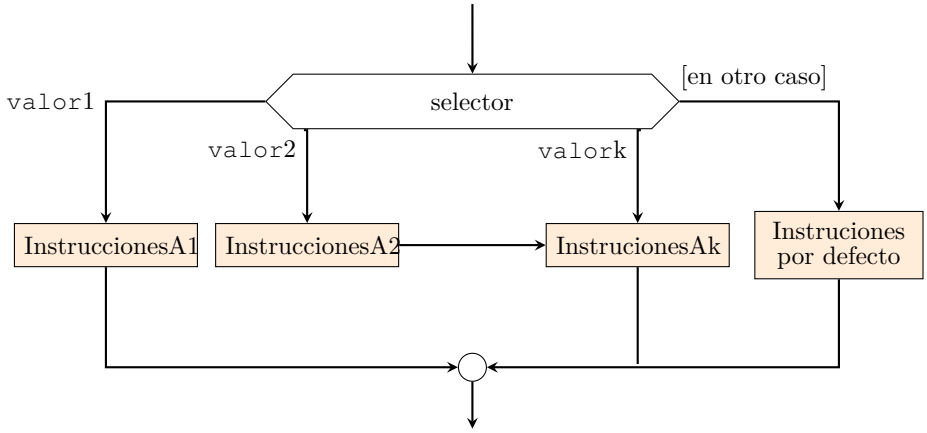


Figura 3.26: Decisiones múltiples - Sin el segundo `break`

Como ya se ha mencionado, las decisiones múltiples son una forma alternativa de escribir ciertas decisiones anidadas. A continuación, se muestran algunos ejemplos para aplicarlas.

.:Ejemplo 3.12. *Reescribir el Ejemplo 3.8 pero esta vez, utilizar decisiones múltiples.*

De acuerdo al análisis realizado para el Ejemplo 3.8, se propone el Programa 3.12.

Programa 3.12: SistemaOperativo

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char opcion;
6
7      printf( "Ingrese un carácter: " );
8      scanf( "%c", &opcion );
9
10     printf( "Su opción es: " );
11
12     switch ( opcion )
13     {
14         case 'a':
15         case 'A': printf( "Android " );
16                   break;
17
```

```
18          case 'i':
19          case 'I': printf( "iOS " );
20                    break;
21
22          default: printf( "Inválida " );
23                   break;
24      }
25
26      return 0;
27  }
```

Explicación del programa:

Se obtiene exactamente el mismo resultado del ejemplo original.

Primera ejecución:

```
Ingrese un carácter: i
Su opción es iOS
```

Segunda ejecución:

```
Ingrese un carácter: A
Su opción es Android
```

Tercera ejecución:

```
Ingrese un carácter: x
Su opción es inválida
```

Explicación del programa:

En este ejemplo se omitió el `break` del primer y tercer valor (líneas 15 y 19), debido a que se necesita ejecutar la misma instrucción de impresión para los casos en donde el valor es una letra mayúscula o minúscula.

Observe que la variable `selector` es de tipo `char`, lo cual implica que cada valor debe ir entre comillas simples (' '); si fuera `int`, solo se escribiría el número.

Aclaración:

Una decisión múltiple es una estructura donde una variable denominada `selector` almacena uno de los valores definidos en la estructura que, al coincidir con uno de los `case` ejecuta el conjunto de instrucciones asociadas a dicho valor.

En Lenguaje C no es posible utilizar decisiones múltiples con operadores relacionales u operadores lógicos; Esto significa que no pueden usarse caracteres como '>', '<', entre otros.

.:Ejemplo 3.13. *Construya un programa en Lenguaje C que permita saber el valor del descuento de un artículo de acuerdo con su tipo. Los tipos disponibles para los artículos y sus descuentos están en la Tabla 3.6.*

Tipo	Descuento
1	12.5 %
2	8.3 %
3	3.2 %
Otro	0.0 %

Tabla 3.6: Opciones para el Ejemplo 3.13

Análisis del problema:

- **Resultados esperados:** un mensaje que muestre el valor del descuento para un artículo de acuerdo al tipo al que pertenezca.
- **Datos disponibles:** el valor del artículo y el tipo al que pertenece dicho artículo.
- **Proceso:** luego de solicitar el valor del artículo y su tipo, se utiliza una estructura de decisión múltiple que permite determinar el porcentaje de descuento para ese artículo según su tipo para, posteriormente, con el porcentaje calcular el valor del descuento e imprimirlo.
- **Variables requeridas:**
  - `tipo`: almacena el tipo de artículo.

- `valor`: almacena el valor del artículo.
- `descuento`: representa el valor del descuento.
- `porcentaje`: almacena el porcentaje de descuento asignado al artículo de acuerdo al tipo.

De acuerdo al análisis planteado, se propone el Programa 3.13.

Programa 3.13: DescuentoArticulo

```
#include <stdio.h>

int main()
{
    char  tipo;
    float valor, descuento, porcentaje;

    printf( "Ingrese el tipo de artículo: " );
    scanf( "%c", &tipo );

    printf( "Ingrese el valor del artículo: " );
    scanf( "%f", &valor );

    switch ( tipo )
    {
        case '1': porcentaje = 0.125;
                  break;

        case '2': porcentaje = 0.083;
                  break;

        case '3': porcentaje = 0.032;
                  break;

        default:  porcentaje = 0.0;
                  break;
    }

    descuento = valor * porcentaje;

    printf( "El valor del descuento es: %.2f ", descuento );

    return 0;
}
```

Explicación del programa:

```
Ingrese el tipo de artículo: 2
Ingrese el valor del artículo: 2500
El valor del descuento es: 207.50
```

Explicación del programa:

Luego de hacer todas las declaraciones básicas que requiere el programa, se solicitan los datos disponibles y se procede a encontrar el descuento que se le debe aplicar al artículo. Observe que la variable `tipo` se utiliza como `selector` y que al ser de tipo `char`, en cada `case`, el valor debe ir entre comillas simples. La instrucción `default` se usó para colocar un cero al porcentaje en el caso de que el tipo no corresponda con ninguno de los valores correctos.

```
14    switch ( tipo )
15   {
16      case '1': porcentaje = 0.125;
17                break;
18
19      case '2': porcentaje = 0.083;
20                break;
21
22      case '3': porcentaje = 0.032;
23                break;
24
25     default: porcentaje = 0.0;
26              break;
27   }
```

Una vez se ha encontrado el porcentaje, se realiza el cálculo del valor del descuento multiplicando el porcentaje encontrado por el valor del artículo. Finalmente, se muestra el valor del `descuento`.

```
29 descuento = valor * porcentaje;
30
31 printf( "El valor del descuento es: %.2f ", descuento );
32
33 return 0;
```

.:Ejemplo 3.14. *Escriba un programa en Lenguaje C que, a partir de la letra inicial del nombre de uno de los departamentos del eje cafetero, muestre el nombre de la capital de ese departamento.*

En caso de que se ingrese otra letra, el programa deberá mostrar que ese departamento no hace parte de dicha zona.

Análisis del problema:

- **Resultados esperados:** un mensaje mostrando el nombre de la capital de uno de los departamentos del eje cafetero o un mensaje

que indique que la letra ingresada no corresponde a uno de los departamentos señalados.

- **Datos disponibles:** la letra inicial del nombre del departamento.
- **Proceso:** después de solicitar al usuario la letra inicial del nombre de un departamento, por medio de una estructura de decisión múltiple se obtiene el nombre de la capital y se imprime. Si la letra ingresada no corresponde a la inicial del nombre de un departamento válido, se muestra un mensaje indicando que lo ingresado no es un departamento del eje cafetero.
- **Variables requeridas:**
  - `departamento`: almacena la letra inicial del nombre del departamento del que se quiere saber su capital.

Conforme al análisis planteado, se propone el Programa 3.14.

Programa 3.14: Capitales

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

int main()
{
    char departamento;

    printf( "Ingrese la letra inicial del departamento: " );
    scanf( "%c", &departamento );

    departamento = toupper ( departamento );

    switch ( departamento )
    {
        case 'Q': // Quindío
                  printf( "La capital es Armenia" );
                  break;

        case 'C': // Caldas
                  printf( "La capital es Manizales" );
                  break;

        case 'R': // Risaralda
                  printf( "La capital es Pereira" );
                  break;

        default: printf( "No es un departamento cafetero" );
                 break;
    }
```

```
30
31      return 0;
32  }
```

Explicación del programa:

Primera ejecución

```
Ingrese el nombre del departamento: Q
La capital del Quindio es Armenia
```

Segunda ejecución

```
Ingrese el nombre del departamento: A
No es un departamento cafetero
```

Explicación del programa:

En la primera parte del programa, se hacen las declaraciones generales, incluyendo las variables que se utilizarán. Se incluye la biblioteca `ctype.h`, ya que ella contiene la función `toupper` que es usada para convertir la letra inicial del nombre del departamento a mayúscula.

Si el programador lo desea, puede realizar los ajustes para trabajar en letras minúsculas, en cuyo caso debe emplear la función `tolower`

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main()
5  {
6      char departamento;
7
8      printf( "Ingrese la letra inicial del departamento: " );
9      scanf( "%c", &departamento );
10
11     departamento = toupper ( departamento );
```

A continuación, se hace uso de una estructura de decisión múltiple o selectiva para determinar el nombre de la capital del departamento ingresado por el usuario. Note que el tipo de dato de la variable `departamento` es `char`, por esta razón, los valores se escriben con comillas simples, por ejemplo: `case 'Q':`, `case 'C':` ...

```
13 switch ( departamento )
14 {
15    case 'Q': // Quindío
16              printf( "La capital es Armenia" );
17              break;
```

```
18
19    case 'C': // Caldas
20              printf( "La capital es Manizales" );
21              break;
22
23    case 'R': // Risaralda
24              printf( "La capital es Pereira" );
25              break;
26
27    default: printf( "No es un departamento cafetero" );
28             break;
29 }
```

Al finalizar la estructura de decisión múltiple, el programa habrá mostrado el nombre de la capital del departamento ingresado; si se ingresó un dato que no concuerda con un departamento del eje cafetero, el programa habrá mostrado un mensaje que indica que ese departamento no es válido.

.:Ejemplo 3.15. *La oficina de incorporación del ejército requiere de un programa que le permita conocer si un aspirante a ingresar a la institución como soldado es apto o no para ingresar. Una persona es apta si cumple los siguientes requisitos:*

- *Si es de género femenino, su estatura debe ser mayor a 1.60 mts y su edad debe estar entre 20 y 25 años.*
- *Si el aspirante es hombre, su estatura debe ser mayor a 1.65 mts y su edad debe estar entre los 18 y 24 años.*

Hombres y mujeres aspirantes deben ser solteros. Escriba un programa en Lenguaje C que permita saber si un aspirante es apto o no para ingresar al ejército.

Aclaración:

Recuerde que un programa puede implementarse de diferentes formas, llegando a una solución correcta. Tenga en cuenta cuál de las posibles implementaciones de un programa puede ser la más eficiente.

Análisis del problema:

- **Resultados esperados:** un mensaje que indique si el aspirante es "Apto" o "No es apto" para ingresar al ejército.
- **Datos disponibles:** el género, estado civil, edad y estatura del aspirante.
- **Proceso:** luego del ingreso de los datos del aspirante, es necesario utilizar las estructuras de decisión para determinar qué requisitos va cumpliendo o no el aspirante y poder saber al final si el aspirante es o no apto.
- **Variables requeridas:**
  - `genero`: almacena el género del aspirante.
  - `estadoCivil`: almacena el estado civil del aspirante.
  - `estatura`: almacena la estatura del aspirante.
  - `edad`: almacena la edad del aspirante.

De acuerdo al análisis planteado, se propone el Programa 3.15.

Programa 3.15: AspiranteEjercito

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

int main()
{
    char  genero;
    char  estadoCivil;
    float estatura;
    int   edad;

    printf( "Género del aspirante (m/f): " );
    scanf( "%c", &genero );
    genero = tolower( genero );

    printf( "Estado civil del aspirante (s/c/v/d/u): " );
    scanf( " %c", &estadoCivil );
    estadoCivil = tolower( estadoCivil );

    printf( "Altura del aspirante : " );
    scanf( "%f", &estatura );

    printf( "Edad del aspirante : " );
    scanf( "%d", &edad );

```

```
25      if( estadoCivil == 's' )
26      {
27          switch ( genero )
28          {
29              case 'f': if ( estatura > 1.6 &&
30                             edad >= 20     && edad <= 25 )
31                        {
32                           printf( "Es apto" );
33                        }
34                        else
35                        {
36                           printf( "No es apto" );
37                        }
38                        break;
39
40              case 'm': if( estatura > 1.65 &&
41                            edad >= 18      && edad <= 24 )
42                        {
43                           printf( "Es apta" );
44                        }
45                        else
46                        {
47                           printf( "No es apta" );
48                        }
49                        break;
50
51              default: printf( "Género incorrecto" );
52                       break;
53          }
54      }
55      else
56      {
57          printf( "No es apto" );
58      }
59
60      return 0;
61  }
```

Explicación del programa:

El programa inicia declarando las variables que se necesitan en las primeras líneas.

```
6       char  genero;
7       char  estadoCivil;
8       float estatura;
9       int   edad;
```

A continuación, se solicitan los datos con la instrucción `printf` y se capturan con la instrucción `scanf`. En esta ocasión se emplea la función `tolower` de la biblioteca `ctype.h` para convertir el carácter ingresado a minúsculas. Esto se hace después de leer el género y el estado civil.

```
11     printf( "Género del aspirante (m/f): " );
12     scanf( " %c", &genero );
13     genero = tolower( genero );
14
15     printf( "Estado civil del aspirante (s/c/v/d/u): " );
16     scanf( " %c", &estadoCivil );
17     estadoCivil = tolower( estadoCivil );
18
19     printf( "Altura del aspirante : " );
20     scanf( "%f", &estatura );
21
22     printf( "Edad del aspirante : " );
23     scanf( "%d", &edad );
```

En la Línea 16, se deja un espacio en la cadena de control del `scanf`, esto para ignorar el carácter asociado al `Enter` en la lectura anterior (género).

Acto seguido, se utiliza una estructura de decisión para determinar si el aspirante es soltero(a) o no lo es. Si se cumple esta condición, se emplea una decisión múltiple para decidir sobre la edad y la estatura según el género. Cada caso considera solo la letra minúscula, dado que el programa garantiza que todo carácter será convertido a minúsculas.

```
25   if( estadoCivil == 's' )
26     {
27         switch ( genero )
28         {
29             case 'f': if ( estatura > 1.6 &&
30                            edad >= 20     && edad <= 25 )
31                       {
32                          printf( "Es apto" );
33                       }
34                       else
35                       {
36                          printf( "No es apto" );
37                       }
38                       break;
39
40             case 'm': if( estatura > 1.65 &&
41                           edad >= 18      && edad <= 24 )
42                       {
43                          printf( "Es apta" );
44                       }
```

```
45                          else
46                          {
47                             printf( "No es apta" );
48                          }
49                          break;
50
51              default: printf( "Género incorrecto" );
52                       break;
53          }
54      }
55      else
56      {
57          printf( "No es apto" );
58      }
```

En caso que el usuario ingrese un género incorrecto, el programa ejecutará la instrucción `default`, allí, se imprimirá el mensaje correspondiente (`"Género incorrecto"`). Por último, si el estado civil no es soltero, el programa muestra un mensaje que indica que el aspirante no es apto.

.:Ejemplo 3.16. *Reescriba el programa del Ejemplo 3.11 pero, esta vez utilice decisiones múltiples y declare la variable* `estrato` *de tipo* `int`*, con el propósito de identificar diferencias y similitudes en su implementación.*

De acuerdo al análisis planteado, se propone el Programa 3.16.

Programa 3.16: FacturaAgua

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int   estrato;
6      float cantidad, cargoFijo, consumo,
7            valorRecoleccion, totalPago;
8
9      printf( "Ingrese estrato socioeconómico del predio: ");
10     scanf( "%d", &estrato);
11
12     printf( "Cantidad de metros consumidos: " );
13     scanf( "%f", &cantidad );
14
15     switch (estrato)
16     {
17         case 1:   cargoFijo = 2500;
18                   consumo = cantidad * 2200;
19                   valorRecoleccion = 5500;
20                   break;
```

```
21
22          case 2:   cargoFijo = 2800;
23                    consumo = cantidad * 2350;
24                    valorRecoleccion = 6200;
25                    break;
26
27          case 3:   cargoFijo = 3000;
28                    consumo = cantidad * 2600;
29                    valorRecoleccion = 7400;
30                    break;
31
32          case 4:   cargoFijo = 3300;
33                    consumo = cantidad * 3400;
34                    valorRecoleccion = 8600;
35                    break;
36
37          case 5:   cargoFijo = 3700;
38                    consumo = cantidad * 3900;
39                    valorRecoleccion = 9700;
40                    break;
41
42          case 6:   cargoFijo = 4400;
43                    consumo = cantidad * 4800;
44                    valorRecoleccion = 11000;
45                    break;
46
47          default: printf("\n Opción incorrecta");
48                   cargoFijo = 0;
49                   consumo = 0;
50                   valorRecoleccion = 0;
51                   break;
52      }
53
54      totalPago = cargoFijo + consumo + valorRecoleccion;
55
56      printf("Valor del cargo fijo: %.2f\n", cargoFijo);
57      printf("Valor del consumo: %.2f\n", consumo);
58      printf("Valor recolección: %.2f\n", valorRecoleccion);
59      printf("Total a pagar: %.2f\n", totalPago);
60
61      return 0;
62  }
```

Explicación del programa:

Complementando la explicación del programa original (Página 176), se puede observar cómo se reemplazó toda la decisión anidada por una decisión múltiple. Cada caso dentro de la decisión múltiple representa un

estrato socioeconómico. Se usó la instrucción (`default`) para cualquier valor ingresado por el usuario diferente a los valores del 1 al 6.

Analice la forma en la que se usa la variable **estrato**, ya que fue declarada de tipo `int`, no requiere del uso de comillas simples al utilizarla en cada `case`.

Actividad 3.3

1. Ajuste, de ser posible, los enunciados de todas las actividades sobre decisiones anidadas para que se puedan resolver mediante decisiones múltiples; o argumente el porque no es posible su conversión.

2. Para los dos enunciados a continuación, escriba un programa en Lenguaje C y construya el diagrama de flujo correspondiente. Resuelva utilizando tanto de decisión anidada como múltiple.

a. Un banco ofrece a sus clientes como producto el CDT. Los intereses que paga el banco dependen del valor ingresado y del periodo al que el cliente abre el producto, de la siguiente manera:

Valor Ingresado	Periodo en días	Porcentaje Interés
Hasta 1000000	90	4 %
Hasta 1000000	180	5 %
Hasta 1000000	360	6 %
Entre 1000001 y 10000000	90	5.5 %
Entre 1000001 y 10000000	180	6.5 %
Entre 1000001 y 10000000	360	7.5 %
Mayor a 10000000	90	6.5 %
Mayor a 10000000	180	7.5 %
Mayor a 10000000	360	8.5 %

Tabla 3.7: Intereses CDT

Determine el valor a pagar por concepto de intereses y el neto a pagarle al cliente, dependiendo del valor que ingresó y el periodo seleccionado.

b. La caseta de un peaje debe realizar el cobro a cada uno de los vehículos que pasan por la vía. El cobro se hace de acuerdo con la siguiente tabla:

Tipo de vehículo	**Valor del peaje**
Automóvil	9000
Campero	10000
Camioneta	12000
Camión	23000
Tractomula	32000

Tabla 3.8: Cobro de peaje

Determine el valor del peaje que se debe cobrar, dependiendo del tipo de vehículo que pasa por la caseta.

CAPÍTULO 4

ESTRUCTURAS DE REPETICIÓN

Controlar la complejidad es la esencia de la programación.

Brian Kernigan

Objetivos del capítulo:

- Utilizar variables de tipo contador, acumulador y centinela en las diferentes estructuras de repetición.
- Conocer la manera en que funcionan las estructuras de repetición, sus características y diferencias.
- Escribir programas que utilizan estructuras de repetición.
- Elaborar pruebas de escritorio con procesos repetitivos.

Para solucionar algunos problemas de programación en los que se llevan a cabo tareas repetitivas, se hace necesario la utilización de las denominadas estructuras cíclicas o de repetición. Estas estructuras permiten que una instrucción o grupo de ellas se ejecuten un determinado número de veces durante la ejecución del programa. Las estructuras de repetición son el objeto de estudio de este capítulo.

En este texto se estudiarán las siguientes estructuras de repetición:

- `while`
- `do` - `while`
- `for`

4.1. Conceptos básicos

Antes de entrar a estudiar cada una de las estructuras repetitivas que se acaban de mencionar, se presentan unas definiciones básicas que son comunes a todas las estructuras y útiles en el desarrollo de programas que requieran la implementación de ciclos.

4.1.1 Contador

Un contador es una variable que se utiliza en un programa, con el propósito de contar ciertas situaciones que se repiten dentro de un ciclo o iteración.

Por ejemplo, es común contar:

- Los *likes* que dan los usuarios a una publicación de Facebook.
- Los vehículos que ingresan a un parqueadero.
- La Cantidad de términos que tiene una serie numérica.
- El número de espectadores que entran a una función de cine.
- Los votos obtenidos por los candidatos en unas elecciones.
- La cantidad de reproducciones de un video en YouTube.

Las variables que hagan las veces de contador, deben declararse como enteras, es decir, `int` en Lenguaje C y deben ser inicializadas (asignarles

un valor) antes del ingreso al ciclo. La inicialización corresponde a la asignación de un valor que corresponde al número desde el cual se requiere que comience el conteo, generalmente se inicializan en 0; aunque puede ser otro valor, dependiendo de la forma en que vaya a ser resuelto el problema de programación.

Ahora bien, al interior del ciclo que usa el contador, debe existir una instrucción que modifique el valor de dicha variable, de la siguiente forma:

```
contador = contador + valorIncremento;
contador = contador - valorDecremento;
```

Recuerde también, que en Lenguaje C las instrucciones anteriores pueden escribirse como aparecen a continuación:

```
contador += valorIncremento;
contador -= valorDecremento;
```

En donde `valorIncremento` y `valorDecremento` se refieren a cualquier cantidad numérica que siempre es constante y que incrementa o decrementa el valor del contador en cada iteración que realice el ciclo.

Para comprender mas claramente el concepto de contador, imagine la cantidad de amigos que tiene en una red social, esta cantidad de amigos se incrementa en uno cada vez que se confirme una solicitud de amistad o se decrementa en esta misma cantidad cuando se elimina un contacto de la lista de amigos.

4.1.2 Acumulador

Así como existen variables que se incrementan o decrementan en cantidades constantes dentro de las estructuras repetitivas, también existen otras variables que se incrementan o decrementan en cantidades variables o no constantes; a este tipo de variables se les denomina acumuladores o totalizadores [Corona and Ancona, 2011].

Una variable de tipo acumulador podría utilizarse para:

- Almacenar las distancias recorridas durante varios viajes.
- Calcular el saldo en una cuenta de ahorros.
- Obtener el valor de una sumatoria de notas que luego puede ser usada para calcular la nota promedio en una asignatura.

- Calcular el total a pagar cuando se compran varios artículos en un almacén.
- Obtener los puntos acumulados por compras en un almacén o establecimiento comercial que ofrezca este beneficio.

Las variables que se utilizan como acumuladores, deberán declararse de tipo `int` o `float`. Al igual que los contadores, los acumuladores deben inicializarse antes del ingreso al ciclo, generalmente se inicializan en 0, aunque también depende de los requerimientos del problema a resolver.

Dentro del ciclo que esté utilizando la variable acumulador, debe existir una instrucción que modifique su valor, la forma general de esta instrucción es la siguiente:

```
acumulador = acumulador + valorIncremento;
acumulador = acumulador - valorDecremento;
```

Que en Lenguaje C, podría escribirse de esta forma:

```
acumulador += valorIncremento;
acumulador -= valorDecremento;
```

Tanto `valorIncremento` como `valorDecremento` se refieren a cualquier cantidad de tipo numérico, que incrementa o disminuye el valor del acumulador en cada iteración de la estructura repetitiva. Tenga presente que, para el caso de los acumuladores, los valores en que se incrementan o disminuyen son cantidades variables.

Para comprender mejor el concepto de acumulador, imagine el saldo en una cuenta de ahorros; este se incrementa con las consignaciones y se disminuye o decrementa con los retiros; como consignaciones y retiros son cantidades variables, se habla de acumulador en lugar de contador. De la misma manera ocurre con los puntos por compras que otorgan algunos almacenes o establecimientos comerciales, incrementan por cada compra que se realice y decrementan en el momento en que se haga un canje por algún artículo o promoción, en cantidades variables.

También es posible utilizar operaciones de multiplicación o división en las instrucciones que modifican un acumulador, su forma general aparece a continuación:

```
acumulador = acumulador * valorIncremento;
acumulador = acumulador / valorDecremento;
```

Aclaración:

Los contadores se declaran de tipo `int` y se incrementan o decrementan en valores constantes.

Los acumuladores pueden ser declarados de tipo `float` o `int` y se incrementan o decrementan en valores variables.

Cuando se deban usar multiplicaciones o divisiones para modificar el valor de un acumulador dentro de un ciclo, es fundamental que el valor del incremento o del decremento no sea 0 (cero), ya que, en el primer caso, la modificación siempre daría 0 y, en el segundo caso, no sería posible para el programa realizar la división.

4.1.3 Bandera

Una variable bandera es una variable que sirve para controlar diferentes acciones dentro de un programa. También recibe el nombre de interruptor, conmutador o centinela.

Si bien es cierto que una variable bandera puede declararse de cualquier tipo de dato, generalmente, en Lenguaje C, se declaran como de tipo entero o char, ya que casi siempre toman uno de dos posibles valores.

Si la variable se declara como entera o `int`, aunque podrá recibir cualquier valor de ese tipo, se debe interpretar que valores posibles podrían ser 1 o 0, que se entienden como verdadero o encendido para el 1 y falso o apagado para el 0. En el caso de que la declaración se haya hecho como de tipo `char`, podría recibir cualquier carácter, pero se podría entender que los valores más comunes son 'S'o 'N', interpretados como Sí o No. No obstante, para estos tipos de datos, el programador está en libertad de asignar los valores que considere convenientes.

Como con las variables de tipo contador y acumulador, es vital inicializar la variable bandera, cuyo valor posteriormente cambiará dependiendo de ciertas condiciones que estarán dadas por la solución del problema. Conforme con lo explicado hasta aquí, el valor inicial que se le asigne a la variable bandera, debe ser uno de los posibles que puede tomar; al presentarse la situación esperada, este valor cambiará su estado al valor contrario, esto es, si se inicializó en verdadero pasará a falso, si fue 1 pasará

a 0 y si fue en 'S' cambiará a 'N'. Luego de ejecutar las instrucciones correspondientes, más adelante en el programa la variable bandera podría retomar su valor inicial.

El uso de las variables de tipo contador, acumulador y bandera se irá ilustrando paulatinamente a lo largo de este capítulo, a medida que se van estudiando las diferentes estructuras de repetición y se introduzcan los ejemplos explicativos.

4.2. Estructura `while`

La primera estructura de repetición que se va a abordar es conocida como estructura `while`, que está presente en todos los lenguajes de programación y, Lenguaje C no es la excepción.

La característica principal de esta estructura es que tiene la condición al inicio del ciclo, lo que le proporciona un comportamiento particular. Esta estructura, como todas las estructuras de repetición, permite ejecutar una instrucción o conjunto de ellas una o más veces, siempre y cuando la condición se cumpla, es decir, sea evaluada como verdadera. Al tener la condición al inicio, si esta no se cumple, es posible que el ciclo no se lleve a cabo ni siquiera una vez.

La forma general de esta estructura de repetición se presenta a continuación.

```
1 Instrucción de inicialización;
2 while( condición )
3 {
4   Instrucción-1;
5   Instrucción-2;
6     ...                    /* Cuerpo del ciclo */
7   Instrucción-n;
8   Instrucción modificadora de condición;
9 }
10 Instrucción externa;
```

Dado que la estructura `while` posee la condición al inicio del ciclo, es fundamental determinar adecuadamente la forma en que se va a inicializar

la variable o variables que serán evaluadas en la condición, ya que de esto depende que se ejecuten o no las instrucciones dentro del ciclo; por esta razón, dentro de la anterior forma general se contempla una Instrucción de inicialización (línea 1). La inicialización también aplica para los contadores y acumuladores (si se están utilizando) que serán modificados dentro del ciclo. La inicialización puede hacerse mediante una asignación directa o solicitando el valor al usuario.

Cuando el programa en su ejecución encuentra la instrucción `while`, evalúa la condición (línea 2), que estará conformada por una expresión relacional o lógica. Si la evaluación da un resultado verdadero, el programa ejecutará el cuerpo del ciclo (líneas de la 4 a la 8); posteriormente, el control del programa vuelve al inicio del ciclo, esto es, a la instrucción `while` y vuelve a evaluar la condición. Este proceso se repetirá hasta que la evaluación de la condición arroje un resultado falso; de ser así, el programa saltará el cuerpo del ciclo e irá a la Instrucción externa (línea 10), que ya no hace parte de la estructura `while`.

En el caso de que el programa evalúe por primera vez la condición del ciclo `while` y su resultado sea falso, el cuerpo del ciclo no se ejecutará ni siquiera una vez y el control lo asumirá la instrucción externa. En conclusión, este ciclo itera (repite las instrucciones que componen el cuerpo del ciclo) mientras el valor de la condición que se evalúe sea verdadero.

Es importante tener en cuenta, que todo ciclo debe terminar de ejecutarse cuando cumpla con la tarea para la cual fue escrito; es por esto que se encuentra la Instrucción modificadora de condición (línea 8), que se escribe para cambiar el estado de la condición. De omitirse esta instrucción, el ciclo nunca terminará de iterar y se obtendrá lo que se conoce como un "Ciclo infinito". Aunque suele escribirse como la última instrucción que compone el cuerpo del ciclo, no necesariamente debe ocupar esa posición.

La estructura de repetición `while` se representa mediante el diagrama de flujo de la Figura 4.1.

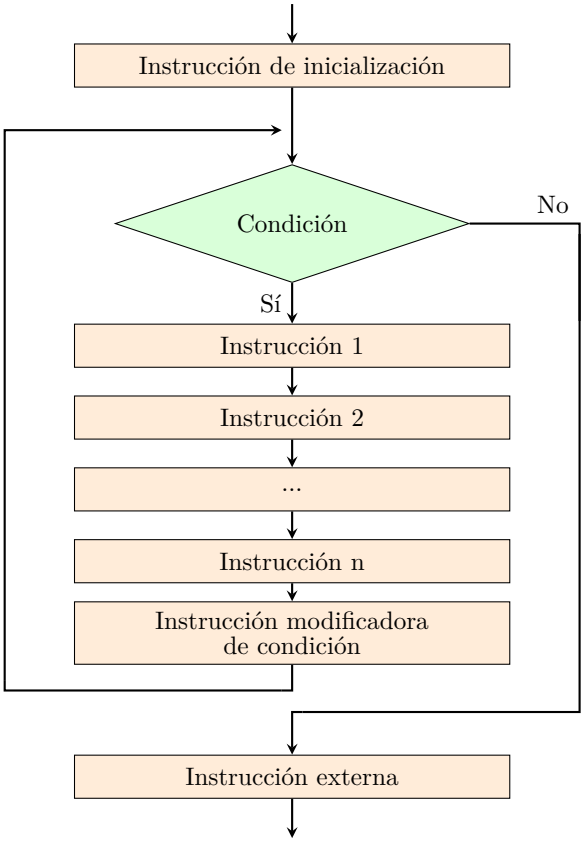


Figura 4.1: Forma general del ciclo `while`

A continuación, puede observarse el funcionamiento del ciclo `while` a través de un programa que imprime los números del 1 al 10.

Aunque es un ejemplo básico, se pretende introducir el concepto de ciclo e iteración con problemas que sean conocidos y fáciles de analizar por el lector.

Se usarán dos versiones, Programas: 4.1 y 4.2, cuya diferencia se puede apreciar en el Diagrama de Flujo 4.2.

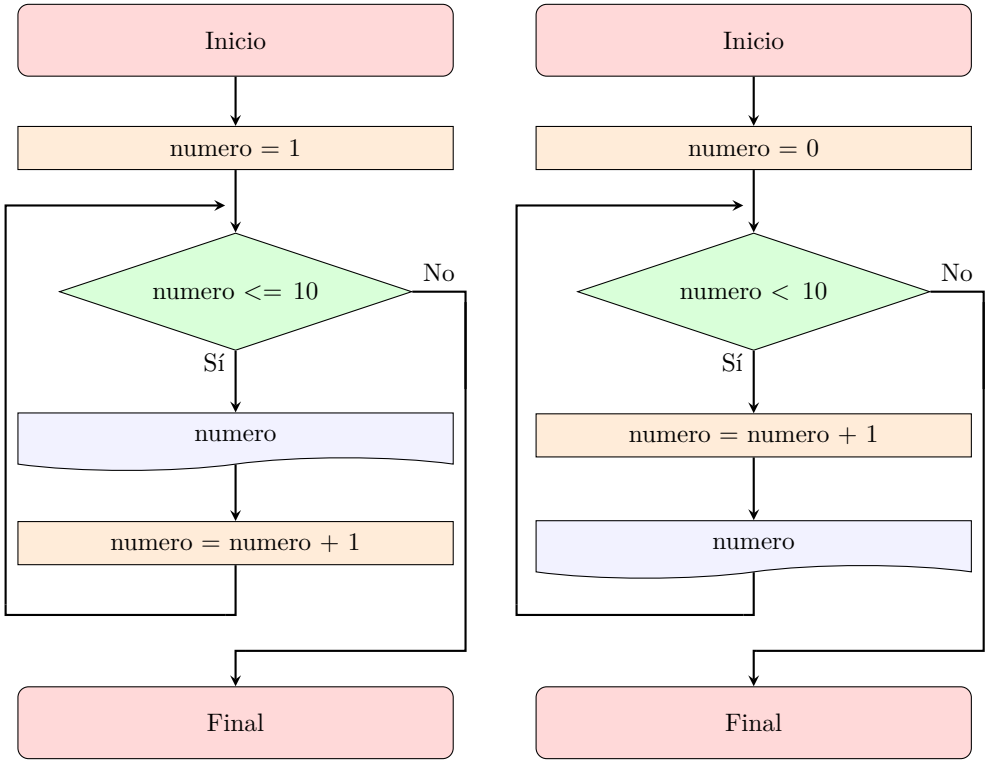


Figura 4.2: Diagrama de flujo Números 1 al 10

Programa 4.1: Numeros1-10-while Ver. 1

```
1  #include <stdio.h>
2
3  int main()
4  {
5     int numero;
6
7     numero = 1;
8
9     while ( numero <= 10 )
10    {
11        printf( "%d\n", numero );
12        numero = numero + 1;
13    }
14
15    return 0;
16 }
```

En la Tabla 4.1 se encuentra la relación de las líneas de código del segmento del programa y las instrucciones que lo conforman.

Línea	Explicación
7	Instrucción de inicialización
9	Condición e inicio del ciclo
11	Cuerpo del ciclo
12	Cuerpo del ciclo e instrucción modificadora de condición
13	Fin del ciclo. Regresa el control a la línea 9.

Tabla 4.1: Explicación del Programa 4.1

En la línea 7 se inicializa en 1 la variable `numero`. En la línea 9 se encuentra la instrucción `while` con su condición. El cuerpo del ciclo lo conforman dos instrucciones (líneas 11 y 12), que se ejecutan mientras que la condición de la línea 9 sea verdadera, es decir, mientras la variable `numero` contenga un valor menor o igual a 10.

Con `printf` se imprimen uno a uno los números del 1 al 10. En otras palabras, esta instrucción se ejecuta diez veces.

La operación `numero = numero + 1`, además de incrementar en 1 el valor de la variable `numero` en cada iteración, cuenta cada iteración y hace el papel de la instrucción modificadora de condición, ya que al llegar a 11 cambia a falso el estado de la condición, puesto que al evaluar la expresión relacional (`numero <= 10`) se obtiene un resultado falso, con lo que finaliza la ejecución del `while`.

Sin dejar de un lado la lógica establecida en el diagrama de la izquierda, de la Figura 4.2, el cuerpo del ciclo de la instrucción `while`, puede tener varias formas, igualmente válidas. Por ejemplo:

Forma 1:

```
1  numero = 1;
2  while ( numero <= 10 )
3  {
4        printf( "%d\n", numero );
5        numero++;
6  }
```

El cambio en esta versión, radica en que para incrementar la variable `numero`, se hizo con el uso del operador especial ++, que permite escribir de forma simplificada el incremento de una variable en una unidad.

Forma 2:

```
numero = 1;
  while ( numero <= 10 )
  {
      printf( "%d\n", numero ++ );
  }
```

En este caso, dentro de la función `printf` se hizo la operación de incremento. Recuerde que cuando la variable precede al operador, primero se ejecuta la acción con la variable y luego se produce el incremento.

Forma 3:

```
numero = 1;
while ( numero <= 10 )
        printf( "%d\n", numero ++ );
```

La misma versión anterior, pero sin el uso de las llaves. Aunque es perfectamente funcional, no es una buena práctica no delimitar las instrucciones que comprenden una estructura, por eso, si se quieren programas más legibles y de fácil entendimiento se recomienda el uso de las llaves.

Otra versión que puede tener el programa, es usando la inicialización en el momento de la declaración del contador:

Forma 4:

```
  int numero = 1;

  while ( numero <= 10 )
  {
      printf( "%d\n", numero );
      numero ++;
  }
```

Aplica para cualquiera de las versiones anteriores: la inicialización del contador se realizó en el momento de la declaración.

De acuerdo a todo lo visto previamente, la forma más simplificada de escribir este código se puede apreciar a continuación:

Forma 5:

```
1  int numero = 1;
2
3  while ( numero <= 10 )
4  {
5          printf( "%d\n", numero ++);
6  }
```

Es imperativo tener presente que cada problema puede tener varias soluciones. Por ejemplo, el siguiente programa también imprime los números del 1 al 10:

Programa 4.2: Numeros1-10-while Ver. 2

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int numero;
6
7      numero = 0;
8      while( numero < 10 )
9      {
10         numero = numero + 1;
11         printf( "%d\n", numero );
12     }
13
14     return 0;
15 }
```

Al comparar el Programa 4.2 (versión 2) con el Programa 4.1 (versión 1) se puede observar lo siguiente:

Línea 7: la inicialización se hizo en 0 y no en 1.

Línea 8: el ciclo se lleva a cabo mientras el valor de `numero` sea menor a 10. En la versión 1 del programa, el ciclo se ejecuta mientras sea menor o igual a 10. Esto se debe a la inicialización que fue en 0 en lugar de 1. Tenga en cuenta que en la versión 1 del programa la variable `numero` alcanza a tomar el valor de 11, mientras que en la versión 2 el valor llega hasta 10.

El cuerpo del programa (Líneas 10 y 11) contiene las expresiones `numero = numero + 1` y `printf ( "%d\n", numero )`, que son las mismas ubicadas en el cuerpo del ciclo del programa de la versión 1, pero en orden diferente, ya que si se dejaban en el orden en que están en la versión 1 del programa, el resultado sería la impresión de los números del 0 al 9 y no del 1 al 10 como se esperaría.

En la Línea 12 finaliza el cuerpo del ciclo y el control regresa a la línea 8 para volver a evaluar la condición y se determine si continúan o paran las iteraciones.

De acuerdo con lo analizado con anterioridad, se puede concluir que hay que tener cuidado con la forma de escribir la condición del `while` y con la inicialización de la variable, por ejemplo:

- `numero = 0;` se utiliza el operador $<$
- `numero = 1;` se utiliza el operador $<=$

También es clave el orden en que se ubiquen las instrucciones dentro del cuerpo del ciclo, una ubicación incorrecta podría dar resultados inesperados.

Al igual que con el programa del Ejemplo 4.1, el cuerpo del ciclo también puede tener varias formas:

Forma 1:

```
1 numero = 0;
2
3 while( numero < 10 )
4 {
5   numero++;
6   printf( "%d\n", numero );
7 }
```

Forma 2:

```
1 numero = 0;
2
3 while( numero < 10 )
4 {
5   printf( "%d\n", ++numero );
6 }
```

Ahora bien, con base a la manera de trabajar del ciclo `while`, que se ejecuta mientras la condición del ciclo sea verdadera, es válido escribir su condición como se ejemplifica en el siguiente segmento de código que imprime los números del 10 al 1, de forma descendente:

Forma 1:

```
numero = 10;

while( numero )
{
   printf( "%d\n", numero );
   numero--;
 }
```

El cuerpo presentado en este ciclo, se ejecuta mientras la variable `numero` tenga un valor diferente a 0; tenga presente que en Lenguaje C, el valor de 0 representa un resultado falso. Cuando la variable `numero` tome este valor, la condición será falsa.

Inicializando el contador en 11 y usando el decremento en la condición, también se logra imprimir los números del 10 al 1:

Forma 2:

```
numero = 11;

while ( --numero )
{
   printf( "%d\n", numero );
}
```

Al usar el operador de decremento antes que la variable (`--numero`), primero se hace la operación y luego se evalúa la condición.

Actividad 4.1

¿Cuál es el resultado arrojado por el siguiente segmento de programa?:

```
numero = 1;

while ( numero <= 10 )
{
  printf( "%d\n", ++ numero );
}
```

Aclaración:

Un problema de programación puede resolverse de múltiples formas, por lo cual, su solución no tiene que ser igual a las planteadas en este libro.

Las variables que hacen parte de la condición del ciclo `while` tienen que inicializarse antes de su primera evaluación.

Las instrucciones que conforman el cuerpo del ciclo se ejecutarán mientras el resultado de la evaluación de la condición sea verdadero. Si al evaluar por primera vez la condición del `while`, el resultado es falso, las instrucciones del cuerpo del ciclo no se ejecutarán ni siquiera una vez.

La evaluación de las expresiones relacionales o lógicas, siempre dará como resultado un valor de 0 o 1, que se interpreta como falso o verdadero, respectivamente.

En las siguientes secciones, se explicarán ejemplos de uso de la estructura repetitiva `while`.

.:Ejemplo 4.1. *Diseñe un programa en Lenguaje C que permita generar e imprimir la siguiente serie de números:* 1 3 5 7 9 11 ... n

El programa recibirá un número entero (n) que indicará la cantidad de términos de la serie.

Análisis del problema:

- **Resultados esperados:** el programa debe generar e imprimir la serie de números: 1 3 5 7 9 11 ... n
- **Datos disponibles:** la cantidad de términos que tendrá la serie (n).
- **Proceso:** como la serie está compuesta por números impares, se empieza la serie en 1 y se incrementa de 2 en 2. Para lograr la serie, se repite el incremento y la impresión hasta la cantidad de términos solicitada. Así mismo, se deben ir contando los términos impresos

para determinar cuándo terminar las iteraciones.

El diagrama de flujo de la Figura 4.3 muestra la solución.

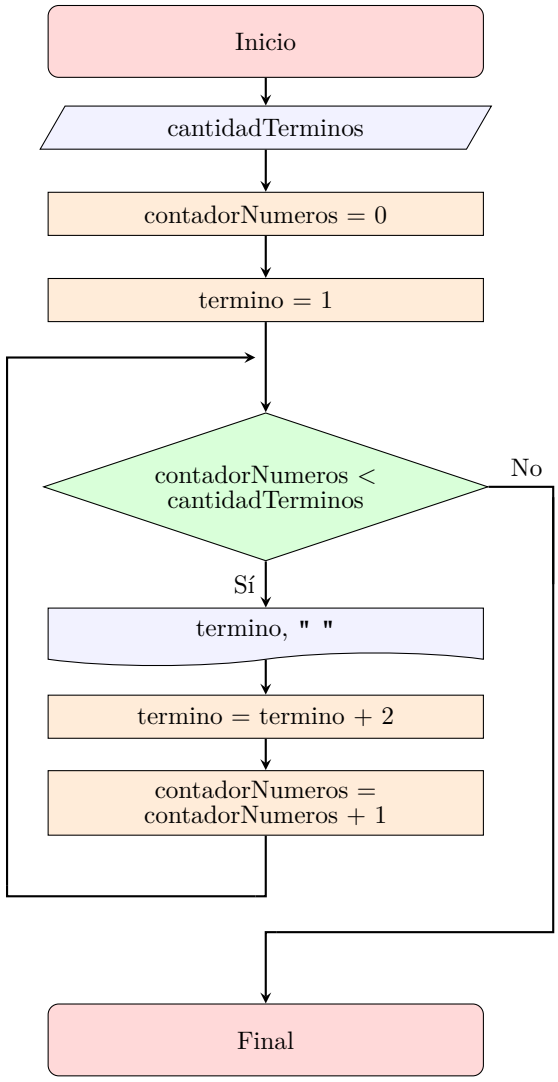


Figura 4.3: Diagrama de flujo Serie

- **Variables requeridas:**
  - `cantidadTerminos`: almacena el número de términos de la serie (n).
  - `contadorNumeros`: variable para controlar la cantidad de términos que se van mostrando, así como la terminación del ciclo.

- `termino`: corresponde a cada uno de los términos de la serie que se van generando y mostrando.

Basados en el anterior análisis, se presenta el Programa 4.3.

Programa 4.3: Serie

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int  cantidadTerminos, contadorNumeros, termino;
6
7    printf( "Ingrese la cantidad de terminos a generar: " );
8    scanf( "%d", &cantidadTerminos );
9
10   contadorNumeros = 0;
11   termino = 1;
12
13   while( contadorNumeros < cantidadTerminos )
14   {
15       printf( "%d ",termino );
16       termino += 2;
17       contadorNumeros++;
18   }
19
20   return( 0 );
21 }
```

Al ejecutar el programa:

```
Ingrese la cantidad de términos a generar: 15
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
```

Explicación del programa:

En este programa la inicialización se realiza a través de tres instrucciones. En la primera, se le solicita la cantidad de términos al usuario (líneas 7 y 8); la segunda inicializa la variable `contadorNumeros` en 0, ubicada en la línea 10. Estas variables conforman la condición del `while`. La primera inicialización es suministrada por el usuario y capturada con `scanf`; la segunda es explicita en el programa con la asignación `contadorNumeros = 0`.

La tercera inicialización está en la línea 11. Allí la variable `termino` recibe el valor de 1, que es el primer número de la serie a generar.

A continuación, se definió la estructura repetitiva `while`, cuya condición se estableció así:

```
while( contadorNumeros < cantidadTerminos )
```

En donde `contadorNumeros` contará la cantidad de números que se imprimirán. Esta variable permitirá que el ciclo se ejecute mientras su valor sea menor al de `cantidadTerminos`.

A modo de ilustración, suponga que se desean imprimir los primeros 15 términos de la serie; la variable `cantidadTerminos` almacenaría este valor. Al hacer la evaluación de la siguiente condición:

$$\texttt{contadorNumeros} < \texttt{cantidadTerminos}$$

Se obtiene un resultado verdadero, debido a que `contadorNumeros` se inicializó en 0.

Mientras la condición sea verdadera se procede a ejecutar las instrucciones del cuerpo del ciclo (líneas 15 a 17). La primera instrucción, en la línea 15, imprime el contenido de la variable `termino`.

En la línea 16, la instrucción `termino += 2`, incrementa la variable `termino` en 2 unidades, para pasar al siguiente término de la serie (inician en 1 y se van incrementando de 2 en 2: $1, 3, 5, 7, 11, \ldots$).

La última instrucción del cuerpo del ciclo `contadorNumeros++`, tiene como objetivo ir contando (de uno en uno) la cantidad de números que se van imprimiendo y a la vez, es la variable que sirve para controlar el número de iteraciones del ciclo; en otras palabras, esta es la instrucción modificadora de la condición.

Con la línea 18 se cierra el ciclo y se devuelve el control al inicio del mismo, es decir, a la instrucción `while`, donde se vuelve a evaluar la condición. Si el valor de la condición es verdadero se repetirá el cuerpo del ciclo. Cuando la variable `contadorNumeros` alcance el valor de la variable `cantidadTerminos`, la condición será falsa y el programa se saldrá del ciclo, pasando a la instrucción que está en la línea 20 para terminar con su ejecución.

Buena práctica:

Se recomienda que los contadores inicien en el valor de 0, excepto en aquellas situaciones que requieran de un valor distinto.

.:Ejemplo 4.2. *El profesor de Lenguaje C requiere de un programa con el cual pueda saber cuántos de los estudiantes aprobaron la materia y cuántos reprobaron, así como el promedio del grupo. Se sabe que el profesor posee el código de cada estudiante y su nota definitiva. Las materias se aprueban con una nota mínima de 3.0.*

Análisis del problema:

- **Resultados esperados:** el programa debe informar siguientes resultados:
  - Número de estudiantes que aprobaron la materia.
  - Número de estudiantes que reprobaron la materia.
  - Nota promedio del grupo.
- **Datos disponibles:** cantidad de estudiantes que conforman el grupo y, por cada estudiante se conoce su código y la nota definitiva que obtuvo en la materia.
- **Proceso:** inicialmente, debe implementarse un ciclo que permita el ingreso de los datos de los estudiantes. Con el uso de decisiones y de contadores se podrá obtener la cantidad de estudiantes que aprobaron y los que reprobaron. Para trabajar un contador se usa la siguiente forma general:

```
contador = contador + valorIncrementar;
```

  En este ejercicio, el valor a incrementar será de 1, que representa cada uno de los estudiantes que aprobaron o reprobaron la materia.

  Adicionalmente, debe obtenerse el promedio general del grupo. En la fórmula del promedio se involucran tres valores:

```
promedio = sumatoria / cantidad;
```

  La `sumatoria` corresponde a la suma de todas las notas definitivas de los estudiantes y la `cantidad` representa el número de estudiantes del curso.

Lo anterior permite inferir que es necesario un cálculo adicional que obtenga la `sumatoria`. Esto implica el uso de un acumulador, concepto analizado en el numeral 4.1.2:

```
acumulador = acumulador + valorIncrementar;
```

El `acumulador` corresponde a la sumatoria y el valor a incrementar es la nota definitiva de cada estudiante:

```
sumatoria = sumatoria + notaDefinitiva;
```

Todos los cálculos que se acaban de mencionar deben ubicarse, ya sea antes, dentro o después del ciclo. Por ejemplo, después de declarar las variables, se solicita la cantidad de estudiantes, dato que indicará la cantidad de iteraciones que tendrá el ciclo. Posteriormente, se inicializan los contadores y los acumuladores necesarios. Seguidamente, se debe ubicar el ciclo con su condición.

Dentro del ciclo se debe solicitar el código del estudiante y la nota definitiva en la materia. Cuando se tenga la nota definitiva se puede determinar si aprobó o reprobó la materia; para ello se utiliza una decisión como se describe con el árbol de decisión de la Figura 4.4.

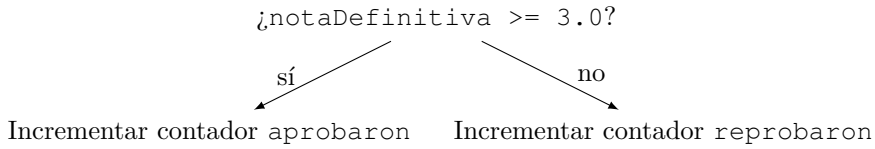


Figura 4.4: Árbol de decisión del Ejemplo 4.2

La sumatoria que acumula las notas de cada estudiante, también debe estar dentro del ciclo.

Al finalizar el ciclo, se obtiene el promedio del grupo y se muestran todos los resultados que debe generar el programa.

Conforme al anterior análisis, se realiza el diagrama de flujo de las Figuras 4.5 y 4.6 donde se muestran las instrucciones a ejecutar.

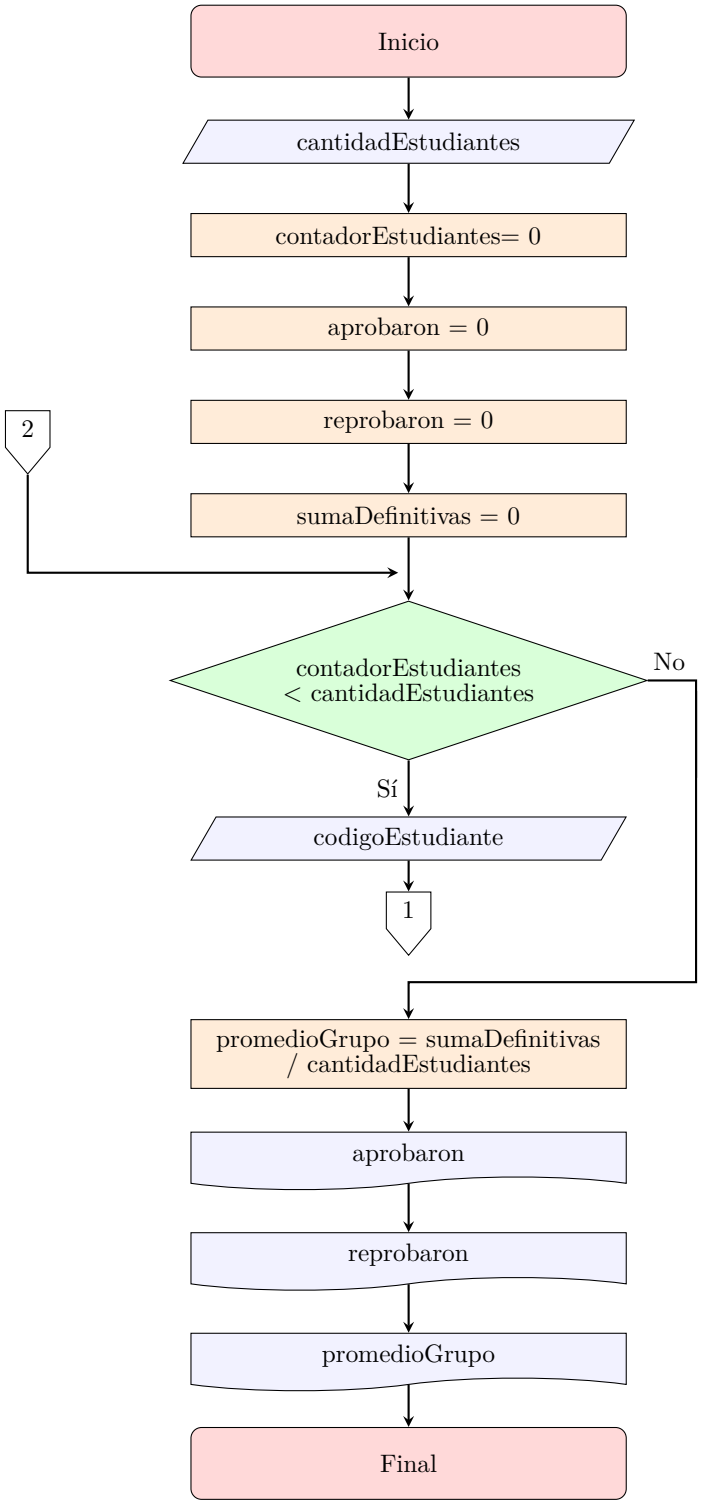


Figura 4.5: Diagrama de flujo Estudiantes - Parte 1

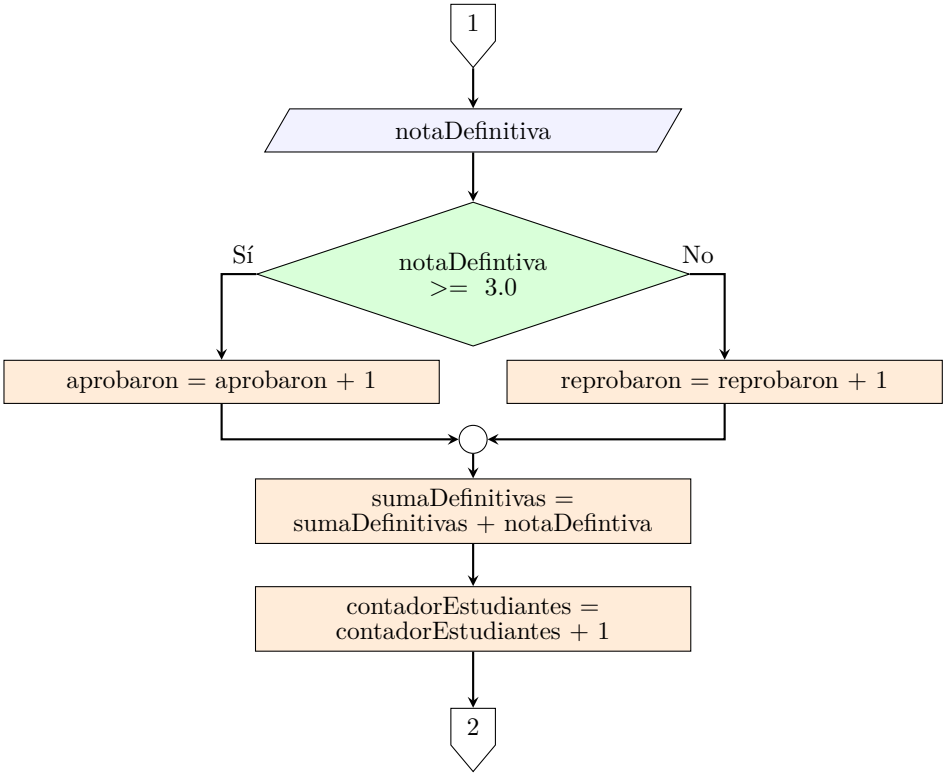


Figura 4.6: Diagrama de flujo Estudiantes - Parte 2

- **Variables requeridas:**
  - `cantidadEstudiantes`: cantidad de estudiantes, condiciona el ciclo.
  - `codigoEstudiante`: código del estudiante a procesar.
  - `notaDefinitiva`: nota obtenida por el estudiante.
  - `contadorEstudiantes`: cuenta cada estudiante que se va procesando.
  - `aprobaron`: contador que se incrementa cada vez que un estudiante aprueba la materia.
  - `reprobaron`: contador que se incrementa cada vez que un estudiante reprueba la materia.
  - `sumaDefinitivas`: acumulador que almacena la suma de las notas de los estudiantes procesados.
  - `promedioGrupo`: almacena el promedio del grupo.

El Programa 4.4 es una de las posibles soluciones a este ejemplo.

Programa 4.4: Estudiantes

```
#include <stdio.h>

int main()
{
  float notaDefinitiva, sumaDefinitivas, promedioGrupo;
  int   cantidadEstudiantes, contadorEstudiantes,
        aprobaron, reprobaron;
  char  codigoEstudiante[ 8 ];

  printf( "Ingrese la cantidad de estudiantes: " );
  scanf( "%d", &cantidadEstudiantes );

  contadorEstudiantes = 0;
  aprobaron = 0;
  reprobaron = 0;
  sumaDefinitivas = 0;

  while ( contadorEstudiantes < cantidadEstudiantes )
  {
    printf( "\nIngrese el código del estudiante: " );
    scanf("%s", codigoEstudiante); ;

    printf( "Ingrese la nota definitiva: " );
    scanf( "%f",&notaDefinitiva );

    if ( notaDefinitiva >= 3.0 )
    {
      aprobaron++;
    }
    else
    {
      reprobaron++;
    }

    sumaDefinitivas += notaDefinitiva;
    contadorEstudiantes++;
  }

  promedioGrupo = sumaDefinitivas / cantidadEstudiantes;

  printf( "\nCantidad que aprobaron: %d \n",aprobaron  );
  printf( "Cantidad que reprobaron: %d \n", reprobaron );
  printf( "Promedio del grupo: %3.1f",promedioGrupo );

  return( 0 );
}
```

Explicación del programa:

Este programa propone el uso de instrucciones repetitivas y de decisión para solucionar el problema.

La instrucción `while` permite repetir las instrucciones dentro del ciclo un determinado número de veces. La instrucción `if` permitirá, en cada iteración, determinar si la materia fue o no aprobada.

Con las instrucciones de las líneas 10 y 11 se ingresará la cantidad de estudiantes para indicar el número de iteraciones del ciclo.

```
10   printf( "Ingrese la cantidad de estudiantes: " );
11   scanf( "%d", &cantidadEstudiantes );
```

La cantidad de estudiantes se utiliza para controlar el ciclo a través de un contador:

```
18   while ( contadorEstudiantes < cantidadEstudiantes )
```

Donde, `contadorEstudiantes` es una variable que va contando los estudiantes que se van procesando, de la siguiente forma:

```
36      contadorEstudiantes++;
```

Recuerde que, dentro del ciclo `while`, esta instrucción es la instrucción modificadora de condición; su contenido se incrementará en 1 en cada iteración. Cuando el contenido de la variable `contadorEstudiantes` se haga igual al de la variable `cantidadEstudiantes`, el ciclo finalizará.

Luego de finalizar el ciclo, se calcula el promedio del grupo así:

```
39 promedioGrupo = sumaDefinitivas / cantidadEstudiantes;
```

Tenga en cuenta que la variable `cantidadEstudiantes`, que actúa como divisor en la anterior expresión, no debe recibir un valor de 0, ya que la división entre cero no está definida. Es fundamental anticiparse a este tipo de situaciones para que no se generen errores en la ejecución del programa.

Conforme a lo anterior, se puede mejorar el programa reescribiendo esta parte de código para obtener el promedio:

```
// Se asume 0 en el promedio cuando la cantidad de
// estudiantes es 0.
if( cantidadEstudiantes >= 1 )
{
  promedioGrupo = sumaDefinitivas / cantidadEstudiantes;
}
else
{
  promedioGrupo = 0;
}
```

Estas instrucciones harían que el promedio se calcule si la cantidad de estudiantes es 1 o más; si el valor es 0 o un valor negativo, el programa asignaría el valor de 0 a la variable `promedioGrupo` y este será el resultado que informaría.

Buena práctica:

Con las instrucciones adecuadas se puede controlar que un programa no genere errores en su ejecución; por ejemplo, cuando se trata de realizar la división entre 0.

Algunos lenguajes de programación poseen instrucciones específicas para el tratamiento de errores y excepciones lo que facilita estas tareas.

Otra buena práctica consiste en definir condiciones que deben garantizarse para que el programa trabaje adecuadamente. Estas condiciones se conocen como precondiciones.

.:Ejemplo 4.3. *Escriba un programa en Lenguaje C que reciba del usuario un número entero. Si el número es positivo, informe la cantidad de cifras que posee y calcule la sumatoria de las mismas; en caso contrario, imprima un mensaje que diga que el número no es positivo.*

Análisis del problema:

- **Resultados esperados:** la cantidad de cifras del número ingresado y la sumatoria de las mismas o, un mensaje en el caso que el número no sea positivo.
- **Datos disponibles:** un número entero.

- **Proceso:** luego del ingreso del número, se determina con una decisión si es positivo, si llega a serlo se separan sus cifras y simultáneamente se van sumando; en caso contrario se informa que el número no es positivo.

  Con la siguiente decisión es posible determinar si un número es positivo (Ver Figura 4.7).

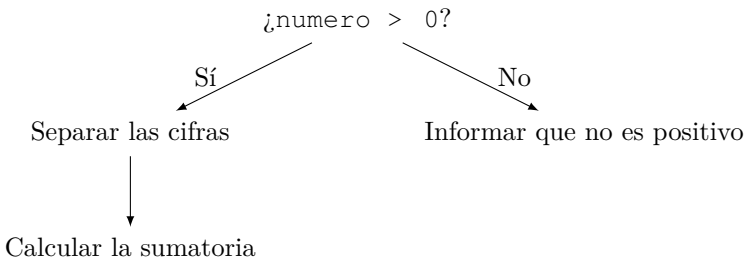


Figura 4.7: Árbol de decisión del Ejemplo 4.3

Obtener las cifras de un número entero que hace parte del sistema decimal, puede hacerse con divisiones enteras sucesivas entre 10. Por ejemplo, se desea separar las cifras del número 8345, esto implica realizar lo siguiente:

Iteración	**Dividendo**	**Divisor**	**Cociente**	**Resto**
1	8345	10	834	5
2	834	10	83	4
3	83	10	8	3
4	8	10	0	8

Tabla 4.2: Descomposición del número 8345 en cifras

Tenga en cuenta el nombre de los términos que hacen parte en una división: dividendo, divisor, residuo (resto) y cociente. En una división entera, al residuo se le conoce como resto.

$$\begin{array}{r|l} \text{dividendo} & \text{divisor} \\ \hline \text{resto} & \text{cociente} \end{array}$$

Como ya se mencionó, la descomposición de un número entero en sus cifras, se lleva a cabo por medio de divisiones sucesivas. Esta operación se realiza tantas veces como cifras tenga el número a descomponer (para este ejemplo es 4). El proceso finaliza cuando el cociente tenga un valor de 0.

Durante la primera iteración, se realiza la operación que se muestra enseguida:

```
8 3 4 5 | 1 0
  3 4   | 8 3 4
    4 5 |
      5 |
```

Note que el residuo o resto de la división es 5, es decir, la última cifra del número, y el cociente es 834. Estos valores obtienen con dos divisiones enteras[1] entre 10.

La primera división que se realizará es el módulo o resto de la división, que utiliza el operador % para hallar el resultado:

```
8345 % 10 = 5
```

La segunda división obtiene el cociente, que va tomando el valor del número original sin la última cifra; ese cociente se convierte en el dividendo en la siguiente iteración. Este cociente se obtiene con una división entera entre 10 usando el operador /:

```
8345 / 10 = 834
```

En la segunda iteración se hacen las mismas operaciones, teniendo en cuenta que el cociente obtenido en la división anterior, pasa a ser el nuevo dividendo:

```
8 3 4 | 1 0
  3 4 | 8 3
    4 |
```

Para obtener el resto de la división se hace la siguiente operación:

```
834 % 10 = 4
```

Para calcular el cociente, se realiza la división entera:

```
834 / 10 = 83
```

[1]Se obtienen resultados sin decimales (fracciones).

Una vez más se deben realizar las divisiones, teniendo en cuenta que el nuevo dividendo es el cociente anterior:

$$\begin{array}{r|l} 8\ 3 & 1\ 0 \\ \cline{2-2} 3 & 8 \end{array}$$

Para obtener el resto de la división y el cociente se hacen las siguientes divisiones:

```
83 % 10 = 3
```

```
83 / 10 = 8
```

Como el cociente aún no es 0, se debe realizar una última iteración:

$$\begin{array}{r|l} 8 & 1\ 0 \\ \cline{2-2} 8 & 0 \end{array}$$

Para obtener el resto de la división y el cociente se hacen las siguientes divisiones:

```
8 % 10 = 8
```

```
8 / 10 = 0
```

- **Variables requeridas:**
  - `numero`: almacena el número ingresado.
  - `copiaNumero`: guarda una copia del número, con objeto de ir eliminando la última cifra. Esta variable representa el cociente dentro de los términos de una división, pero recuerde que ese cociente pasa a ser el dividendo en la siguiente iteración.
  - `contadorCifras`: contador de las cifras del número.
  - `sumaCifras`: acumula la suma de las cifras del número.
  - `cifra`: almacena una a una las cifras en las que se va descomponiendo el número. Dentro de los términos de una división representa al residuo o al resto de la división entera.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 4.8 y se escribe el Programa 4.5.

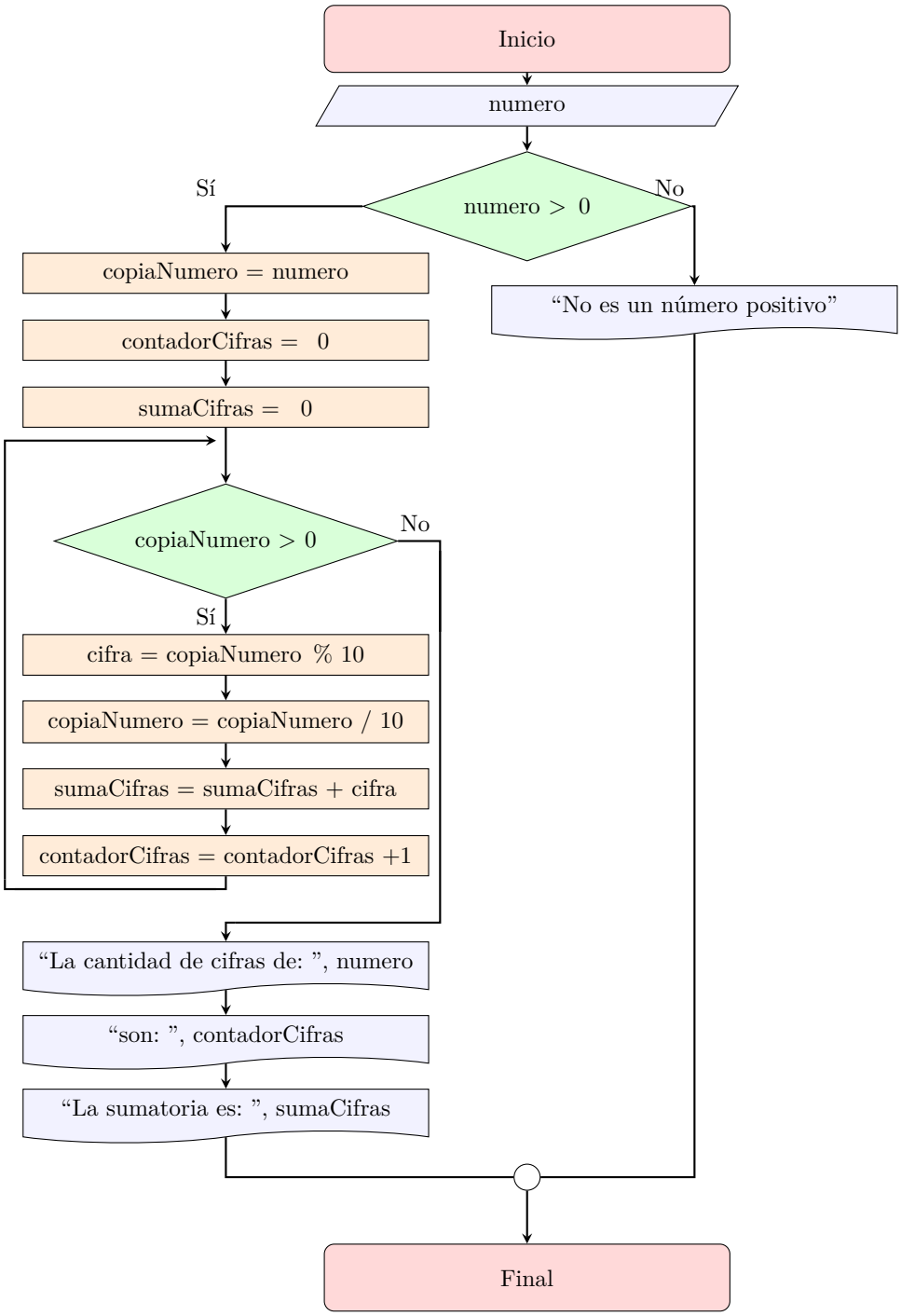


Figura 4.8: Diagrama de flujo del Programa `CifrasNumero`

Programa 4.5: CifrasNumero

```
1  #include <stdio.h>
2
3  int main ()
4  {
5    int numero, cifra, contadorCifras, sumaCifras,copiaNumero;
6
7    printf( "Digite un número entero: " );
8    scanf( "%d", &numero );
9
10   if( numero > 0 )
11   {
12     copiaNumero = numero;
13     contadorCifras = 0;
14     sumaCifras = 0;
15
16     while ( copiaNumero > 0 )
17     {
18       cifra = copiaNumero % 10;
19       copiaNumero = copiaNumero / 10;
20       sumaCifras = sumaCifras + cifra;
21       contadorCifras++;
22     }
23
24     printf( "\nLa cantidad de cifras de: %d es de: %d\n",
25               numero, contadorCifras );
26     printf( "La sumatoria de las cifras es: %d\n",
27               sumaCifras );
28   }
29   else
30   {
31     printf( "\nNo es un número positivo" );
32   }
33
34   return 0;
35 }
```

Al ejecutar el programa:

Primera ejecución:

```
Digite un número entero: 459

La cantidad de cifras de: 459 es de: 3
La sumatoria es: 18
```

Segunda ejecución:

```
Digite un número entero: -459

No es un número positivo
```

Explicación del programa:

Luego de ingresar el número en la variable `numero` se le hace una copia en la variable `copiaNumero`, con el propósito de conservar el valor original y poder imprimirlo al final junto con los resultados. Esto indica que la descomposición de cifras se realiza sobre la copia y no sobre la variable que tiene el valor original.

La solución de este problema utliza en conjunto la estructura condicional `if` y la estructura repetitiva `while`, donde el ciclo `while` hace parte del grupo de instrucciones que se ejecutan cuando la condición del `if` es verdadera.

El ciclo `while` hace lo siguiente en cada iteración:

- Evalúa la condición (línea 16)
- Separa una cifra del número (línea 18)
- Elimina la última cifra al número (línea 19)
- Acumula la sumatoria de las cifras (línea 20)
- Cuenta las cifras (línea 21)
- Retorna el control al `while` (línea 22)

Todo esto se lleva a cabo mientras el valor de la variable `copiaNumero` sea mayor a 0.

Si cuando se evalúa la condición `if( numero > 0 )` su resultado es falso, el cuerpo del ciclo `while` no se lleva a cabo y, en su lugar imprimiría el mensaje: “No es un número positivo”.

En este ejercicio se encuentran dos aspectos importantes a tener en cuenta:

- Las iteraciones del ciclo no se condicionaron con un contador, sino con un acumulador:

```
22 while ( copiaNumero > 0 )
```

- La instrucción modificadora de condición se realiza con una división, la cual se ejecuta en cada iteración mientras la condición sea verdadera, reduciendo el valor de la variable `copiaNumero`:

```
24 copiaNumero = copiaNumero / 10;
```

Buena práctica:

Cuando los datos de entrada se vayan a modificar dentro del programa, es recomendable utilizar otra variable para copiarlos y así mantener su valor original.

Aclaración:

Las estructuras vistas hasta ahora pueden anidarse unas dentro de otras, es decir, una estructura de decisión puede hacer parte del cuerpo o conjunto de instrucciones de una estructura de repetición, o viceversa.

.:Ejemplo 4.4. *Entre las curiosidades de las matemáticas, se encuentra lo que se denomina número de Armstrong o número narcisista, el cual se define como un número cuyas cifras elevadas a la potencia n y sumadas dan como resultado el mismo número, siendo n la cantidad de cifras del número dado.*

Para ilustrar el caso, suponga el número 407*, que tiene* 3 *cifras. Este es un número de Armstrong ya que:*

$$4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407$$

Otros números de Armstrong son: 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, entre otros.

Ahora bien, se requiere de un programa en Lenguaje C que determine si un número entero positivo ingresado por el usuario es o no un número de Armstrong.

Análisis del problema:

- **Resultados esperados:** mostrar si el número ingresado es o no un número de Armstrong.
- **Datos disponibles:** el número entero positivo ingresado por el usuario.
- **Proceso:** luego de capturar el número ingresado por el usuario, se obtienen las cifras que lo componen y, cada una de ellas se eleva a la n, donde n representa la cantidad de cifras. A continuación, se suman estos resultados. Por último, a través de una estructura de decisión, se determina si el número ingresado es igual a la sumatoria calculada; si esto ocurre, se muestra un mensaje informando que el número es un Armstrong. (Ver Figura 4.9).

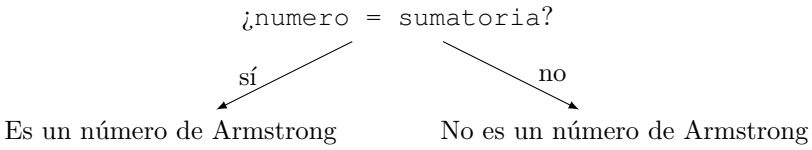


Figura 4.9: Árbol de decisión del Ejemplo 4.4

- **Variables requeridas:**
  - `numero`: almacena el número que ingresa el usuario.
  - `copiaNumero`: almacena una copia del número para conservar el valor original ingresado por el usuario.
  - `contadorCifras`: permitirá saber la cantidad de cifras que componen el número.
  - `cifra`: almacenará las cifras del número.
  - `sumaCifras`: almacenará la suma de las cifras ya elevadas a la potencia n.

Las Figuras 4.10 y 4.11 muestran el Ejemplo 4.4 solucionado con un diagrama de flujo.

Como el diagrama el diagrama ocupa dos páginas, se usó un conector que las relaciona.

Buena práctica:

Se deben usar conectores en la misma página y a otras páginas para enlazar las diversas partes de un diagrama de flujo extenso.

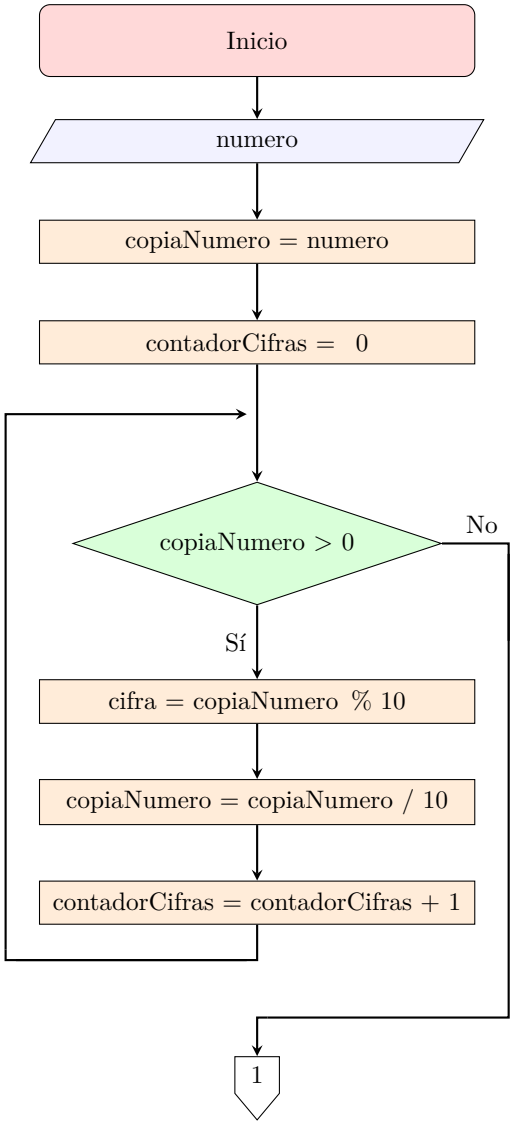


Figura 4.10: Diagrama del flujo del Programa `Armstrong` - Parte 1

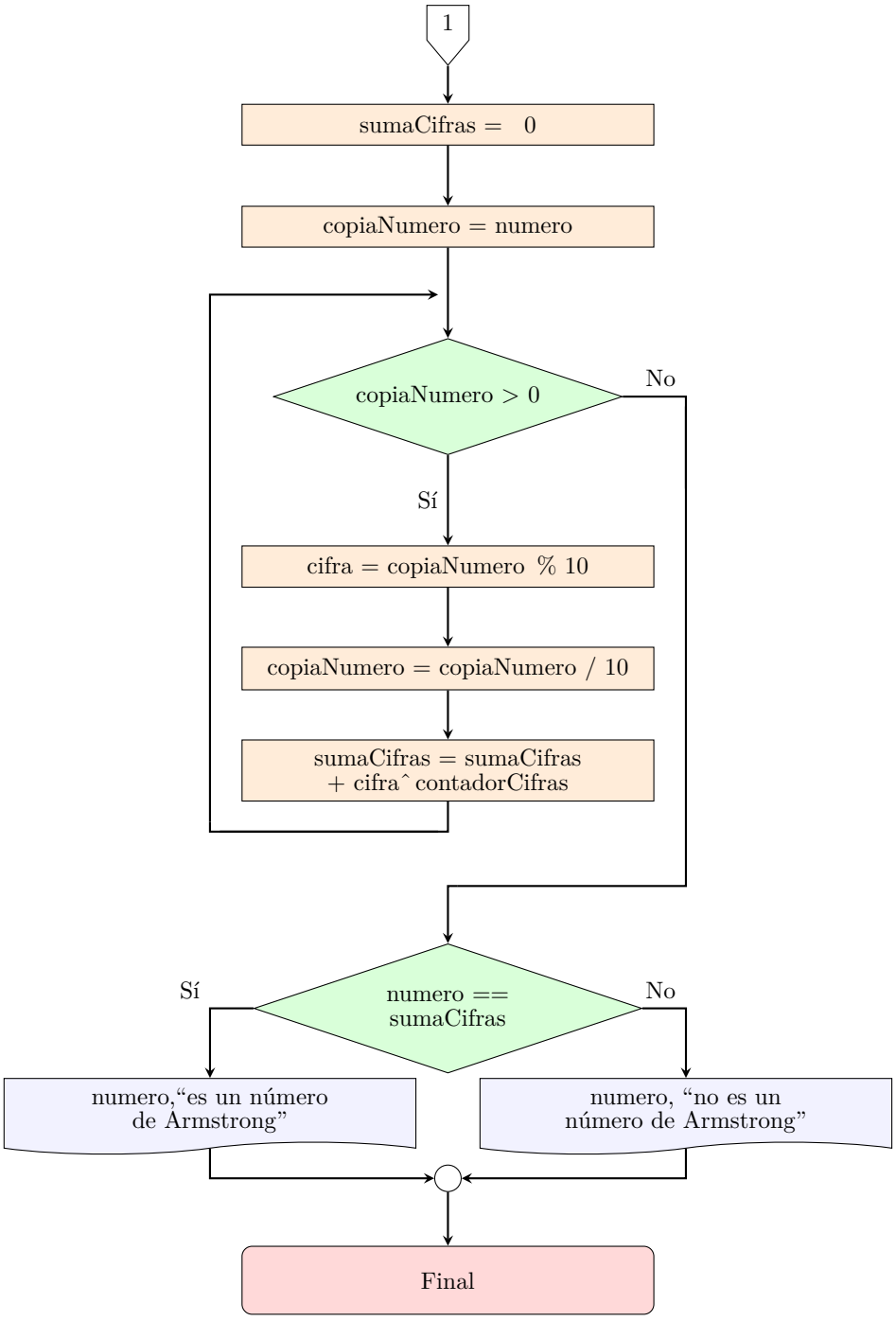


Figura 4.11: Diagrama del flujo del Programa `Armstrong` - Parte 2

Enseguida se muestra el código en Lenguaje C que resuelve el problema que se está tratando. (Programa 4.6).

Programa 4.6: Armstrong

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

int main ()
{
    int long numero, copiaNumero, sumaCifras;
    int      contadorCifras, cifra;

    printf( "Escriba el número a analizar: " );
    scanf( "%ld", &numero );

    copiaNumero = numero;
    contadorCifras = 0;

    while (copiaNumero > 0)
    {
        cifra = copiaNumero % 10;
        copiaNumero = copiaNumero / 10;
        contadorCifras = contadorCifras + 1;
    }

    sumaCifras = 0;
    copiaNumero = numero;

    while( copiaNumero > 0 )
    {
        cifra = copiaNumero % 10;
        copiaNumero = copiaNumero/10;
        sumaCifras = sumaCifras + pow(cifra,contadorCifras);
    }

    if ( numero == sumaCifras )
    {
        printf("%ld es un número de Armstrong",numero );
    }
    else
    {
        printf("%ld no es un número de Armstrong", numero );
    }

    return 0;
}
```

Al ejecutar el programa:

Primera ejecución:

```
Escriba el número a analizar: 1634
1634 es un número de Armstrong.
```

Segunda ejecución:

```
Escriba el número a analizar: 14321
14321 no es un número de Armstrong.
```

Explicación del programa:

Dentro de las bibliotecas que se incluyen se encuentra `math.h`, necesaria para poder usar la función `pow` en el cálculo de pontencias.

En este programa se utilizaron dos estructuras repetitivas `while` independientes. Independientes significa que, primero se ejecuta una en su totalidad y luego se ejecuta la otra.

El primer ciclo `while`, se encarga de contar la cantidad de cifras o dígitos que tiene el número que ingresó el usuario; esto permite saber el valor de la potencia (`contadorCifras`) y poder elevar cada uno de los dígitos. La cuenta de las cifras usa el mismo procedimiento explicado en el Ejemplo 4.3.

El segundo ciclo `while` calcula la sumatoria de los dígitos elevados a la potencia `n` (`contadorCifras`).

Posterior a la ejecución de los dos ciclos, hay una estructura de decisión `if`, que tiene por objetivo determinar si el número suministrado es o no un Armstrong.

Si se analizan los dos ciclos `while` detalladamente, se puede observar que poseen instrucciones parecidas y que ambos separan las cifras, el primero para determinar cuantas cifras tiene el número y el segundo para elevarlas a la potencia y sumarlas.

.:Ejemplo 4.5. *Construya un programa en Lenguaje C que calcule el Máximo Común Divisor (MCD[2]) y el Mínimo Común Múltiplo (mcm[3]) de dos números enteros positivos, utilizando el algoritmo de Euclides. El programa debe ejecutarse hasta que el usuario decida que no desea continuar.*

El programa formulado por Euclides permite encontrar el MCD y el mcm de dos números enteros, `a` *y* `b` *(*`a` $>$ `b`*), a través de los siguientes pasos:*

1. *Se divide* `a` *en* `b` *y se obtiene cociente* c_1 *y resto* r_1*. El cociente no se tendrá en cuenta en la solución del problema.*
2. *Si* r_1 *es diferente de 0, se divide* `b` *en* r_1*, y se obtiene el cociente* c_2 *y el resto* r_2*.*
3. *Si* r_2 *es diferente de 0, se divide* r_1 *en* r_2*, y se obtiene un nuevo cociente y un nuevo residuo.*
4. *Estos pasos se repiten mientras que* r_n *sea diferente de 0.*
5. *El resto anterior (*r_{n-1}*), esto es, el último divisor corresponde al MCD de* `a` *y* `b`*.*

De de la misma forma, se sabe que:

$$mcm = \frac{a * b}{MCD}$$

Análisis del problema:

- **Resultados esperados:** el programa debe mostrar el MCD y el mcm de dos números enteros.
- **Datos disponibles:** los dos números enteros que ingresa el usuario. (`a` y `b`).

[2]El MCD es el mayor número que divide exactamente a dos o más números.

[3]El mcm es el número más pequeño, que no sea 0, que es múltiplo de dos o más números.

- **Proceso:** después de ingresar los dos números (`a` y `b`), se determina si se cumple la condición que `a > b`, de lo contrario, se intercambian los valores de las dos variables (Ver Figura 4.12).

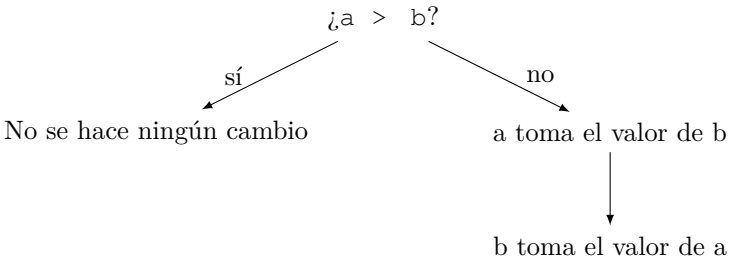


Figura 4.12: Árbol de decisión del Ejemplo 4.5

Para ilustrar el procedimiento descrito anteriormente, se explicará un ejemplo. Imagine que se quiere encontrar el MCD y el mcm de 532 y 112.

1. Suponga `a = 532` y `b = 112`. Lo primero será dividir `a` entre `b`, obteniendo un resto de 84 (r_1=`84`).

$$\frac{532}{112}$$

2. El primer resto (r_1) es distinto a 0, entonces se divide `b` entre `r1`, obteniendo un resto de 28 (r_2=`28`).

$$\frac{112}{84}$$

3. El segundo resto (r_2) es distinto a 0, por lo cuál se hace una nueva división de r_1 entre r_2, obteniendo un resto de 0 (r_3=`0`).

$$\frac{84}{28}$$

4. El nuevo resto (r_3) es igual a 0, por lo tanto no se llevan a cabo nuevas divisiones.
5. El resto anterior (r_{n-1}), es decir, r_2 o el divisor de la última división realizada, cuyo valor es `28`, es el MCD.

De acuerdo con los cálculos que se acaban de explicar en el ejemplo, se nota que, al repetir cada división, el divisor se convierte en el nuevo dividendo y cada nuevo resto se torna en el nuevo divisor.

Luego de hallar el MCD de los dos números, se calcula el mcm con la fórmula propuesta en el enunciado.

En vista de que el problema solicita que se puedan hacer cálculos hasta que el usuario determine que ya no quiere continuar, el programa requiere de otra estructura de repetición que contenga a la estructura de repetición que calculará el MCD y el mcm.

- **Variables requeridas:** los nombres de las variables se asignarán conforme a la fórmula matemática y a las recomendaciones establecidas para el nombre de identificadores.

  - `a` y `b`: almacenarán los números a los que se les encontrará el MCD y el mcm.

  - `dividendo` y `divisor`: guardarán copias del valor de a y b para hacer los cálculos y que no se pierdan los valores originales. `divisor` tendrá el MCD en la última ejecución del ciclo.

  - `resto`: contiene el resto de la división entera. El resto se convertirá en el divisor en la próxima iteración que se lleve a cabo. Servirá también para controlar la repetición del ciclo donde se calculará el MCD.

  - `mcm`: contendrá el mínimo común múltiplo de los dos números ingresados al programa.

  - `seguir`: variable bandera o centinela que recibirá una respuesta del usuario, para saber si se hace o no un nuevo cálculo. Permitirá controlar el ciclo externo.

Las Figuras 4.13 y 4.14 muestran la solución del Ejemplo 4.5 con el uso de un diagrama de flujo.

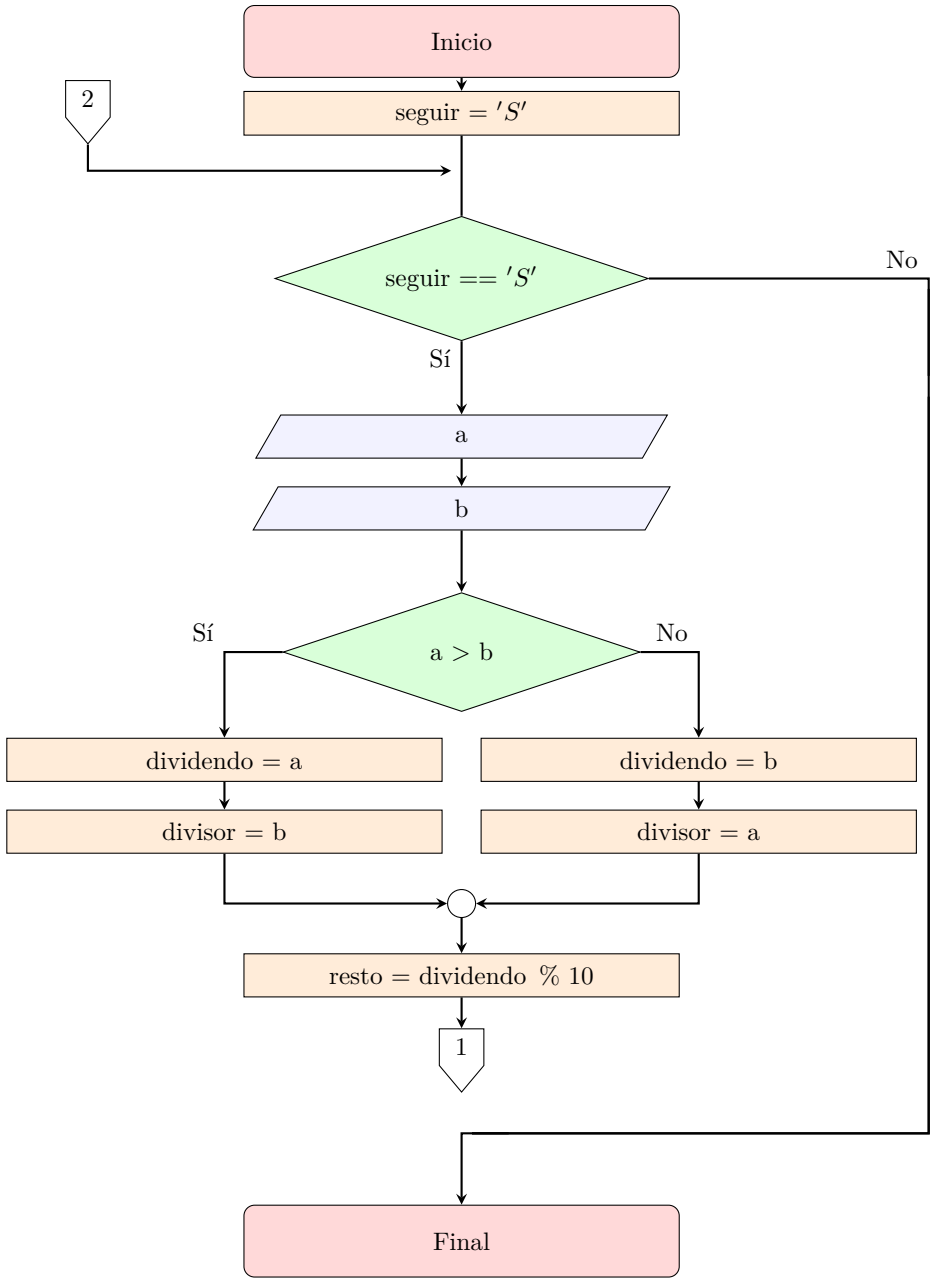


Figura 4.13: Diagrama del flujo del Programa `Euclides` - Parte 1

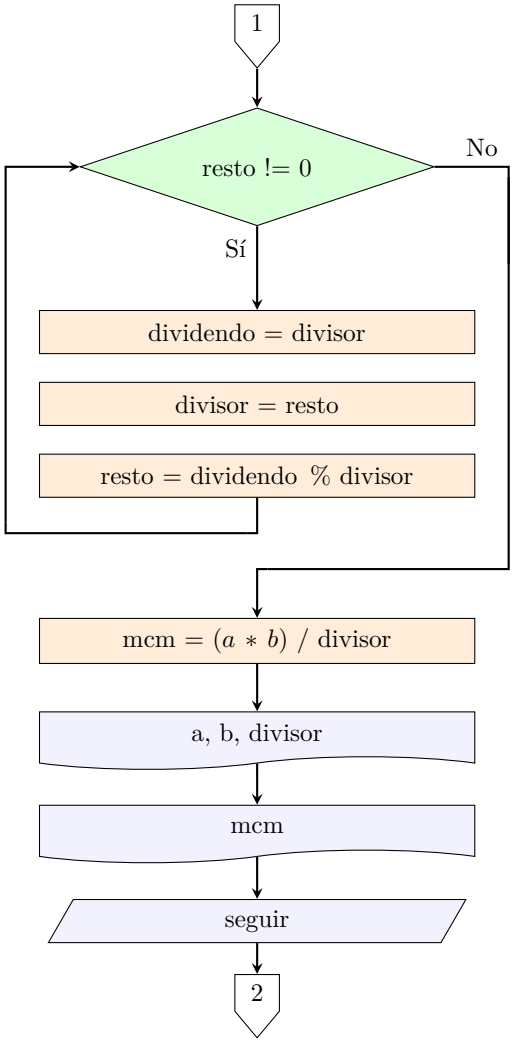


Figura 4.14: Diagrama del flujo del Programa `Euclides` - Parte 2

Antes de presentar el código del programa que soluciona este ejemplo, se hará la siguiente aclaración en lo que concierne al manejo de ciclos anidados.

Aclaración:

Para comprender mejor la solución propuesta a este ejercicio, se explicará el concepto de ciclos anidados.

Se dice que en un programa hay ciclos anidados cuando existe un ciclo que contiene a otro u otros. El ciclo que contiene a otro se denomina ciclo externo y al que está contenido se le denomina ciclo interno. En un programa es posible anidar cualquier cantidad de ciclos (Ver Figura 4.15).

Los ciclos anidados funcionan de manera muy simple. En la Figura 4.15 se observa la `Condición 1`, que determina el inicio del ciclo externo. Siempre y cuando esta condición se cumpla, se ejecutarán las instrucciones dentro de este ciclo, incluyendo al ciclo interno, el cual es controlado por la `Condición 2`. Siempre que esta `Condición 2` sea verdadera, se ejecutarán las instrucciones dentro del ciclo interno. Cuando la `Condición 2` ya no sea verdadera, el ciclo interno termina y se vuelve al ciclo externo continuando con el resto de sus instrucciones; cuando se han ejecutado todas las instrucciones del ciclo externo, se regresa a la `Condición 1` para evaluar si todavía es verdadera y volver a ejecutarlo por completo, incluyendo una nueva iteración del ciclo interno, siempre y cuando la `Condición 2` sea verdadera.

En el momento que la `Condición 1` ya no sea evaluada como verdadera, el ciclo externo finaliza su ejecución y el control del programa continúa con las instrucciones que se encuentren debajo de este ciclo.

Una vez haya analizado la forma de trabajar de las estructuras repetitivas anidadas, observe el Programa 4.7 con la solución a este ejemplo.

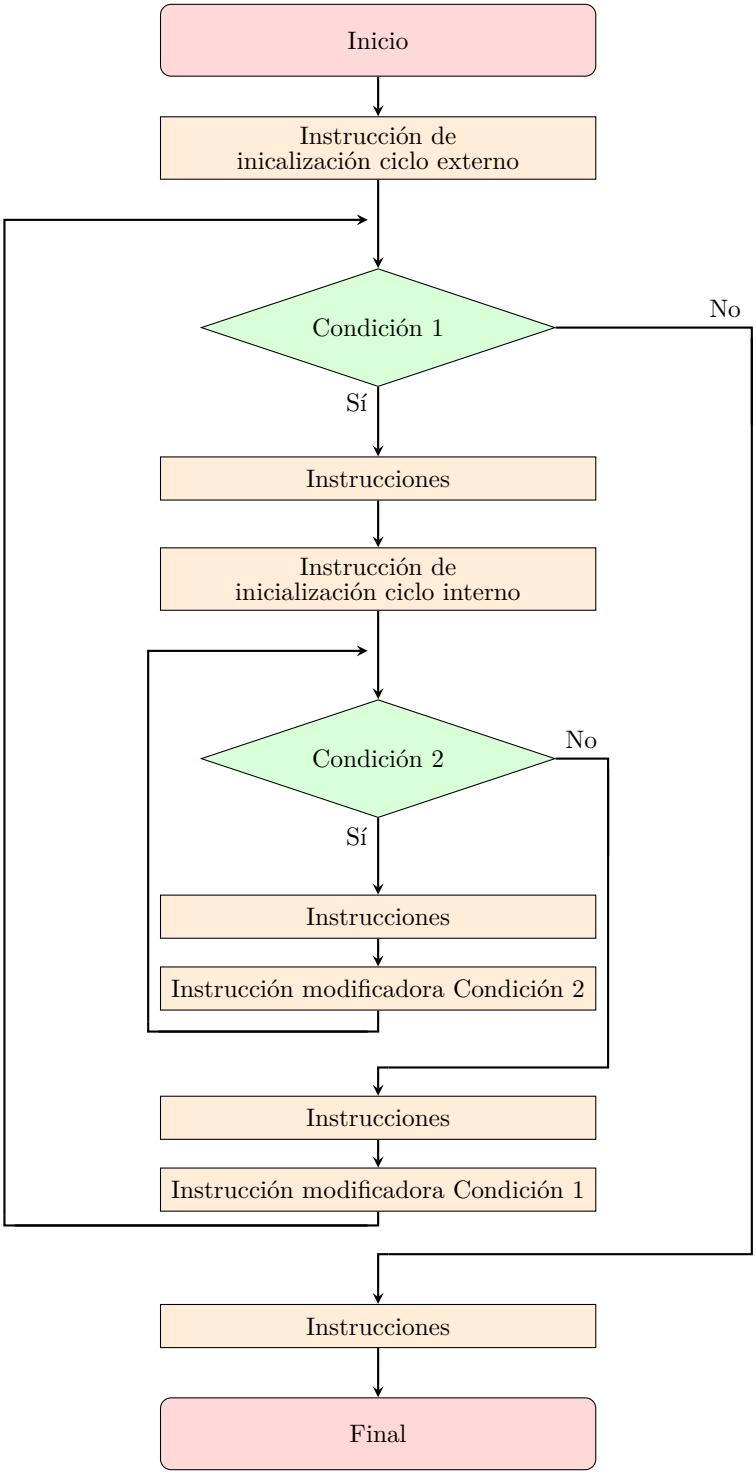


Figura 4.15: Ciclos Anidados

Programa 4.7: Euclides

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

int main ()
{
  int  a, b, dividendo, divisor, resto, mcm;
  char seguir;

  seguir = 'S';
  while (seguir == 'S' )
  {
    printf( "Ingrese el primer número: " );
    scanf( "%d", &a );

    printf( "Ingrese el segundo número: " );
    scanf( "%d", &b );

    if( a > b )
    {
      dividendo = a;
      divisor   = b;
    }
    else
    {
      dividendo = b;
      divisor   = a;
    }

    resto = dividendo % divisor;

    while ( resto != 0 )
    {
      dividendo = divisor;
      divisor   = resto;
      resto     = dividendo % divisor;
    }

    mcm = (a * b) / divisor;

    printf( "\n\nEl máximo común divisor de: %d y %d es: %d
        \n", a, b, divisor );
    printf( "El mínimo común múltiplo es: %d \n", mcm );
    printf( "\nDesea realizar nuevos cálculos [S] o [N]?:");
    seguir = toupper( getchar() );
  }

  return 0;
}
```

Al ejecutar el programa:

```
Ingrese el primer número: 532
Ingrese el segundo número: 112

El máximo común divisor de:  532 y 112 es: 28
El mínimo común múltiplo es: 2128

Desea realizar nuevos cálculos [S] o [N]?: N
```

Explicación del programa:

Este Ejercicio 4.7 se resolvió mediante la implementación de dos estructuras repetitivas `while` anidadas.

El ciclo externo lo conforman las líneas de la 10 a la 44. El ciclo interno va desde la línea 31 a la 36.

El ciclo externo se ejecuta primero desde la línea 10 hasta la 29. Si la condición en la línea 31 se cumple, el programa ingresa a ejecutar las instrucciones del ciclo interno.

En el momento en que la condición del ciclo interno ya no sea verdadera, el programa se sale del ciclo interno y regresa al externo (líneas 37 a la 43). Posteriormente, el programa regresa a la línea 10 y evalúa la condición nuevamente, si el resultado es verdadero se ingresa otra vez al ciclo externo y se ejecutan de nuevo todas las instrucciones, incluyendo el ciclo interno. Este proceso se repite siempre y cuando las dos condiciones de las estructuras `while` sean verdaderas.

Recuerde que, en este programa la estructura externa controla la repetición de todo el proceso con el uso de la variable `seguir`, cuyo valor inicial es 'S', para que, al evaluar por primera vez la condición del primer `while` (línea 10) se obtenga un resultado verdadero y se ingrese al cuerpo de este ciclo.

Las líneas 42 y 43 corresponden a la instrucción modificadora de la condición del ciclo externo, ya que dependiendo de lo que el usuario ingrese, este ciclo continuará o no su ejecución. Es posible que la respuesta del usuario a la pregunta se ingrese en minúscula, pero con la función `toupper` que hace parte de la biblioteca `ctype.h`, el dato almacenado siempre estará en mayúscula.

Cuando el usuario ingrese una respuesta positiva a la pregunta de la línea 42, se solicitarán dos nuevos valores para `a` y `b`, y se repetirá todo el proceso que calcula MCD y el mcm. El programa terminará cuando la

respuesta a la pregunta sobre si se desea continuar sea negativa.

El ciclo interno implementado también con `while` tiene como propósito calcular las respectivas divisiones mientras la variable resto posea un valor distinto a 0.

Aclaración:

En las estructuras repetitivas anidadas, hasta que el ciclo interno no termine de ejecutarse, el ciclo externo no retomará el control del programa.

4.2.1 Prueba de escritorio

Previamente en el Capítulo 2, se explicó que una prueba de escritorio es una estrategia para hacerle seguimiento a cada línea escrita en un programa y determinar si está funcionando adecuadamente o no. Recuerde que una prueba de escritorio se implementa mediante una tabla de verificación.

En la siguiente sección se explicará una prueba de escritorio para un programa implementado con el ciclo `while`.

.:Ejemplo 4.6. *El siguiente programa multiplica dos números enteros positivos a través de sumas sucesivas, realice la respectiva prueba de escritorio, utilizando una tabla de verificación.*

Programa 4.8: Multiplicacion

```
1  #include <stdio.h>
2
3  int main ()
4  {
5    int multiplicando, multiplicador, producto, contador;
6
7    printf( "Ingrese el multiplicando: ");
8    scanf(  "%d", &multiplicando );
9
10   printf( "Ingrese el multiplicador: ");
11   scanf(  "%d", &multiplicador );
12
13   producto = 0;
14   contador = 0;
15   while ( contador < multiplicador )
16   {
17     producto += multiplicando;
18     contador ++;
```

```
19    }
20
21    printf( "El producto es: %d", producto );
22
23    return 0;
24 }
```

Al ejecutar el programa:

```
Ingrese el multiplicando: 75
Ingrese el multiplicador: 6
El producto es: 450
```

Explicación de la prueba de escritorio:

Algunas variables se ubican antes de la tabla (valores iniciales), o después de ella, cuando se trate de valores de salida. La tabla de verificación suele tener columnas donde se plasman las expresiones lógicas o relacionales, ya que ayudan al seguimiento de las instrucciones del programa.

Para esta prueba se requiere de dos datos, el multiplicando y el multiplicador. Se van a tomar dos valores fáciles de trabajar, como se observa en la Tabla 4.3.

`multiplicando = 75`
`multiplicador = 6`

producto	contador	contador< multiplicador
0	0	0 < 6 (V)
75	1	1 < 6 (V)
150	2	2 < 6 (V)
225	3	3 < 6 (V)
300	4	4 < 6 (V)
375	5	5 < 6 (V)
450	6	6 < 6 (F)

`producto = 450`

Tabla 4.3: Prueba de escritorio - Programa 4.6

En este ejemplo, las variables `multiplicando`, `multiplicador`, `producto` y `contador`, empiezan conteniendo los valores 75, 6, 0 y 0 respectivamente.

Posteriormente, se evalúa la condición del ciclo `while` (línea 15), en este caso, $0 < 6$, lo que arroja un resultado verdadero, por tanto, se ejecuta el cuerpo del ciclo.

Dentro del ciclo, se lleva a cabo la operación `producto = producto + multiplicando`, que a través de las iteraciones, va efectuando

las sumas sucesivas del multiplicando. Por cada iteración, la variable `producto` aumenta su valor en 75.

La instrucción:

```
18 contador ++;
```

incrementa la variable `contador` en 1. Luego, el programa retorna el control a la condición que está al principio del ciclo y así poder determinar si se lleva a cabo otra iteración. Este proceso es reiterativo mientras `contador` tenga un valor menor al de `multiplicador`. Cuando la condición de la línea 15 sea $6 < 6$, el resultado será falso y el programa ya no ejecutará el cuerpo del ciclo sino lo que hay después de este, es decir, imprimirá el producto.

Buena práctica:

La tabla de verificación debe ser clara, fácil de comprender y debe registrar cada uno de los cálculos realizados, así como las evaluaciones efectuadas a las condiciones.

Es recomendable asignar valores fáciles de procesar a las variables que reciben los datos de entrada al programa.

Actividad 4.2

Implemente un programa en Lenguaje C, para cada uno de los siguientes enunciados.

1. Suponga que tiene una población de, máximo 500 habitantes, encuentre cuántos son mayores y cuántos menores de edad. De cada uno de ellos se conoce la edad en años.

2. Genere e imprima los múltiplos de 3 que se encuentren entre 6 y n, donde n tiene que ser superior a 6.

3. En una compañía que tiene varias sucursales a nivel nacional, una o varias en cada departamento y solo una por ciudad, se requiere de un censo que permita conocer la siguiente información de sus empleados:

Porcentaje total de personas que tienen estudios de primaria, secundaria, profesional, maestría o doctorado. Se tiene en cuenta solamente, el nivel más alto de estudio. Porcentaje total de mujeres con posgrado, con relación a todas las mujeres de la compañía. Cantidad de hombres con estudios de solo primaria en cada departamento. Cantidad de hombres y mujeres en cada sucursal que hayan recibido su título profesional antes de cumplir 25 años de edad. De cada empleado se conoce el número de identificación y el nombre, adicional a los datos que se requieren en la solución del problema.

4. Usando la estructura repetitiva `while`, elabore un programa, que genere e imprima las letras del abecedario de la siguiente forma:

```
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
Y X W V U T S R Q P O N M L K J I H G F E D C B A
X W V U T S R Q P O N M L K J I H G F E D C B A
W V U T S R Q P O N M L K J I H G F E D C B A
V U T S R Q P O N M L K J I H G F E D C B A
U T S R Q P O N M L K J I H G F E D C B A
T S R Q P O N M L K J I H G F E D C B A
S R Q P O N M L K J I H G F E D C B A
R Q P O N M L K J I H G F E D C B A
Q P O N M L K J I H G F E D C B A
P O N M L K J I H G F E D C B A
O N M L K J I H G F E D C B A
N M L K J I H G F E D C B A
M L K J I H G F E D C B A
L K J I H G F E D C B A
K J I H G F E D C B A
J I H G F E D C B A
I H G F E D C B A
H G F E D C B A
G F E D C B A
F E D C B A
E D C B A
D C B A
C B A
B A
A
```

4.3. Estructura do - while

Al igual que la estructura while, esta estructura hace que una instrucción o un conjunto de ellas se ejecuten una o más veces.

La característica principal del ciclo do-while es que la condición se encuentra al final del mismo, lo cual garantiza que sus instrucciones se ejecuten por lo menos una vez, ya que no existe un impedimento para que el control del programa ingrese al cuerpo del ciclo.

En las siguientes líneas de código, se muestra la estructura general del ciclo do-while.

```
1 Instrucción de inicialización;
2 do
3 {
4   Instrucción-1;
5   Instrucción-2;
6     ...                    /* Cuerpo del ciclo */
7   Instrucción-n;
8   Instrucción modificadora de condición;
9 } while( condición );
10 Instrucción externa;
```

Esta forma general se interpreta así:

La Instrucción de inicialización está conformada por una o más instrucciones que dan un valor inicial a las variables que oficiarán como contadores o acumuladores; puede darse el caso de programas que no requieran de ellos. Las variables que hacen parte de la condición del ciclo do - while se podrían inicializar también dentro del ciclo; esto dependiendo de las características del problema a resolver.

El cuerpo del ciclo se encuentra delimitado por las palabras reservadas do, y while (condición); si cuando se evalúa la condición, el resultado es verdadero, el control del programa regresa hasta la palabra do y se repite la ejecución de las instrucciones que conforman el cuerpo del ciclo. La condición será una expresión relacional o lógica.

Las iteraciones finalizarán cuando la evaluación de la condición genere un resultado falso; en ese momento el control del programa irá a la Instrucción externa, esto es, la que esté debajo de la palabra reservada while, la cual no hace parte del ciclo.

El ciclo do - while, también requiere de la Instrucción modificadora de condición (línea 8), que, en un momento dado cambia el estado de la

condición. Si esta instrucción no está presente, se obtendrá un ciclo infinito, debido a que las iteraciones "nunca terminarán". Aunque generalmente se ubica como la última instrucción del cuerpo del ciclo, no necesariamente tiene que ocupar esa posición.

La Figura 4.16 corresponde al diagrama de flujo de la estructura de repetición `do - while`.

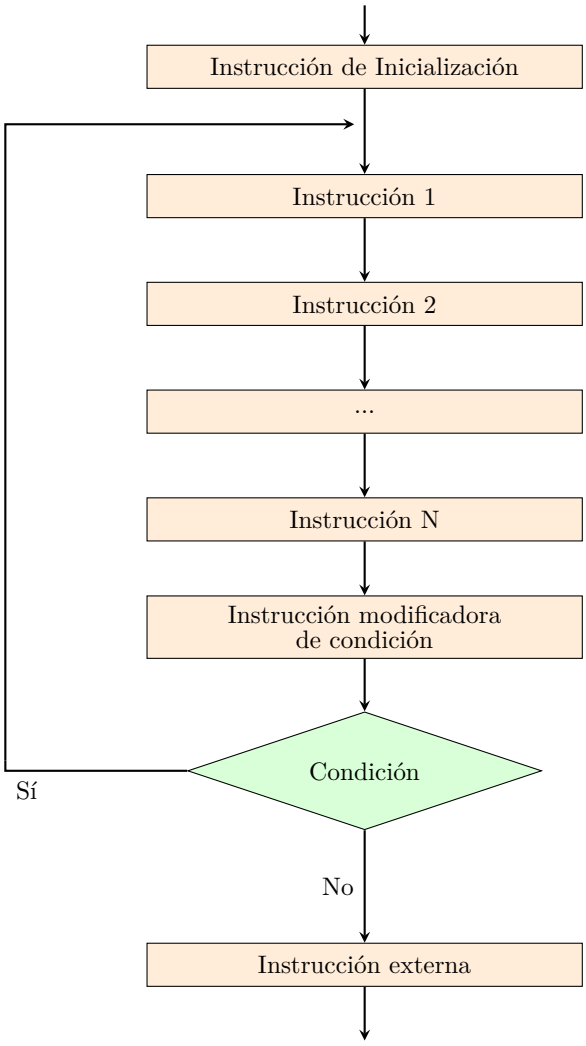


Figura 4.16: Estructura de repetición `do - while`

A continuación se explicarán algunos ejemplos del uso del ciclo `do - while`.

.:Ejemplo 4.7. *Este programa genera e imprime los números del 1 al 10, usando el ciclo* `do-while`.

Los diagramas de flujo de la Figura 4.17 representan las dos versiones creadas para resolver el problema (Programas 4.9 y 4.10).

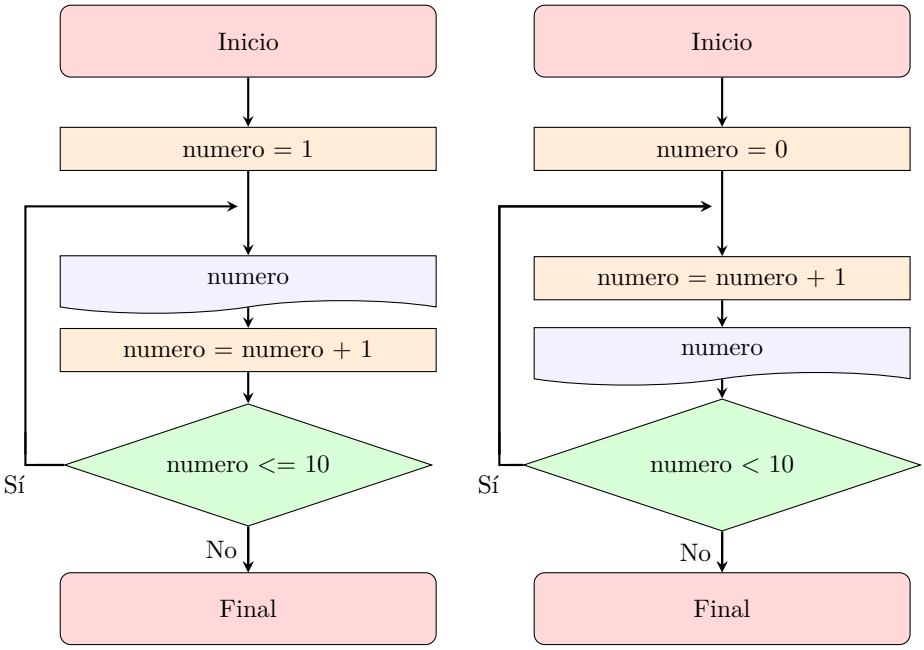


Figura 4.17: Imprimen los números del 1 al 10 (`do-while`)

Programa 4.9: Numeros-1-10-do-while Ver. 1

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int numero;
6
7    numero = 1;
8
9    do
10   {
11       printf("%d\n", numero);
12       numero = numero + 1;
13   } while ( numero <= 10 );
14
15   return 0;
16 }
```

Este mismo problema fue implementado, en la sección anterior, usando el ciclo `while`.

La Tabla 4.4 identifica las partes que conforman el segmento del Programa 4.9.

Línea	Explicación
7	Instrucción de inicialización
9	Inicio del ciclo
11	Cuerpo del ciclo
12	Cuerpo del ciclo e instrucción modificadora de condición
13	Fin del ciclo y condición. Regresa el control a la línea 9.

Tabla 4.4: Explicación programa 4.7

En la línea 7, la variable `numero` se inicializa en 1. En la línea 9 está el inicio del ciclo, identificado con `do`. Las líneas 11 y 12 componen el cuerpo del ciclo. La línea 13 representa el final del ciclo y la condición. El ciclo deja de iterar cuando la condición sea falsa.

La instrucción `printf` se lleva a cabo 10 veces, mostrando los números del 1 al 10, uno por uno con cada iteración.

La operación `numero = numero + 1` además de incrementar en uno el valor de la variable `numero` en cada iteración, representa también la instrucción modificadora de condición, ya que cuando `numero` tome el valor de 11, la condición `(numero <= 10)` será falsa, lo que terminará la ejecución del `do-while`.

Como ya se ha manifestado en repetidas ocasiones, un problema puede ser solucionado de diferentes maneras, lo cual genera nuevas versiones de los programas.

Es por eso, que otra posible solución para este ejemplo, es la que se muestra en el Programa 4.10:

Programa 4.10: Numeros-1-10-do-while Ver. 2

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int numero;
6
7    numero = 0;
8
9    do
10   {
11       numero = numero + 1;
12       printf("%d\n", numero);
13   } while ( numero < 10 );
14
15   return 0;
16 }
```

Al comparar la segunda versión de este programa (Programa 4.10) con la primera (Programa 4.9) se observan algunos aspectos importantes:

Línea 7: la inicialización es distinta, se hizo en 0.

Línea 9: es igual en ambas, corresponde al inicio del ciclo `do - while`

Líneas 11 y 12: Las instrucciones que conforman el cuerpo del ciclo son las mismas en ambas versiones, pero en orden inverso; esto garantiza que se impriman los números del 1 al 10, como se espera para este programa.

Línea 13: en la segunda versión de este programa, el ciclo se ejecuta mientras el valor de `numero` sea menor a 10, en lugar de menor o igual a 10, como se escribió en la primera versión. Esto es así, porque la inicialización fue en 0 y no en 1. Tenga presente que en la primera versión del programa la variable `numero` llega a tomar el valor de 11, en la segunda versión llega hasta 10.

Aclaración:

Las instrucciones que componen el cuerpo del ciclo `do-while`, se ejecutarán al menos una vez.

.:Ejemplo 4.8. *Escriba un programa en Lenguaje C que genere y muestre la siguiente serie:* 1 3 5 7 9 11 ... *n*

Análisis del problema:

El análisis de este problema es el mismo del Ejemplo 4.1, allí se solucionó con la estructura `while`. Aquí se propone la solución con el ciclo `do-while`, como se muestra en el Programa 4.11.

Programa 4.11: SerieVer.2

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int contadorNumeros, cantidadTerminos, termino;
6
7    printf( "Ingrese la cantidad de términos a generar: " );
8    scanf ( "%d", &cantidadTerminos );
9
10   contadorNumeros = 0;
11   termino = 1;
12
13   do
14   {
15     printf( "%d ", termino );
16     termino = termino + 2;
17     contadorNumeros = contadorNumeros + 1;
18   } while ( contadorNumeros <= cantidadTerminos );
19
20   return 0;
21 }
```

Al ejecutar el programa:

```
Ingrese la cantidad de términos a generar: 15
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
```

Explicación del programa:

La explicación de la forma en que se resolvió este programa es muy similar a la propuesta para el Programa 4.3. La única diferencia está en el ciclo utilizado. El Programa 4.3 empleó el ciclo `while` cuya condición está al inicio del ciclo. En este programa se empleó el ciclo `do-while`, con la condición al final; por lo demás, las instrucciones son mismas, así como los resultados.

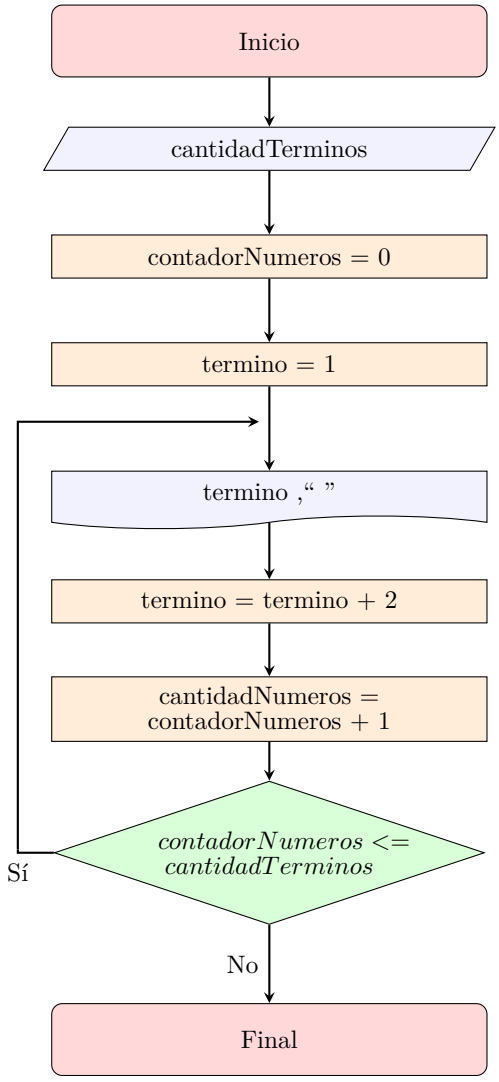


Figura 4.18: Diagrama de flujo del Programa `Serie Versión 2`

Aclaración:

Tanto el ciclo `while` y el `do-while` se ejecutan mientras la evaluación de la condición arroje un resultado verdadero.

El `while` evalúa la condición al comienzo, el `do-while` la evalúa al final.

.:Ejemplo 4.9. *El profesor de la asignatura de Programación para dispositivos móviles, desea conocer la opinión de sus estudiantes sobre la plataforma sobre la cual desean trabajar. Las opciones disponibles son Android e iOS. Cada estudiante podrá votar ingresando su código y la plataforma que prefiere. En caso de que se ingrese una plataforma diferente, se informará y no se tendrá en cuenta el voto.*

Análisis del problema:

- **Resultados esperados:** mostrar el nombre de la plataforma que obtuvo la mayor votación, incluyendo la cantidad de votos. Cuando un estudiante elija una opción diferente a las plataformas dadas, se informará esta situación.
- **Datos disponibles:** se conoce el código y la plataforma de predilección de cada estudiante, así mismo se posee la respuesta que proporciona el usuario a la pregunta sobre continuar o no con el proceso de votación.
- **Proceso:** se solicita a cada estudiante su código y la plataforma de su predilección; esto implica realizar iteraciones. Paralelamente, se van contando los votos emitidos por cada plataforma. (Ver Figura 4.19).

  El usuario determinará cuando terminar con el proceso.

  Una vez finalizadas las votaciones, el programa mostrará la cantidad de votos obtenidos por cada plataforma y se determinará cuál fue la ganadora, para ello se hará uso de una estructura de decisión. Como es posible que haya empate en la votación, se debe contemplar esta posibilidad y mostrar un mensaje . (Ver Figura 4.20).

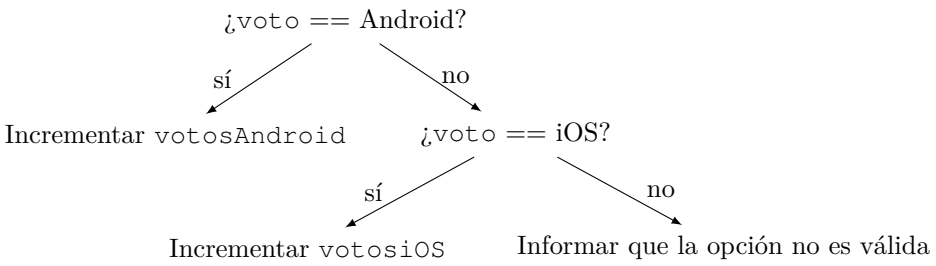


Figura 4.19: Árbol de decisión del Ejemplo 4.9 - Parte 1

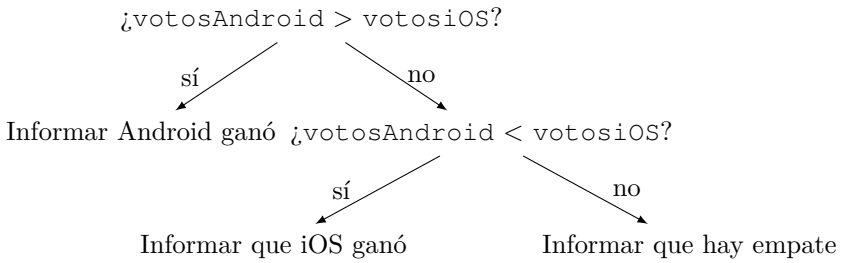


Figura 4.20: Árbol de decisión del Ejemplo 4.9 - Parte 2

- **Variables requeridas:**
  - `codigo`: representa el código del estudiante.
  - `voto`: plataforma votada por el estudiante.
  - `votosAndroid`: cuenta los votos por la plataforma Android.
  - `votosiOS`: cuenta los votos por la plataforma iOS.
  - `seguir`: recibe la respuesta del usuario sobre si continuar el proceso de votación o no.

Las Figuras 4.21 y 4.22 muestran la solución del Ejemplo 4.4 a través de diagramas de flujo.

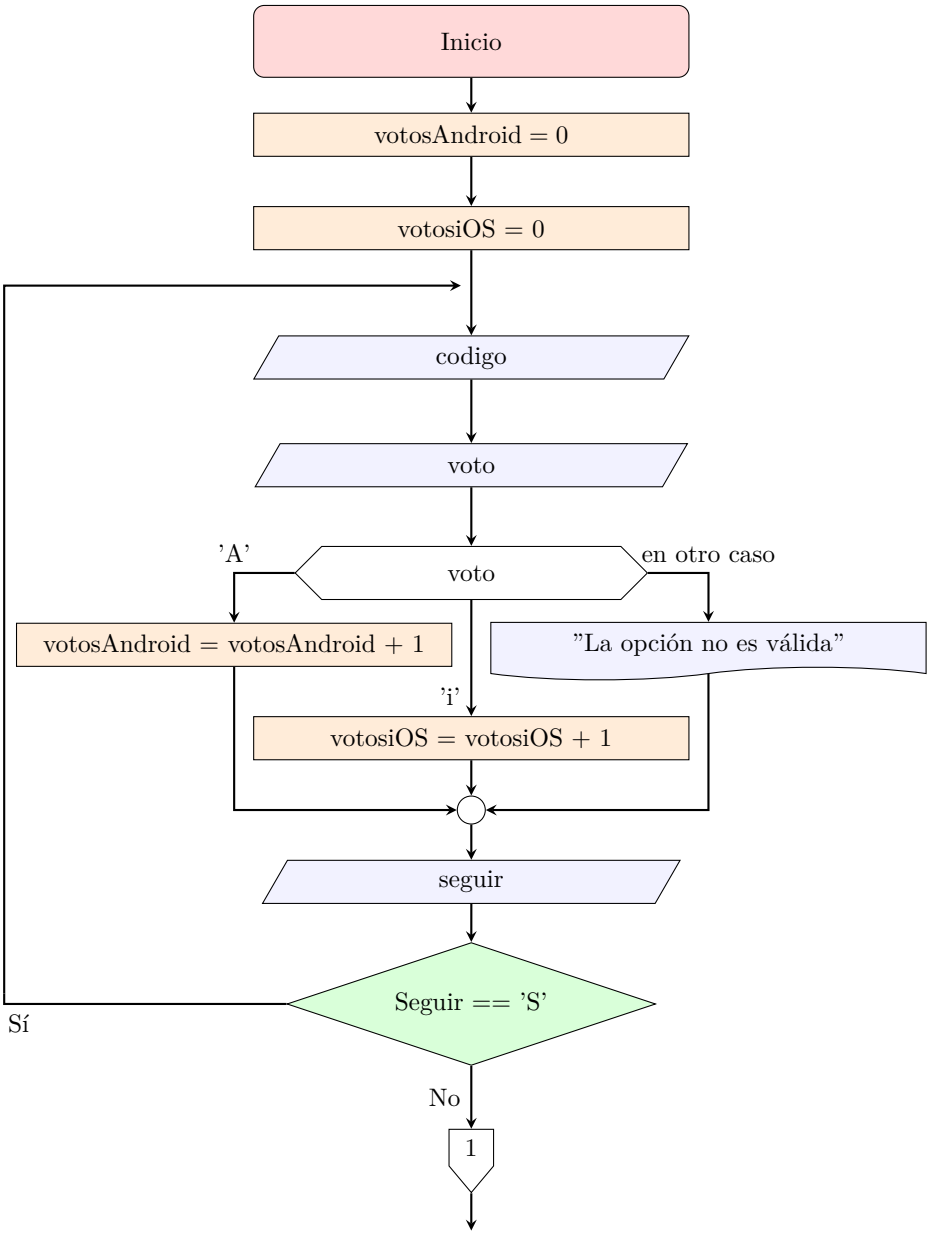


Figura 4.21: Diagrama de flujo del Programa `Plataformas` - Parte 1

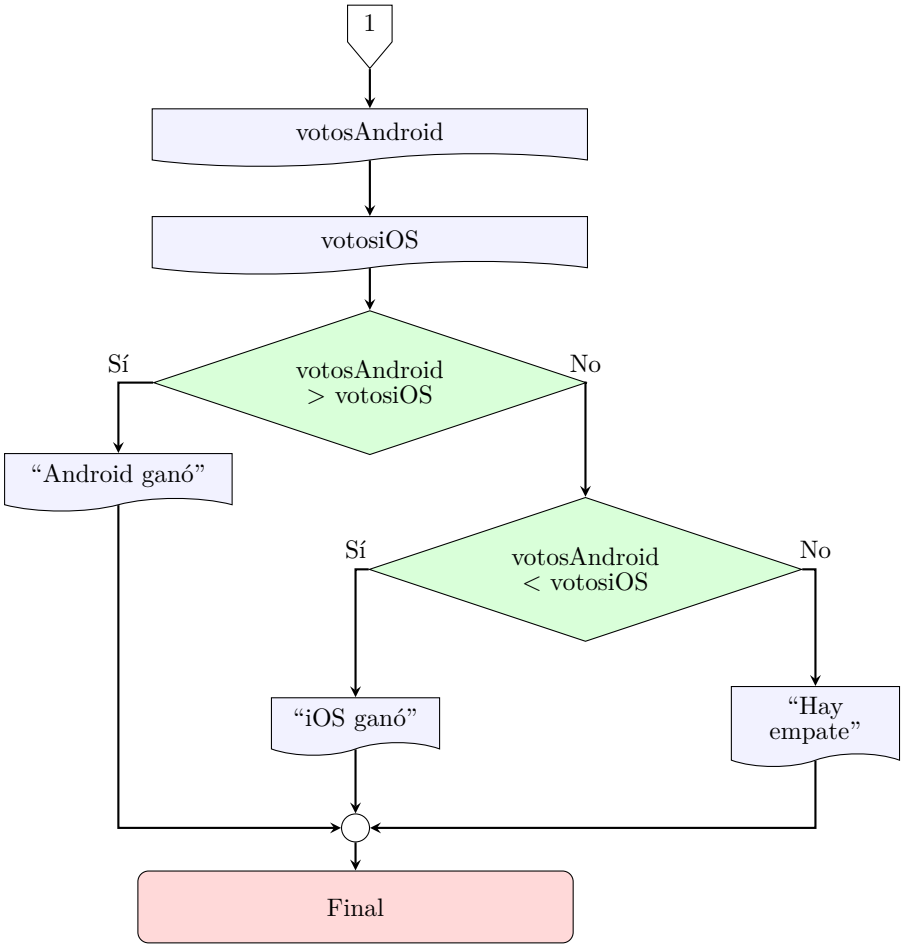


Figura 4.22: Diagrama de flujo del Programa `Plataformas` - Parte 2

En el Programa 4.12 se muestra el código del Programa de plataformas.

Programa 4.12: Plataformas

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char seguir, voto;
6    char codigo[ 20 ];
7    int  votosAndroid, votosiOS;
8
9    votosAndroid = 0;
10   votosiOS     = 0;
```

```
11    do
12    {
13      printf( "Ingrese el código del estudiante: " );
14      scanf("%s", codigo );
15
16      printf( "PLATAFORMAS DISPONIBLES\n" );
17      printf( "[A]Android\n" );
18      printf( "[i]OS\n\n" );
19      printf( "Elija su opción: " );
20      scanf( " %c", &voto );
21
22      switch ( voto )
23      {
24        case 'A':
25        case 'a': votosAndroid = votosAndroid + 1;
26             break;
27
28        case 'I':
29        case 'i': votosiOS = votosiOS + 1;
30              break;
31
32        default: printf( "La opción no es válida" );
33      }
34      printf( "Desea realizar un nuevo voto [S] [N]?: " );
35      scanf ( " %c", &seguir );
36    } while ( seguir == 'S' || seguir == 's' );
37
38    printf( "Votos por Android: %d\n", votosAndroid );
39    printf( "Votos por iOS: %d\n", votosiOS );
40
41    if( votosAndroid > votosiOS )
42    {
43      printf( "Android ganó" );
44    }
45    else
46    {
47      if( votosAndroid < votosiOS )
48      {
49        printf( "iOS ganó" );
50      }
51      else
52      {
53        printf( "Hay empate" );
54      }
55    }
56
57    return 0;
58  }
```

Explicación del programa:

Luego de declarar las variables se inicializaron los contadores `votosAndroid` y `votosiOS` en cero.

Con la instrucción `do` se inicia el proceso iterativo. El ciclo `do - while` contiene las instrucciones que leen el código del estudiante y su elección de plataforma.

El estudiante vota por Android ingresando una 'A' o 'a', y vota por iOS ingresando una 'I'o 'i'. Luego de ingresar el voto, este se contabiliza, para ello se empleó una estructura `switch` con tres opciones:

```
22      switch ( voto )
23      {
24        case 'A':
25        case 'a': votosAndroid = votosAndroid + 1;
26              break;
27
28        case 'I':
29        case 'i': votosiOS = votosiOS + 1;
30              break;
31
32        default: printf( "La opción no es válida" );
33      }
```

Si el usuario ingresa un carácter diferente a los contemplados para las plataformas, el mensaje "La opción no es válida" se mostrará y el voto no se contabilizará.

Posterior al `switch` se encuentra la instrucción modificadora de condición:

```
36      printf( "Desea realizar un nuevo voto [S] [N]?: " );
37      scanf ( " %c", &seguir );
```

Analice cómo en esta solución no fue necesario inicializar la variable `seguir` antes del ingreso al ciclo, ya que previo a la condición presente en el `while`, el valor que toma esta variable es ingresado por el usuario y capturado con (`scanf`).

Con la respuesta que entregue el usuario, se evalúa la condición:

```
38    } while ( seguir == 'S' || seguir == 's' );
```

Si su resultado es verdadero, el ciclo repite el proceso. Si la condición es falsa, la ejecución del ciclo finaliza y se continúa con la instrucción que está por debajo del `while`.

Por último, el programa muestra los votos obtenidos por cada plataforma (líneas 38 y 39) e informa cuál es la ganadora; para ello, se utilizó una estructura de decisión anidada (líneas 41 a 55) que contempla también la posibilidad del empate.

.:Ejemplo 4.10. *Construya un programa en Lenguaje C, que calcule el factorial de un número entero ingresado por el usuario.*

El factorial de un número, corresponde al producto del número por todos sus antecesores hasta uno. Se sabe que los números negativos no tienen factorial y que el factorial de 0 es 1. Matemáticamente, el factorial se representa con un signo de exclamación al lado del número (n!).

***Por ejemplo**:*

$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040$

o se puede expresar como:

$7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040.$

A $7!$ *se le llama* 7 *factorial, o también el factorial de* 7*.*

Análisis del problema:

- **Resultados esperados:** el factorial del número ingresado o un mensaje que informe que el número no tiene factorial, si el número es negativo.
- **Datos disponibles:** el número al que se le va a calcular el factorial.
- **Proceso:** se obtiene el número que porporciona el usuario. Luego mediante una estructura de decisión se determina si el número es negativo, en cuyo caso se informa que el número no tiene factorial; de lo contrario se calcula el factorial y se muestra en pantalla.

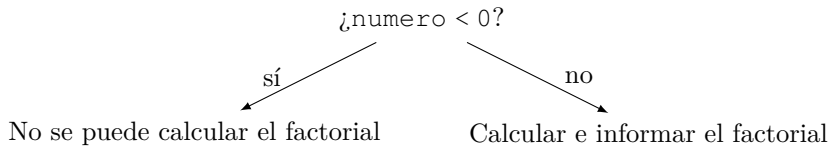


Figura 4.23: Árbol de decisión del Ejemplo 4.10

El cálculo del factorial requiere de un proceso cíclico. Para comprender este proceso se analizará el cálculo del factorial de 7

$(7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040)$, de acuerdo a como se muestra en la Tabla 4.5.

Inferiores	**Cálculo**	**n!**
1	1	1
2	1! x 2	2
3	2! x 3	6
4	3! x 4	24
5	4! x 5	120
6	5! x 6	720
7	6! x 7	5040

Tabla 4.5: Cálculo de 7!

La columna rotulada como `Inferiores` muestra los números inferiores a 7 hasta él inclusive. En la columna `Cálculo` se observa el proceso acumulativo que se hace para el cálculo del factorial (`n!`); el factorial anterior es multiplicado por el número que está en la misma fila en la primera columna. La tercera columna acumula los resultados en cada iteración o cálculo realizado; al final se obtiene el resultado: $7! = 5040$.

Con el fin de diseñar el proceso descrito en la Tabla 4.5 y en el párrafo anterior, se hace necesario utilizar una estructura de repetición que se ejecute mientras que un contador que inicie en 1 llegue hasta el valor del número al que se le calculará el factorial.

- **Variables requeridas:**
  - `numero`: almacena el número al que se le calculará el factorial (n).
  - `factorial`: contendrá el resultado del cálculo del factorial ($n!$).
  - `inferiores`: variable que almacenará los valores desde 1 hasta el número al que se le calculará el factorial. También será parte de la condición que controlará el ciclo que realizará los cálculos.

La Figura 4.24 representa la solución del Ejemplo 4.10 a través de un diagrama de flujo.

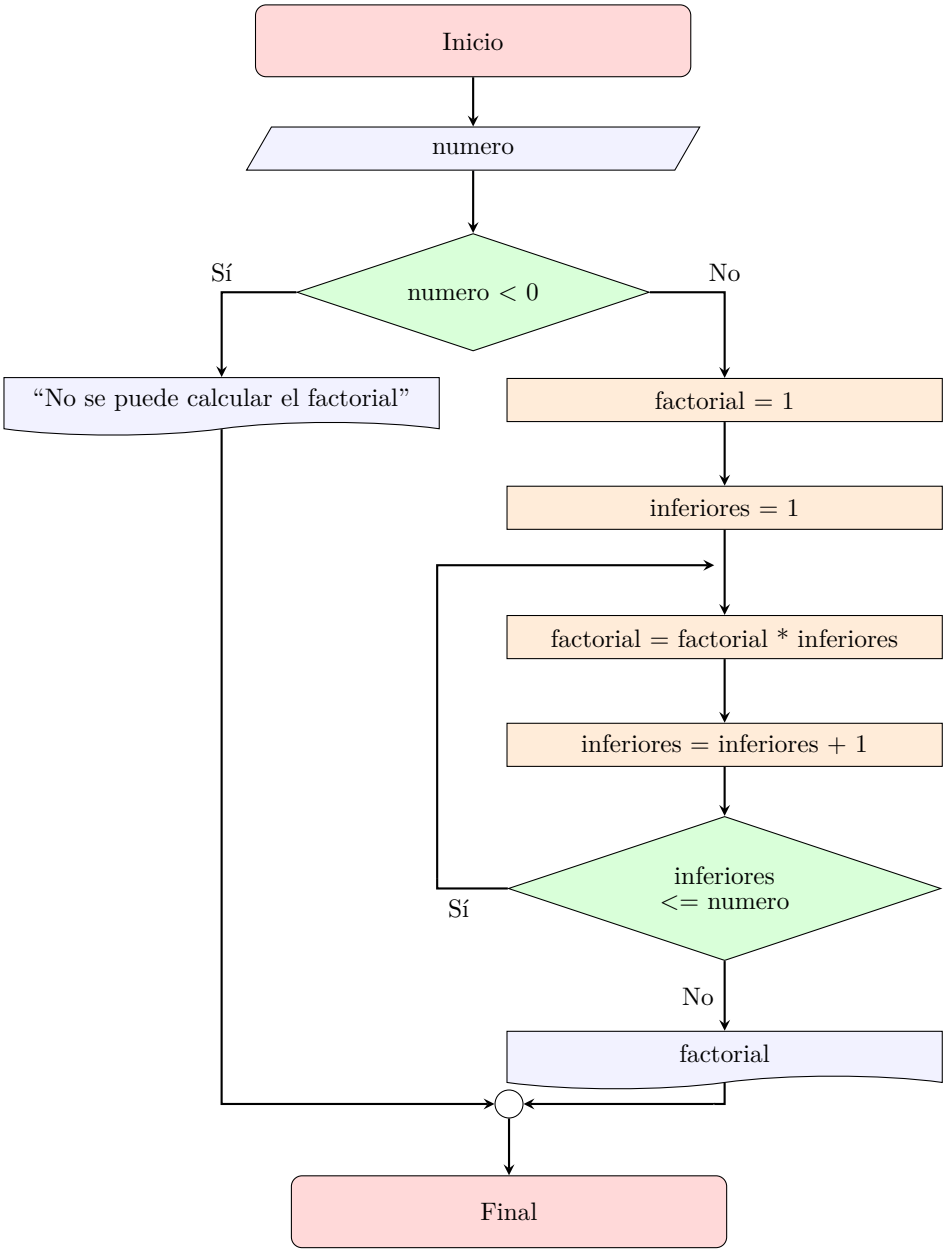


Figura 4.24: Diagrama de flujo del Programa `FactorialNumero`

La solución de este ejercicio en Lenguaje C se propone en el Programa 4.13.

Programa 4.13: FactorialNumero

```c
#include <stdio.h>

int main()
{
  int numero, factorial, inferiores;

  printf( "Ingrese el número para el factorial: " );
  scanf( "%d", &numero );

  if( numero < 0 )
  {
    printf( "No se puede calcular el factorial" );
  }
  else
  {
    factorial  = 1;
    inferiores = 1;
    do
    {
      factorial  = factorial * inferiores;
      inferiores = inferiores + 1;
    } while( inferiores <= numero );

    printf( "Factorial de %d es %d", numero, factorial );
  }
  return 0;
}
```

Al ejecutar el programa:

Primera ejecución:

```
Ingrese el número para el factorial: 9
Factorial de 9 es: 362880
```

Segunda ejecución:

```
Ingrese el número para el factorial: 0
Factorial de 0 es: 1
```

Tercera ejecución:

```
Ingrese el número para el factorial: -12
No se puede calcular el factorial
```

Explicación del programa:

Luego del ingreso del número al que se le va a calcular el factorial, se analiza si este es negativo (línea 10), si es así, se informa que para los números negativos el factorial no se puede calcular. Si el número es 0 o positivo, se procede a calcular el factorial mediante un ciclo `do-while`.

Al interior del ciclo, el acumulador `factorial` se incrementa con multiplicaciones sucesivas; es por esto que se inicializó en 1 en la línea 16. Si el acumulador se hubiese inicializado en 0, todos las multiplicaciones darían 0.

En cada iteración del ciclo `do - while`, se va calculando el factorial por medio de la expresión `factorial = factorial * inferiores` que se encuentra en la línea 20. En la variable `inferiores` se tienen los números inferiores, los cuales son multiplicados para calcular el factorial. Las iteraciones se llevan a cabo mientras el contenido de la variable `inferiores` sea menor o igual a `numero` (`while(inferiores <= numero)` (línea 22)).

.:Ejemplo 4.11. *Escriba un programa usando Lenguaje C que permita resolver la siguiente expresión matemática:*

$$expresion = 1 + \frac{x^2}{2!} - \frac{x^3}{4!} + \frac{x^4}{6!} - \frac{x^5}{8!} + \cdots \pm \frac{x^n}{(2(n-1))!}$$

Donde n es la cantidad de términos a calcular.

Análisis del problema:

- **Resultados esperados:** imprimir el resultado de la expresión.
- **Datos disponibles:** se debe saber el valor de x y la cantidad de términos (n).
- **Proceso:** inicialmente se obtienen del usuario los valores de x y de n.

  El cálculo de esta serie necesita un proceso iterativo para ir generando y acumulando el valor de los términos. Se debe empezar con el primer término en 1, cada uno de los siguientes es una división donde el numerador es x elevada a un exponente; este exponente empieza en 2 y se va incrementando de 1 en 1 hasta n. El denominador corresponde

al factorial de los valores pares en forma creciente y consecutiva, iniciando en 2 en el segundo término de la expresión, llegando hasta $2(n-1)$. Recuerde que el factorial requiere de otro proceso repetitivo, como se describió en el Ejemplo 4.10. Lo que se acaba de explicar, lleva a concluir que se necesita del uso de dos estructuras repetitivas anidadas.

Otro elementos a resaltar en la expresión, es que está conformada por sumas y restas sucesivas y alternas; a partir del segundo término, si el exponente de x es par se debe sumar y si es impar se debe restar.

- **Variables requeridas:**
  - `x`: almacena el valor que ingresa el usuario y sirve de base para el calculo de la expresión..
  - `n`: corresponde a la cantidad de términos que tiene la expresión.
  - `contadorTerminos`: variable que cuenta la cantidad de términos que se van generando en la expresión; hará parte de la condición del ciclo externo y servirá también como exponente para elevar x.
  - `inferiores`: almacena los valores desde 1 hasta el valor del denominador en cada uno de los términos de la serie. Hará parte de la condición del ciclo interno que calculará el factorial del denominador.
  - `denominador`: almacena el denominador de cada término de la expresión. Hará parte de la condición del ciclo que calculará su factorial.
  - `factorial`: acumula el factorial que se le calcule al denominador de la expresión.
  - `expresion`: es un acumulador que guardará el valor del cálculo de la expresión. Tiene incrementos y decrementos dependiendo del valor del exponente que tenga x.

Conforme al anterior análisis, se realiza el diagrama de flujo de las Figuras 4.25 y 4.26 y se escribe el Programa 4.14.

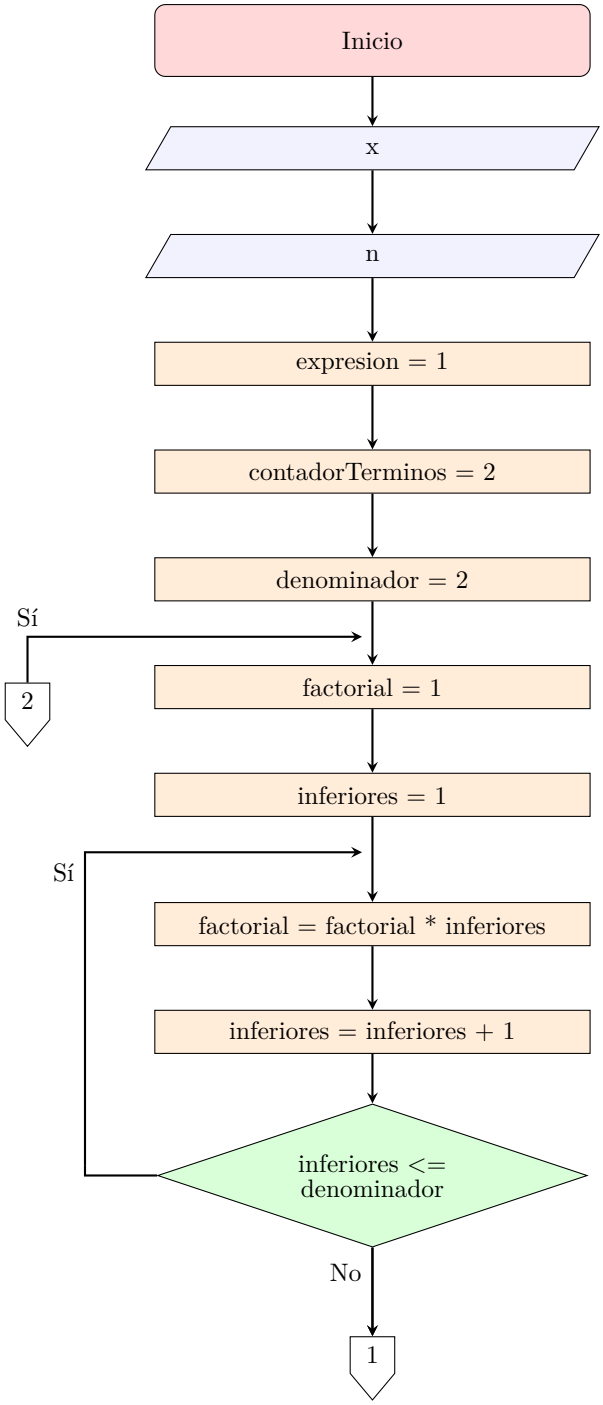


Figura 4.25: Diagrama de flujo `ExpresionMatematica` - Parte 1

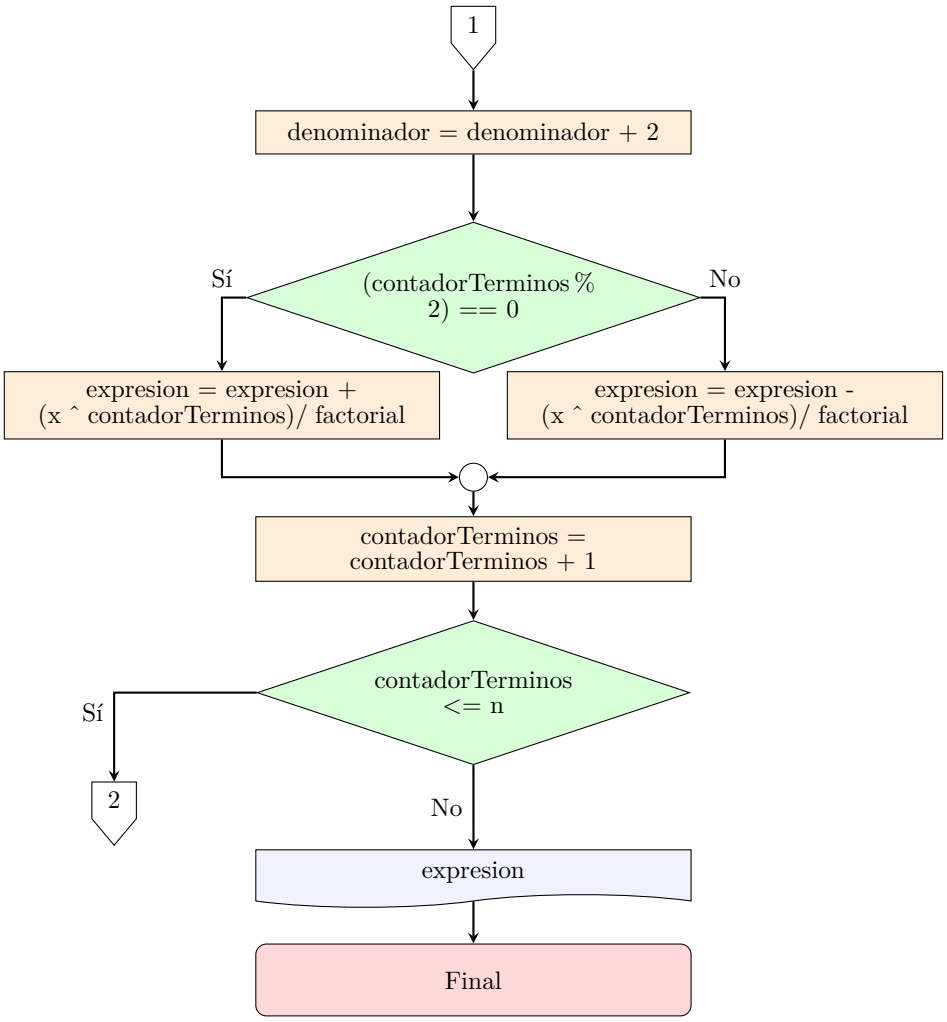


Figura 4.26: Diagrama de flujo `ExpresionMatematica` - Parte 2

Programa 4.14: ExpresionMatematica

```
1  #include <stdio.h>
2  #include <math.h>
3
4
5  int main()
6  {
7    long   x, n, contadorTerminos, inferiores,
8             denominador, factorial;
9    float expresion;
10
11   printf( "Ingrese el valor para x: " );
12   scanf( "%ld", &x );
```

```
13
14    printf( "Ingrese la cantidad de términos (n): " );
15    scanf( "%ld", &n );
16
17
18    expresion = 1;
19    contadorTerminos = 2;
20    denominador = 2;
21
22    do
23    {
24      factorial  = 1;
25      inferiores = 1;
26
27      do
28      {
29        factorial  = factorial  * inferiores;
30        inferiores = inferiores + 1;
31      } while( inferiores <= denominador );
32
33      denominador = denominador + 2;
34
35      if( (contadorTerminos % 2) == 0 )
36      {
37        expresion=expresion+pow(x,contadorTerminos)/factorial;
38      }
39      else
40      {
41        expresion=expresion-pow(x,contadorTerminos)/factorial;
42      }
43
44      contadorTerminos = contadorTerminos + 1;
45    } while( contadorTerminos <= n );
46
47    printf( "El valor de la expresión es: %.2f", expresion );
48
49    return 0;
50  }
```

Al ejecutar el programa:

Primera ejecución:

```
Ingrese el valor para x: 3
Ingrese la cantidad de términos (n): 6
El valor de la expresión es: 4.48
```

Segunda ejecución:

```
Ingrese el valor para x: 5
Ingrese la cantidad de términos (n): 7
El valor de la expresión es: 9.09
```

Explicación del programa:

Se usaron dos estructuras de repetición `do-while` anidadas. El ciclo externo se utilizó para ir generando cada uno de los términos de la expresión, desde el segundo hasta el término n. El ciclo interno se encarga de calcular el factorial de los denominadores para cada uno de los términos.

Los acumuladores y contadores se inicializaron así:

`expresion = 1`; (línea 18): se inicializó en 0 que es el valor del primer término de la expresión.

La generación de los términos y los cálculos empezarán entonces, a partir del segundo término, por lo tanto las inicializaciones siguientes en el programa fueron: `contadorTerminos = 2` y `denominador = 2` (líneas 19 y 20 respectivamente). Recuerde que `contadorTerminos`, también se usó como el exponente de x.

Luego de obtener el factorial[4] con el ciclo interno, se incrementa o decrementa el acumulador de la expresión; por esto se usó la decisión de la línea 35, que determina si el exponente de x (`contadorTerminos`) es par[5]. Con los pares se hace un incremento (línea 37), con los impares se hace un decremento (línea 41).

Aclaración:

Uno de los usos del ciclo `do-while` es en procesos de validación de datos de entrada, donde se condiciona al usuario para que ingrese datos acordes a los valores establecidos en los requisitos del problema.

Por ejemplo, una nota de un estudiante debe estar entre cero y cinco. Esto implica hacer una validación que haga que el usuario ingrese notas en este rango.

[4]Este proceso se explicó en el Programa 4.13.

[5]Todo número par da 0 como resultado del resto (módulo), en la división entera entre 2.

Validación de datos de entrada

Los programas escritos hasta el momento asumen que el usuario ha ingresado datos de forma correcta o, en su defecto tienen mensajes que informan de su ingreso erróneo. Tomando como caso el Ejemplo 4.9 donde el usuario debe elegir entre Android o iOS ingresando una 'A', 'a' o una 'I', 'i'; allí se muestra el mensaje "La opción no es válida" si el usuario digita un valor diferente a los establecidos. Lo ideal, aparte de informar la situación, es que el programa no permita el ingreso de datos erróneos, es decir, validar la entrada de datos.

La estructura repetitiva `do - while` es ideal para hacer este proceso, puesto que primero ejecuta el cuerpo del ciclo y luego revisa la condición.

Para hacer validaciones con la estructura `do - while` se recomienda seguir estos pasos:

1. Solicitar el dato
2. Escribir la palabra reservada `do`
3. Capturar el dato
4. Escribir la instrucción `while` y su condición.

Basado en lo anterior y pensando en validar un dato numérico, se procede como se muestra en el segmento de programa que se presenta a continuación, que valida el ingreso de un valor entre un rango (`MINIMO` y `MAXIMO`); en el caso de que el valor esté por fuera del rango, la condición del ciclo se hace verdadera y se solicita nuevamente el valor.

```
1  printf( "Ingrese un valor numérico: " );
2
3  do
4  {
5    scanf( "%d", &valor );
6  } while ( valor < MINIMO || valor > MAXIMO );
```

Un ejemplo de validación es mostrado en el Programa 4.15 que solicita el ingreso de una edad entre 18 y 90 años.

Programa 4.15: ValidacionValorNumerico

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int edad;
6
7   printf( "Ingrese la edad (entre 18 y 90 años): " );
8
9   do
10  {
11    scanf( "%d", &edad );
12  } while ( edad < 18 || edad > 90 );
13
14  printf ( "La edad ingresada es %d", edad );
15
16  return 0;
17 }
```

En este ejemplo (Programa 4.15) se solicita el ingreso de una edad entre 18 y 90 años, luego, se inicia el ciclo con la palabra reservada `do`, a continuación, se lee la variable `edad` que guardará el valor digitado.

Finalmente, se encuentra la condición `while (edad < 18 || edad > 90)`, que verifica si el valor ingresado está dentro del rango para aceptarlo en el programa o, por el contrario, volverlo a solicitar.

Imagine que se ingresa una edad de 10 años, la variable `edad` almacena este valor, al evaluar la condición se observa que:

```
edad < 18 || edad > 90
  10 < 18 || 10 > 90
Verdadero || Falso
      Verdadero
```

Como finalmente la condición es verdadera, el ciclo se vuelve a repetir, solicitando nuevamente el valor de la edad, hasta que el usuario ingrese un valor válido, es decir, un valor entre 18 y 90.

En el segmento del Programa, `MINIMO` y `MAXIMO` conforman un rango de valores que aceptará el programa como entrada, pero puede ocurrir que se solicite solo uno de los valores, por ejemplo, si el problema requiere un valor positivo mayor a cero, la validación puede hacerse como se muestra en el Programa 4.16.

Programa 4.16: ValidacionValorInferior

```
#include <stdio.h>

int main()
{
  int valor;

  printf( "Ingrese un valor numérico: " );

  do
  {
    scanf( "%d", &valor );
  } while ( valor < 1 );

  return 0;
}
```

Con respecto a los datos de tipo `char`, es imprescindible validar cada carácter por separado. En la porción de código que se presenta a continuación, se analiza esta situación.

```
  char letra;

  printf( "Ingrese una letra: " );

  do
  {
    scanf( " %c", &letra );
  } while ( letra != 'letra1' && letra != 'letra2' );

  printf ( "El valor ingresado es %c", letra );
```

Imagine que se desea validar el ingreso de una 'S' o 'N' tanto mayúsculas como minúsculas, esto se ilustra en el Programa 4.17.

Programa 4.17: ValidacionDatoChar

```
#include <stdio.h>

int main()
{
  char seguir;

  printf( "¿Desea continuar (S / N)?: " );

  do
  {
    scanf( " %c", &seguir );
  } while ( seguir != 's' && seguir != 'n' &&
```

```
13              seguir != 'S' && seguir != 'N' );
14
15    printf ( "El valor ingresado es %c", seguir );
16
17    return 0;
18 }
```

La condición escrita valida el ingreso de la letra 'S' o la letra 'N' en mayúscula o en minúscula. Por supuesto, `seguir` debe ser de tipo `char`.

Recuerde que Lenguaje C posee las funciones `toupper` y `tolower`, de la biblioteca `ctype.h`. La primera convierte un carácter a mayúscula y la segunda a minúscula. Con el uso de una de estas funciones ya no será necesario colocar en la condición la letra tanto en mayúscula como en minúscula. Observe el Programa 4.18.

Programa 4.18: ValidacionTipoChar

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main()
5  {
6    char seguir;
7
8    printf( "¿Desea continuar (S/N)?: " );
9
10   do
11   {
12     scanf( " %c", &seguir );
13     seguir = tolower( seguir );
14   } while ( seguir != 's' && seguir != 'n' );
15
16   printf ( "El valor ingresado es %c", seguir );
17
18   return 0;
19 }
```

Otra de las validaciones importantes es la de las cadenas. La validación de un arreglo de caracteres, se enfoca generalmente a que no se reciban valores vacíos (Ver Programa 4.19)

Programa 4.19: ValidacionCadena

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6    char cadena[ 20 ];
7
8    printf( "Ingrese una cadena: " );
9
10   do
11   {
12     scanf("%d", cadena);
13
14   } while ( strlen (cadena)  ==  0 );
15
16   printf ( "La cadena ingresada es %s", cadena );
17
18   return 0;
19 }
```

Note cómo se utilizó en el Programa 4.19 la función `strlen` de la biblioteca `string.h` (linea 14), que determina la longitud del arreglo `cadena`; si esta longitud es cero, no se ha ingresado nada y la condición es verdadera, lo que obliga a que el ciclo se repita. Si una variable de tipo cadena no tiene almacenado ningún dato, su longitud es 0, por el contrario, si tuviera almacenado algo, por ejemplo el nombre `"Liliana"`, su longitud sería de 7.

La expresión relacional `while(strlen(cadena) == 0)`, determina si la longitud del dato almacenado en la variable `cadena` es 0. Si esto es verdadero, la variable está vacía y el ciclo debe repetirse para leer nuevamente un dato. La condición tomará el valor de falso cuando la cadena tenga al menos 1 carácter.

En este ejemplo se usó la función `scanf` para leer la cadena, en los siguientes ejemplos se analizarán otras funciones que permiten capturar este tipo de datos.

Una alternativa para validar una cadena consiste en verificar si en la primer posición de la cadena está el carácter de fin de cadena (`'\n'`). Imagine validar el ingreso de un nombre de esta manera, hay que proceder de acuerdo al Programa 4.20

Programa 4.20: ValidacionNombre

```
1 #include <stdio.h>
2
3 int main()
4 {
5   char nombre[ 30 ];
6
7   printf( "Ingrese un nombre: " );
8
9   do
10  {
11    fgets( nombre, 30, stdin );
12
13  } while ( nombre[ 0 ] == '\n' );
14
15  printf ( "El nombre ingresado es %s", nombre );
16
17  return 0;
18 }
```

Otra validación podría condicionar la entrada a una cantidad mínima de caracteres. Por ejemplo, si se desea que la cadena tenga al menos 10 caracteres, la condición se plantearía de la siguiente forma:

```
while( strlen( cadena ) < 10 )
```

De forma parecida, se puede establecer una cantidad mínima y máxima de caracteres a leer en una variable de tipo cadena. La siguiente condición valida que no se vaya a dejar la variable llamada `nombre` con un dato en blanco y que máximo tenga 15 caracteres:

```
while( strlen( nombre ) == 0 ||  strlen( nombre ) > 15 )
```

También se puede validar que el contenido de una variable corresponda o no a ciertos valores. Ver Programa 4.21.

Programa 4.21: ValidacionContenidoCadena

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6   char cadena[ 20 ];
7
8   printf( "Ingrese una cadena: " );
9
```

```
10   do
11   {
12     fgets( cadena, 20, stdin );
13   } while ( strcmp( cadena, "texto1") != 0 &&
14             strcmp( cadena, "texto2") != 0);
15
16   printf ( "El texto ingresado es %s", cadena );
17
18   return 0;
19 }
```

Se observa cómo en el Programa 4.21 se usa la función `strcmp`, que hace parte de la biblioteca `string.h` y que compara dos cadenas: la variable denominada `cadena` con una cadena de texto, que en la condición es "texto1" y "texto2".

Ahora que ya se sabe como validar la entrada de datos a un programa en Lenguaje C, se analizarán algunos ejemplos que hacen uso de validaciones.

.:Ejemplo 4.12. *Los campesinos rusos tienen un método de multiplicación particular que consiste en tomar los dos factores de la operación (multiplicando y multiplicador) y disponerlos cada uno en una columna. El primer factor se va multiplicando sucesivamente por 2. Al mismo tiempo, en la segunda columna, al segundo factor se le van aplicando divisiones enteras entre 2. Estos cálculos se hacen hasta que el segundo factor llegue a 1. El siguiente paso consiste en sumar todos los números de la primera columna que estén al frente de un número impar de la segunda columna. El resultado que se obtiene es el producto del multiplicando por el multiplicador.*

Enseguida se ilustra este método ruso, con el producto de 3 por 19, cuyo resultado es 57.

Primer factor	Segundo factor	Impar	Producto
3	19	Sí	3
6	9	Sí	6
12	4	No	
24	2	No	
48	1	Sí	48
		Total suma:	57

Tabla 4.6: Ejemplo ilustrativo - Ejemplo 4.12

A partir de la anterior explicación, implemente un programa en Lenguaje C que halle el producto de dos números enteros entre 0 y 10000.

Análisis del problema:

- **Resultados esperados:** el producto de dos números enteros ingresados.

- **Datos disponibles:** multiplicando y multiplicador.

- **Proceso:** se leen multiplicando y del multiplicador, teniendo en cuenta que sus valores deben estar entre 0 y 10000[6].

  Luego del ingreso de los datos, se implementará un ciclo en el que, en cada iteración se multiplique por 2 el primer factor o multiplicando; de forma simultánea se hacen divisiones enteras entre 2 del segundo factor o multiplicador. Por cada iteración, se analiza si cada uno de los valores que va tomando el segundo factor es impar y así proceder a realizar la acumulación del primer factor. Las iteraciones terminarán cuando las divisiones del segundo factor lo lleven a un valor de 1.

- **Variables requeridas:**

  - `multiplicando` y `multiplicador`: almacenan los dos números a multiplicar.

  - `factor1` y `factor2`: almacenan una copia del valor del multiplicando y del multiplicador para no afectar los valores originales ingresados.

  - `producto`: corresponde al resultado del producto de los dos números ingresados.

Conforme al análisis realizado, se propone el Programa 4.22.

[6]Tenga en cuenta que este rango se estableció a modo de ejemplo.

Programa 4.22: MultiplicacionRusa

```
#include <stdio.h>

int main()
{
  int multiplicando, multiplicador, producto,
      factor1, factor2;

  printf( "Ingrese el multiplicando: " );
  do
  {
     scanf( "%d", &multiplicando );
  } while( multiplicando < 0 || multiplicando > 10000 );

  printf( "Ingrese el multiplicador: " );
  do
  {
     scanf( "%d", &multiplicador );
  } while( multiplicador < 0 || multiplicador > 10000 );

  factor1  = multiplicando;
  factor2  = multiplicador;
  producto = 0;

  do
  {
     if ( factor2 % 2 != 0 )
     {
       producto = producto + factor1;
     }

     factor1 = factor1 * 2;
     factor2 = factor2 / 2;

  } while( factor2 >= 1 );

  printf( "Producto de %d x %d = %d", multiplicando,
          multiplicador, producto );

  return 0;
}
```

Al ejecutar el programa:

```
Ingrese el multiplicando: 5
Ingrese el multiplicador: 17

Producto de 5 x 17 = 85
```

Explicación del programa:

Se solicita el ingreso de multiplicando y el multiplicador y se validan con dos estructuras cíclicas. La instrucción `scanf` está dentro del cuerpo de un ciclo `do-while` para ambas variables; esto se hizo con el objetivo de repetir la lectura de los datos, en el caso de que valor digitado no se encuentre en el rango de 0 y 10000.

El acumulador declarado como `producto` se inicializó en 0; allí se almacena el cálculo de la multiplicación. Dependiendo del resultado de una estructura de decisión, se incrementa mediante sumas sucesivas dentro del ciclo:

```
26      if ( factor2 % 2 != 0 )
27      {
28        producto = producto + factor1;
29      }
```

La estructura de decisión anterior, se usó para saber si el valor de la variable `factor2` es impar. Esta evaluación se lleva a cabo por cada iteración.

Luego de la decisión hay una instrucción que duplica `factor1`; después, `factor2` es dividido entre 2.

A continuación en el programa (Linea 34) se evalúa la expresión relacional `while( factor2 >= 1 )` y puede ocurrir una de dos cosas: primero, si la expresión es falsa el ciclo termina, se informa el resultado y el programa finaliza; segundo, si la expresión es verdadera, el programa retorna a la instrucción `do` para hacer otra iteración. Todo se repite mientras `factor2` no tome el valor de 0, puesto que en este caso la condición pasaría a ser falsa.

Aclaración:

La condición:
`while( factor2 >= 1 )`, es equivalente a
`while( factor2 > 0 )`.

Cualquiera de las dos podría ser usada para resolver el problema.

.:Ejemplo 4.13. *Diseñe un programa en Lenguaje C, que simule el funcionamiento de una calculadora con las 4 operaciones básicas (suma, resta, división y multiplicación). Las operaciones deben llevarse a cabo a medida que se van ingresando los datos, de igual forma se debe ir mostrando el resultado parcial. El programa terminará cuando se ingrese el signo igual (=).*

Debe contemplar la situación de tratar de dividir entre 0.

Análisis del problema:

- **Resultados esperados:** mostrar el resultado de las operaciones realizadas. En caso de dividir entre cero, mostrar un mensaje de error.
- **Datos disponibles:** números para calcular las operaciones, operadores básicos ('+', '-' ,'/' y '*') y el signo '=' para terminar la ejecución.
- **Proceso:** se solicita el ingreso de un primer número, luego uno de los operadores básicos o el signo '='. Si el operador ingresado no es el signo '=', se solicita un segundo número; dependiendo del operador que el usuario haya ingresado, se realiza la operación respectiva, mostrando su resultado. El programa se repite mientras el usuario no ingrese el signo '='.
- **Variables requeridas:**
  - `numero`: almacena los números que intervienen en las operaciones.
  - `calculo`: acumula los resultados de las operaciones.
  - `operador`: almacena el operador o el signo '=' que ingrese el usuario.
  - `bandera`: en caso que se trate de dividir entre 0, tomará el valor de falso y se mostrará un mensaje de error; si su valor es verdadero, indica que el programa imprimirá el resultado de las operaciones.

De acuerdo con el análisis realizado, se propone el Programa 4.23.

Programa 4.23: Calculadora

```
1  #include <stdio.h>
2
3  #define Verdadero 1
4  #define Falso     0
5
6  int main()
7  {
8    int   numero, calculo;
9    char  operador;
10   int   bandera;
11
12   printf( "Digite un número: " );
13   do
14   {
15     scanf( "%d", &numero );
16   } while( numero < -200000 || numero > 200000 );
17
18   calculo = numero;
19   bandera = Verdadero;
20   do
21   {
22     printf( "Digite un operador: " );
23     do
24     {
25       scanf ( " %c", &operador );
26     } while( operador != '+' && operador != '-' &&
27              operador != '/' && operador != '*' &&
28              operador != '=' );
29
30     if( operador != '=' )
31     {
32       printf( "Digite otro número: " );
33       do
34       {
35         scanf( "%d", &numero );
36       } while( numero < -200000 || numero > 200000 );
37
38       switch ( operador )
39       {
40         case '+': calculo = calculo + numero;
41                   break;
42
43         case '-': calculo = calculo - numero;
44                   break;
45
46         case '*': calculo = calculo * numero;
47                   break;
48
```

```
49          case '/':
50                      if( numero == 0 )
51                      {
52                         printf( "Error. División entre 0.\n");
53                         bandera = Falso;
54                      }
55                      else
56                      {
57                         calculo =  calculo / numero;
58                      }
59        }
60
61        if( bandera == Verdadero )
62        {
63          printf( "%d\n\n", calculo );
64        }
65      }
66    }while( operador != '=' );
67
68    return 0;
69 }
```

Explicación del programa:

El programa utiliza varias estructuras `do-while` independientes y anidadas. También hace uso de estructuras de decisión.

Varias de las estructuras `do-while` se usaron para validar los datos ingresados. Se hicieron dos validaciones para el ingreso de los dos números cuyos valores deben estar entre -200000 y 200000. Este rango es meramente ilustrativo. La otra validación condiciona al usuario a que solo pueda ingresar uno de los siguientes caracteres: '+','-' ,'*' y '/' o el signo '=', con los que se realiza una de las operaciones o se termina la ejecución del programa.

La variable `calculo` que guardará el resultado de las operaciones que se lleven a cabo, se inicializó con el valor del primer número ingresado. Así mismo, se hace la inicialización de la variable `bandera` con el valor Verdadero, el cuál cambiará si se trata de dividir entre 0.

A continuación, se ubica otro ciclo `do - while`, cuyas primeras instrucciones solicitan y leen un operador para determinar qué proceso se hace con el valor almacenado en la variable `calculo`. Si se ingresa un signo '=' se termina el proceso y se informa el resultado. Si por el contrario, el valor suministrado corresponde a uno de los operadores básicos, se solicita el segundo número; si el operador es uno de los siguientes '+', '-' o '*' se

lleva a cabo esa operación. Si el operador es '/', existe una decisión que determina si se puede realizar la división.

```
49          case '/':
50                      if( numero == 0 )
51                      {
52                       printf( "Error. División entre 0.\n" );
53                       bandera = Falso;
54                      }
55                      else
56                      {
57                         calculo =  calculo / numero;
58                      }
```

Esta decisión corresponde al caso '/', que hace parte de una estructura `switch`. Cuando se ingresa el operador de división se verifica el valor del número con el propósito de informar la situación de división entre 0.

Note que, cuando el número es 0, se imprime un mensaje informando la situación (línea 52) y `bandera` cambia su valor a Falso, para no imprimir el resultado del cálculo (línea 63). Finalmente, si la variable `operador` tiene almacenado el signo '=', la condición `while (operador != '=')` dará un resultado falso y se terminará la ejecución del ciclo y también del programa.

Ahora bien, si el número es diferente de 0, se realiza la división, se muestra el contenido de la variable `calculo` (línea 63) y se verifica la condición del ciclo `do- while`; en caso de tener un resultado verdadero se realiza otra iteración.

Por último, analice cómo la variable `operador` tiene un doble propósito: además, de determinar el tipo de operación a realizar con los números ingresados, hace el papel de centinela que controla la condición del ciclo `do-while` (Linea 66).

.:Ejemplo 4.14. *Se dice que un número es perfecto si la suma de sus divisores propios positivos resulta igual al número. Un ejemplo de número perfecto es el* 28*, dado que:* $1 + 2 + 4 + 7 + 14 = 28$.

Escriba un programa en Lenguaje C que reciba un número entero positivo e determine se es o no un número perfecto. El programa se ejecutará hasta que el usuario decida lo contrario.

Análisis del problema:

- **Resultados esperados:** un mensaje que informe si el número ingresado es perfecto o no.
- **Datos disponibles:** un número entero positivo que será ingresado por el usuario.
- **Proceso:** lo primero será validar que el número ingresado sea mayor que cero.

  Luego de tener el número, se ejecuta un proceso iterativo que determine si cada uno de los números menores a él es su divisor propio; esto se puede confirmar si al dividir el mayor entre el menor se obtiene una división exacta (con residuo 0). La siguiente decisión determina si un número es divisor de otro:

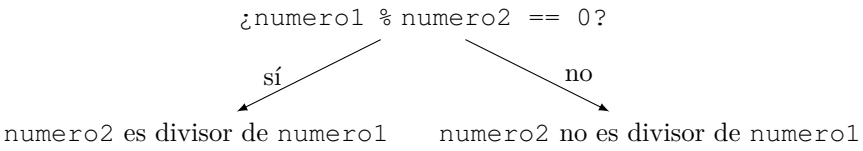


Figura 4.27: Árbol de decisión del Ejemplo 4.14

  Ahora, si la condición da un resultado verdadero, se incrementa un acumulador con la suma de todos los números menores que sean divisores del número ingresado. Después de este ciclo, se determina si el valor del acumulador es igual al número, lo que permitirá concluir que el número es perfecto, en caso contrario, no lo será. Tenga en cuenta que el 1 no es perfecto, ya que no tiene divisores propios menores a él.

  Por último, dado que el programa terminará cuando el usuario así lo decida, todo el proceso anterior debe estar dentro de otra estructura repetitiva de forma anidada.

- **Variables requeridas:**

  - `numero`: almacena el número ingresado al programa.
  - `numeroMenor`: almacena los números menores al número ingresado, para determinar si son sus divisores propios.
  - `sumaDivisores`: se usa para acumular la suma de los divisores del número ingresado.

- `seguir`: guarda la respuesta del usuario sobre si continuar o no con la ejecución del programa.

De acuerdo al análisis que se acaba de realizar, se propone el Programa 4.24.

Programa 4.24: Perfecto

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4
5  int main()
6  {
7    int numero, numeroMenor, sumaDivisores;
8    char seguir;
9
10   seguir = 'S';
11
12   while ( seguir == 'S' )
13   {
14     printf (" Ingrese un número entero positivo: " );
15     do
16     {
17       scanf ("%d", &numero );
18     } while( numero < 1 );
19
20     sumaDivisores = 0;
21     numeroMenor   = 1;
22
23     do
24     {
25       if( numero  % numeroMenor == 0 )
26       {
27         sumaDivisores = sumaDivisores + numeroMenor;
28       }
29       numeroMenor = numeroMenor + 1;
30     } while( numeroMenor <  numero );
31
32     if( sumaDivisores == numero && numero != 1 )
33     {
34       printf( "%d es un número perfecto.\n", numero );
35     }
36     else
37     {
38       printf( "  %d no es un número perfecto.\n", numero );
39     }
40
41     printf( "Desea continuar [S] [N]?: " );
```

```
42      do
43      {
44        scanf( " %c", &seguir );
45
46        seguir = toupper( seguir );
47      } while( seguir != 'S' && seguir != 'N'  );
48    }
49
50    return 0;
51  }
```

Al ejecutar el programa:

Primera ejecución:

```
Ingrese un número entero positivo: 496
496 es un número perfecto.
Desea continuar [S] [N]?: S
```

Segunda ejecución:

```
Ingrese un número entero positivo: 57
57 no es un número perfecto.
Desea continuar [S] [N]?: N
```

Explicación del programa:

El programa consta de varios ciclos anidados. El ciclo externo se escribió con una estructura `while`; al interior de este, se anidaron 3 ciclos `do-while` y se ubicaron dos estructuras de decisión. El primer ciclo `do-while` (líneas 15 a 18) valida el ingreso de un número positivo. El segundo (líneas 23 a 30) genera los números menores al número dado y acumula la suma de sus divisores propios. Analice cómo la estructura de decisión dentro de este ciclo, no contiene parte falsa:

```
25        if( numero  % numeroMenor == 0 )
26        {
27          sumaDivisores = sumaDivisores + numeroMenor;
28        }
```

La condición de esta decisión contiene una división y una comparación de igualdad. La división (`numero % numeroMenor`) calcula el resto de la división entre el número y cada uno de los números menores a él; el resultado se compara con el 0. Si la comparación es verdadera, la división es exacta, lo que indica que el número menor es un divisor del número estudiado, por tanto, se acumula el valor del número menor en la variable `sumaDivisores`.

A continuación en el programa se incrementa la variable `numeroMenor` (línea 29); se evalúa la condición del ciclo `do - while (numeroMenor < numero)`, que determina si el ciclo se lleva a cabo una vez más o se termina su ejecución.

Cuando este ciclo finalice, la siguiente estructura de decisión determina si el número es perfecto o no:

```
32      if( sumaDivisores == numero && numero != 1 )
```

El tercer ciclo `do-while` (líneas 42 a 47), valida que la variable `seguir`, reciba como dato de entrada una 'S' o una 'N'. La función `toupper` convierte a mayúscula el carácter ingresado por el usuario.

Después de esta condición, el programa retorna el control al inicio del ciclo `while` (línea 12), allí se determina si se lleva a cabo o no otra iteración. Si no se hace otra iteración, el programa termina.

4.3.1 Prueba de escritorio

En esta sección se hará la prueba de escritorio a un ejemplo que utiliza un ciclo `do - while`:

.:Ejemplo 4.15. *Para el Programa 4.25 que calcula la división entera entre dos números mediante restas sucesivas, realice la respectiva prueba de escritorio.*

Programa 4.25: Division

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int dividendo, divisor, cociente, resto;
6
7    dividendo = 38;
8    divisor   = 6;
9    cociente  = 0;
10
11   do
12   {
13     dividendo = dividendo - divisor;
14     cociente  = cociente + 1;
15   } while( dividendo >= divisor );
16
```

```
17    resto = dividendo;
18
19    printf( "El cociente es: %d\n", cociente );
20    printf( "El resto es: %d\n", resto );
21
22    return 0;
23 }
```

La prueba de escritorio para el Ejemplo 4.15 se puede observar en la Tabla 4.7.

divisor = 6

dividendo	cociente	dividendo>= divisor
38	0	
32	1	32 >= 6 (verdadero)
26	2	26 >= 6 (verdadero)
20	3	20 >= 6 (verdadero)
14	4	14 >= 6 (verdadero)
8	5	8 >= 6 (verdadero)
2	6	2 >= 6 (falso)

cociente = 6

resto = 2

Tabla 4.7: Prueba de escritorio - Programa 4.15

Las dos primeras columnas de la tabla representan los valores de las variables `dividendo` y `cociente`, que se van actualizando por cada iteración hasta llegar al valor que se espera; la tercera columna se utiliza para evaluar la condición del ciclo `do-while`.

Para este ejemplo, se trabajará con 38, 6 y 0, que se ubican en la tabla en las variables `dividendo`, `divisor` y `cociente`.

Luego, en el programa se encuentra la instrucción `do`, que marca el principio del ciclo. Dentro de este, la primera instrucción reduce la variable `dividendo` en 6, que es el valor del `divisor`: `(dividendo = dividendo - divisor)`; la segunda instrucción incrementa en 1 el valor del `cociente`, contando la cantidad de restas que se van haciendo, que equivalen al cociente de la división de estos dos números. Los resultados de estas operaciones se van registrando en la tabla de verificación a medida que se van obteniendo.

Mas abajo está la condición del ciclo que, al evaluarla da un resultado verdadero, como se ve en la tabla (`32 >= 6 (verdadero)`). En este punto, el programa retorna a la instrucción `do` y se repite el proceso.

El procedimiento que se acaba de describir se repite mientras el valor de `dividendo` sea mayor o igual al de `divisor`. Cuando `dividendo` toma el valor de 2 y se vuelve a evaluar la condición del ciclo `do-while`, se tiene que `2 >= 6` y se obtiene un resultado falso. Por lo anterior, el ciclo deja de ejecutarse y se lleva a cabo la siguiente operación: `resto = dividendo`, lo que guarda el 2 en `resto`. Remítase a la tabla para hacerle seguimiento paso a paso al ejemplo.

Por último, el programa imprime los valores almacenados en `cociente` y en `resto`.

Actividad 4.3

Basado en los siguientes enunciados, implemente los respectivos programas en Lenguaje C, que den solución a los problemas planteados.

1. Lea un número entero positivo, descompóngalo en cada una de sus cifras y con ellas genere el número invertido. Por ejemplo, si el número a leer es el 5432, el resultado será 2345.

2. Dado un número menor o igual a 50, calcule su factorial mediante sumas sucesivas.

3. El cajero de un restaurante desea controlar el flujo de caja en un día de trabajo cualquiera. Antes de abrir al público, el gerente del establecimiento le entrega la base para el día, la cual consiste en una suma de dinero que debe registrar en la caja y con la cual se espera pueda desempeñarse sin contratiempos. Durante su jornada tendrá ingresos por concepto de las ventas que se realicen, también habrán salidas de caja para la compra de insumos o gastos eventuales que deban realizarse.

Se espera un programa, que reciba el registro de cada una de las operaciones a medida que vayan sucediendo. El cajero también requiere un informe del saldo en caja después de cada registro. El programa deberá dar un mensaje de alerta en el caso que el saldo sea inferior o igual al 15 % de la base asignada. Al cierre del restaurante, se requiere los saldos finales (saldo en caja, ingresos y egresos) y la cantidad de cada una de las operaciones realizadas.

4.4. Estructura de repetición `for`

La estructura de repetición `for` se constituye en una alternativa a los ciclos `while` y `do-while`, cuando se requiere que una instrucción o un conjunto de ellas se ejecuten repetidas veces.

Así como el ciclo `while`, el ciclo `for` es un ciclo condicionado al inicio, lo que quiere decir que su ejecución está determinada por la evaluación de su condición. Lo anterior significa que el cuerpo del ciclo no se ejecute, ni siquiera una vez, cuando al evaluar la condición, esta resulte falsa.

Este ciclo es ideal en aquellos problemas en que se conoce de antemano el número de iteraciones que se deben ejecutar.

La forma general de la estructura de repetición `for` se muestra a continuación:

```
1 Instrucción de inicialización;
2
3 for ( var = valor1; condición; modificación de var)
4 {
5   Instrucción 1;
6   Instrucción 2;
7
8   Instrucción n;
9 }
10
11 Instrucción externa;
```

- Instrucción de inicialización: asigna un valor inicial a los contadores o acumuladores que se modificarán dentro del ciclo. Dependiendo del problema, puede no ser necesaria. También puede usarse en la inicialización de las variables que controlan el ciclo.
- La línea 3, es la cabecera del ciclo, marca su inicio.
- var: variable declarada por el programador de tipo int o char. Esta variable es la de control del ciclo.
- valor1: valor que se le asigna a var, puede ser otra variable o una constante.
- condición: cualquier condición válida en Lenguaje C, mientras sea verdadera, se ejecuta el cuerpo del ciclo.
- modificación de var: es una expresión que permite modificar el valor de la variable de control. Puede ser con incrementos o decrementos en cualquier valor.

- Instrucción externa: es la instrucción que se ejecutará cuando termine de iterar el cuerpo del ciclo.

Aclaración:

No todos los problemas que se pueden solucionar utilizando indistintamente la estructura `while` o `do-while`, tienen solución con la estructura `for`.

El ciclo `for`, funciona de la siguiente manera: se inicializa la variable de control con el valor que se haya asignado (`var = valor1`), luego se evalúan la `condición`; si la evaluación da un resultado verdadero, se ingresa al ciclo y las instrucciones que componen el cuerpo del mismo son ejecutadas hasta llegar a la llave que cierra el ciclo, la cual devuelve el control al inicio de este. Al regresar a la instrucción `for`, se ejecuta la instrucción `modificación de var`, incrementando o decrementando la variable de control; nuevamente se evalúa la condición, volviendo a iterar si el resultado es verdadero o, pasando a la primera instrucción que está por debajo de la llave que cierra el ciclo (`Instrucción externa`), en caso contrario, para que el programa continúe de allí en adelante.

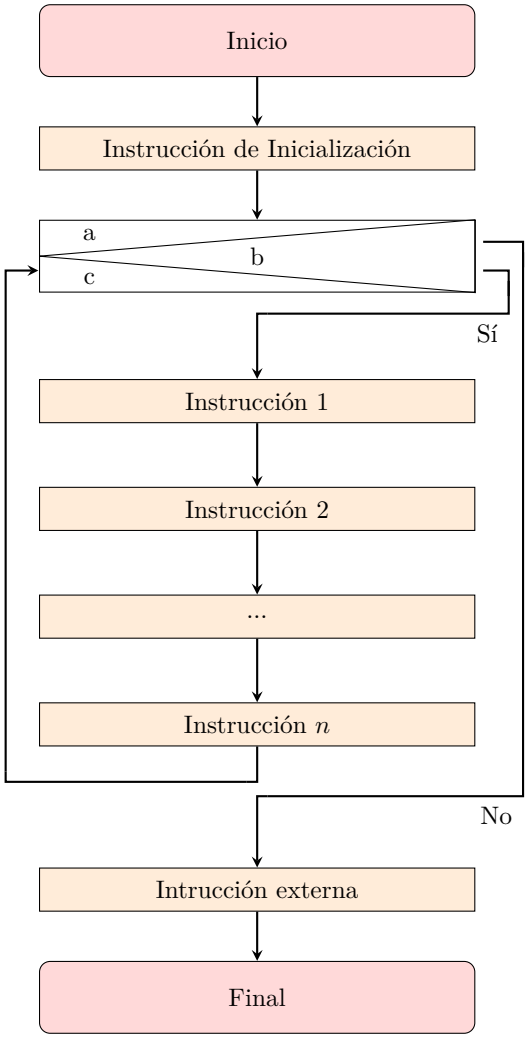
Aclaración:

Al encontrarse por primera vez una estructura repetitiva `for`, el programa inicializa la variable de control con el valor asignado, realizando luego la evaluación de la condición. La modificación no se ejecuta.

En la segunda y más iteraciones, se ejecuta la evaluación de la condición y la modificación de var; la inicialización no se vuelve a realizar.

El diagrama de flujo que representa la estructura `for` usa la notación presentada en la Figura 4.28 [Villalobos, 2014].

Aunque esta es la forma general de un ciclo `for`, Lenguaje C, presenta algunas variaciones en cuanto a la configuración de esta estructura de repetición. Luego del primer ejemplo se explicarán en detalle.



a: var = valor1. b: condición. c: modificación de var

Figura 4.28: Forma general del ciclo `for`

Para explicar el funcionamiento del ciclo `for` se empleará el mismo primer ejemplo usado con las dos estructuras anteriores, de esta manera, se podrán apreciar similitudes y diferencias.

El Programa 4.26 generará e imprimirá los números del 1 al 10, en la Figura 4.29 se puede analizar la solución usando el ciclo `for`.

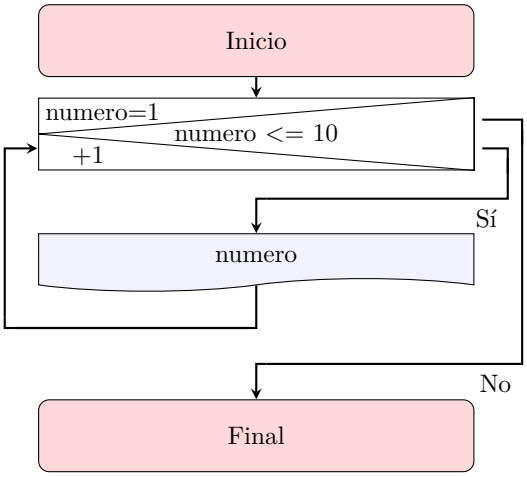


Figura 4.29: Diagrama de flujo para el Programa 4.26

Programa 4.26: Numeros-1-10-for Ver. 1

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int numero;
6
7    for ( numero = 1 ; numero <= 10 ; numero++ )
8    {
9      printf ( "%d\n", numero );
10   }
11
12   return 0;
13 }
```

A continuación, se van a identificar sus partes de acuerdo a la forma general expresada en párrafos anteriores.

- Línea 7: Instrucción de inicio del ciclo.
- Línea 9: Cuerpo del ciclo.

- Línea 10: Fin del ciclo. Retorna el control a la línea 7.

La línea 7 corresponde al inicio o cabecera del ciclo, donde se realizan las siguientes acciones:

1. La variable `numero` se inicializa en 1. Esta inicialización solo se realiza la primera vez que se ejecuta el ciclo.

2. Se evalúa la condición, la cual es *numero* <= 10.

3. Posterior a la ejecución del cuerpo del ciclo, se incrementa la variable `numero` en 1 unidad (`numero++`).

La línea 9 conforma el cuerpo del ciclo, que tiene la instrucción `printf` e imprime el valor de `numero`. La línea 10 es el final del ciclo y retorno a la línea 7 donde se incrementa la variable `numero` antes de volver a evaluar a condición del ciclo.

Las iteraciones finalizarán cuando `numero` tome el valor de 11, así al evaluar la condición será falsa en ese momento.

Aclaración:

El bloque de instrucciones o cuerpo del ciclo `for`, se ejecutará solamente si la evaluación de la condición es verdadera.

El valor inicial asignado a la variable de control y su incremento o decremento, pueden establecerse mediante una variable o una constante.

Otras formas generales del ciclo `for`:

Como se enunció anteriormente, la forma general del ciclo `for`, se puede expresar de diferentes maneras. Para ilustrarlo, se hará con base al ejemplo anterior (Programa 4.26), imprimiendo los números del 1 al 10.

En primera instancia, la modificación de la variable de control se puede omitir de la cabecera del ciclo y realizar dentro del cuerpo del mismo:

```
1 for ( numero = 1; numero <= 10; )
2 {
3   printf ("%d\n", numero);
4   numero ++;     // Modificación de la variable de control
5 }
```

Otra variante de la forma general, es la inicialización de la variable de control antes de la cabecera del ciclo:

```
1 numero = 1; // Inicialización de la variable de control
2
3 for ( ; numero <= 10; numero ++ )
4 {
5   printf ("%d\n",numero);
6 }
```

La cual, también puede realizarse en el momento de la declaración:

```
1 int numero = 1;   // Declaración e inicialización de la
      variable de control.
2
3 for ( ; numero <= 10; numero ++ )
4 {
5   printf("%d\n", numero);
6 }
```

De igual forma, se pueden combinar las dos formas anteriores: la inicialización y la modificación de la variable de control, por fuera de la cabecera del ciclo.

```
1 int numero = 1;   // Declaración e inicialización de la
      variable de control.
2
3 for ( ; numero <= 10; )
4 {
5   printf ("%d\n", numero);
6   numero ++;  // Modificación de la variable de control
7 }
```

Aclaración:

Cuando se vaya a realizar la inicialización o la modificación de la variable de control, por fuera de la cabecera del ciclo, se debe usar un signo de punto y coma (;) en su lugar.

Así mismo, la modificación a la variable de control, no necesariamente debe hacerse en una unidad. La siguiente porción de código imprime los números del 5 al 50, es decir, los primeros 10 múltiplos de 5, iniciando en el 5:

```
1 for ( numero = 5; numero <= 50; numero += 5 )
2 {
3   printf("%d\n", numero);
4 }
```

Como se observa, la modificación a la variable de control, se realizó con incrementos de a 5 unidades (`numero +=5`).

Por otro lado, en la condición del ciclo no necesariamente debe intervenir la variable que se inicializó en su cabecera. En el siguiente ejemplo, que también muestra en pantalla los primeros 10 múltiplos del 5, se ilustra este escenario:

```
1 x = 1;
2
3 for ( numero = 5; x <= 10; numero += 5 )
4 {
5   printf("%d\n",numero);
6   x ++;
7 }
```

La inicialización, el cambio de valor y la impresión de resultados, se hizo con la variable `numero`, pero la condición se planteó con la variable `x`. Note, que mientras la variable `numero` va tomando valores de 5 en 5, la variable `x` va tomando valores desde el 1 hasta el 10. Como `x` es la variable de la condición, el ciclo se repite 10 veces.

Además de la anterior forma del ciclo, es igualmente válido tener la inicialización de varias variables en la cabecera del ciclo:

```
1 for (x = 1, numero = 5; x <= 10; numero += 5 )
2 {
3   printf("%d\n",numero);
4   x ++;
5 }
```

En esta porción de código, se aprecia que tanto la inicialización de las variables `x` en 1 y `numero` en 5 (`x = 1, numero = 5;`), se realizaron en la cabecera del ciclo. La coma (,) es usada como separador de cada inicialización. El punto y coma (;) se usa para separar cada una de las partes de la cabecera del ciclo.

De manera semejante, se puede hacer con la modificación a la variable:

```
1 for ( x = 1, numero = 5; x <= 10; x ++, numero += 5 )
2 {
3   printf("%d\n",numero);
4 }
```

En estas líneas de código, se puede notar que las variables `x` y `numero` están inicializadas en la cabecera del ciclo, y que de la misma manera se hacen los cambios de valor.

Actividad 4.4

Cuál sería el resultado de la siguiente porción de código:

```
1 for (x = 2, numero = 4; x + numero < 16; x ++, numero ++ )
2 {
3   printf("%d \n",x + numero);
4 }
```

.:Ejemplo 4.16. *Escriba un programa en Lenguaje C que genere e imprima la siguiente serie:* 1 3 5 7 9 11 ... *n*

Utilice esta vez, la estructura de repetición `for`.

Análisis del problema:

Este problema ya se solucionó anteriormente, con las dos estructuras cíclicas estudiadas. El análisis completo de este problema está en el Ejemplo 4.1. El Programa 4.3 muestra la solución usando el ciclo `while` y el Ejemplo 4.8 aplica el ciclo `do-while`, la solución con la estructura `for` se puede apreciar en el Programa 4.27.

Conforme al análisis previo, se muestra el diagrama de flujo de la solución en la Figura 4.30.

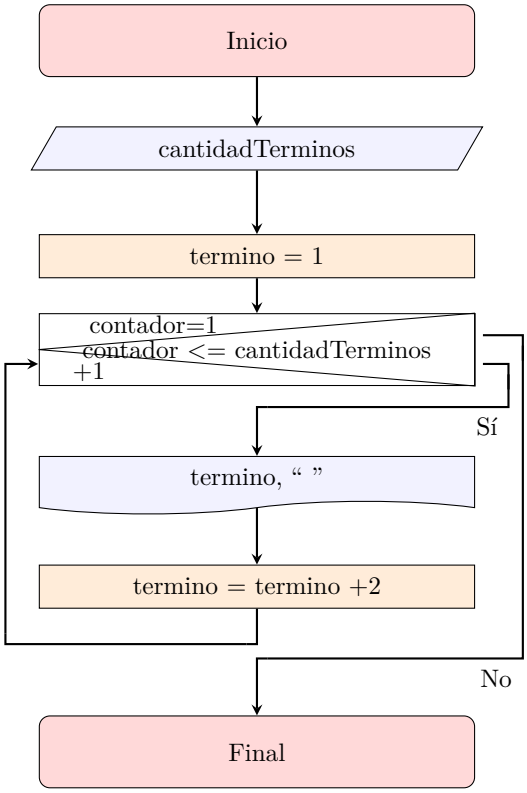


Figura 4.30: Diagrama de flujo del Programa `Serie Versión 3`

Programa 4.27: Serie Ver. 3

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int cantidadTerminos, contador, termino;
6
7      printf( "Ingrese la cantidad de términos a generar: " );
8      scanf( "%d", &cantidadTerminos );
9
10     termino = 1;
11
12     for ( contador = 1 ;
13           contador <= cantidadTerminos ;
14           contador++ )
15     {
16         printf( "%d, ", termino );
17         termino += 2;
18     }
19
20     return 0;
21 }
```

Explicación del programa:

La parte inicial del programa es igual a la de los Ejemplos 4.1 y 4.8. La primera diferencia está en la inicialización de las variables, acá solo es necesario inicializar la variable `termino`.

Luego se encuentra el inicio del ciclo:

```
12     for ( contador = 1 ;
13           contador <= cantidadTerminos ;
14           contador++ )
```

Esta instrucción inicializa la variable `contador` en 1, pero solo se hace la primera vez que se ejecute el `for`. Con la condición `contador <= cantidadTerminos`, se indica que las iteraciones deben hacerse mientras `contador` sea menor o igual a `cantidadTerminos`. Luego de producirse la primera ejecución del cuerpo del ciclo, en cada iteración, el programa suma 1 unidad a `contador`.

Si al evaluar la condición, se obtiene un resultado verdadero, se muestra el valor de `termino` que luego se incrementa en dos unidades. Cuando la condición de un resultado falso, se termina el ciclo y el programa va a la línea 20, la cual culmina con la ejecución del programa.

.:Ejemplo 4.17. *Construya un programa en Lenguaje C que imprima tres columnas de números, de acuerdo con la Tabla 4.8, donde el valor de n será proporcionado por el usuario.*

1	1	2
2	4	6
3	9	12
4	16	20
5	25	30
...	...	...
n	...	...

Tabla 4.8: Tabla de datos del Ejemplo 4.17

Análisis del problema:

- **Resultados esperados:** el programa debe generar la tabla del enunciado e imprimirla.
- **Datos disponibles:** el valor de n, que representa la cantidad de filas o números.
- **Proceso:** luego de capturar la cantidad de números, se generan las filas realizando los cálculos correspondientes.

  La primera columna de la tabla la forman los números desde 1 hasta `n`, de manera consecutiva. La segunda columna corresponde a los cuadrados de cada uno de los números de la primera; la última columna es la suma de los números que están en la primera y segunda columna en cada fila.

  La generación de esta tabla requiere de un proceso iterativo que empieza con un contador en 1 y finaliza con el valor de `n`; en cada iteración, se calcula el cuadrado del contador y la suma correspondiente, luego se hace la impresión de los resultados.
- **Variables requeridas:**

  - `cantidadNumeros`: almacena el total de filas o números que formarán la tabla (n).
  - `numero`: variable que va almacenando los números desde 1 hasta la cantidad que se ingresó; a partir de ella se obtendrá el cuadrado y la suma, que conforman las otras dos columnas.

Conforme al análisis realizado, se propone el Programa 4.28.

Programa 4.28: Tabla

```
#include <stdio.h>

int main()
{
  int cantidadNumeros, numero;

  printf( "Ingrese la cantidad de números: " );
  do
  {
    scanf( "%d", &cantidadNumeros );
  } while( cantidadNumeros < 0 );

  for( numero = 1 ; numero <= cantidadNumeros ; numero++ )
  {
      printf( "%5d %5d %5d\n", numero,
                               numero * numero,
                               numero + numero * numero );
  }

  return 0;
}
```

Explicación del programa:

Luego de declarar las variables requeridas, se solicitó al usuario la cantidad de términos y se validó para que no se ingresaran números negativos.

```
  int cantidadNumeros, numero;

  printf( "Ingrese la cantidad de números: " );
  do
  {
    scanf( "%d", &cantidadNumeros );
  } while( cantidadNumeros < 0 );
```

A continuación, se crea la tabla utilizando un ciclo `for`, que inicializa la variable `numero` con el valor de 1 e itera mientras que el valor que contiene esta variable sea menor o igual al valor contenido en la variable `cantidadNumeros`.

```
13   for( numero = 1 ; numero <= cantidadNumeros ; numero++ )
14   {
15       printf( "%5d %5d %5d\n", numero,
16                                numero * numero,
17                                numero + numero * numero );
18   }
```

En cada iteración que haga el ciclo `for`, se muestra el valor de la variable `numero`, su cuadrado y la suma de estos dos valores. Al finalizar el ciclo con su llave, se regresa a la instrucción `for` y se incrementa la variable `numero` en 1. Allí se evalúa la condición, mientras el valor de `numero` sea menor o igual al de `cantidadNumeros`, se vuelve a ejecutar el cuerpo del ciclo.

El cuerpo del ciclo solo tiene la instrucción `printf`. Analice cómo los cálculos se hacen dentro de ella:

```
15       printf( "%5d %5d %5d\n", numero,
16                                numero * numero,
17                                numero + numero * numero );
```

Esta es una forma adecuada de calcular e imprimir, sin embargo, se podría hacer de otra manera, incluyendo nuevas variables que almacenen los cálculos, como se ve en la siguiente porción de código:

```
    for( numero = 1 ; numero <= cantidadNumeros ; numero++ )
    {
       cuadrado = numero * numero;
       suma     = numero + cuadrado;
       printf( "%5d %5d %5d\n", numero, cuadrado, suma );
    }
```

Para el usuario final, las dos soluciones le resuelven el problema, pero, desde el punto de vista de la eficiencia, hay diferencias, por ejemplo: la primera solución hace dos veces la misma operación `numero * numero`, mientras que la segunda solución solo realiza esta operación una vez. En un programa con pocas iteraciones, esto no tiene gran impacto, pero para un programa que lleve a cabo muchas iteraciones, se puede notar una gran diferencia.

.:Ejemplo 4.18. *Diseñe un programa en Lenguaje C que calcule y muestre el resultado de la siguiente función:*

$$f(x) = x^3 + x^2 - 5$$

Para `x` *con valores desde* 0 *hasta* `n`*, con incrementos de* 2.

Análisis del problema:

- **Resultados esperados:** el programa debe mostrar cada uno de los valores que va tomando `x`, con el resultado que genere el cálculo de la función.
- **Datos disponibles:** el valor de `n`, que corresponde al máximo valor que tomará `x`.
- **Proceso:** primero se solicita el valor de `n` o máximo que tendrá `x`. A continuación, se hace el cálculo de la función y se imprimen los resultados esperados. Lo anterior se lleva a cabo dentro de un ciclo donde `x` empieza en 0 y llega hasta el valor máximo (`n`) ingresado por el usuario; en cada iteración `x` se incrementa en 2.
- **Variables requeridas:**
  - `valorX`: almacena el valor máximo que tomará `x` ingresado por el usuario.Representa la `n`.
  - `funcion`: almacena el cálculo de la función.
  - `x`: variable de control del ciclo, tomará valores entre 0 y `n`.

Conforme al análisis realizado, se propone el Programa 4.29.

Programa 4.29: FuncionX

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

int main()
{
  int valorX, x, funcion;

  printf( "Ingrese el máximo valor para x: " );
  do
  {
    scanf( "%d", &valorX );
  }while ( valorX < 0 );
  // Se valida para que no reciba valores negativos

  // Cálculo de la función
  for( x = 0; x <= valorX; x += 2 )
  {
    funcion = pow (x,3) + pow (x,2)- 5;
    printf( "\nPara x = %d, f(x) = %d", x, funcion );
  }
  return 0;
}
```

Al ejecutar el programa:

```
Ingrese el máximo valor para x: 10

Para x = 0  fx(x) = -5
Para x = 2  fx(x) = 7
Para x = 4  fx(x) = 75
Para x = 6  fx(x) = 247
Para x = 8  fx(x) = 571
Para x = 10 fx(x) = 1095
```

Explicación del programa:

Luego de la declaración de variables, se solicita el ingreso del valor máximo que tendrá x, denominado $valorX$. Este ingreso es validado con un ciclo `do - while` para que no acepte números negativos.

```
6   int valorX, x, funcion;
7
8   printf( "Ingrese el máximo valor para x: " );
9   do
10  {
11    scanf( "%d", &valorX );
12  }while ( valorX < 0 );
```

La instrucción `for( x = 0; x <= valorX; x += 2 )`, inicializa x en 0 y la incrementa en 2 en cada iteración, siempre y cuando sea menor o igual a `valorX`.

En cada iteración del ciclo, se calcula la función y se muestra el resultado. En este cálculo se incluye la función `pow` para calcular la potencia, su uso justifica el haber incluido la bibliteca `math.h` al inicio del programa.

```
16  for( x = 0; x <= valorX; x += 2 )
17  {
18    funcion = pow (x,3) + pow (x,2)- 5;
19    printf( "\nPara x = %d, f(x) = %d", x, funcion );
20  }
```

.:Ejemplo 4.19. *El Código Americano Estándar para el Intercambio de Información, conocido como código ASCII, por sus siglas en inglés (American Standard Code for Information Interchange), sirve para unificar la representación de los caracteres alfanuméricos y códigos de control en los computadores. Antes del surgimiento de este código, cada familia de computadores utilizaba una regla diferente para la representación.*

ASCII está compuesto por 256 códigos[7]; van desde el 0 al 255, en su representación numérica decimal. En la Tabla 4.9 se pueden observar 25 de ellos.

ASCII	Carácter	ASCII	Carácter	ASCII	Carácter	ASCII	Carácter
48	'0'	49	'1'	50	'2'	51	'3'
52	'4'	53	'5'	54	'6'	55	'7'
56	'8'	57	'9'	58	':'	59	';'
60	'<'	61	'='	62	'>'	63	'?'
64	'@'	...		...		...	
...		...		...		...	
91	'['	92	'\'	93	']'	94	'^'
...		...		...		...	
123	'{'	124	'|'	125	'}'	126	'~'

Tabla 4.9: Tabla de datos del Ejemplo 4.17

Los primeros 32 códigos (del 0 al 31) son conocidos como códigos de control, que se usan para controlar algunos dispositivos, por ejemplo, el salto de línea, la tecla escape, entre otros. Estos códigos no se pueden imprimir.

Conforme a lo anterior, construya un programa en Lenguaje C que genere e imprima los códigos ASCII desde el 32 al 255, tanto en su representación numérica decimal como su carácter equivalente. Deben mostrarse en orden inverso.

Análisis del problema:

- **Resultados esperados:** mostrar la lista de los códigos ASCII, desde el 255 hasta el 32.
- **Datos disponibles:** no es necesario solicitarle ningún dato al usuario.
- **Proceso:** mediante el uso de un ciclo, que itere desde el número 255 hata el 32 y que vaya disminuyendo de 1 en 1, se generan e imprimen los códigos ASCII.
- **Variables requeridas:**
  - `decimal`: variable de control del ciclo. Iniciará en 255 e irá decrementando en 1 unidad en cada iteración. Cada número decimal es la representación numérica de cada uno de los códigos.

[7]En esta dirección de internet, puede consultar todos los códigos ASCII: `http://www.asciitable.com/`. De igual forma, como dato curioso, a través de esta otra `http://www.asciiarte.com/`, encontrará dibujos realizados con estos caracteres.

- `ascii`: almacenará el carácter respectivo en cada iteración.

Conforme al análisis realizado, se propone el Programa 4.30.

Programa 4.30: Codigos

```
#include <stdio.h>

int main()
{
  int  decimal;
  char ascii;

  // Generación e impresión de los código ASCII
  for( decimal = 255; decimal >= 32; decimal-- )
  {
    ascii = decimal;
    printf("\n  %d = %c  ",decimal, ascii );

    if (decimal % 20 == 0)
      getchar();  // Hace una pausa cada 20 impresiones

  }

  return 0;
}
```

Explicación del programa:

Como este programa tiene una función muy puntual, no requiere del ingreso de datos y muestra siempre los mismos resultados al ejecutarse.

La instrucción `for(decimal = 255; decimal >= 32; decimal-- )`, inicializa la variable `decimal` en 255, la decrementa en 1 en cada iteración y se ejecuta mientras `decimal` sea mayor o igual a 32.

Dentro del ciclo está la instrucción: `ascii = decimal`. La variable `ascii` es de tipo `char` y la variable `decimal` es de tipo `int`; dado que para Lenguaje C los caracteres no son más que representaciones de los números decimales de la tabla ASCII, esta asignación con el signo igual ('=') es totalmente válida. Otros lenguajes necesitan de alguna función especial para realizar esta operación. Al hacer esta asignación, internamente el computador hace la conversión entre los tipos de datos, almacenando en `ascii` el carácter correspondiente.

La siguiente línea en el cuerpo del ciclo, muestra tanto el valor decimal como su correspondiente carácter:

```
12      printf("\n   %d = %c   ",decimal, ascii );
```

Posterior a la impresión, el control del programa vuelve a la instrucción `for` (línea 9), decrementando previamente la variable `decimal` en 1 unidad, nuevamente se evalúa la condición, si el valor de `decimal` es mayor o igual a 32, el ciclo da otra vuelta. Cuando la condición sea falsa, el ciclo termina y se ejecuta la instrucción que hay debajo de la llave que cierra el ciclo `for`, finalizando el programa.

Aclaración:

En Lenguaje C, también es posible imprimir los caracteres Unicode, usando la función `printf` y empleando \u seguido de un código, por ejemplo, para imprimir un corazón puede usar:

```
printf ( "\u2665");
```

(por ahora no funciona en Windows, solo en los Unix o similares)

Para otros código puede consultar:
`https://unicode-table.com/es/#latin-1-supplement`

Para imprimir la tabla con los 65535 caracteres (`0xffff` en base 16 o hexadecimal), en un sistema Unix o similar, puede usar:

```
1  #include <stdio.h>
2  #include <locale.h>
3
4  int main ()
5  {
6     int i;
7
8     setlocale(LC_ALL, "");
9
10    // Imprime 65535 caracteres Unicode
11    for ( i = 0 ; i <= 0xffff; i++ )
12    {
13        printf( "%4x -> %5lc\n", i, i );
14    }
15 }
```

El código imprime por cada línea: el código del carácter en hexadecimal `%x` (necesario para usar \u) y el carácter Unicode, pero esta vez empleando (`%lc`).

.:Ejemplo 4.20. *Elabore un programa en Lenguaje C, que genere y muestre las letras del abecedario de la siguiente forma:*

```
Z
Z Y
Z Y X
Z Y X W
Z Y X W V
Z Y X W V U
Z Y X W V U T
Z Y X W V U T S
Z Y X W V U T S R
Z Y X W V U T S R Q
Z Y X W V U T S R Q P
Z Y X W V U T S R Q P O
Z Y X W V U T S R Q P O N
Z Y X W V U T S R Q P O N M
Z Y X W V U T S R Q P O N M L
Z Y X W V U T S R Q P O N M L K
Z Y X W V U T S R Q P O N M L K J
Z Y X W V U T S R Q P O N M L K J I
Z Y X W V U T S R Q P O N M L K J I H
Z Y X W V U T S R Q P O N M L K J I H G
Z Y X W V U T S R Q P O N M L K J I H G F
Z Y X W V U T S R Q P O N M L K J I H G F E
Z Y X W V U T S R Q P O N M L K J I H G F E D
Z Y X W V U T S R Q P O N M L K J I H G F E D C
Z Y X W V U T S R Q P O N M L K J I H G F E D C B
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Análisis del problema:

- **Resultados esperados:** mostrar las letras del alfabeto de la forma en que lo solicita el problema.
- **Datos disponibles:** no es necesario el ingreso de ningún dato.
- **Proceso:** observe cómo se debe imprimir el alfabeto, de la `Z` a la `A`. La primera fila la conforma solo la letra `Z`, la segunda fila las letras `Z` y `Y`, la tercera fila las letras `Z Y` y `X`; cada nueva fila agrega una letra más. La impresión termina cuando se complete todo el alfabeto.

  Este proceso bebe usar dos ciclos anidados. El ciclo interno mostrará las letras de cada fila, mientras que el ciclo externo realizará el avance de una fila a la otra.

  Para resolver el problema, ambos ciclos deberán trabajar con decrementos, iniciando en la letra `'Z'` hasta la letra `'A'`.

- **Variables requeridas:**
  - `letraFila`: maneja el ciclo externo. Permitirá el avance de las filas.
  - `letra`: maneja el ciclo interno. Almacenará las letras que se van imprimiendo en cada fila.

Conforme al anterior análisis, se realiza el diagrama de flujo con la solución (Ver Figura 4.31) y se escribe el Programa 4.31.

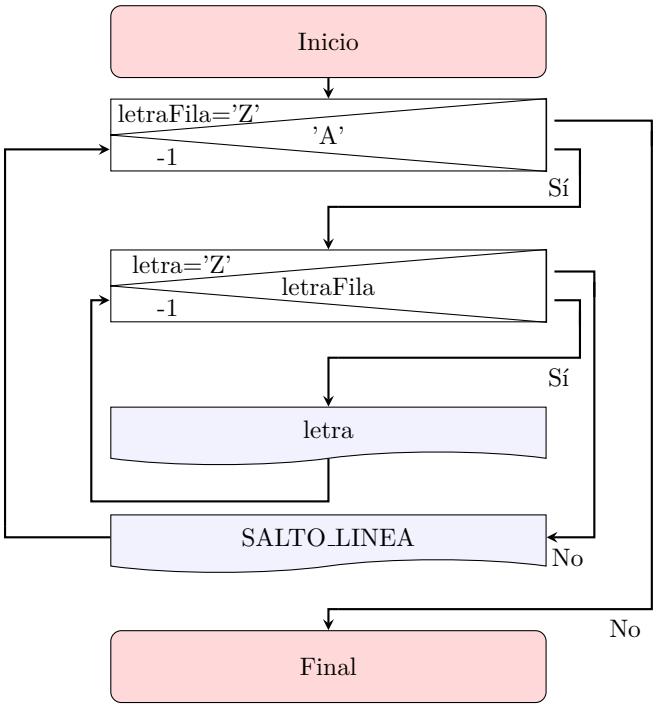


Figura 4.31: Diagrama de flujo del programa `Alfabeto`

Programa 4.31: Alfabeto

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char letraFila, letra;
6
7    // Generación del alfabeto
8    for( letraFila = 'Z'; letraFila >=  'A'; letraFila-- )
9    {
10     for( letra = 'Z' ; letra >= letraFila; letra-- )
11     {
12       printf( "%c ",letra );
13     }
14
15     // Salto de línea
16     printf( "\n" );
17   }
18
19   return 0;
20 }
```

Explicación del programa:

Para resolver el problema, se utilizaron dos ciclos `for` anidados.

El primer ciclo o ciclo externo presenta la siguiente instrucción:

```
8    for( letraFila = 'Z'; letraFila >= 'A'; letraFila-- )
```

Este ciclo inicializa la variable `letraFila` con la letra 'Z' y por cada iteración decrementa una letra hasta llegar a la 'A'. Cuando se ingresa al ciclo `for` interno, este inicializa `letra` = 'Z' y la decrementa en cada iteración hasta alcanzar el valor de `letraFila`. Se imprime igualmente en cada iteración, el valor de la variable `letra`, que en la primera iteración sería la letra 'Z'. Luego de que el ciclo interno termina se produce un salto de línea, es decir, se pasa al siguiente renglón.

```
8    for( letraFila = 'Z'; letraFila >=  'A'; letraFila-- )
9    {
10     for( letra = 'Z' ; letra >= letraFila; letra-- )
11     {
12       printf( "%c ",letra );
13     }
14
15     // Salto de línea
16     printf( "\n" );
17   }
```

En Lenguaje C, el salto de línea se hace con:

```
16      printf( "\n" );
```

Cuando el programa vuelve a la instrucción `for` del ciclo externo, `letraFila` se decrementa y su valor pasa a ser 'Y', como su condición sigue siendo verdadera, ya que aún no ha llegado a 'A', se vuelve a ejecutar el ciclo interno que, esta vez imprime la 'Z' y luego la 'Y' en la misma línea. En esta ocasión el ciclo se ejecuta dos veces. Luego se lleva a cabo el salto de línea.

En la tercera iteración del ciclo externo, la variable `letraFila` ha pasado su valor a 'X', y la condición sigue siendo verdadera. Nuevamente se ejecuta el ciclo interno, ahora ejecuta 3 iteraciones: en la primera imprime la letra 'Z', en la segunda la letra 'Y' y en la tercera la letra 'X', todas en la misma línea. Después, se produce el salto de línea.

En este momento, el programa debe estar mostrando lo siguiente:

```
Z
Z Y
Z Y X
```

El proceso sigue mientras la variable `letraFila` del ciclo externo, no tome el valor de la letra A y no se haya impreso todo el alfabeto dentro del ciclo interno.

.:Ejemplo 4.21. *Escriba un programa en Lenguaje C que simule el comportamiento de un reloj digital durante un día, con horas, minutos y segundos. Debe trabajarse con un formato de 24 horas.*

Análisis del problema:

- **Resultados esperados:** mostrar el funcionamiento simulado de un reloj digital en formato de 24 horas: HH:MM:SS.
- **Datos disponibles:** este ejemplo no requiere del ingreso de ningún dato.
- **Proceso:** este programa necesita de 3 ciclos anidados. El ciclo externo manejará las horas, el ciclo intermedio los minutos y el ciclo interno los segundos. La horas van desde 0 hasta 23; los segundos y minutos van desde 0 hasta 59, cada que uno de ellos llegue a 59 se cambiará al minuto u hora siguiente, respectivamente.

- **Variables requeridas:**
  - `hora`: controla el ciclo externo, tomará valores entre 0 y 23.
  - `minuto`: controla el ciclo intermedio, tomará entre 0 y 59.
  - `segundo`: controla el ciclo interno, toma valores entre 0 y 59.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 4.32 y se escriben los programas 4.32 y 4.33.

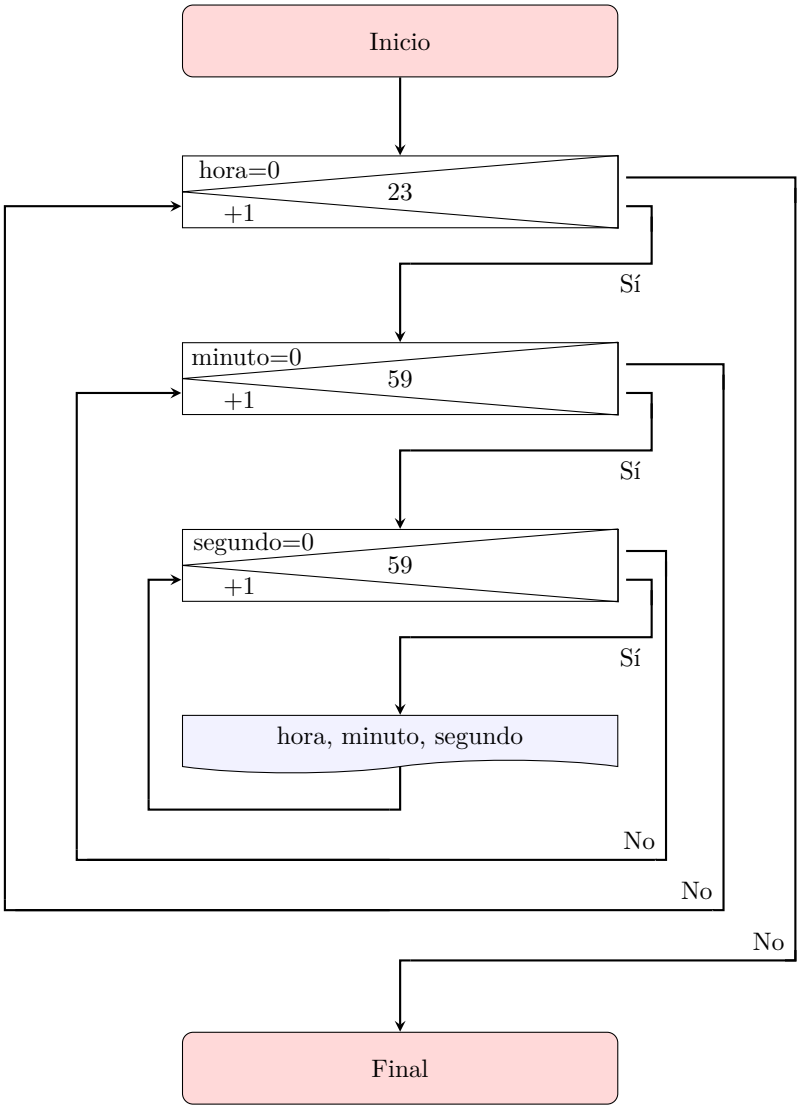


Figura 4.32: Diagrama de flujo del Programa `Reloj`

Programa 4.32: Reloj - solo para sistemas Windows

```
#include <stdio.h>
#include <windows.h> // Necesaria para Sleep

int main()
{
  int horas, minutos, segundos;
  WORD pausa = 100; // 100 ms

  for( horas = 0; horas <= 23; horas++ )
  {
    for( minutos = 0; minutos <= 59;  minutos++ )
    {
      for( segundos = 0; segundos <= 59;  segundos++ )
      {
        printf( "\n%2d:%2d:%2d", horas, minutos, segundos );
        Sleep( pausa );
      }
    }
  }

  return 0;
}
```

Programa 4.33: Reloj - solo para sistemas Unix como MacOS/Linux/...

```
#include <stdio.h>
#include <unistd.h> // Necesaria para usleep

int main()
{
  int horas, minutos, segundos;
  int pausa = 100; // 100 ms

  for( horas = 0; horas <= 23; horas++ )
  {
    for( minutos = 0; minutos <= 59;  minutos++ )
    {
      for( segundos = 0; segundos <= 59;  segundos++ )
      {
        printf( "\n%2d:%2d:%2d", horas, minutos, segundos );
        usleep( pausa );
      }
    }
  }

  return 0;
}
```

Explicación del programa:

Como ya se mencionó, se escribieron 3 ciclos anidados.

Estos ciclos funcionan así: el ciclo externo inicializa la variable `hora` en 0, luego el control pasa al ciclo intermedio que inicializa la variable `minuto` en 0 y le pasa el control al ciclo interno el cual inicializa la variable `segundo` en 0; con la instrucción `printf`, este último ciclo imprime el resultado y ubica una instrucción que obliga al computador a hacer una pausa, `Sleep` para los sistemas operativos Windows y `usleep` para los sistemas operativos basados en Unix. El ciclo interno realiza 60 iteraciones conforme a las instrucciones:

```
13        for( segundos = 0; segundos <= 59;  segundos++ )
14        {
15          printf( "\n%2d:%2d:%2d", horas, minutos, segundos );
16          // Instrucción para pausar (Sleep o usleep)
17        }
```

Cuando la variable `segundo` llegue a 60, la condición `(segundo <= 59)` será falsa y la ejecución de este ciclo interno finaliza. En este momento, se vería la siguiente salida del programa:

```
00:00:00
00:00:01
00:00:02
...
00:00:59
```

Cuando el ciclo interno termina, el control regresa al ciclo intermedio, que incrementa el valor de `minuto` en 1 unidad. La condición `(minuto <= 59)` sigue siendo verdadera y el ciclo interno vuelve a iniciar, repitiéndose todo el proceso que se acaba de describir.

Como la variable `minuto` tiene almacenado el valor de 1, en este momento, la instrucción `printf` muestra lo siguiente:

```
00:01:00
00:01:01
00:01:02
...
00:01:59
```

Cuando la variable `minuto` se haya incrementado 60 veces, el control del programa retornará al ciclo externo, que incrementa la variable `hora` en 1. Nuevamente se pasa el control al ciclo intermedio, que vuelve a empezar la variable `minutos` en 0 y le cede el control al ciclo interno, inicializando

la variable `segundo` en 0 y se repite el proceso de mostrar los segundos y minutos para una nueva hora.

Cuando las condiciones de los tres ciclos den un resultado falso, en forma simultánea, las iteraciones terminarán. La última salida que mostrará el programa es:

```
23:59:59
```

.:Ejemplo 4.22. *Construya un programa en Lenguaje C que le permita a un niño repasar las tablas de multiplicar, del 1 al 20. El programa genera aleatoriamente la tabla a repasar y preguntará los resultados, desde la fila uno hasta la fila 10; si la respuesta es correcta se mostrará un mensaje de felicitaciones, de lo contrario mostrará el resultado, acompañado de un mensaje que informe que el niño no acertó.*

Por cada tabla que el niño repase el programa dará una calificación, de acuerdo al número de respuestas correctas (por cada respuesta correcta se asigna un punto), basada en la escala de la Tabla 4.10.

Aciertos	Valoración
De 0 a 5	Insuficiente
6 o 7	Aceptable
8 o 9	Sobresaliente
10	Excelente

Tabla 4.10: Tabla de valoración - Ejercicio 4.22

El niño podrá repasar la cantidad de veces que quiera.

Análisis del problema:

- **Resultados esperados:** mostrar un mensaje de acuerdo a la respuesta del niño y la calificación obtenida por cada tabla repasada.
- **Datos disponibles:** el número (entre 1 y 20) de la tabla que desea repasar. Este número es generado aleatoriamente por el programa.
- **Proceso:** el programa necesita dos procesos repetitivos anidados. El ciclo interno calcula la tabla del número que el computador genere, desde la fila 1 hasta la fila 10. Así mismo, este ciclo mostrará un mensaje de acuerdo a la respuesta del niño como se aprecia en la Figura 4.33.

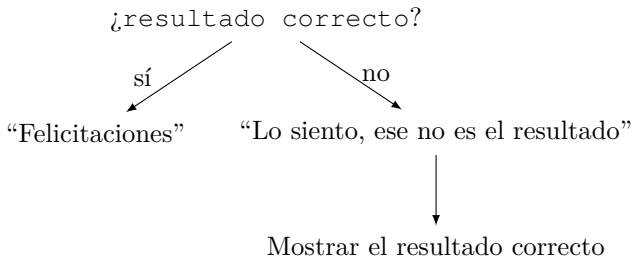


Figura 4.33: Árbol de decisión del Ejemplo 4.21 - Respuesta

Esta estructura de decisión, también contará los aciertos o desaciertos que el niño tenga al responder.

Por último, el programa informará la valoración obtenida en cada tabla de multiplicar, de acuerdo al número de aciertos, basado en los rangos que aparecen en la Tabla 4.10. Las decisiones que se deben plantear se pueden ver en la Figura 4.34.

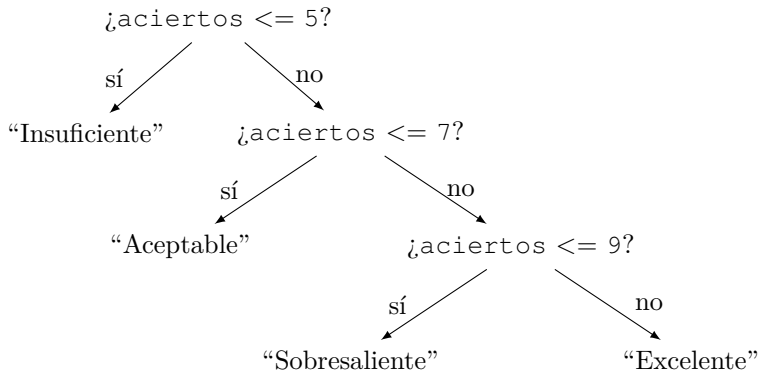


Figura 4.34: Árbol de decisión del Ejemplo 4.21 - Valoración

El proceso que se acaba de mencionar debe enmarcarse en otro ciclo. En este ciclo externo, se solicitará la tabla que el niño repasará, se inicializan los contadores de `aciertos` y `desaciertos` y se preguntará si se desea calcular una nueva tabla.

- **Variables requeridas:**
  - `tabla`: almacena el número que se quiere repasar.
  - `contadorFilas`: servirá para contar las filas de cada tabla, tomará valores entre 1 y 10.
  - `producto`: almacena el cálculo del resultado de cada fila de la tabla.
  - `respuesta`: almacena la respuesta que el niño ingresa como resultado de la multiplicación.
  - `aciertos`: cuenta las respuestas correctas.
  - `desaciertos`: cuenta las respuestas incorrectas.
  - `seguir`: almacena la respuesta sobre si se desea continuar o terminar con la ejecución del programa.

Las Figuras 4.35, 4.36, 4.37 y 4.38 ilustran la solución del Ejemplo 4.22 utilizando un diagrama de flujo.

Aclaración:

Como se puede apreciar en el análisis de este ejemplo, su desarrollo es un poco complejo y esto se avidenciará en los siguientes diagramas de flujo y en la implementación del programa.

Cuando se tiene este tipo de problemas, suele suceder que los diagramas de flujo sean extensos y difíciles de leer. Con los que se muestran a continuación, se quiere ilustrar el uso de conectores, tanto a la misma página así como a página diferente. Los conectores facilitan el seguimiento de la lógica en diagramas de gran tamaño.

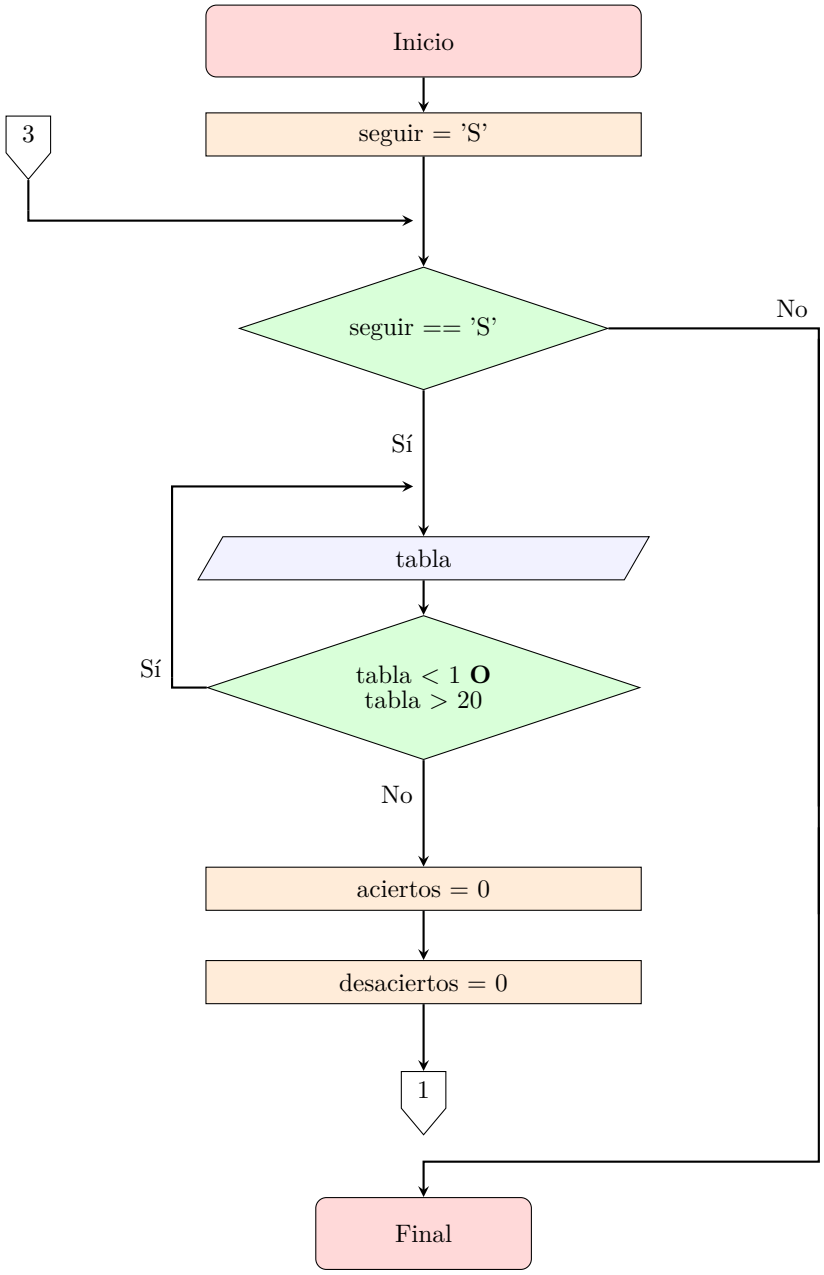


Figura 4.35: Diagrama de flujo del Programa `JuegoTablas` - Parte 1

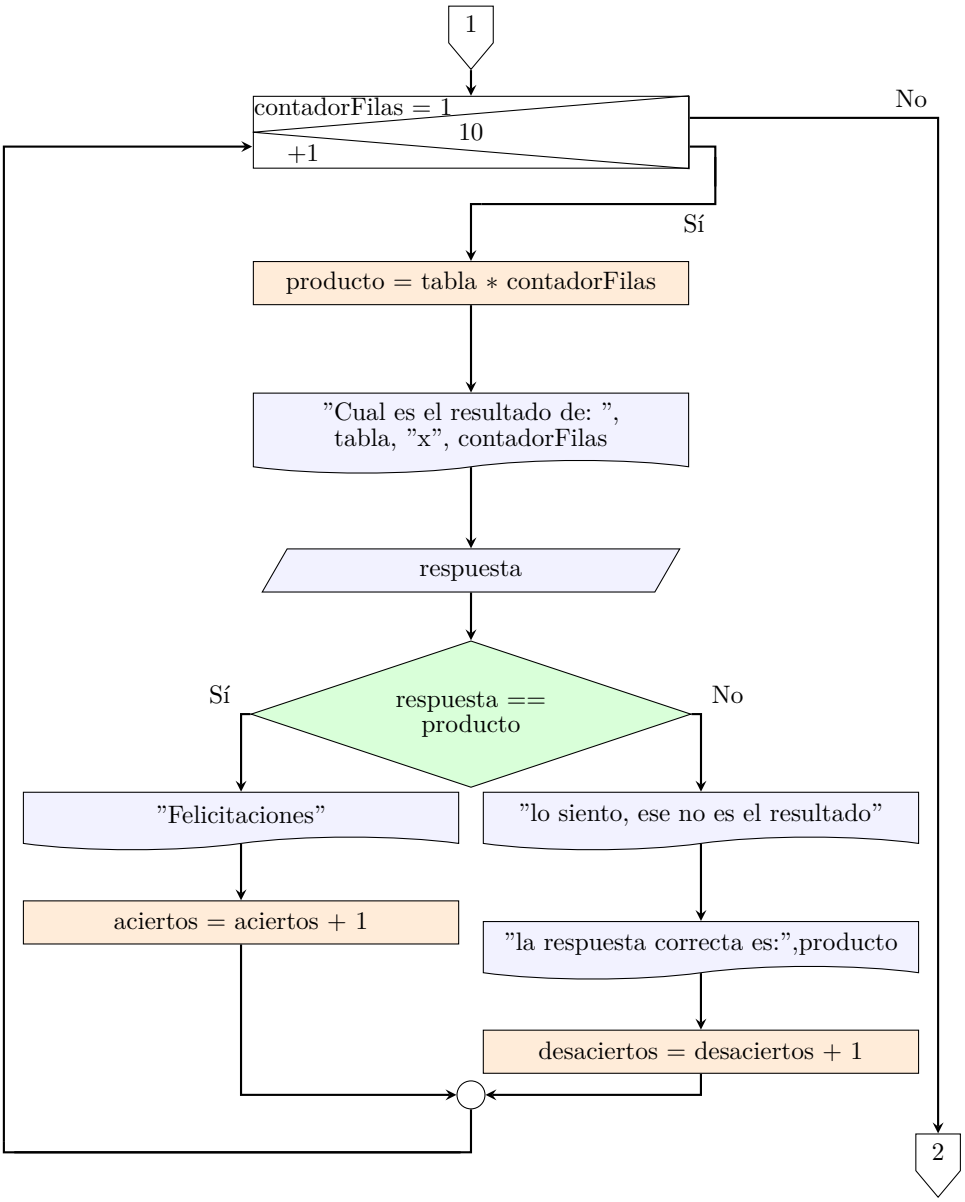


Figura 4.36: Diagrama de flujo del Programa `JuegoTablas` - Parte 2

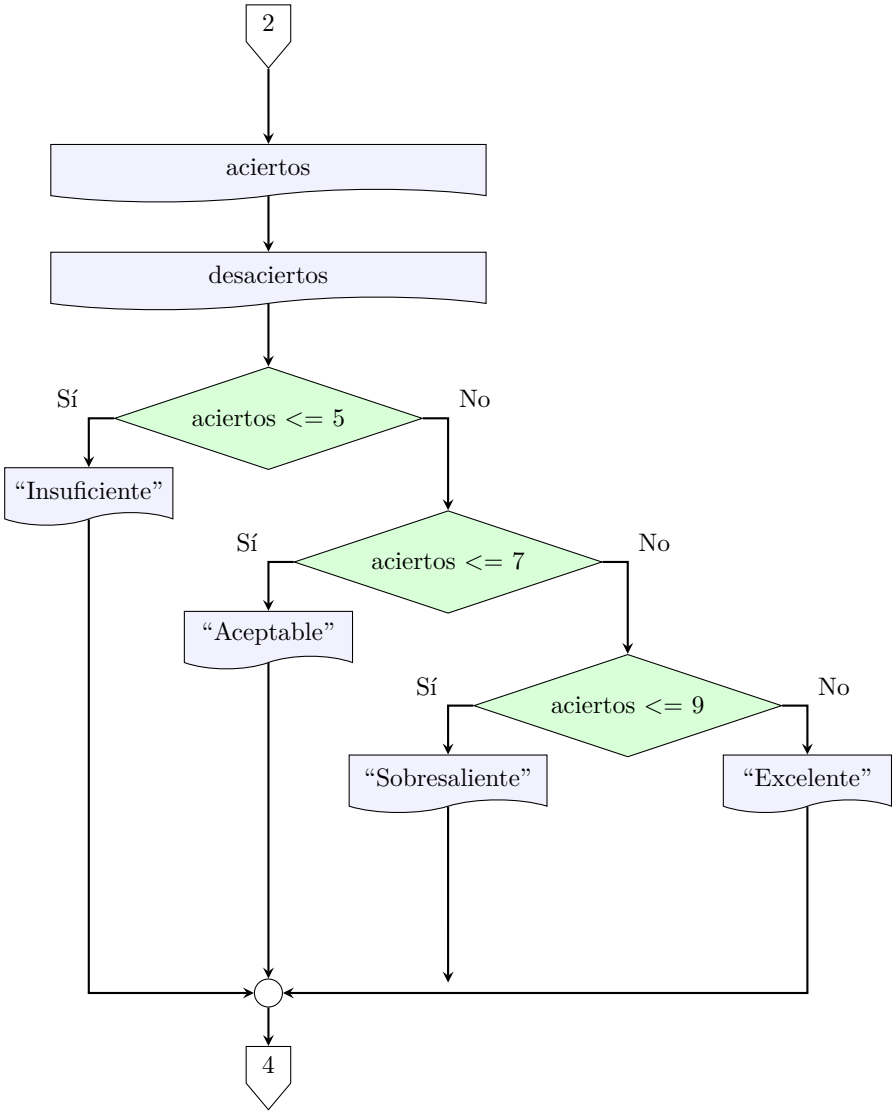


Figura 4.37: Diagrama de flujo del Programa `JuegoTablas` - Parte 3

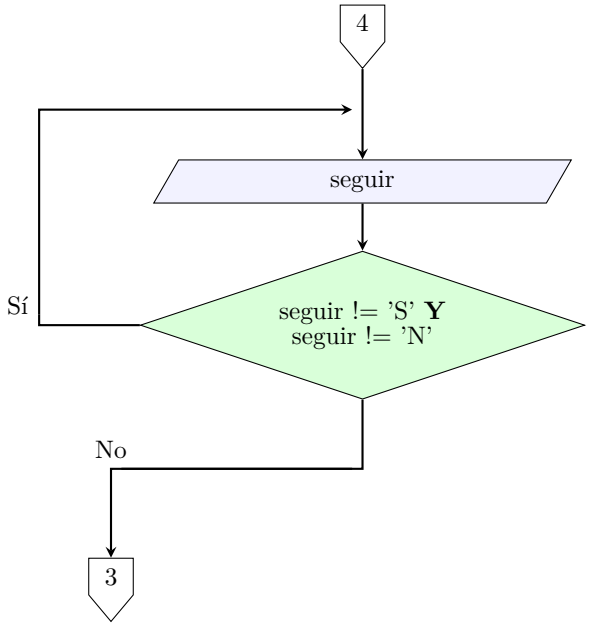


Figura 4.38: Diagrama de flujo del Programa `JuegoTablas` - Parte 4

El Programa 4.34 corresponde al código en Lenguaje C, solución al juego de las tablas de multiplicar.

Programa 4.34: JuegoTablas. (Versión para Windows)

```
1  #include <stdio.h>
2  #include <ctype.h>
3  #include <time.h>
4  #include <windows.h> // Para Unix usar: <stdlib.h>
5
6  int main()
7  {
8    int  tabla, contadorFilas, producto,
9         respuesta, aciertos, desaciertos;
10   char seguir;
11
12   srand( time(NULL) ); // Valor semilla para "aleatorios"
13
14   seguir = 'S';
15
16   while ( seguir == 'S' )
17   {
18     system( "cls" ); // Para Unix usar: "clear"
19
```

```
20      tabla = 1 + rand()  % 20; // "Aleatorio" entre 1 y 20
21
22      aciertos = 0;
23      desaciertos = 0;
24
25      for( contadorFilas = 1; contadorFilas <= 10;
26           contadorFilas++ )
27      {
28        producto = tabla * contadorFilas;
29
30        printf("\n\nEscriba el resultado de %2d x %2d = ",
              tabla, contadorFilas);
31        scanf( "%d", &respuesta );
32
33        if( respuesta == producto )
34        {
35          printf( "\nAcertaste. Felicitaciones." );
36          aciertos++;
37        }
38        else
39        {
40          printf( "Lo siento, ese no es el resultado \n\n" );
41          printf( "La respuesta correcta es: %d", producto );
42          desaciertos++;
43        }
44      }
45
46      printf( "\n\n\nAciertos: %d", aciertos );
47      printf( "\nDesaciertos: %d\n\n\n", desaciertos );
48
49      if ( aciertos <= 5 )
50        printf( "Insuficiente" );
51      else
52        if ( aciertos <= 7 )
53          printf( "Aceptable" );
54        else
55          if ( aciertos <= 9 )
56            printf( "Sobresaliente" );
57          else
58            printf( "Excelente" );
59
60      printf( "\n\n\n¿Desea volver a jugar [S] o [N]?: " );
61      do
62      {
63        seguir = getchar();
64        seguir = toupper( seguir );
65      } while( seguir != 'S' && seguir != 'N' );
66    }
67
```

```
68    return 0;
69 }
```

Explicación del programa:

En este programa se utilizaron las 3 estructuras repetitivas que se analizaron durante el capítulo. Cada una de ellas realiza una tarea específica y, en conjunto resuelven el problema.

El ciclo `while`, permite repetir todo el proceso. Este ciclo se llevará a cabo mientras el usuario responda con una letra `'S'` a la pregunta: `"¿Desea volver a jugar [S] o [N]?:"`. Esta estructura repetitiva, corresponde al ciclo externo del programa.

Las instrucciones que se encuentran en las líneas 12 y 20, generan un número pseudoaleatorio entre 1 y 20, incluyendo ambos valores. La instrucción en la línea 12 genera la semilla, siempre diferente al ejecutar el programa, base para que la instrucción de la línea 20 genere el número.

```
20      tabla = 1 + rand()  % 20; // "Aleatorio" entre 1 y 20
```

La función `srand` usa el número de segundos transcurridos desde el 1 de enero de 1970 hasta el momento actual, a partir de la función `time(NULL)`.

```
20    srand( time(NULL) ); // Valor semilla para "aleatorios"
```

De la misma manera, este ciclo externo valida la respuesta a la pregunta final, para que la variable `seguir` acepte únicamente la letra `'S'` o `'N'`, convirtiéndola a mayúscula con la función `toupper`.

```
60      printf( "\n\n\n¿Desea volver a jugar [S] o [N]?: " );
61      do
62      {
63        seguir = getchar();
64        seguir = toupper( seguir );
65      } while( seguir != 'S' && seguir != 'N' );
```

Se encuentra un ciclo `for` anidado dentro del ciclo `while` externo. Al interior de este ciclo `for` se calcula la tabla, se le pregunta al usuario sobre el resultado de la multiplicación y se verifica mediante una estructura de decisión si la respuesta fue correcta; en caso afirmativo se muestra un mensaje de "Felicitaciones" y se cuenta el acierto. Si la respuesta fue incorrecta, se muestra un mensaje informando del error junto con el resultado correcto y se incrementa el contador de desaciertos.

```
22      aciertos = 0;
23      desaciertos = 0;
24
25      for( contadorFilas = 1; contadorFilas <= 10;
26           contadorFilas++ )
27      {
28        producto = tabla * contadorFilas;
29
30        printf("\n\nEscriba el resultado de %2d x %2d = ",
             tabla, contadorFilas);
31        scanf( "%d", &respuesta );
32
33        if( respuesta == producto )
34        {
35          printf( "\nAcertaste. Felicitaciones." );
36          aciertos++;
37        }
38        else
39        {
40          printf( "Lo siento, ese no es el resultado \n\n" );
41          printf( "La respuesta correcta es: %d", producto );
42          desaciertos++;
43        }
44      }
```

El proceso que se acaba de describir se lleva a cabo 10 veces, que corresponden al número de filas de la tabla. En el momento en que el ciclo `for` finalice su ejecución, el programa regresa al ciclo externo, esto es, al ciclo `while`.

A continuación, el programa muestra la cantidad de `aciertos` y de `desaciertos` que tuvo el niño.

```
46      printf( "\n\n\nAciertos: %d", aciertos );
47      printf( "\nDesaciertos: %d\n\n\n", desaciertos );
```

Para asignar la calificación se usó un conjunto de decisiones anidadas, que también hacen parte del ciclo externo.

```
49      if ( aciertos <= 5 )
50        printf( "Insuficiente" );
51      else
52        if ( aciertos <= 7 )
53          printf( "Aceptable" );
54        else
55          if ( aciertos <= 9 )
56            printf( "Sobresaliente" );
57          else
58            printf( "Excelente" );
```

En la última parte del cuerpo del ciclo `while` se hace la pregunta para saber si el usuario desea volver a jugar o no, la respuesta es almacenada en la variable `seguir`; luego el control regresa al principio del ciclo `while` donde se evalúa la condición:

```
16   while( seguir == 'S' )
```

Si la condición da un resultado verdadero, todo el proceso es ejecutado nuevamente. Cuando el usuario responda con una letra `'N'` a la pregunta sobre volver a jugar, el juego finaliza.

.:Ejemplo 4.23. *Una Universidad requiere de un programa en Lenguaje C que le permita obtener cierta información de los estudiantes de primer semestre que acaban de terminar el periodo académico; entre dicha información está:*

- *El mayor y el menor promedio general.*
- *El número de estudiantes que aprobaron y reprobaron, así como la cantidad de estudiantes que quedaron en situación condicional.*
- *Porcentaje de estudiantes que quedaron excluidos por bajo rendimiento; de igual forma, el porcentaje que aprobaron el periodo, con relación al total de ellos.*
- *La cantidad de estudiantes que aprobaron y reprobaron el periodo por cada uno de los grupos. Los grupos se identifican como Grupo A, Grupo B, Grupo C y así sucesivamente.*
- *Para cada estudiante es necesario informar su nota definitiva acompañada de un mensaje que especifique su situación académica.*

Dentro del reglamento estudiantil de la universidad se contemplan las siguientes reglas:

1. *Cualquier estudiante que curse el primer semestre debe cursar 6 asignaturas o materias.*
2. *Las notas se califican en el rango de 0.0 a 5.0.*
3. *Una asignatura se aprueba con una nota definitiva mayor o igual a 3.0.*

4. *Un periodo académico se aprueba, si la media aritmética de las notas definitivas de las 6 materias, es igual o superior a 3.0.*

   Nota: *en el caso de que un estudiante sea excluido por bajo rendimiento, perderá el periodo académico, sin importar el promedio obtenido.*

5. *Una vez finalizado el semestre, el estudiante será clasificado en una de las siguientes categorías:*

   a) *Excluido por bajo rendimiento.*

   b) *Condicional.*

   c) *Continúa de manera normal.*

6. *Un estudiante será excluido por bajo rendimiento, si cumple una o ambas de las siguientes condiciones:*

   a) *Si la media aritmética obtenida en el periodo académico, es inferior a 2.0.*

   b) *Si pierde más del 50 % de las asignaturas que ha cursado.*

7. *Un estudiante queda en situación condicional, si no fue expulsado por bajo rendimiento y si su media aritmética del periodo académico está entre 2.0 y 2.9.*

8. *Si el estudiante no se encuentra en ninguna de las dos situaciones anteriores, se considera que continúa de manera normal.*

Análisis del problema:

- **Resultados esperados:** el programa debe mostrar la información que se le solicita a nivel de cada estudiante, a los estudiantes de primer semestre y a nivel de los grupos.
  - Por cada estudiante se debe mostrar:
    - Su n ota definitiva y un mensaje que indique la situación académica en que queda el estudiante.
  - Por cada grupo de primer semestre, se requiere conocer:
    - Número de estudiantes que aprobaron el periodo.
    - Número de estudiantes que reprobaron el periodo.
  - En general, por todos los estudiantes se debe mostrar:

  - Mayor promedio.
  - Menor promedio.
  - Número de estudiantes que aprobaron el periodo.
  - Número de estudiantes que reprobaron el periodo.
  - Porcentaje de estudiantes que aprobaron el periodo, con relación al total de estudiantes.
  - Porcentaje de estudiantes excluidos con relación al total de estudiantes.
  - Número de estudiantes que quedaron en situación condicional.

- **Datos disponibles:** por cada uno de los estudiantes se conoce su código, nombre y la nota definitiva (entre 0.0 y 5.0) de cada una de las 6 asignaturas que cursó durante el periodo.

  Cada grupo se identifica así: Grupo A, Grupo B, Grupo C, ... Se sabe también, en qué grupo está cada estudiante.

  Es importante tener en cuenta las normas que provee el reglamento estudiantil, ya que estas brindan claridad sobre los procesos a realizar.

- **Proceso:** basados en la información que se ha presentado hasta aquí, se puede observar que lo solicitado debe presentarse agrupado de la siguiente forma:

  1. Por estudiante.
  2. Por grupo.
  3. General, es decir, del total de los estudiantes.

  El promedio del periodo de cada estudiante se calculará así: con una variable se lee la nota definitiva de cada materia y, a través un ciclo que itere 6 veces se leen las notas y se acumulan; finalizado el ciclo se calcula el promedio.

  El ciclo que se acaba de mencionar, estará anidado dentro de un ciclo intermedio, que manejará la información relacionada con los estudiantes y, este a su vez, hará parte de un ciclo externo que manipulará la información de los grupos.

  La Figura 4.39 ilustra de forma general el diseño de la solución, y puede interpretarse de la siguiente manera:

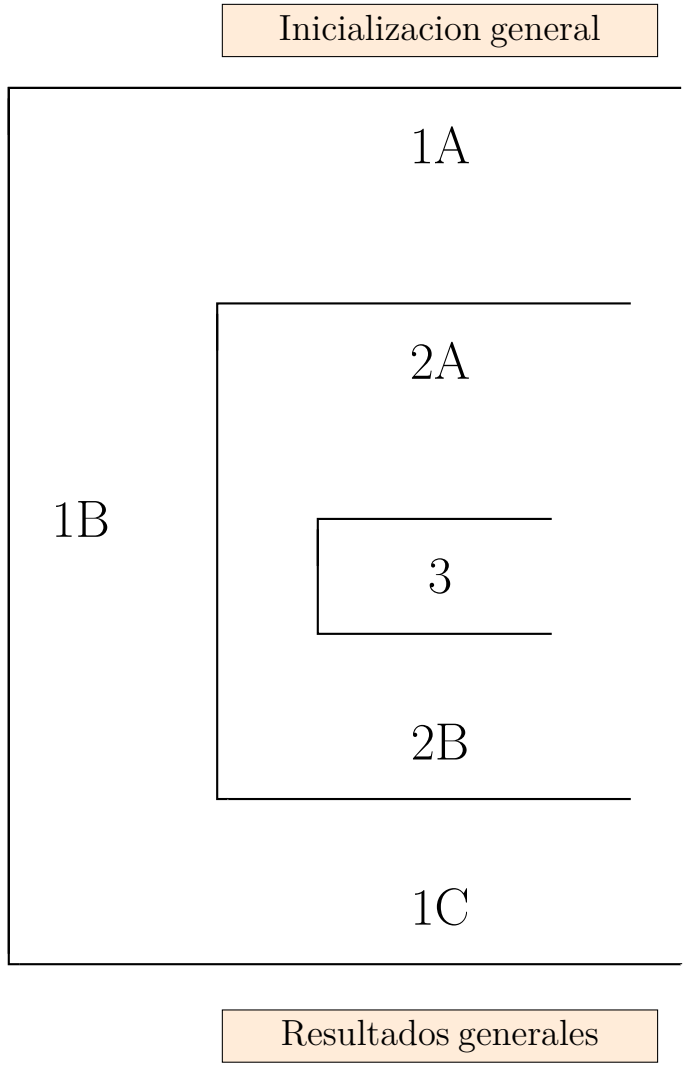


Figura 4.39: Forma general para el Ejemplo 4.23

La zona que se denominó "Inicialización general", permitirá inicializar las variables que se usarán en la condición del ciclo externo, en caso de que se requieran para su ejecución. Igualmente, los contadores de los datos generales que solicitan, por ejemplo, el total de estudiantes, estudiantes en situación condicional, los que aprobaron o reprobaron el promedio y los excluidos se inicializarán allí.

Así mismo, dentro del ciclo externo aparecen las siguientes zonas: `1A` y `1B` y `1C`.

La zona `1A`, contiene los contadores (que deberán inicializarse) que se incrementarán dentro del ciclo intermedio, que se encuentra en la Zona 1B. Estos contadores guardarán el número de estudiantes que aprobaron o reprobaron en cada uno de los grupos. Además, allí se dirá el grupo al que se le van a procesar los datos de sus estudiantes.

La zona `1B`, que está compuesta por las zonas `2A`, `2B` y `3`, contiene el ciclo intermedio, el cual procesará los datos de los estudiantes; dentro de él, habrá un ciclo interno (zona 3), en el que se hará la lectura de las 6 notas definitivas.

Finalizando el ciclo externo, está la zona `1C`; allí se da la información de los grupos, se incrementan algunos contadores generales así como la identificación de cada grupo (Grupo A, Grupo B...).

En el ciclo intermedio están las zonas `2A`, `2B` y `3`. Este ciclo se encargará de los datos de cada estudiante. En la zona 2A, se solicitarán los datos de los alumnos: código, nombre y las seis notas. Recuerde que las notas irán dentro de un ciclo, que estará anidado dentro del ciclo intermedio. En la zona 2A también se inicializan el contador de asignaturas reprobadas y el acumulador de la sumatoria de las notas definitivas, con el cuál se calculará el promedio del periodo.

La zona `3` está entre las zonas `2A` y `2B` y se encargará de procesar las notas definitivas de cada materia.

La zona `2B` servirá para:

- Calcular el promedio del periodo a partir de una media aritmética.
- Encontrar el mayor y menor promedio.
- Mostrar la situación académica de cada estudiante.

- Determinar el número de estudiantes excluidos por bajo rendimiento, los que quedan en situación condicional y los que aprobaron el periodo.

El tercer ciclo que forma el programa, está ubicado en la zona 3. Este ciclo debe iterar 6 veces, en cada ejecución realiza 3 tareas: leer la nota definitiva de cada una de las 6 materias que cursó el estudiante, incrementar la sumatoria de notas definitivas y determinar si el estudiante perdió alguna materia; si esto último llega a ocurrir, se incrementa un contador que, ya en la zona 2B, permitirá decidir si el estudiante es excluido por perder más del 50 % de las materias.

Por último, se ubica la zona llamada "Resultados generales", en la que se hacen los siguientes cálculos e informes globales:

- Obtener los porcentajes generales.
- Mostrar los datos del mayor y menor promedio, número de estudiantes que aprobaron y reprobaron el promedio; porcentaje de estudiantes que aprobaron y el de los que quedaron excluidos por bajo rendimiento y, finalmente informará el número de estudiantes que quedaron en situación condicional.

De acuerdo a las características que presenta este problema, puede afirmarse que las estructuras `while` o `do-while` se adecuan mejor para implementar los ciclos externo e intermedio. El ciclo externo se construirá entonces con una estructura `while` y el intermedio con una estructura `do-while`; ambos ciclos tendrán una pregunta de continuar o no con la ejecución. El ciclo interno, que se encuentra en la zona 3, se diseñará con una estructura `for`, puesto que es la más adecuada, ya que se conoce el número de iteraciones a realizar, seis.

- **Variables requeridas:**
  - Para los estudiantes:
    - `codigo`: almacena el código de cada estudiante.
    - `nombre`: almacena el nombre de cada estudiante.
    - `definitiva`: nota definitiva obtenida por cada estudiante en cada materia.
    - `sumaDefinitivas`: almacena la sumatoria de las 6 notas definitivas del estudiante.
    - `promedioEstudiante`: almacena el cálculo del promedio de las 6 notas definitivas por estudiante.

  - `reprobadas`: número de materias que reprobó el estudiante. Permitirá determinar si el estudiante es expulsado por reprobar más del 50 % de las materias.

- Para cada grupo:
  - `grupo`: identifica a cada uno de los grupos. Inicia en la letra A y se va incrementando por cada uno (Grupo A, Grupo B, ...).
  - `aprobaronGrupo`: número de estudiantes que aprobaron el periodo por grupo.
  - `reprobaronGrupo`: número de estudiantes que reprobaron el periodo por grupo.
- A nivel general:
  - `condicionalGeneral`: almacena el número de estudiantes que quedan en situación condicional.
  - `estudiantesGeneral`: contará el número de estudiantes que se procesen.
  - `aprobaronGeneral`: número total de estudiantes que aprobaron el periodo académico.
  - `reprobaronGeneral`: número total de estudiantes que reprobaron el periodo académico.
  - `excluidosGeneral`: cantidad de estudiantes excluidos por bajo rendimiento.
  - `mayorPromedioGral`: almacena el mayor promedio entre todos los estudiantes.
  - `menorPromedioGral`: almacena el menor promedio entre todos los estudiantes.
  - `porcentajeAprobaronGral`: amacena el cáclulo del porcentaje de estudiantes que aprobaron, con relación a la población total.
  - `porcentajeExcluidosGral`: almacena el cálculo del porcentaje de estudiantes que fueron excluidos por bajo rendimiento, con relación a la población total.
- Para controlar los ciclos:
  - `seguir`: almacena la respuesta del usuario sobre seguir o no. Controla los ciclos externo e intermedio.
  - `materia`: contador para el ciclo que leerá las 6 notas definitivas de cada estudiante.

Conforme al análisis realizado, se propone el Programa 4.35.

Programa 4.35: Universidad

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <windows.h> // Para Unix usar: <stdlib.h>

int main()
{
  int condicionalGeneral, estudiantesGeneral,
      aprobaronGeneral,   reprobaronGeneral,
      excluidosGeneral;
  int aprobaronGrupo, reprobaronGrupo, reprobadas, materia;

  float mayorPromedioGral, menorPromedioGral, definitiva,
        sumaDefinitivas, promedioEstudiante,
        porcentajeAprobaronGral,
        porcentajeExcluidosGral;

  char seguir, grupo;
  char codigo[12], nombre[30];

  // Zona Inicialización general
  seguir = 'S';
  grupo  = 'A';

  condicionalGeneral = 0;
  estudiantesGeneral = 0;
  aprobaronGeneral   = 0;
  reprobaronGeneral  = 0;
  excluidosGeneral   = 0;

  // Ciclo externo que controla los ciclos
  while (seguir == 'S')
  {
    // Zona 1A
    system( "cls" ); // En Unix usar: "clear"

    printf( "Grupo %c: \n\n", grupo );

    aprobaronGrupo  = 0;
    reprobaronGrupo = 0;

    // Ciclo intermendio que procesa los datos del
       estudiante
    // Inicio zona 1B
    do
    {
      // Zona 2A
      printf( "Digite los datos del estudiante\n\n" );
```

```
48        printf( "Código: " );
49        do
50        {
51            fgets( codigo, 12, stdin );
52        } while ( codigo [0] == '\n' );
53
54        printf( "Nombre: " );
55        do
56        {
57            fgets( nombre, 30, stdin );
58        } while ( nombre [0] == '\n' );
59
60        sumaDefinitivas = 0;
61        reprobadas = 0;
62
63        // Ciclo interno, procesa las notas de cada estudiante
64        for( materia = 1; materia <= 6; materia++ )
65        {
66          // Zona 3
67          printf( "Nota definitiva - Materia %d: ", materia );
68          do
69          {
70            scanf("%f", &definitiva);
71          } while (definitiva < 0.0 || definitiva > 5.0);
72
73          sumaDefinitivas += definitiva;
74
75          // Para saber si lo excluyen
76          if ( definitiva < 3.0 )
77              reprobadas ++;
78        }
79
80        // Zona 2B
81        promedioEstudiante = sumaDefinitivas / 6;
82        printf( "\n Su promedio del periodo es: %.1f",
              promedioEstudiante );
83
84        if ( estudiantesGeneral == 0 )
85        {
86          mayorPromedioGral = promedioEstudiante;
87          menorPromedioGral = promedioEstudiante;
88        }
89        else
90        {
91          if ( promedioEstudiante > mayorPromedioGral )
92              mayorPromedioGral = promedioEstudiante;
93          else
94            if ( promedioEstudiante < menorPromedioGral )
95                menorPromedioGral = promedioEstudiante;
```

```
96        }
97
98        // Determina situación académica del estudiante
99        // Verifica quienes quedan excluidos.
100       if ( promedioEstudiante < 2.0 || reprobadas > 3 )
101       {
102         printf( "\n\n %s queda excluido por bajo rendimiento
              ", nombre );
103         excluidosGeneral++;
104         reprobaronGrupo++;
105       }
106       else
107         if ( promedioEstudiante < 3.0 )
108         {
109           printf( "\n\n %s queda en situación condicional",
                nombre );
110           condicionalGeneral++;
111           reprobaronGrupo++;
112         }
113         else
114         {
115           printf( "\n\n %s continúa normalmente", nombre );
116           aprobaronGrupo++;
117         }
118
119       estudiantesGeneral++;
120
121       printf("\n\n\n Hay más estudiantes en este grupo [S] o
              [N]?: ");
122       do
123       {
124         seguir = getchar();
125         seguir = toupper(seguir);
126       }while( seguir != 'S' && seguir != 'N' );
127
128       system( "cls" );
129     } while ( seguir == 'S' );
130
131     // Final zona 1B
132     // Zona 1C
133     // Cálculos e información por grupo...
134
135     printf( "\n Cantidad de estudiantes del grupo [%c] que
            aprobaron el periodo: %d", grupo, aprobaronGrupo);
136     printf( "\n Cantidad de estudiantes del grupo [%c] que
            reprobaron el periodo: %d", grupo, reprobaronGrupo);
137
138     aprobaronGeneral  += aprobaronGrupo;
139     reprobaronGeneral += reprobaronGrupo;
```

```
140
141     grupo ++;
142
143     printf( "\n\n\n Hay más grupos [S] o [N]?: " );
144     do
145     {
146       seguir = getchar();
147       seguir = toupper( seguir );
148     }while( seguir != 'S' && seguir != 'N' );
149   } // Fin del mientras
150
151   // Zona de resultados generales
152
153   porcentajeAprobaronGral = aprobaronGeneral * 100 /
          estudiantesGeneral;
154   porcentajeExcluidosGral = excluidosGeneral * 100 /
          estudiantesGeneral;
155
156   printf( "RESULTADOS GENERALES\n" );
157   printf( "Mayor promedio: %.1f\n", mayorPromedioGral );
158   printf( "Menor promedio: %.1f\n", menorPromedioGral );
159   printf( "Cantidad que aprobaron: %d\n", aprobaronGeneral);
160   printf( "Cantidad que reprobaron: %d\n",reprobaronGeneral);
161   printf( "Aprobación: %.1f%%\n", porcentajeAprobaronGral );
162
163   printf( "De los %d estudiantes fueron excluidos %d,
          equivalente al %.1f%%\n", estudiantesGeneral,
          excluidosGeneral, porcentajeExcluidosGral );
164
165   printf( "Cantidad de estudiantes que quedaron en situación
          condicional: %d\n", condicionalGeneral );
166
167   return 0;
168 }
```

Explicación del programa:

En este programa se utilizaron varias estructuras de repetición anidadas. También se ubicaron varias estructuras de decisión.

Se usó una variable bandera o centinela denominada `seguir`, para controlar la ejecución de dos ciclos. La línea 22, inicializa la variable `seguir = 'S'` con el propósito de que se pueda ingresar al ciclo externo `while` (línea 32), ya que la condición será verdadera. En las líneas 122 a 126, se lee esta variable y se determina si el proceso se repite o no. De la misma manera, las líneas 105 a la 109 hacen lectura, pero en esta ocasión, para verificar la repetición o no del ciclo intermedio `do-while`; dado que

este último ciclo tiene la condición al final, no fue necesario inicializar esta variable para que se lleve a cabo una primer iteración.

En la línea 23, se inicializa la variable grupo con el valor de `'A'`, púes los grupos se identifican como Grupo A, Grupo B, Grupo C, etc. En la línea 141, esta variable se incrementa en 1; como es de tipo carácter toma el siguiente valor, lo que significa que pasa a `'B'`, en la siguiente iteración toma el valor de `'C'` y continuará con las siguientes letras del alfabeto mientras se estén ingresando nuevos grupos.

El ingreso de todos los datos, está condicionado utilizando ciclos `do-while` para validar lo ingresado y su funcionamiento es como se ha explicado en programas anteriores.

Aunque se usaron los tres tipos de estructuras cíclicas vista en el capítulo, hubiese sido igualmente válido usar solo uno o dos de ellos, eso sí, respetando las estructuras y las restricciones del problema.

4.4.1 Prueba de escritorio

A continuación se estudiarán las pruebas de escritorio para un programa con ciclo `for` mediante un ejemplo ilustrativo.

.:Ejemplo 4.24. *Lleve a cabo la prueba de escritorio para el Programa 4.36.*

Programa 4.36: Tabla1

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int indice;
6
7    for(indice = 0; indice <= 30; indice += 5)
8    {
9      if ( indice % 10 == 0 )
10     {
11       printf( " \n%d", indice );
12     }
13   }
14
15   return 0;
16 }
```

La Tabla 4.11 presenta la tabla de verificación para el Programa 4.36.

indice	**indice<= 30**	**indice % 10 == 0**	imprimir **indice**
0	0 <= 30 (V)	0 == 0 (V)	0
5	5 <= 30 (V)	5 == 0 (F)	
10	10 <= 30 (V)	0 == 0 (V)	10
15	15 <= 30 (V)	5 == 0 (F)	
20	20 <= 30 (V)	0 == 0 (V)	20
25	25 <= 30 (V)	5 == 0 (F)	
30	30 <= 30 (V)	0 == 0 (V)	30
35	35 <= 30 (F)		

Tabla 4.11: Prueba de escritorio - Programa 4.36

Explicación de la prueba de escritorio:

Se presenta un ciclo `for`, donde la variable `indice` se inicializa en 0; mientras la condición del ciclo, `indice <= 30`, sea verdadera, el ciclo itera.

En cada iteración, mediante la condición de la estructura `Si(indice % 10 == 0)`, se pregunta que si al dividir el valor de `indice` entre `10`, deja como resto el valor de `0`. En pocas palabras, está preguntando que si el valor de `indice` es múltiplo de `10`. Si la condición es verdadera, entonces se procede a realizar la impresión del valor de `indice`.

.:Ejemplo 4.25. *En este Programa 4.37, se utilizan de forma anidada, los tres ciclos abordados en el capítulo. El propósito es conocer cuál es el resultado que muestra el programa; por esto, una prueba de escritorio constituye una forma eficiente de hacerle seguimiento.*

A continuación está el código del programa escrito en Lenguaje C:

Programa 4.37: Tabla2

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int a, b, c;
6
7   a = 1;
8   b = 3;
```

```
9   while ( a < 5 )
10  {
11    do
12    {
13      for( c = 1; c <= b ; c++ )
14      {
15        printf(" \n%d  %d   %d", a, b, c );
16      }
17      b = b - 2;
18    } while ( b >= 1 );
19    b = 2;
20    a = a + b;
21  }
22  return 0;
23 }
```

La tabla de verificación para el anterior programa es la siguiente:

a	**b**	**c**	**c<= b**	**a< 5**	**b>= 1**	imprimir **a, b, c**
1	3		1 < 5 (V)			
		1		1 <= 3 (V)		1, 3, 1
		2		2 <= 3 (V)		1, 3, 2
		3		3 <= 3 (V)		1, 3, 3
		4		4 <= 3 (F)		
	1				1 >= 1 (V)	
		1		1 <= 1 (V)		1, 1, 1
		2		2 <= 1 (F)		
	-1				−1 >= 1 (F)	
	2					
3			3 < 5 (V)			
		1		1 <= 2 (V)		3, 2, 1
		2		2 <= 2 (V)		3, 2, 2
		3		3 <= 2 (F)		
	0				0 >= 1 (F)	
	2					
5			5 < 5 (F)			

Tabla 4.12: Prueba de escritorio - Programa 4.37

En la Tabla 4.12, cada columna representa una de las variables del programa. Así mismo, se ubicaron las condiciones de cada uno de los ciclos y la instrucción `printf`.

Al evaluar por primera vez la condición del ciclo `while`, se obtiene un resultado verdadero, de esta forma se ejecuta su cuerpo que está conformado por un ciclo `do-while`, el cual, al tener la condición al final,

no tien restricciones para ingresar a su cuerpo por primera vez.

Al interior del ciclo `do-while`, se encuentra un ciclo `for`, cuya variable de control `c` empieza en 1; como su condición es verdadera `(1 <= 3)`, se ejecuta su cuerpo .

Al ingresar al ciclo `for`, la ejecución del programa itera allí hasta que la variable `c` tome un valor de 4. Cuando esto suceda, este ciclo termina y se regresa al ciclo `do-while` donde dos unidades son restadas a la variable `b`. Al encontrar el ciclo `while`, se evalúa la condición, y se tiene que `1 >= 1`. Entonces, el ciclo `do-while` realiza otra iteración; nuevamente la variable `c` del ciclo `for`, toma el valor de 1 y se incia otra serie de iteraciones; en el momento que la variable `c` tome el valor de 2, la condición será falsa (`c <= b`, esto es, `2 <= 1 (F)`). De nuevo, el control del programa regresa al ciclo `do-while`, donde la variable `b` se decrementa en 2 unidades y llegando al valor de -1; en este momento, la condición del ciclo `while` arroja un resultado falso (`-1 >= 1 (F)`), lo que hace que se regrese al ciclo `while`. Ahora, la variable `b` toma el valor de 2 y la variable `a` el valor de 3. Al llegar al final del ciclo `while` se vuelve a su inicio donde está la condición y se evalúa (`a <5`, es decir, `3 <5 (V)`), dando un resultado verdadero. Nuevamente se ingresa al ciclo `do-while` y por obvias razones al ciclo `for`, el índice `c` vuelve a empezar en 1 y este ciclo se ejecuta dos veces. Una vez más el ciclo `do-while` tiene el control, al ejecutar la instrucción `b = b - 2`, la variable `b` toma el valor de 0 con lo cual la condición del ciclo `do - while` se vuelve falsa. Por tercera vez el control lo asume el ciclo `while`, la variable `b` toma el valor de 2 y la variable `a` se incrementa con el valor de `b`, almacenando un 5; al llegar al final del `while` el programa regresa al principio y vuelve a evaluar la condición (`a <5`), obteniendo un resultado falso, lo que termina con la ejecución de los tres ciclos y, por supuesto, del programa.

Note cómo en la tabla, las tres condiciones de los tres ciclos terminan con un valor de falso: `3 <= 2 (F)`, `0 >= 1 (F)` y `5 < 5 (F)`. Recuerde que los ciclos se ejecutan mientras la condición sea verdadera.

En cada iteración que se lleve a cabo con el ciclo `for` se imprimen los valores de `a`, `b` y `c`.

En el momento en que el ciclo intermedio `do-while` no se ejecute, tampoco se ejecutará el `for`.

Cuando el ciclo externo `while` deje de iterar, los otros dos ciclos tampoco lo volverán a hacer.

Actividad 4.5

1. Para el siguiente programa, genere la respectiva prueba de escritorio:

```c
#include <stdio.h>

int main ()
{
  int numero, divisor;

  numero = 6 * 4;

  for(divisor = 1 ;divisor <= numero / 2 ;divisor++)
  {
    if ( numero / divisor * divisor == numero )
    {
      printf ( "%d\n", divisor );
    }
  }

  return 0;
}
```

2. Usando la instrucción `for`, escriba las porciones de código para los siguientes enunciados:

   a) La variable x con un valor inicial de 4, un valor final de 40 e incrementos de a 1.

   b) La variable x que va desde 100 a 20, disminuyendo de 1 en 1.

   c) La variable x que inicia en 10, incrementando de 5 en 5, hasta llegar a 200.

3. Escriba un programa en Lenguaje C, para el siguiente enunciado: dados los extremos de un intervalo [M, N], encuentre la sumatoria de los números pares y de los impares que pertenezcan a él.

4. Implemente un programa en Lenguaje C, que simule el funcionamiento de un temporizador, que reciba como entrada una cantidad de minutos y segundos. En el momento que falten 5 minutos para cumplir el tiempo, deberá dar un mensaje de alerta, cuando finalice mostrará el siguiente mensaje “Tiempo fuera”. Debe funcionar máximo para 1 hora.

Aclaración:

Las tres estructuras repetitivas vistas permiten hacer casi las mismas cosas. Por supuesto que cada estructura reetitiva puede funcionar mejor en determinadas situaciones: El ciclo `while` es ideal cuando se tienen problemas donde el cuerpo del ciclo puede que se ejecute o no, dependiendo de una condición. Por su parte, el ciclo `do-while`, se usa en aquellos procesos que se ejecutan al menos una vez. El ciclo `for`, es el más adecuado si se conoce exactamente el número de iteraciones que se deben realizar. [Trejos, 2017]

Actividad 4.6

La presente actividad es un repaso de todo el capítulo.

1. Responda las siguientes preguntas:

   a) ¿Cuáles son las estructuras repetitivas condicionadas al comienzo?

   b) ¿De las estructuras repetitivas estudiadas en este capítulo, cuál o cuáles de ellas pueden llegar a no ejecutarse y por qué?

   c) ¿De las estructuras repetitivas estudiadas en este capítulo, cuál o cuáles de ellas se ejecutan por lo menos una vez y por qué?

   d) ¿Qué es lo que se conoce como iteración?

   e) Mencione dos situaciones en las que usaría una variable bandera.

   f) ¿Qué diferencias encuentra entre un acumulador y un contador?

   g) ¿Qué sucedería si Usted no escribe la instrucción modificadora de condición dentro de un ciclo `while`?

   h) ¿Para validar la entrada de un dato (lectura) en un programa, cuál es el ciclo ideal? Justifique su respuesta.

   i) Dentro de la cultura popular siempre se ha dicho que, si alguien no puede dormir, debe contar ovejas hasta que logre conciliar el sueño. Suponga que le piden a Usted, que mediante un diagrama de flujo represente esta situación.

j) De las 3 estructuras repetitivas estudiadas en este capítulo, ¿cuál es la menos indicada para hacerlo? Justifique su respuesta.

2. Reescríba cada uno de los ejemplos vistos en este capítulo, esta vez usando una estructura de ciclo diferente. En caso de no ser posible el cambio, justifique su respuesta.

3. En 1937, el matemático alemán Lothar Collatz, enunció la conjetura de Collatz, también conocida como el problema de Ulam, conjetura 3n + 1, entre otros.

   Collatz enunció que, a partir de cualquier número natural, siempre se obtiene la unidad. Para ello se hace el siguiente procedimiento:

   Tome un número n y ejecute las siguientes operaciones:

   Si n es par, halle la división entera entre 2. Si n es impar, multiplíquelo por 3 y súmele 1.

   Con el resultado que obtenga, repita las operaciones anteriores, hasta obtener 1 como respuesta. Ejemplos:

   n = 13, se obtienen los siguientes resultados:

   40, 20, 10, 5, 16, 8, 4, 2, 1.

   n = 6, se obtienen los siguientes resultados:

   3, 10, 5, 16, 8, 4, 2, 1.

   Otra de las curiosidades de esta conjetura, es que cuando se llegue a 1 y se apliquen nuevamente las fórmulas, obtendrá la secuencia 4, 2, 1 de forma infinita.

   El programa que Usted diseñe, debe solicitar un número y aplicar el anterior concepto, imprimiendo los resultados que se obtienen hasta llegar a la unidad.

4. En el Ejemplo 4.14 se desarrolló un programa que determina si un número es o no perfecto. A partir de esa solución, diseñe un nuevo programa que lea un número `n` e imprima los números perfectos entre 1 y `n`.

5. Imprima los primeros 10 múltiplos sucesivos de 3 en orden descendente, a partir de un número `n` que será ingresado por el usuario y que representará el menor valor. Si el número `n`, no es múltiplo de 3, debe llevarse al múltiplo más cercano superior. Por ejemplo, si el número ingresado es el 28, al no ser múltiplo de 3 debe llevarse al siguiente múltiplo superior, o sea, 30. Entonces la salida del programa se visualizaría así: 57, 54, 51, 48, 45, 42, 39, 36, 33, 30.

6. Para el siguiente programa, realice la respectiva prueba de escritorio o tabla de verificación:

```c
#include <stdio.h>

int main ()
{
  int r, p, g;
  r = 4;
  while (r >= 2)
  {
    for ( p = 1 ; p <= 2 ; p++ )
    {
      g = 2;
      do
      {
        printf ( "%d %d\n", p, g );
        g = g + p;
      } while (g <= 4);
    }
    r = r - 2;
  }
  return 0;
}
```

CAPÍTULO 5

PROCEDIMIENTOS Y FUNCIONES

> Controlar la complejidad es la esencia de la programación.
>
> Brian Kernigan

Objetivos del capítulo:

- Aprender a construir procedimientos y funciones en Lenguaje C.
- Manipular datos por medio de la direcciones de memoria (punteros).
- Construir programas para la solución de problemas mediante el uso de funciones y procedimientos.
- Identificar los parámetros de cada procedimiento / función adecuadamente.
- Diferenciar entre el paso de parámetros por valor y por referencia.

Antes de desarrollar el concepto de procedimiento y función, es imprescindible retomar el concepto de la memoria visto en la Página 27 e ilustrado mediante un ejemplo en la Tabla 1.7 (Página 31). Allí, se mencionó que la memoria de un dispositivo de procesamiento (ej: computador), en realidad es un conjunto finito de celdas (cuadros), cada una, con una dirección de memoria (celda), y un contenido; algunas de estas celdas, pueden tener asociado un identificador (nombre) con el cual se puede acceder al contenido de la celda, ya sea para consultar su valor o para alterarlo. En esta sección se explicará como acceder y modificar el contenido de la memoria por medio de la dirección asociada, mediante el uso de los "Punteros".

5.1. Punteros o Apuntadores

Un puntero es una variable que almacena una dirección de memoria y por ello, se suele decir que se "apunta" a dicha posición de memoria. La forma de diferenciar en el momento de la declaración, una variable tradicional de un puntero, es por el uso o no de un asterisco, por ejemplo:

```
int a, *b;
```

En el ejemplo se puede observar cómo se declaran dos variables (`a` y `b`), la primera es una variable a un tipo de dato entero (`int`), mientras que la segunda es un puntero a un tipo de dato entero.

```
a = 4;
b = &a;
```

A la variable (`a`), por ejemplo, se le puede asignar el valor de 4; mientras que a la variable (`b`) se le asigna la dirección de la variable (`a`). Note que para conocer la dirección de la memoria asociada a un identificador, basta con usar el signo `et` (o en inglés *ampersand*) (`&`). Incluso como los punteros son variables, también es posible conocer su dirección de memoria y de ser necesario, se puede crear otro puntero que apunte a él; en este caso se requiere de un puntero doble (Ver el Capítulo 6: Vectores y matrices, para mayor información).

Ahora que (`b`) "apunta" a la variable (`a`), es posible alterar su contenido, sin usar el identificador de (`a`).

```
*b = 60 + *b;
```

Cuando se emplea el asterisco al lado del identificador de un puntero, se debe entender que lo que se desea conocer/modificar en realidad es la celda de memoria que está siendo apuntada por él.

Para el ejemplo, la expresión `60 + *b`, se debe entender como `60 + 4`, debido a que el contenido de la celda apuntada por (`b`), es el contenido de la variable (`a`); luego al realizar la asignación `*b = 64` (resultado de la expresión), dicho valor será almacenado en la celda apuntada por él, es decir, la celda asociada al identificador de `a`. Para comprender mejor el concepto, vea el Programa 5.1.

Programa 5.1: PunterosConcepto

```
1  #include <stdio.h>
2
3  int main ()
4  {
5      int a, b, *c, *d;
6
7      a = 19;
8      b = 73;
9      c = &a;
10     d = &b;
11
12     printf ( "Variable a:\n" );
13     printf ( "\t\t (%p) es la dirección en memoria\n", &a );
14     printf ( "\t\t (%d) es el valor almacenado\n\n",    a );
15
16     printf ( "Variable b:\n" );
17     printf ( "\t\t (%p) es la dirección en memoria\n", &b );
18     printf ( "\t\t (%d) es el valor almacenado\n\n",    b );
19
20     printf ( "Variable c:\n" );
21     printf ( "\t\t (%p) es la dirección en memoria\n", &c );
22     printf ( "\t\t (%p) es el valor almacenado\n",      c );
23     printf ( "\t\t (%d) es el valor apuntado\n\n",     *c );
24
25     printf ( "Variable b:\n" );
26     printf ( "\t\t (%p) es la dirección en memoria\n", &d );
27     printf ( "\t\t (%p) es el valor almacenado\n",      d );
28     printf ( "\t\t (%d) es el valor apuntado\n\n",     *d );
29
30     *c = 5;
31     *d = 31;
32
33     printf ( "Valor de a es %d y el del b es %d\n", a, b );
34     return 0;
35 }
```

Al ejecutar el programa se obtiene:

```
Variable a:
     (0x7ffeeedf0688) es la dirección en memoria
     (19) es el valor almacenado

Variable b:
     (0x7ffeeedf0684) es la dirección en memoria
     (73) es el valor almacenado

Variable c:
     (0x7ffeeedf0678) es la dirección en memoria
     (0x7ffeeedf0688) es el valor almacenado
     (19) es el valor apuntado

Variable b:
     (0x7ffeeedf0670) es la dirección en memoria
     (0x7ffeeedf0684) es el valor almacenado
     (19) es el valor apuntado

Valor de a es 5 y el del b es 31
```

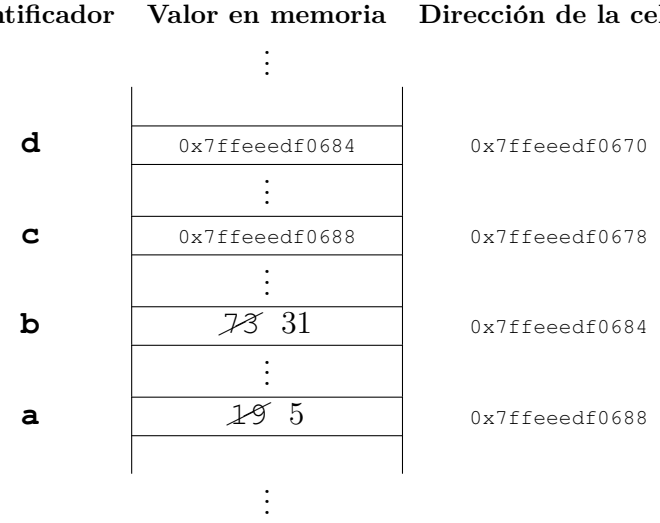


Tabla 5.1: Representación gráfica de la memoria - Programa 5.1

Aclaración:

Es de aclarar que las direcciones de memoria asignadas a las variables, cambian de un sistema computacional a otro, incluso, pueden variar entre ejecuciones diferentes en el mismo sistema.

Observe que, cuando se desea imprimir una dirección de memoria, en la instrucción `printf`, se emplea "`%p`". La forma correcta de imprimir una dirección de memoria es en base 16 (hexadecimal), en lugar de usar base 10 (decimal). Sin embargo, si se desea, imprimirlo en decimal, basta con forzar el formato a un `unsigned long`, tal y como se presenta a continuación:

```
printf ( "Dirección de a es: (%lu)", (unsigned long) &a );
```

Aclaración:

En las variables tradicionales, el tipo de dato es importante para poder conocer la cantidad de memoria que se requiere para almacenar el respectivo dato (Ver la sección de "Tipos de datos" en la Página 22).

En los punteros, el tipo de dato es fundamental para poder desplazarse correctamente por la memoria a partir de la posición almacenada por el puntero. A esto se le conoce como "Aritmética de punteros" y permite realizar ciertas operaciones aritméticas con ellos, como por ejemplo la suma:

```
c = &a + 5;
```

Para el ejemplo, si (`c`) es un puntero y (`a`) es una variable de tipo (`int`), el (`+5`) se refiere a la celda del quinto entero posterior (en términos de la dirección de la memoria) a la posición de la variable (`a`). Entonces, si un `int` ocupa 4 Bytes, el desplazamiento será de 20 Bytes (5 * 4). Pero si, por ejemplo, el tipo de dato fuera `double`, el desplazamiento en Bytes, sería mayor.

En el libro se hará un uso limitado de esta aritmética, solo para algunos ejemplos del Capítulo 6: Vectores y matrices. Se deja al lector, la búsqueda del significado y uso de la palabra reservada `const` (constante) para los punteros.

Finalmente, si se desea indicar que un puntero no "apunta" a ninguna posición de memoria, se debe emplear la constante `NULL`, por ejemplo:

```
c = NULL;
```

Solo para ilustrar, el Programa 5.2 presenta un uso básico de la aritmética de punteros.

Programa 5.2: AritmeticaPunteros

```c
#include <stdio.h>

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

    a  = 19;
    b  = 73;

    c  = &a - 1;
    *c = 31;


    printf ( "Variable a:\n" );
    printf ( "\t\t (%p) es la dirección en memoria\n", &a );
    printf ( "\t\t (%d) es el valor almacenado\n\n",    a );

    printf ( "Variable b:\n" );
    printf ( "\t\t (%p) es la dirección en memoria\n", &b );
    printf ( "\t\t (%d) es el valor almacenado\n\n",    b );

    printf ( "Variable c:\n" );
    printf ( "\t\t (%p) es la dirección en memoria\n", &c );
    printf ( "\t\t (%p) es el valor almacenado\n",      c );
    printf ( "\t\t (%d) es el valor apuntado\n\n",     *c );

    return 0;
}
```

Al ejecutar el programa se obtiene:

```
Variable a:
     (0x7ffee82ed668) es la dirección en memoria
     (19) es el valor almacenado

Variable b:
     (0x7ffee82ed664) es la dirección en memoria
     (31) es el valor almacenado

Variable c:
     (0x7ffee82ed658) es la dirección en memoria
     (0x7ffee82ed664) es el valor almacenado
     (31) es el valor apuntado
```

Identificador	Valor en memoria	Dirección de la celda
	⋮	
c	0x7ffee82ed664	0x7ffee82ed658
	⋮	
b	~~73~~ 31	0x7ffee82ed664
	⋮	
a	19	0x7ffee82ed668
	⋮	

Tabla 5.2: Representación gráfica de la memoria - Programa 5.2

Analice como en las Figuras 5.1 y 5.2, el programa asignó las direcciones de memoria en un orden inverso al orden de la declaración de las mismas.

5.2. Procedimiento

Si dentro de un programa se tiene un bloque de instrucciones con un propósito bien definido, dicho bloque es un candidato perfecto a convertirse en un procedimiento. Así, un procedimiento es un conjunto de instrucciones con una tarea bien definida y un nombre (identificador) que se utiliza para "invocar" (ejecutar) este bloque de instrucciones. La invocación puede ser hecha el número de veces que sea necesario dentro del programa, incluso dentro del mismo procedimiento[1]. Es deseable que el nombre indique la acción o propósito del procedimiento por medio de un verbo y algún complemento que da sentido al verbo, por ejemplo, `insertarNombre`, `sumarVentas`, `verificarCorreo`.

Entre los motivos para usar procedimientos están: el poder descomponer un programa en partes más pequeñas que, en principio, sean más simples de comprender que el todo; por tanto, para resolver un problema se emplean uno o varios procedimientos que en conjunto realizan la tarea deseada. Otro motivo, es la reutilización de código que implica el poder utilizar las instrucciones de un procedimiento tantas veces como sea necesario dentro del programa. Un tercer motivo, es que en términos generales, el código queda más organizado, lo que en un futuro facilitará su mantenimiento.

[1]Utilizado para crear, por ejemplo, soluciones recursivas.

La forma general de esta estructura es presentada en el siguiente segmento de un programa.

```
1 void verboComplemento ( [lista de parámetros] )
2 {
3   Instrucción1;
4   Instrucción2;
5   ...
6   Instrucciónk;
7 }
```

Los procedimientos suelen requerir de datos para poder realizar su tarea; en la estructura general se puede notar que `[lista de parámetros]` indica el lugar en donde se deben enumerar los tipos y las variables que reciben los datos enviados al procedimiento. Estas variables se conocen como parámetros y su existencia está limitada al procedimiento, se dice entonces que, el alcance o ámbito de los parámetros es local. Se emplean, en la forma general, los corchetes (`[ ]`) para indicar que la lista puede estar vacía.

El orden en que se definen los parámetros solo influye en el orden en que los argumentos (valores enviados al procedimiento) se deben escribir, sin alterar la funcionalidad.

Buenas prácticas:

- La cantidad de parámetros debe ser razonable y acorde con la funcionalidad.
- El tamaño de un procedimiento no debería ser mayor a una página, para facilitar su seguimiento.

Aclaración:

Todas las variables declaradas en un procedimiento tienen un alcance local, es decir, una vez el procedimiento termine, sus variables dejan de existir. Como los parámetros son variables locales, sus nombres son independientes de los nombres de las variables utilizadas para enviar los valores como argumentos al momento de invocar el procedimiento; es decir, no existe conflicto entre las variables locales y las variables externas al procedimiento.

.:Ejemplo 5.1. *Diseñe un programa con procedimientos que permita mostrar el mensaje de "Bienvenida".*

Análisis del problema:

- **Resultados esperados:** imprimir el mensaje "Bienvenida".
- **Datos disponibles:** Ninguno.
- **Proceso:** imprimir el mensaje solicitado.
- **Variables requeridas:** Ninguna.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 5.1 y se escribe el Programa 5.3.

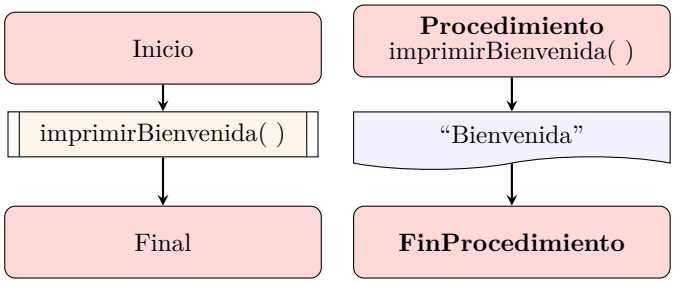


Figura 5.1: Diagrama de flujo del Programa `Mensaje`

Observe la forma cómo se representó la invocación de un procedimiento (un rectángulo con dos pequeñas líneas a los extremos). En el momento de la invocación siempre se deben usar los paréntesis, para dejar explícita la información que se desea enviar al procedimiento (los argumentos); en la figura están vacíos ya que el procedimiento no declaró ningún parámetro.

Programa 5.3: Mensaje

```
1  #include <stdio.h>
2
3  void imprimirBienvenida();
4
5  int main()
6  {
7      imprimirBienvenida();
8
9      return 0;
10 }
```

```
11
12 void imprimirBienvenida()
13 {
14     printf( "Bienvenida" );
15 }
```

Al ejecutar el programa:

```
Bienvenida
```

Explicación del programa:

El Lenguaje C requiere que los procedimientos estén previamente declarados antes de utilizarlos. Una forma de hacer esto es declarando la "firma" del procedimiento antes del `main`, en la zona de encabezados del programa.

```
3 void imprimirBienvenida();
```

Aclaración:

La "firma" de un procedimiento, visto de la forma general, se refiere a la palabra reservada `void`, seguida del nombre del procedimiento, la declaración de los parámetros encerrados entre paréntesis y un punto y coma:

```
1 void verboComplemento ( [lista de parámetros] );
```

Los procedimientos, a diferencia de las funciones del Lenguaje C, no hacen uso de la palabra reservada `return` dentro del cuerpo del procedimiento, como si lo hacen estas últimas, tal y como se explicará más adelante.

Por otro lado, la única instrucción del programa que se está analizando, es precisamente la invocación del procedimiento `imprimirBienvenida()`, la cual indica que se deben ejecutar todas las líneas definidas con este nombre, para el caso, la línea 14.

```
14     printf( "Bienvenida" );
```

.:Ejemplo 5.2. *Diseñe un programa con procedimientos que permita ingresar el nombre de dos personas y mostrar el mensaje de saludo "Hola" seguido del nombre de cada una de ellas de forma independiente.*

Análisis del problema:

- **Resultados esperados:** dos mensajes de saludo, uno por cada nombre ingresado.
- **Datos disponibles:** los dos nombres de las personas.
- **Proceso:** solicitar al usuario los nombres de las dos personas, luego invocar dos veces un procedimiento que imprima el saludo del nombre que se enviará como argumento.
- **Variables requeridas:**
  - `nombre1`: nombre de la primera persona.
  - `nombre2`: nombre de la segunda persona.

  Parámetros del procedimiento:
  - `nombre`: nombre de la persona a saludar.

Conforme al anterior análisis, se realiza el diagrama de flujo de la Figura 5.2 y se escribe el Programa 5.4.

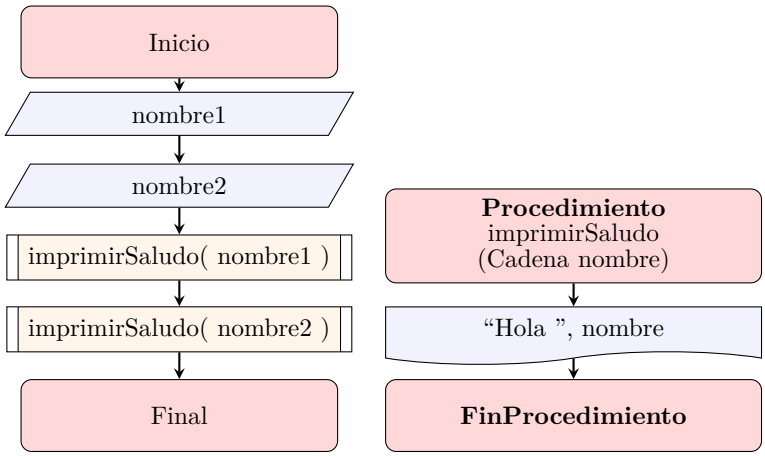


Figura 5.2: Diagrama de flujo del Programa `Mensaje2`

Aclaración:

Los diagramas de flujo son una herramienta importante para la representación gráfica de un programa, pero como se ha podido evidenciar, usar ambas representaciones es redundante, por tal motivo, solo se usará una u otra representación.

Programa 5.4: Mensaje2

```
1  #include <stdio.h>
2
3  void imprimirSaludo();
4
5  int main()
6  {
7      char nombre1 [ 50 ];
8      char nombre2 [ 50 ];
9
10     printf ( "Ingrese el primer nombre: " );
11     fgets( nombre1, sizeof( nombre1 ), stdin );
12
13     printf ( "Ingrese el segundo nombre: " );
14     fgets( nombre2, sizeof( nombre2 ), stdin );
15
16     imprimirSaludo( nombre1 );
17     imprimirSaludo( nombre2 );
18
19     return 0;
20 }
21
22 void imprimirSaludo(char nombre [])
23 {
24     printf( "Hola  %s\n", nombre );
25 }
```

Al ejecutar el programa:

```
Ingrese el primer nombre: Diana Milena
Ingrese el segundo nombre: Ana Sofia
Hola Diana Milena
Hola Ana Sofia
```

Explicación del programa:

Lo primero es declarar las variables y solicitar los datos disponibles (líneas 7 a la 14); posteriormente se invoca dos veces el procedimiento `imprimirSaludo`.

```
16   imprimirSaludo( nombre1 );
17   imprimirSaludo( nombre2 );
```

Es importante comprender que, con el nombre (`imprimirSaludo`) se identifica el procedimiento a invocar, luego por medio de los parámetros, se envía al procedimiento los datos que él requiera. Estos datos, son una **copia** del valor de los argumentos, para el caso, la dirección en donde comienza el `nombre1` y `nombre2` respectivamente. Los valores de los argumentos se almacenan en los parámetros del procedimiento; en este caso, solo es la dirección en donde se encuentra el nombre que se desea imprimir.

El tipo del parámetro (`char nombre []`) coincide plenamente con el tipo del argumento, esto es, con el valor de la variable `nombre1` y `nombre2`, las cuales están declaradas como: (`char nombre1[ 50 ]`) y (`char nombre2[ 50 ]`).

En el Programa 5.4 se observa el uso de la función `fgets`, la cual es algo compleja, así que se le recomienda al lector hacer uso de `#define` para declarar una forma abreviada para esta o cualquier otra función. Ver Programa 5.5.

Programa 5.5: Mensaje2

```
1  #include <stdio.h>
2
3  #define leerCadena(x) fgets( x, sizeof( x ), stdin )
4
5  void imprimirSaludo();
6
7  int main()
8  {
9      char nombre1 [ 50 ];
10     char nombre2 [ 50 ];
11
12     printf ( "Ingrese el primer nombre: " );
13     leerCadena( nombre1 );
14
15     printf ( "Ingrese el segundo nombre: " );
16     leerCadena( nombre2 );
17
18     imprimirSaludo( nombre1 );
19     imprimirSaludo( nombre2 );
20
21     return 0;
22 }
23
24
```

```
25 void imprimirSaludo(char nombre [])
26 {
27     printf( "Hola %s", nombre );
28 }
```

.:Ejemplo 5.3. *Diseñe un programa con procedimientos que permita intercambiar el valor de dos variables enteras.*

Análisis del problema:

- **Resultados esperados:** el contenidos intercambiado de dos variables.
- **Datos disponibles:** los dos números enteros.
- **Proceso:** solicitar al usuario los dos números enteros, luego invoca un procedimiento que intercambie los valores, para luego imprimir los nuevos valores..
- **Variables requeridas:**
  - `x`: variable que almacena el primer número.
  - `y`: variable que almacena el segundo número.

  Internos al procedimiento se requiere el parámetro:
  - `*a`: la dirección de memoria del primer valor.
  - `*b`: la dirección de memoria del segundo valor.

De acuerdo al análisis planteado, se propone el Programa 5.6.

Programa 5.6: Punteros

```
1  #include <stdio.h>
2
3  void intercambiar( int *a, int *b );
4
5  int main()
6  {
7     int x, y;
8
9     printf ( "Ingrese el primer valor: " );
10    scanf  ( "%d", &x );
11
12    printf ( "Ingrese el primer valor: " );
13    scanf  ( "%d", &y );
14
15    intercambiar ( &x, &y );
```

```
16
17    printf ( "Valores intercambiados son: %d y %d\n", x, y );
18
19    return 0;
20 }
21
22 void intercambiar( int *a, int *b )
23 {
24     int temp;
25
26     temp = *a;
27     *a   = *b;
28     *b   = temp;
29 }
```

Al ejecutar el programa:

```
Ingrese el primer valor: 76
Ingrese el primer valor: 45
Los valores intercambiados son: 45 y 76
```

Explicación del programa:

Lo más importante a observar en el ejemplo, es la necesidad de requerir las direcciones de memoria de las dos variables a intercambiar, y como estas ilustran el paso de parámetros por referencia.

Aclaración:

Hay que recordar que **una copia del valor** de los argumentos se envía al procedimiento, por eso, si se enviara una copia del valor de las variables al procedimiento y se intercambiarían sus valores en los parámetros, este cambio **NO** afectaría los valores de las variables definidas en el `main`. Por otro lado, al enviar una copia de la dirección de las variables, el procedimiento, ahora sí puede alterar el valor de las variables originales usando las respectivas direcciones.

.:Ejemplo 5.4. *Diseñe un programa con procedimientos que permita imprimir dos secuencias de números: "3 6 9 12 ... $3 * n$" y "5 10 15 20 ... $5 * m$", en donde n y m representan la cantidad de términos de la serie.*

Análisis del problema:

- **Resultados esperados:** la impresión de las dos secuencias de números.
- **Datos disponibles:** el valor de n y el de m para poder generar las dos secuencias de números.
- **Proceso:** primero se le solicita al usuario los valores de n y m, luego se invoca dos veces un procedimiento que imprima una secuencia dado un valor base (3 o 5) y el valor final (n o m).

  En términos generales, el procedimiento requiere dos parámetros, un valor base (*base*) y la cantidad (*cantidad*) de términos. Con esta información se imprime la serie:

  $$1 * base \;\; 2 * base \;\; 3 * base \;\; \ldots \;\; cantidad * base$$

  Ahora si:

base = 3 y *cantidad* = **4**	*base* = 5 y *cantidad* = **6**
$1*3 \;\; 2*3 \;\; 3*3 \;\; \mathbf{4}*3$	$1*5 \;\; 2*5 \;\; 3*5 \;\; 4*5 \;\; 5*5 \;\; \mathbf{6}*5$
3 6 9 12	5 10 15 20 25 30

- **Variables requeridas:**
  - `n`: cantidad de términos de la primera serie.
  - `m`: cantidad de términos de la segunda serie.

  Variables locales internas al procedimiento:

  - `base`: valor de la base para generar la serie (parámetro 1).
  - `cantidad`: número de términos de la serie (parámetro 2).
  - `i`: variable que controla el ciclo que va desde 1 hasta `cantidad` se requiere.
  - `termino`: variable que almacena el término actual de la serie `termino = i * base`.

De acuerdo al análisis planteado, se propone el Programa 5.7.

Programa 5.7: Serie

```
1  #include <stdio.h>
2
3  void imprimirSerie( int base, int cantidad );
4
5  int main()
6  {
7      int n, m;
8
9      printf( "Cantidad de términos de la primera serie: " );
10     scanf( "%i", &n );
11
12     printf( "Cantidad de términos de la segunda serie: " );
13     scanf( "%i", &m );
14
15     imprimirSerie( 3, n );
16     imprimirSerie( 5, m );
17
18     return 0;
19 }
20
21 void imprimirSerie( int base, int cantidad )
22 {
23     int termino;
24     int i;
25
26     for( i = 1 ; i <= cantidad ; i++ )
27     {
28         termino = i * base;
29         printf( "%d ", termino );
30     }
31
32     printf( "\n" );
33 }
```

Al ejecutar el programa:

```
Cantidad de términos de la primera serie: 4
Cantidad de términos de la segunda serie: 6
3 6 9 12
5 10 15 20 25 30
```

Explicación del programa:

En las primeras líneas del programa (líneas de la 3 a la 10) se declaran las variables y se solicitan los datos disponibles.

Luego se invoca dos veces el procedimiento `imprimirSerie` para generar las respectivas series. La diferencia entre ellas es el valor de la base (3 y 5) y la cantidad de términos (`n` y `m`).

```
15   imprimirSerie( 3, n );
16   imprimirSerie( 5, m );
```

Lo interesante es que no se requieren procedimientos diferentes, para imprimir las dos series. El procedimiento generaliza la impresión de cualquier serie en donde se conoce un valor de la `base` y una `cantidad` de términos.

```
21 void imprimirSerie( int base, int cantidad )
```

Para la primera invocación:

- el parámetro `base` toma el valor de 3.
- el parámetro `cantidad` toma el valor de la variable `n`.

Para la segunda invocación:

- el parámetro `base` toma el valor de 5.
- el parámetro `cantidad` toma el valor de la variable `m`.

Una vez los parámetros tomen sus valores, se ejecutan las instrucciones para imprimir la respectiva serie.

```
23      int termino;
24      int i;
25
26      for( i = 1 ; i <= cantidad ; i++ )
27      {
28          termino = i * base;
29          printf( "%d ", termino );
30      }
```

En este caso la variable `termino` podría ser eliminada haciendo lo siguiente:

```
        int i;

        for( i = 1 ; i <= cantidad ; i++ )
        {
            printf( "%d ", (i * base) );
        }
```

Aclaración:

El uso apropiado de procedimientos facilita no solo la reutilización de código, sino que se puede ganar legibilidad al simplificar la cantidad de líneas de la parte central del programa o de otros procedimientos. Usar procedimientos permite "ocultar" o encapsular funcionalidades y, por medio de las invocaciones se tiene acceso a ellas. Cada procedimiento se puede estudiar y comprender aisladamente y luego se crea un programa y otros procedimientos que haciendo las respectivas invocaciones resuelven un problema mayor. Para el caso, el procedimiento imprime una serie, pero su doble invocación resolvió el problema inicial.

.:Ejemplo 5.5. *Se desea crear un procedimiento genérico que permita imprimir la coordenadas (x, y) de un punto cualquiera en un plano cartesiano, con su respectivo nombre del punto. (Ver Figura 5.3).*

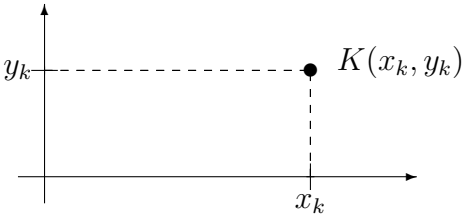


Figura 5.3: Coordenada de un punto K

Análisis del problema:

- **Resultados esperados:** la impresión del nombre del punto ("K") con sus respectivas coordenadas `x`, `y`.
- **Datos disponibles:** el valor de las coordenadas `x`, `y`.
- **Proceso:** solicitar al usuario las coordenadas del punto "K" y luego se invoca el procedimiento para imprimirlo.
- **Variables requeridas:**
  - `xk`: valor de la coordenada `x` del punto "K".
  - `yk`: valor de la coordenada `y` del punto "K".

  Internos al procedimiento se necesitan:

- `nombre`: nombre del punto arbitrario.
- `x`: valor de la coordenada `x` del punto arbitrario.
- `y`: valor de la coordenada `y` del punto arbitrario.

De acuerdo al análisis planteado, se propone el Programa 5.8.

Programa 5.8: Punto

```
1  #include <stdio.h>
2
3  void imprimirPunto( char nombre, float x, float y );
4
5  int main()
6  {
7      float xk, yk;
8
9      printf( "Ingrese el valor de x del punto K: " );
10     scanf( "%f", &xk );
11
12     printf( "Ingrese el valor de y del punto K: " );
13     scanf( "%f", &yk );
14
15     imprimirPunto( 'K', xk, yk );
16
17     return 0;
18 }
19
20 void imprimirPunto( char nombre, float x, float y )
21 {
22     printf( "El punto %c ", nombre );
23     printf( "tiene coordenadas  [%.2f , %.2f]\n", x , y );
24 }
```

Al ejecutar el programa:

```
Ingrese el valor de x del punto K: 5
Ingrese el valor de y del punto K: 73
El punto K tiene coordenadas ( 5.00, 73.00 )
```

```
Ingrese el valor de x del punto K: 31
Ingrese el valor de y del punto K: 5
El punto K tiene coordenadas ( 31.00, 5.00 )
```

Explicación del programa:

En la línea 7 se declaran las variables del programa y posteriormente se leen los datos disponibles (líneas 9 a la 13). Luego ee invoca el procedimiento para imprimir el punto (`imprimirPunto`), enviando la información que él requiere (nombre del punto, las coordenadas `x`, `y`).

Observe que los parámetros (`nombre`, `x`, `y`) toman los valores "K", el valor de la coordenada x del punto k (`xk`) y el valor de la coordenada y del punto k (`yx`).

.:Ejemplo 5.6. *Usando el procedimiento del punto anterior, diseñe un programa que permita imprimir los datos de dos puntos "T" y "S" (Ver Figura 5.4).*

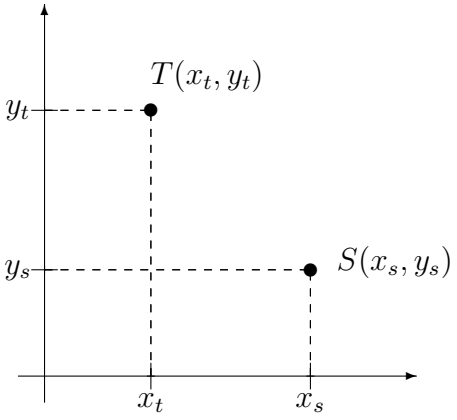


Figura 5.4: Puntos S, T en el plano cartesiano

Análisis del problema:

- **Resultados esperados:** la impresión del nombre del punto ("T" y "S") con sus respectivas coordenadas `x`, `y`.
- **Datos disponibles:** el valor de las coordenadas `x`, `y` de ambos puntos.
- **Proceso:** solicitar al usuario las coordenadas de los puntos "T" y "S" y luego se invoca dos veces el procedimiento para imprimir la información.
- **Variables requeridas:**

  - `xt`: valor de la coordenada `x` del punto "T".
  - `yt`: valor de la coordenada `y` del punto "T".
  - `xs`: valor de la coordenada `x` del punto "S".
  - `ys`: valor de la coordenada `y` del punto "S".

  Internos al procedimiento se necesitan:

  - `nombre`: nombre del punto arbitrario.

- `x`: valor de la coordenada `x` del punto arbitrario.
- `y`: valor de la coordenada `y` del punto arbitrario.

Conforme al anterior análisis, se realiza el diagrama de flujo de las Figuras 5.5 y 5.6 y se escribe el Programa 5.9.

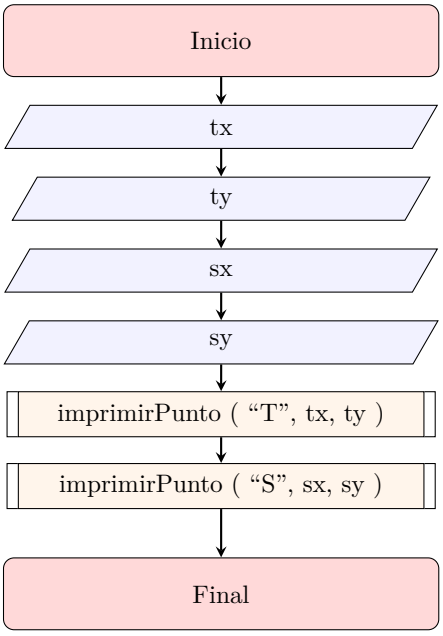


Figura 5.5: Diagrama de flujo del Programa `Puntos` - Parte 1

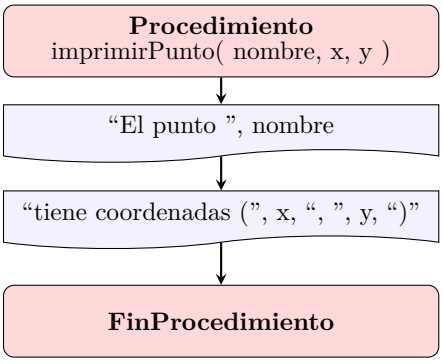


Figura 5.6: Diagrama de flujo del Programa `Puntos` - Parte 2

Programa 5.9: Puntos

```
#include <stdio.h>

void imprimirPunto(char nombre, float x, float y );

int main()
{
    float xt, yt, xs, ys;

    printf( "Ingrese el valor de x del punto T: " );
    scanf( "%f", &xt);

    printf( "Ingrese el valor de y del punto T: " );
    scanf( "%f", &yt);

    printf( "Ingrese el valor de x del punto S: " );
    scanf( "%f", &xs);

    printf( "Ingrese el valor de y del punto S: " );
    scanf( "%f", &ys );

    imprimirPunto( 'T', xt, yt );
    imprimirPunto( 'S', xs, ys );

    return 0;
}

void imprimirPunto( char nombre, float x, float y )
{
    printf( "El punto %c ", nombre );
    printf( "tiene coordenadas  [%.2f , %.2f]\n", x , y );
}
```

Al ejecutar el programa:

```
Ingrese el valor de x del punto T: 31
Ingrese el valor de y del punto T: 5
Ingrese el valor de x del punto S: 19
Ingrese el valor de y del punto S: 73
El punto T tiene coordenadas ( 31.00, 5.00 )
El punto S tiene coordenadas ( 19.00, 73.00 )
```

```
Ingrese el valor de x del punto T: 10
Ingrese el valor de y del punto T: 27
Ingrese el valor de x del punto S: 1
Ingrese el valor de y del punto S: 31
El punto T tiene coordenadas ( 10.00, 27.00 )
El punto S tiene coordenadas ( 1.00, 31.00 )
```

Explicación del programa:

Observe que, adicional a la declaración y lectura de los datos disponibles, se hace la invocación al mismo procedimiento pero enviando la información correspondiente en cada caso. Note que el procedimiento `imprimirPunto` no sufrió modificación alguna, esta es una de las fortalezas del uso de procedimientos en los programas, el poder reutilizar código.

5.3. Funciones

Una función es un tipo especial de procedimiento que entrega (retorna) un valor como resultado. Algunos autores definen los procedimientos como funciones que no retornan ningún tipo de valor, aunque en principio esa definición es correcta, en realidad es imprecisa desde el punto de vista del concepto matemático de función del cual se deriva este término.

La declaración de una función es similar a la de un procedimiento, con la diferencia que no se escribe `void` antes del nombre de la función, sino el tipo del dato que la función retorna.

La forma general de esta estructura es presentada en el segmento del Programa:

```
1  tipoDato verboComplemento ( [lista de parámetros] )
2  {
3    Instrucción1;
4    Instrucción2;
5    ...
6    Instrucciónn;
7
8    return valorARetornar;
9  }
```

Aclaración:

En capítulos anteriores del libro, ya se han utilizado varias funciones propias del Lenguaje C y que fueron citadas en el primer capítulo cuando se habló de los archivos de cabecera, entre las funciones utilizadas se encuentran: `scanf`, `fgets`, `printf`, `sin`, `cos`, `tan`, `sqrt`, `main`, entre otras.

.:Ejemplo 5.7. *Diseñe un programa que permita determinar la distancia entre dos puntos "T" y "S" (Ver Figura 5.7).*

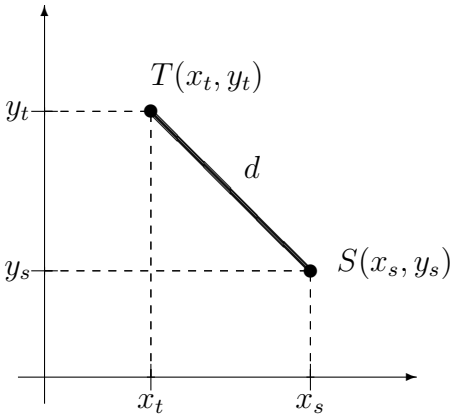


Figura 5.7: Distancia entre dos puntos T y S

Análisis del problema:

- **Resultados esperados:** la distancia entre los puntos "T" y "S".
- **Datos disponibles:** las coordenadas `x`, `y` de ambos puntos.
- **Proceso:** solicitar al usuario que ingrese las coordenadas `x`, `y` de ambos puntos. Posteriormente se procede a calcular la distancia entre los puntos usando la ecuación: $d = \sqrt{(x_s - x_t)^2 + (y_s - y_t)^2}$
- **Variables requeridas:**
  - `xt`: valor de la coordenada `x` del punto "T".
  - `yt`: valor de la coordenada `y` del punto "T".
  - `xs`: valor de la coordenada `x` del punto "S".
  - `ys`: valor de la coordenada `y` del punto "S".
  - `distancia`: valor de la distancia entre los puntos "T" y "S".

  Internos al procedimiento `calcularDistancia` se necesitan:

  - Parámetros:
    - `x1`: valor de la coordenada `x` del primer punto.
    - `y1`: valor de la coordenada `y` del primer punto. arbitrario.
    - `x2`: valor de la coordenada `x` del segundo punto.

- y2: valor de la coordenada y del segundo punto.

- d: distancia entre dos puntos (variable local).

De acuerdo al análisis planteado, se propone el Programa 5.10.

Programa 5.10: DistanciaPuntos

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

float calcularDistancia( float x1, float y1,
                         float x2, float y2 );

int main()
{
    float xt, yt, xs, ys, distancia;

    printf( "Ingrese el valor de x del punto T: ");
    scanf( "%f", &xt);

    printf( "Ingrese el valor de y del punto T: ");
    scanf( "%f", &yt);

    printf( "Ingrese el valor de x del punto S: ");
    scanf( "%f", &xs);

    printf( "Ingrese el valor de y del punto S: ");
    scanf( "%f", &ys);

    distancia = calcularDistancia(xt, yt, xs, ys);

    printf( "Distancia entre T y S es %.2f", distancia );

    return 0;
}

float calcularDistancia( float x1, float y1,
                         float x2, float y2 )
{
   float d;

   d = sqrt( pow( (x2-x1), 2.0 ) + pow( (y2-y1), 2.0 ) );

   return d;
}
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese el valor de x del punto T: 10
Ingrese el valor de y del punto T: 4
Ingrese el valor de x del punto S: 6
Ingrese el valor de y del punto S: 7
Distancia entre T y S es 5.00
```

Segunda ejecución

```
Ingrese el valor de x del punto T: 10
Ingrese el valor de y del punto T: 27
Ingrese el valor de x del punto S: 1
Ingrese el valor de y del punto S: 31
Distancia entre T y S es 9.85
```

Explicación del programa:

En las primeras líneas (de la 9 a la 21) se declaran las variables y se leen los datos disponibles, luego se invoca el procedimiento `calcularDistancia` enviando la información disponibles, es decir, se invoca con las coordenadas del punto "T" y el punto "S". El procedimiento recibe esta información en sus respectivos parámetros `x1`, `y1`, `x2`, `y2` para luego aplicar la formula de distancia y retornar el valor calculado mediante la variable local `d`. La función puede ser usado para calcular la distancia entre cualquier par de puntos (ver el siguiente ejemplo).

.:Ejemplo 5.8. *Diseñe un programa que calcule e imprima el perímetro*[2] *de un triangulo dadas las coordenadas de cada uno de sus vértices. (Ver Figura 5.8).*

Análisis del problema:

- **Resultados esperados:** el perímetro de un triángulo dados los tres vértices del mismo, denominados arbitrariamente "P", "Q", y "R".
- **Datos disponibles:** las coordenadas `x`, `y` de cada uno de los tres vértices.
- **Proceso:** se le solicita al usuario que ingrese las coordenadas `x`, `y` de cada uno de los tres vértices, luego se determina la distancia entre ellos, invocando la función:

[2]El perímetro de una figura, se calcula como la suma de las longitudes de los lados de la figura. La longitud se puede determinar calculando la distancia entre los puntos extremos de cada lado (vértices).

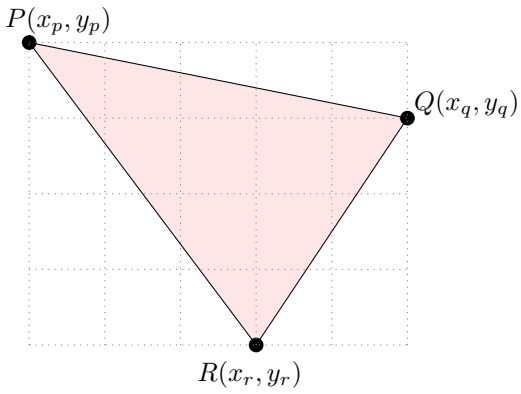


Figura 5.8: Triángulo de vértices P, Q y R

```
d1 = calcularDistancia( x1, y1, x2, y2 );
d2 = calcularDistancia( x1, y1, x3, y3 );
d3 = calcularDistancia( x2, y2, x3, y3 );
```

Posteriormente, se realiza la suma de las distancias para encontrar el valor del perímetro:

```
perimetro   = d1 + d2 + d3;
```

- **Variables requeridas:**

  - `xt`: valor de la coordenada `x` del punto "P".
  - `yt`: valor de la coordenada `y` del punto "P".
  - `xs`: valor de la coordenada `x` del punto "Q".
  - `ys`: valor de la coordenada `y` del punto "Q".
  - `xs`: valor de la coordenada `x` del punto "R".
  - `ys`: valor de la coordenada `y` del punto "R".
  - `perimetro`: valor del triangulo con vértices "P", "Q" y "R".

  Internos al procedimiento `calcularPerimetro` se necesitan:

  - Parámetros:
    - `x1`: valor de la coordenada `x` del primer punto.
    - `y1`: valor de la coordenada `y` del primer punto.
    - `x2`: valor de la coordenada `x` del segundo punto.
    - `y2`: valor de la coordenada `y` del segundo punto.
    - `x3`: valor de la coordenada `x` del tercer punto.
    - `y3`: valor de la coordenada `y` del tercer punto.

- Variables locales:
  - `d1`: valor de distancia entre el primer y segundo punto.
  - `d2`: valor de distancia entre el primero y tercer punto.
  - `d3`: valor de distancia entre el segundo y tercer punto.
  - `perimetro`: valor del perímetro calculado como la suma de las distancias entre los puntos.

Internos al procedimiento `calcularDistancia` se necesitan:

- Parámetros:
  - `x1`: valor de la coordenada `x` del primer punto.
  - `y1`: valor de la coordenada `y` del primer punto. arbitrario.
  - `x2`: valor de la coordenada `x` del segundo punto.
  - `y2`: valor de la coordenada `y` del segundo punto.
- `d`: distancia entre dos puntos.

De acuerdo al análisis planteado, se propone el Programa 5.11.

Programa 5.11: Perimero

```
1  #include <stdio.h>
2  #include <math.h>
3
4  float calcularDistancia( float x1, float y1,
5                           float x2, float y2 );
6  float calcularPerimetro( float x1, float y1,
7                           float x2, float y2,
8                           float x3, float y3 );
9
10 int main()
11 {
12     float xp, yp, xq, yq, xr, yr, perimetro;
13
14     printf( "Ingrese el valor de x del punto P: ");
15     scanf( "%f", &xp);
16
17     printf( "Ingrese el valor de y del punto P: ");
18     scanf( "%f", &yp);
19
20     printf( "Ingrese el valor de x del punto Q: ");
21     scanf( "%f", &xq);
22
23     printf( "Ingrese el valor de y del punto Q: ");
24     scanf( "%f", &yq);
25
26     printf( "Ingrese el valor de x del punto R: ");
```

```
27      scanf( "%f", &xr);
28
29      printf( "Ingrese el valor de y del punto R: ");
30      scanf( "%f", &yr);
31
32      perimetro = calcularPerimetro(xp, yp, xq, yq, xr, yr);
33
34      printf( "El perímetro del triángulo es: %.2f ",
            perimetro );
35
36      return 0;
37  }
38
39  float calcularDistancia( float x1, float y1,
40                           float x2, float y2 )
41  {
42     float d;
43
44     d = sqrt(pow((x2-x1),2) + pow((y2-y1),2));
45
46     return d;
47  }
48
49  float calcularPerimetro( float x1, float y1,
50                           float x2, float y2,
51                           float x3, float y3 )
52  {
53      float p, d1, d2, d3;
54
55      d1 = calcularDistancia( x1, y1, x2, y2 );
56      d2 = calcularDistancia( x1, y1, x3, y3 );
57      d3 = calcularDistancia( x2, y2, x3, y3 );
58
59      p = d1 + d2 + d3;
60
61      return p;
62  }
```

Al ejecutar el programa:

Primera ejecución

```
Ingrese el valor de x del punto P: 0
Ingrese el valor de y del punto P: 4
Ingrese el valor de x del punto Q: 5
Ingrese el valor de y del punto Q: 3
Ingrese el valor de x del punto R: 3
Ingrese el valor de y del punto R: 0
El perímetro del triangulo es 13.70
```

Segunda ejecución

```
Ingrese el valor de x del punto P: 5
Ingrese el valor de y del punto P: 3
Ingrese el valor de x del punto Q: 5
Ingrese el valor de y del punto Q: 7
Ingrese el valor de x del punto R: 8
Ingrese el valor de y del punto R: 7
El perímetro del triángulo es 12.00
```

Explicación del programa:

Lo primero es observar que en este ejemplo se emplearon dos funciones, más la parte central del programa. En la parte central (`main`) se declaran y se solicitan todos los datos disponibles (líneas de la 12 a la 30), luego se invoca la función que calcula el perímetro del triángulo (línea 32) y finalmente se imprime el resultado obtenido (línea 34).

Hasta este punto no es necesario conocer cómo se calcula el perímetro, debido a que todo está "oculto" dentro el procedimiento (encapsulado). Por tanto, la parte central se limita a solicitar la información, calcular los resultados esperados e imprimirlos.

Por otro lado, la función `calcularPerimetro` recibe la información de las coordenadas de los tres vértices y procede a calcular el perímetro mediante la suma de las distancias entre los vértices. Pero estas distancias son calculada mediante la función `calcularDistancia`.

```
49 float calcularPerimetro( float x1, float y1,
50                          float x2, float y2,
51                          float x3, float y3 )
52 {
53     float p, d1, d2, d3;
54
55     d1 = calcularDistancia( x1, y1, x2, y2 );
56     d2 = calcularDistancia( x1, y1, x3, y3 );
57     d3 = calcularDistancia( x2, y2, x3, y3 );
58
59     p = d1 + d2 + d3;
60
61     return p;
62 }
```

La función `calcularDistancia` recibe la información de los dos vértices arbitrarios y calcula la distancia entre ellos, así, al invocar tres veces la función con la información apropiada (líneas de la 55 a la 57) es posible calcular todas las distancias necesarias.

```
39 float calcularDistancia( float x1, float y1,
40                          float x2, float y2 )
41 {
42    float d;
43
44    d = sqrt(pow((x2-x1),2) + pow((y2-y1),2));
45
46    return d;
47 }
```

Aclaración:

Aunque en apariencia los programas se vuelven más largos, el hacer uso de procedimientos / funciones permite la solución de problemas cada vez más complejos, al permitir atacar la complejidad del problema al dividiéndolos en componentes más pequeños y fáciles de manejar.

.:Ejemplo 5.9. *Construya un programa que reciba un número entero positivo y que, a través de funciones determine: la cantidad de cifras que tiene el número, si este es o no un número compuesto y, a partir de este número ingresado, encontrar el n número triangular. Valide el ingreso del número con una función para que este sea positivo. Los resultados deben mostrarse mediante la invocación de un procedimiento.*

- *Un número compuesto es aquel que tiene más de dos divisores propios, es decir, existen otros números que lo dividen aparte del 1 y de si mismo.*
- *Un número triangular es un número con el que se puede formar un triángulo equilátero utilizando sus unidades. Por ejemplo, el número 6 es un número triangular, así como el 10. Para determinar el n número triangular se emplea la fórmula:* $T(n) = n * (n + 1)/2$

Análisis del problema:

- **Resultados esperados:** mostrar la cantidad de cifras que tiene el número ingresado, si este número es o no compuesto y el `n` número triangular.
- **Datos disponibles:** se conoce el número ingresado.

- **Proceso:** Utilizando una primera función, se valida el ingreso del número para que sea positivo. A continuación, se invoca una función que determine la cantidad de cifras que posee el número ingresado. Así mismo, se invoca luego otra función que permite determinar si el número es o no compuesto. Posteriormente se invoca una tercera función que, a partir del número ingresado determina el n número triangular. Los resultados arrojados por estas tres funciones se muestran invocando un procedimiento tres veces.
- **Variables requeridas:** Internos al procedimiento `main` se necesitan:

  - `numero`: corresponde al número ingresado por el usuario.
  - `cantidadCifras`: almacena la cantidad de cifras del número.
  - `nTriangular`: almacena el enésimo número triangular.
  - `esCompuesto`: almacena si el número es o no compuesto.

  Internos a la función `validarNumero` se necesitan:

  - Variables locales:
    - `n`: valor del número ingresado.

  Internos a la función `contarCifras` se necesitan:

  - Parámetros:
    - `n`: valor del número que tiene las cifras.
  - Variables locales:
    - `contador`: cuenta el número de cifras.

  Internos a la función `determinarCompuesto` se necesitan:

  - Parámetros:
    - `n`: valor del número objeto del cálculo.
  - Variables locales:
    - `esComp`: almacena si el número es compuesto o no.
    - `i`: variable de control del ciclo.
    - `contador`: cuenta las divisiones exactas.

Internos a la función `obtenerTriangular` se necesitan:

- Parámetros:
  - `n`: valor del número objeto de cálculo.
- Variables locales:
  - `num`: almacena el resultado de la fórmula.

Internos al procedimiento `mostrarResultados` se necesitan:

- Parámetros:
  - `mensaje`: cadena con el mensaje que se quiere mostrar.
  - `n`: resultado del cálculo realizado.

De acuerdo al análisis planteado, se propone el Programa 5.12.

Programa 5.12: CompuestoTriangular

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

int  validarNumero       ( );
int  contarCifras        ( int n );
bool determinarCompuesto ( int n );
int  obtenerTriangular   ( int n );
void mostrarResultados   ( char *mensaje, int n );

int main()
{
  int  numero, cantidadCifras, nTriangular;
  bool esCompuesto;

  numero         = validarNumero ( );
  cantidadCifras = contarCifras ( numero );
  esCompuesto    = determinarCompuesto ( numero );
  nTriangular    = obtenerTriangular ( numero );

  mostrarResultados ( "Número de cifras: ", cantidadCifras);

  if ( esCompuesto )
  {
      mostrarResultados("ES compuesto ", numero );
  }
  else
  {
      mostrarResultados("NO es compuesto ", numero );
  }

```

```
31    mostrarResultados(" El enésimo número triangular es: ",
          nTriangular );
32
33    return 0;
34  }
35
36  int validarNumero( )
37  {
38    int n;
39
40    do
41    {
42        printf( "Ingrese un número positivo: " );
43        scanf( "%d", &n );
44
45        if( n <= 0 )
46        {
47           printf("El número no es positivo. \n");
48        }
49    }while( n <= 0 );
50
51    return n;
52  }
53
54
55  int contarCifras( int n )
56  {
57    int contador;
58
59    contador = 0;
60
61    while ( n > 0 )
62    {
63      n = n / 10;
64      contador++;
65    }
66
67    return contador;
68  }
69
70  bool determinarCompuesto( int n )
71  {
72    bool esComp;
73    int  i, contador;
74
75    esComp   = false;
76    contador = 0;
77
78
```

```
79    for( i = 1 ; i <= n ; i++ )
80    {
81      if ( n % i == 0 )
82      {
83        contador++;
84      }
85    }
86
87    if ( contador > 2 )
88    {
89      esComp = true;
90    }
91
92    return esComp;
93  }
94
95  int obtenerTriangular( int n )
96  {
97    int num;
98
99    num = ( n * ( n + 1 ) ) / 2;
100
101   return num;
102 }
103
104 void mostrarResultados( char *mensaje, int n )
105 {
106   printf( "%s %d \n\n", mensaje, n );
107 }
```

Explicación del programa:

En este programa, se puede observar cómo se declaran las funciones y el procedimiento en la sección de encabezados, lo que hará que el preprocesador de Lenguaje C cargue estas instrucciones y el programa sea más veloz.

Dentro de la función `main`, se declaran las variables requeridas que recibirán los resultados retornados por las funciones y que se mostrarán a través del procedimiento `mostrarResultados`.

La función `validarNumero` se utiliza para ingresar el número con el que se va a trabajar en el resto del programa y garantizar que este sea positivo. Usa principalmente un ciclo `do..while` para realizar la validación.

La función `contarCifras` se encarga de hacer la cuenta de las cifras que posee el número que es pasado como parámetro. En ella se emplea un

ciclo `while` con el que se hacen divisiones sucesivas que, en cada iteración va llevando la cuenta de las cifras del número.

La función `determinarCompuesto` permite saber si el número que se pasa como parámetro es un número compuesto. Esto lo lleva a cabo realizando divisiones al número por sus antecesores y contando las divisiones exactas. En caso de que hayan más de dos divisiones exactas, se concluye que el número sí es compuesto. Esta función retorna un resultado lógico, es decir, un `true` un `false`. Es importante recordar que, para declarar variables y funciones lógicas, en Lenguaje C se emplea la biblioteca `stdbool.h` y el tipo de datos será `bool`.

La función `obtenerTriangular` aplica la fórmula vista en el análisis del problema y con ella encuentra el enésismo número triangular pasado como parámetro.

El procedimiento `mostrarResultados` tiene dos parámetros: el mensaje que se desea mostrar y el número obtenido por las funciones anteriores. Esta función se reutiliza dos veces para mostrar los resultados esperados del programa.

5.4. Temas complementarios para profundizar

Esta sección pretende dejar en el lector una semilla para que continúe investigando sobre todas las capacidades del Lenguaje C y así pueda hacer uso de todo el potencial que ofrece el lenguaje en sus futuros proyectos. A continuación se presentan ejemplos sobre algunos temas que están por fuera del alcance del libro, pero que se espera sirvan como motivación para el lector:

- Uso de enumeraciones.
- Uso de variables estáticas.
- Punteros a procedimientos / funciones.
- Parámetros variables en procedimientos / funciones.

Adicional a estos temas, existen otros temas que no son ilustrados en el libro, como por ejemplo el uso de las palabras reservadas: `auto`, `continue`, `extern`, `register`, `union`, `volatile`, así como temas más avanzados, como por ejemplo: la creación de interfaces gráficas, acceso a dispositivos hardware, uso de paralelismo y recursividad, estructuras

de datos, comunicación entre procesos de un mismo sistema y remotos; entre muchos otros temas que pueden ser desarrollados con el Lenguaje C, una vez que el lector ya tenga cierto dominio del lenguaje. Nuevamente se motiva al lector a continuar con el aprendizaje del lenguaje una vez adquiera cierto dominio del mismo.

5.4.1 Uso de enumeraciones

En muchas ocasiones, aparece la necesidad de usar una variable que tome un valor entre varios valores preestablecidos, como por ejemplo, con el uso de las variables lógicas que pueden tomar sólo dos valores: verdadero o falso. Para lograr esto, se puede hacer uso de constantes (usando `const` o `define` si son constantes simbólicas), tal y como fue estudiado en el primer capítulo.

El problema de este enfoque es que las variables pueden tomar valores no considerados, por ejemplo, si se asume que 1 es de verdadero y 0 es falso, ¿cómo se interpreta si se asigna algún otro valor, como el 3? Bajo este enfoque, el Lenguaje C no hace ninguna verificación. Las enumeraciones son empleadas para resolver este problema.

.:Ejemplo 5.10. *Crear un programa que permita almacenar e imprimir el estado civil (soltero, casado, unión libre, otro) de dos empleados.*

```
1  #include <stdio.h>
2
3  typedef enum
4     { SOLTERO, CASADO, UNION_LIBRE, OTRO } EstadoCivil;
5
6  int main ()
7  {
8      EstadoCivil empleado1, empleado2;
9
10     empleado1 = CASADO;
11     empleado2 = UNION_LIBRE;
12
13     printf ( "Estado civil: %d\n", empleado1 );
14     printf ( "Estado civil: %d\n", empleado2 );
15
16     return 0;
17 }
```

Ejecución

```
Estado civil: 1
Estado civil: 2
```

Lo primero que se debe observar es que se ha creado un nuevo tipo de dato (enumeración) llamada `EstadoCivil` que acepta solo los valores allí indicados.

```
3 typedef enum
4 { SOLTERO, CASADO, UNION_LIBRE, OTRO } EstadoCivil;
```

Este nuevo tipo de dato se puede emplear para indicar como valor de retorno de una función, por ejemplo:

```
  EstadoCivil obtenerEstadoCivilEmpleado ( int idEmpleado );
```

Es de aclarar que el Lenguaje C asigna automáticamente los enteros de 0, 1, 2, ... a cada uno de los valores enumerados, salvo que el programador defina explícitamente otros valores. Esto quiere decir que para el lenguaje, `CASADO` equivale a 1 y es así como será almacenado, en lugar de almacenar la palabra "CASADO", esto ahorra memoria, entre otras ventajas como por ejemplo el desempeño.

Como internamente las enumeraciones son números, es posible asignar por ejemplo, el valor 30, el cual claramente no tiene significado; es por eso que al momento de programar, se debe emplear el nombre del valor en lugar de emplear el equivalente numérico, así, si por error se intenta escribir:

```
10 empleado1 = CASADA;
```

Compilación

```
  static.c:10:17:
  error: use of undeclared identifier 'CASADA'; did you mean '
     CASADO'?
  empleado1 = CASADA;
              ^~~~~~
              CASADO
  static.c:4:16: note: 'CASADO' declared here
  { SOLTERO, CASADO, UNION_LIBRE, OTRO } EstadoCivil;
             ^
  1 error generated.
  Ha fallado la compilación.
```

El Lenguaje C producirá un error al momento de compilar el programa, impidiendo que se asigne un estado no considerado en la lista válida, que para el caso es: `CASADO`.

5.4.2 Uso de variables estáticas

Como ya fue explicado en este mismo capítulo, las variables creadas dentro de un procedimiento / función son de ámbito local, es decir, que una vez que el procedimiento / función termine, dichas variables desaparecen y por tanto, los valores que fueron asignados a ellas se pierden.

Cuando una variable es marcada como estática, el compilador del Lenguaje Conserva el valor de la variable para un uso futuro cuando el procedimiento / función sea nuevamente ejecutado. Estas variables requieren de un valor inicial que se asigna solo en la primera invocación.

.:Ejemplo 5.11. *Crear un programa que permita generar un consecutivo autoincrementable, por medio de una función. Es decir, cada vez que se invoque la función, se debe obtener el siguiente consecutivo a partir del valor anterior y partiendo desde 1.*

```
1  #include <stdio.h>
2
3  int obtenerConsecutivo ( );
4
5  int main ()
6  {
7    int i;
8
9    for ( i = 0 ; i < 5 ; i++ )
10   {
11     printf ( "Consecutivo: %d\n", obtenerConsecutivo(  ) );
12   }
13
14   return 0;
15 }
16
17 int obtenerConsecutivo ( )
18 {
19   static int valor = 0;
20
21   valor++;
22
23   return valor;
24 }
```

Ejecución

```
Consecutivo: 1
Consecutivo: 2
Consecutivo: 3
Consecutivo: 4
Consecutivo: 5
```

Si se omitirá el modificador `static`, el resultado sería:

```
Consecutivo: 1
Consecutivo: 1
Consecutivo: 1
Consecutivo: 1
Consecutivo: 1
```

Esto debido a que en cada invocación, la variable tomaría nuevamente el valor de 0, para luego incrementarlo en una unidad, obteniendo siempre 1.

5.4.3 Punteros a procedimientos / funciones

Así como las variables, las funciones también tienen asociadas una dirección de memoria, la cual se puede enviar como el valor de un argumento a un procedimiento / función, para que internamente se pueda invocar a la función de la dirección de memoria indicada.

.:Ejemplo 5.12. *Crear un programa que permita aplicar la función de `sumar` o `restar` a dos valores enteros ingresados por el usuario. Tanto la función a utilizar como los valores, deben ser enviados como argumentos a una tercera función que los utilizará para obtener el resultado correspondiente. Independiente de la función aplicada, al resultado obtenido se le debe multiplicar por dos.*

```
1  #include <stdio.h>
2
3  int sumar  ( int a, int b );
4  int restar ( int a, int b );
5
6  int operar ( int (*funcion)(), int a, int b );
7
8  int main()
9  {
10     int x, y, suma, resta;
11
12     printf( "Ingrese x: " );
```

```
13      scanf( "%d", &x );
14
15      printf( "Ingrese y: " );
16      scanf( "%d", &y );
17
18      suma  = operar( sumar,  x, y );
19      resta = operar( restar, x, y );
20
21      printf ( "Suma  : %5d\n", suma );
22      printf ( "Restar: %5d\n", resta  );
23  }
24
25  int operar ( int (*funcion)(), int a, int b )
26  {
27     return funcion ( a , b ) * 2;
28  }
29
30  int sumar ( int a, int b )
31  {
32    return a + b;
33  }
34
35  int restar ( int a, int b )
36  {
37    return a - b;
38  }
```

Ejecución

```
Ingrese x: 9
Ingrese y: 4
Suma  :    26
Restar:    10
```

Lo realmente nuevo en el ejemplo, es el uso de un puntero a una función:

```
6  int operar ( int (*funcion)(), int a, int b );
```

y la forma como se utiliza el puntero para invocar la función a la que referencia en el momento de su uso. Para el caso de la línea 18, `funcion` referencia `sumar`, mientras que en la línea 19, `funcion` referencia `restar`. Así, independiente de la función que se ejecute, al resultado obtenido se le multiplica por dos; retornando finalmente el valor obtenido.

```
25  int operar ( int (*funcion)(), int a, int b )
26  {
27    return funcion ( a , b ) * 2;
28  }
```

5.4.4 Parámetros variables en procedimientos / funciones

El último tema complementario del libro, consiste en declarar funciones con un número arbitrario de parámetros, tal y como sucede por ejemplo, con la función `printf` de la biblioteca `stdio.h`.

Para ilustrar esto, se han creado dos ejemplo:

- Imprimir una lista arbitraria de nombres dados como argumentos.
- Obtener el menor valor entero que se ha enviado como argumento de una función.

.:Ejemplo 5.13. *Crear un programa que permita imprimir cualquier cantidad de nombres ingresados manualmente como argumentos de un procedimiento.*

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  void imprimirNombre ( int n, ... );
5
6  int main ()
7  {
8    imprimirNombre ( 1, "Robinson" );
9    imprimirNombre ( 2, "Robinson", "Orlando" );
10   imprimirNombre ( 3, "Robinson", "Orlando", "Julian" );
11 }
12
13 void imprimirNombre ( int n, ... )
14 {
15   va_list parametros;
16   int i;
17
18   printf ("Nombre: ");
19
20   va_start( parametros, n );
21
22   for ( i = 0 ; i < n ; i++ )
23   {
24     printf ( "%s ", va_arg( parametros, char * ) );
25   }
26
27   printf ( "\n" );
28
29   va_end( parametros );
30 }
```

Ejecución

```
Nombre: Robinson
Nombre: Robinson Orlando
Nombre: Robinson Orlando Julian
```

Para poder crear este tipo de procedimientos / funciones con parámetros variables, es necesario usar la biblioteca `stdarg.h` y emplear tres puntos (...) dentro de la definición de los parámetros para indicarle al compilador del Lenguaje C que será un función de este tipo.

Dentro del procedimiento / función se debe emplear una variable de tipo `va_list`, así como utilizar las macros: `va_start`, `va_arg` y `va_end` tal y como se ilustra en el ejemplo.

Por su parte, `va_start`, permite el acceso a los `n` parámetros; `va_arg` obtiene el siguiente parámetro; y finalmente `va_end` finaliza la trasferencia de los parámetros.

Un punto a resaltar es que cuando se emplea `va_arg`, se debe indicar el tipo de dato de los parámetros que se ingresarán, para el primer ejemplo es `char *` debido a que son direcciones de memoria a nombres.

```
24 valor = va_arg( parametros, char * );
```

En el siguiente ejemplo se ilustra el uso con un tipo de dato entero.

```
23 valor = va_arg( parametros, int );
```

.:Ejemplo 5.14. *Crear un programa que permita determinar el menor valor entero que se ha enviado como argumento de una función. La función puede recibir cualquier cantidad arbitraria de argumentos.*

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int obtenerMenor ( int n, ... );
5
6  int main ()
7  {
8   printf ( "El menor: %d\n", obtenerMenor ( 1, 73  ));
9   printf ( "El menor: %d\n", obtenerMenor ( 2, 73, 31 ));
10  printf ( "El menor: %d\n", obtenerMenor ( 3, 73, 5, 31 ));
11 }
12
13 int obtenerMenor ( int n, ... )
14 {
15   va_list parametros;
```

```
16    int i, menor, valor;
17
18    va_start( parametros, n );
19
20    menor = va_arg( parametros, int );
21    for ( i = 0 ; i < n - 1 ; i++ )
22    {
23      valor = va_arg( parametros, int );
24
25      if ( valor < menor )
26      {
27        menor = valor;
28      }
29    }
30
31    va_end( parametros );
32
33    return menor;
34  }
```

Ejecución

```
El menor: 73
El menor: 31
El menor: 5
```

Buenas prácticas:

Dejar lo más claro posible cada elemento del programa:

- Documente cuando sea conveniente y sin abusar de este recurso.
- Defina cuál es la responsabilidad de cada función / procedimiento. Evite que ellas tengan más de una responsabilidad.
- Use nombres para las funciones / procedimiento acordes con la responsabilidad asociada.
- Evite exagerar en la cantidad de parámetros de un función / procedimiento necesita.
- Controle la longitud de todas y cada una de las funciones / procedimiento para evitar que superen una página, de ser el caso, delegue parte de la responsabilidad en otras funciones / procedimientos.

5.5. Creando una biblioteca

El Lenguaje C consta de un conjunto de bibliotecas internas que agrupan funciones, procedimientos y constantes, tal y como se ha podido experimentar en el transcurso de libro. El Lenguaje C también permite que el programador pueda crear sus propias bibliotecas, que le faciliten la creación de nuevos proyectos, sin la necesidad de comenzar cada vez desde "cero", es decir, empleando solo las bibliotecas propias del lenguaje.

Como una biblioteca es en realidad un conjunto de recursos que se pueden emplear dentro de un programa, entonces, la biblioteca requiere ser vinculada al programa que la está utilizando en el proceso de compilación. Esta vinculación se puede hacer directamente en la mayoría de los IDE (Entornos Integrados de Desarrollo) como **Geany**, o se puede realizar manualmente. Ambas formas son ilustradas en el Anexo A.

Para crear una biblioteca es importante separar las declaraciones (interfaces) de las funciones que ofrece, de la implementación de dichas interfaces. Las primeras se realizan en un archivo de extensión (`.h`), mientras que el segundo se realiza en un archivo tradicional (`.c`), en el cual, de ser necesario, se pueden incluir otras bibliotecas, tal y como si se tratara de un programa tradicional, con la excepción de no poseer una función principal (`main`).

5.5.1 Estructura general de un archivo cabecera (`.h`)

```
1 #ifndef PALABRA
2
3   #define PALABRA
4
5   // Declaración de las interfaces de las funciones y
6   // procedimientos de la bibliteca, al igual que
7   // las posibles constantes
8
9 #endif
```

En la estructura general, se recomienda utilizar **`#ifndef`** y **`#define`** para garantizar que, si el programador llegara incluir, por accidente, dos o más veces el archivo de cabecera, el compilador solo utilizará la primera e ignorará las demás. Para lograr esto, la primera instrucción es una "decisión especial" que se evalúa en el momento de la compilación y no en tiempo de ejecución; así, cuando el programador lo compila, el compilador decide si incluye o no este código.

Esta decisión en particular dice que se deben incluir las líneas siguientes, solo si **NO** está definida la palabra especificada; de no estar definida, indicaría que es la primera inclusión, por lo tanto, en la siguiente línea se declara dicha palabra (así en la próxima inclusión será falsa la decisión) y posteriormente, se declaran todas las interfaces de las funciones, procedimientos y/o constantes de las que conste la biblioteca.

Por ejemplo, suponga que desea crear una biblioteca con las funciones de las operaciones básicas de la aritmética (sumar, restar, multiplicar y dividir) sobre números enteros. Entonces, el archivo de cabecera debería quedar como se muestra a continuación, el cual se llamará `aritmetica.h`

```
#ifndef ARTIMETICA_ENTERA

  #define ARTIMETICA_ENTERA

  int sumar       ( int a, int b );
  int restar      ( int a, int b );
  int multiplicar ( int a, int b );
  int dividir     ( int a, int b );
#endif
```

Una vez defino el archivo de cabecera, se debe crear un archivo con una implementación de todas estas definiciones, tal y como si se tratara de un programa tradicional.

5.5.2 Implementación de un archivo cabecera (`.c`)

Se puede ver a continuación, que la implementación es un conjunto de procedimientos/funciones como los ya estudiados. Sin embargo, es importante aclarar que se realiza esta separación para facilitar la creación de diferentes implementaciones de la biblioteca, sin que ello afecte los programas que la están usando, debido a que cada nueva versión de la biblioteca DEBE respetar las interfaces definidas en la cabecera, interfaces que utilizan todos los programas que la emplean. El siguiente código que se denomina `aritmetica.c` ilustra lo mencionado anteriormente.

```
  int sumar ( int a, int b )
  {
     return a + b;
  }

  int restar ( int a, int b )
  {
     return a - b;
  }
```

```
10
11
12    int multiplicar ( int a, int b )
13    {
14       return a * b;
15    }
16
17    int dividir ( int a, int b )
18    {
19       return a / b;
20    }
```

Ahora solo resta utilizar la biblioteca.

5.5.3 Utilizando la biblioteca

El siguiente código presenta un ejemplo que hace uso de la biblioteca que anteriormente fue definida. Lo interesante en esto, es que a partir de este momento, no hay necesidad de re-escribir la implementación de las funciones: `sumar`, `restar`, `multiplicar` y `dividir`, en todos los programas que hagan uso de ellas, es decir, se usan de manera transparente, tal y como se usan las funciones propias del Lenguaje C, siempre y cuando se incluya el respectivo archivo de cabecera y se compile de manera apropiada (Ver Anexo A).

```
1  #include <stdio.h>
2  #include "aritmetica.h"
3
4  int main ()
5  {
6    int x, y;
7
8    printf("Ingrese el primer valor: ");
9    scanf( "%d", &x );
10
11   printf("Ingrese el segundo valor: ");
12   scanf( "%d", &y );
13
14   printf("La suma es         %d\n", sumar ( x, y ));
15   printf("La resta es       %d\n", restar ( x, y ) );
16   printf("La multiplicación es  %d\n", multiplicar ( x, y ));
17   printf("La diviisión es      %d\n", dividir ( x, y ));
18
19   return 0;
20 }
```

A continuación se presentan una serie de ejercicios para ser resueltos mediante programas que hagan uso de funciones / procedimientos. Cada ejercicio debe estar acompañado de su respectivo análisis y ejemplos ilustrativos de su funcionamiento. En algunos casos, el lector deberá investigar las fórmulas necesarias para llegar a la solución.

Actividad 5.1

1. Diseñe un programa con funciones y procedimientos que permita determinar el área de un hexágono.

2. Diseñe un programa con funciones y procedimientos que permita que un arquitecto pueda determinar fácilmente el área total de un parque infantil que esta diseñando. El arquitecto puede ajustar cada una de las longitudes de las zonas del parque para ver como el cambio afecta el área total. El parque está formado por: 3 zonas en forma de hexágono en donde los niños juegan con arena, 2 zonas circulares para reuniones y 4 zonas rectangulares para los juegos mecánicos.

3. Diseñe un programa con funciones y procedimientos que permita determinar las coordenadas de un triángulo si se conocen las longitudes de cada uno de los tres lados.

4. Diseñe un programa con funciones y procedimientos que indique el valor del descuento de un artículo el cual es del 5% solo si el artículo tiene un costo superior al $150.000.

5. Diseñe un programa con funciones y procedimientos que indique si la llave de un tanque de agua debe ser abierta o cerrada. El tanque debe estar siempre entre 250 y 450 litros.

6. Diseñe un programa con funciones y procedimientos que dado un número entero entre 0 y 20 diga si es o no un número primo.

   Recuerde que los números primeros menores o iguales a 20 son: $2, 3, 5, 7, 11, 13, 17, 19$.

7. Diseñe un programa con funciones y procedimientos que indique si un estudiante ganó o perdió un curso después de presentar los cinco trabajos asociados al mismo (Notas entre 0.0 y 5.0). Los trabajos tienen igual peso sobre la nota final y se gana el curso si la nota definitiva es superior a 3.5

8. Diseñe un programa con funciones y procedimientos que permita saber si una ecuación cuadrática tiene o no solución.

   Recuerde que una ecuación cuadrática se define como:

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

   Y se dice que tiene solución si el valor a calcular la raíz cuadrada es mayor o igual a cero y el valor de a es diferente de cero.

9. Diseñe un programa con funciones y procedimientos que indique si un número entero x se encuentra por dentro o por fuera el intervalo cerrado-cerrado $[minimoValor, maximoValor]$

   Por ejemplo: Si los valores mínimos y máximo son 3 y 7 respectivamente, el valor 5 está dentro, mientras que el valor de 8 está por fuera del intervalo.

10. Diseñe un programa con funciones y procedimientos que indique si un número entero x se encuentra por dentro o por fuera de tres intervalos abierto-abierto cuyo rangos no se interceptan entre sí y sus límites son ingresados por el usuario.

CAPÍTULO 6

VECTORES Y MATRICES

La función de un buen software es hacer que lo complejo aparente ser simple.

Grady Booch

Objetivos del capítulo:

- Conocer la forma como se declaran e inicializan vectores y matrices en Lenguaje C.
- Aprender a leer e imprimir los elementos almacenados en vectores y matrices.
- Construir programas que sean capaces de buscar elementos en vectores y matrices.
- Hacer operaciones con varios vectores y matrices.

En los capítulos previos se han utilizado variables que sólo almacenan un dato al mismo tiempo, lo que quiere decir que si la variable es reutilizada, el valor que había inicialmente se perderá. No obstante, los diversos lenguajes de programación proveen de variables que permiten almacenar varios datos del mismo tipo simultáneamente; estas variables reciben el nombre de arreglos. Lenguaje C también ofrece al programador esta posibilidad.

Imagine que es necesario almacenar las estaturas de 50 personas, las edades de los 30 estudiantes de un grupo o el peso de 40 pacientes de una clínica. Una alternativa para estos problemas consistiría en declarar 50 variables para almacenar las estaturas, 30 variables para las edades y 40 para los pesos, pero esto es totalmente inviable. En esta clase de problemas, cobran importancia las llamadas variables **Variables suscritas** que almacenan la ubicación en la memoria del computador (referencia) donde se crearon muchos espacios o celdas que almacenarán múltiples datos (llamado **Arreglo**).

De este modo, puede afirmarse que un arreglo es un conjunto de espacios o celdas de memoria usados para almacenar temporalmente muchos datos del mismo tipo. De esta manera, los lenguajes de programación, incluyendo C permiten almacenar muchos datos, por ejemplo las estaturas de 50 personas en una misma variable.

En programación hay varios tipos de arreglos, por ejemplo: arreglos unidimensionales también llamados **Vectores**, arreglos bidimensionales o **Matrices** y arreglos multidimensionales. Un vector es una fila de datos, una matriz es una tabla con filas y columnas y los arreglos multidimensionales tomarían otras formas.

Para acceder a un arreglo y a sus celdas es necesario declarar una variable suscrita con su respectivo tipo de dato que será el mismo para todas las celdas.

6.1. Vectores

Un vector o arreglo unidimensional es un conjunto finito de celdas que almacenan datos del mismo tipo. Con propósitos didácticos, imagine un arreglo unidimensional como una fila en la que cada celda corresponde a un espacio de memoria que almacena un dato. Cada celda o espacio de memoria se identifica con un número llamado índice. La Figura 6.1 ilustra la forma en que se vería un vector de 10 posiciones:

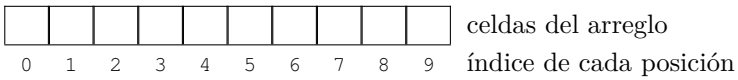


Figura 6.1: Diagrama de un Vector

En Lenguaje C los índices de un arreglo inician en cero. El último número índice representa a la última celda, y es menor en uno al tamaño del arreglo, es decir, el número de celdas que tiene el arreglo.

6.1.1 Declaración de un vector

Para poder utilizar un vector (o arreglo en general) es necesario la declaración de una variable suscrita que lo referencie. Una variable suscrita se diferencia de las demás por tener en su declaración un juego de corchetes [ ].

```
int    edadEstudiante [ n ];
double estaturaPersona[ n ];
char   nombreEmpleado [ n ][ longitud ];
char   letrasDocumento[ n ];
```

Es importante aclarar que, luego de declarar la variable suscrita, se debe especificar el tamaño del vector; esto permitirá determinar la cantidad de celdas, casillas o espacios de memoria que tendrá el vector y el tipo de dato. Lo anterior se puede hacer de dos maneras: usando la función llamada `dimensionarI()`, `dimensionarD()`, `dimensionarC()` y `dimensionarS()` (estos elementos son propios del libro y para que puedan ser utilizados se requiere que el programa incluya la biblioteca `#include <stdlib.h>`, cada lenguaje tiene su forma particular de crear los arreglos); o en el momento de declarar la variable suscrita.

Crear un arreglo de enteros (`int`):

```
#define dimensionarI(n) (int *) malloc(sizeof(int)*n)
```

La función `malloc` de la biblioteca `stdlib.h` permite reservar una cantidad determinada de Bytes en la memoria de computador, en este caso el tamaño del tipo del arreglo (**`sizeof(int)`**), multiplicado por la cantidad de elementos (`n`). Un programador puede reservar memoria para diversas cosas, pero de momento, solo se empleará para crear arreglos y matrices. El resultado de la función `malloc` debe ser convertido al tipo de puntero necesario, para el ejemplo, un puntero a un entero `(int *)`.

Si por el contrario, se desea crear un arreglo para almacenar números reales (`double`), se debe realizar el ajuste correspondiente:

```
#define dimensionarD(n) (double *) malloc(sizeof(double)*n)
```

Crear un arreglo de caracteres (`char`):

```
#define dimensionarC(n) (char *) malloc(sizeof(char)*n)
```

Crear un vector de cadenas (vector de vectores tipo (`char`):

```
char **dimensionarS ( int n, int longitud )
{
  char **vector;

  vector =  (char **) malloc ( sizeof(char*) * n);

  for ( int i = 0 ; i < n ; i++ )
  {
    vector[i] = dimensionarC ( longitud );
  }

  return vector;
}
```

Observe que, para este último caso, es necesario reservar memoria para el arreglo principal (tamaño `n`) y luego por cada posición, de allí el ciclo, se debe reservar memoria para cada una de las cadenas de la longitud indicada.

Independiente del tipo de dato, siempre es indispensable eliminar (liberar) el vector creado en la memoria del computador.

```
  free ( nombreVariableVector );
```

Solo para un vector de cadenas (`char **`);

```
void freeS ( char **vector, int n )
{
  for ( int i = 0 ; i < n ; i++ )
  {
    free ( vector [ i ] );
  }

  free( vector );
}
```

Creación de arreglos con `dimensionarI`:

1. Se declara una variable suscrita llamada `edadEstudiante` y se le da un tamaño de 6 posiciones para datos de tipo `int`. (Ver Figura 6.2).

```
int *edadEstudiante;

edadEstudiante = dimensionarI( 6 );
```

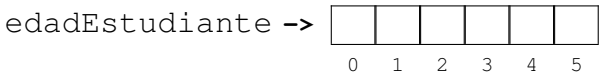


Figura 6.2: Vector `edadEstudiante`

Después de usarlo se procede a liberar la memoria.

```
free ( edadEstudiante );
```

2. Se declara una variable suscrita llamada `estaturaPersona` y se le da un tamaño de 4 posiciones para datos de tipo `double`. (Ver Figura 6.3).

```
double *estaturaPersona;

estaturaPersona = dimensionarD( 4 );
```

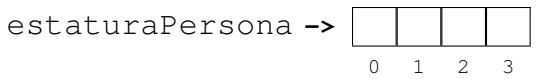


Figura 6.3: Vector `estaturaPersona`

Después de usarlo se procede a liberar la memoria.

```
free ( estaturaPersona );
```

3. Se declara una variable suscrita llamada `letrasDocumento` y se le da un tamaño de 50 posiciones con datos de tipo `char`. (Ver Figura 6.4).

```
char *letrasDocumento;

letrasDocumento = dimensionarC( 50 )
```

Después de usarlo se procede a liberar la memoria.

```
free ( letrasDocumento );
```

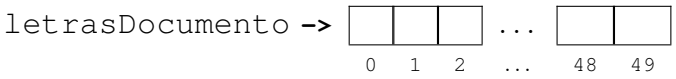


Figura 6.4: Vector `letrasDocumento`

4. Se declara una variable suscrita llamada `nombreEmpleado` y se le da un tamaño de 3 posiciones para caracteres.(Ver Figura 6.5).

```
char **nombreEmpleado;

nombreEmpleado = dimensionarS( 3 )
```

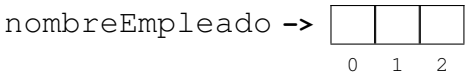


Figura 6.5: Vector `nombreEmpleado`

Después de usarlo se procede a liberar la memoria.

```
freeS ( nombreEmpleado );
```

La función `dimensionar()` no requiere que se envíe una constante numérica como argumento, también se puede usar el contenido de una variable o constante de tipo `int`. Por ejemplo:

```
int *edadEstudiante;
int  cantidadPersonas

cantidadPersonas = 20
edadEstudiante   = dimensionarI( cantidadPersonas );
```

Aclaración:

Aunque se puede declarar una variable en cualquier parte del programa, sin ser un error de lógica, lo ideal sería realizar siempre las declaraciones al inicio del algoritmo o de los procedimientos o funciones.

Algunos ejemplos de creación de vectores al declarar la variable:

1. Se declara una variable suscrita llamada `semestreEstudiante` con un tamaño de 35 posiciones que almacena datos de tipo `int`. (Ver Figura 6.6).

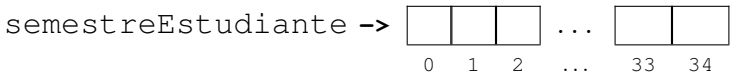


Figura 6.6: Vector `semestreEstudiante`

```
int semestreEstudiante [ 35 ];
```

2. Se declara una variable suscrita denominada `correoElectronico` con un tamaño de 30 posiciones que almacena datos de tipo cadena, con una longitud de 20 caracteres (posiciones). (Ver Figura 6.7).

```
char correoElectronico [ 30 ][ 20 ];
```

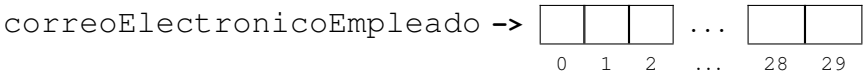


Figura 6.7: Vector `correoElectronicoEmpleado`

3. Se declara una variable suscrita denominada `sueldoPersona` con una longitud de 100 posiciones para datos de tipo `double`. (Ver Figura 6.8).

```
float sueldoPersona [ 100 ]
```

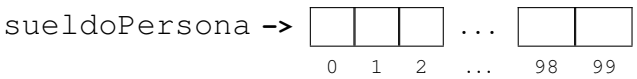


Figura 6.8: Vector `sueldoPersona`

Observe cómo, en los casos anteriores se dió tamaño a cada una de las variables suscritas declaradas.

Buena práctica:

Siempre que se vaya a leer el contenido de un vector, es fundamental estar seguro de que este posee datos almacenados. Lenguaje C no asigna ningún valor inicial a los arreglos, por tanto, si se lee un vector al que no se le hayan ingresados datos, se pueden obtener resultados imprevistos.

Para finalizar esta explicación, es importante aclarar que, una vez se haya declarado un arreglo, ya se puede acceder a él para almacenar datos o, realizar alguna operación con ellos.

6.1.2 Almacenamiento de datos en un vector

El almacenamiento de los datos se realiza en cada una de las celdas que conforman el vector, identificando dichas celdas a través de su índice. Imagine que tiene un vector ya declarado de tipo `int`, llamado `numero` en el que va a almacenar 5 números; el almacenamiento se haría de la siguiente manera:

```
int numero[ 5 ];

numero[ 0 ] = 4;
numero[ 1 ] = 2;
numero[ 2 ] = 15;
numero[ 3 ] = 8;
numero[ 4 ] = 3;
```

Estas asignaciones ubican en la primera posición del vector `numero` el número 4, en la segunda posición un 2, en la tercera un 15, en la cuarta un 8 y en la quinta un 3. De manera gráfica, el arreglo luciría como en la Figura 6.9.

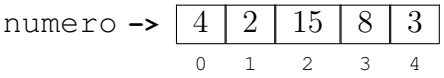


Figura 6.9: Ejemplo almacenamiento en un vector

También puede almacenar en una celda de un vector un dato ya almacenado en una variable, teniendo en cuenta que la variable y el vector deben ser del mismo tipo de dato. Por ejemplo, suponga que tiene una variable entera denominada `edad` en la que se ha almacenado un 20. El contenido de esta variable puede copiarse en una posición cualquiera del vector de la siguiente forma:

```
int numero[ 5 ], edad;

edad = 20;
numero[ 4 ] = edad;
```

Por medio de esta instrucción el valor almacenado en la variable `edad` se copie en la posición 4 del vector.

Al momento de la declaración del vector, se pueden almacenar varios datos, de la siguiente forma:

```
int numero[ ] = {7, 21, 93, 48, 5};
```

Con la anterior instrucción, el número 7 se almacenará en la primera posición del vector `numero` (índice 0), 21 en la posición 2 (índice 1), 93 en la posición 3 (índice 2), 48 en la posición 4 y 5 en la posición 4 de este arreglo (índices 3 y 4). (Ver Figura 6.10).

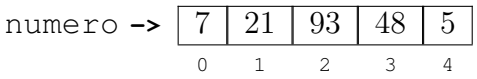


Figura 6.10: Ejemplo de asignación directa

Ahora bien, el usuario puede ingresar datos y almacenarlos en un vector. Aunque puede realizar una lectura manual para cada posición, mediante un ciclo se automatiza el proceso de almacenamiento para todas las celdas. Imagine que se van a leer, por ejemplo, 20 números enteros:

```
#include <stdio.h>
#define N 20

int main ()
{
  int i;
  int numero[ N ];

  for ( i = 0 ; i < N ; i++ )
  {
      printf ( "Ingrese el número %d: " , (i + 1) );
      scanf ( "%d", &numero[ i ] );
  }

  for ( i = 0 ; i < N ; i++ )
  {
      printf ( "%d\n", numero[ i ] );
  }

  return 0;
}
```

En este segmento de programa, la variable `i` corresponde al índice que permitirá acceder a las diversas posiciones del arreglo. El ciclo `for` iterará 20 veces, lo que coincide con el tamaño del arreglo. Con `printf()` se solicitará al usuario el ingreso de los números. La variable `i` ubicada al final de la función `printf()`, luego del mensaje, que está en paréntesis y es sumada con 1, permitirá saber el número que se va ingresando. Con `scanf()` se capturará el número ingresado y se almacenará en la posición del vector `numero` representada por el índice.

6.1.3 Recuperación de datos almacenados en un vector

Si se conoce la posición del vector en que se encuentra el dato que se quiere recuperar, basta con usar una instrucción que lo recupere. Por ejemplo, imagine que necesita el dato almacenado en la posición 3 del vector `numero`. (Ver Figura 6.11).

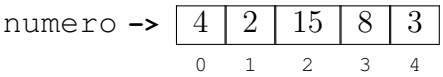


Figura 6.11: Ejemplo recuperación de datos

Con la siguiente instrucción se puede mostrar:

```
printf( "El tercer elemento es %d" , numero[ 3 ] );
```

Al ejecutar esta instrucción y, suponiendo que los datos almacenados son los de la figura anterior, el dato que se mostrará es el número 15.

Este dato puede almacenarse en otra variable, por supuesto, del mismo tipo de dato que el arreglo de donde proviene el dato. Imagine que el dato de la última posición del arreglo que se observa a continuación se va acopiar en la variable `n`:

```
int n, numero[] = {7, 21, 93, 5, 48};

n = numero[ 4 ];
```

Al ejecutar esta instrucción, el valor 48 se copiará a la variable `n`. Ahora, es importante aclarar que el 48 no se borrará de la posición 4 de `numero`.

En muchos problemas de programación, es necesario recuperar todos los datos almacenados en el arreglo. En estas situaciones, hay que recorrer el vector utilizando un ciclo, que va de la primera hasta la última posición. Suponga que se necesita sumar los valores almacenados en el arreglo `numero` y mostrar al usuario este resultado. Lo anterior se puede lograr con el segmento de código siguiente:

```
#include <stdio.h>

int main ()
{
  int numero[  ] = {7, 21, 93, 48, 5};
  int i, suma;
```

```
    suma = 0;
    for ( i = 0 ; i < 5 ; i++ )
    {
        suma = suma + numero [ i ];
    }

    printf ( "La suma de los elementos es: %d", suma );

    return 0;
}
```

Note algunos aspectos importantes en la porción de código que se acaba de escribir:

- La variable `suma` es de tipo `int` lo que coincide con el tipo de datos del arreglo `numero`.
- `suma` se inicializa en 0, con el propósito de almacenar en ella la suma de los números que están almacenados en el arreglo.
- El ciclo `for` tiene la variable `i` como contador e índice, y recorre el vector desde la posición 0 hasta el tamaño menos uno (4).
- La instrucción dentro del ciclo va sumando el valor de la posición actual de `numero` con los valores de las posiciones anteriores, es decir, esta variable es un acumulador.

  ```
  suma = suma + numero[ i ]
  ```

- El total de la `suma` es mostrado al final del programa mediante la función `printf()`.

De aquí en adelante, se mostrarán programas de ejemplo que utilizan arreglos unidimensionales o vectores. Los primeros programas tendrán una estructura secuencial; posteriormente, los programas que siguen, harán uso de funciones y procedimientos.

.:Ejemplo 6.1. *Diseñe un programa en Lenguaje C que almacene en un vector de tamaño 4, números enteros que estén entre 10 y 20 y que, posteriormente determine cuántas veces quedó almacenado en el vector el número 14.*

Análisis del problema:

- **Resultados esperados:** determinar cuántas veces quedó almacenado en el vector el número 14.

- **Datos disponibles:** los cuatro números enteros que se van a ingresar sabiendo que deben estar en el intervalo 10 - 20.
- **Proceso:** se ingresan los cuatro datos y se almacenan en el vector, luego este se recorre posición por posición buscando el número 14, desde la primera posición hasta la última; en el momento en que haya coincidencia, se incrementa un contador que, se ha inicializado en cero antes del ciclo que hace la búsqueda. Finalmente, se muestra la cantidad de veces que se encontró el número 14.
- **Variables requeridas:**
  - `numero`: arreglo o vector que almacenará los números que el usuario ingrese.
  - `i`: variable que controla el ciclo y que representa el índice para acceder a las posiciones del arreglo.
  - `contador14`: contador que guarda el número de veces en que se repite el número 14 en el arreglo.
  - `repetir`: variable de tipo bandera que sirve para saber si es necesario solicitar nuevamente un dato según cumpla o no cumpla con el criterio de estar entre 10 y 20.

De acuerdo al análisis planteado, se propone el programa 6.1.

Programa 6.1: Numero14

```
1  #include <stdio.h>
2
3  #define VERDADERO 1
4  #define FALSO     0
5
6  int main()
7  {
8      const int MAX = 4;
9
10     int numero[ MAX ], contador14, i;
11     int repetir;
12
13     for( i = 0 ; i < MAX ; i++ )
14     {
15        do
16        {
17           printf( "Ingrese el número %i: ", i );
18           scanf( "%i", &numero[ i ] );
19
20           repetir = FALSO;
```

```
21            if( numero[ i ] < 10 || numero[ i ] > 20 )
22            {
23               printf( "Este número no es válido\n\n" );
24               repetir = VERDADERO;
25            }
26
27         } while( repetir == VERDADERO );
28      }
29
30      contador14 = 0;
31      for( i = 0 ; i < MAX ; i++ )
32      {
33         if ( numero[ i ] == 14 )
34         {
35             contador14++;
36         }
37      }
38
39      printf( "El número 14 está %d  veces", contador14 );
40
41      return 0;
42  }
```

Al ejecutar el programa:

```
Ingrese el número 0: 14
Ingrese el número 1: 2
Este número no es válido
Ingrese el número 2: 18
Ingrese el número 3: 20
Ingrese el número 4: 14
El número 14 está 2 veces
```

Explicación del programa:

Luego de declarar las variables y constante (MAX) necesarias (líneas 8 a 11), se pasa a ingresar los números y almacenarlos en el arreglo. Observe cómo se utiliza la constante tanto para dar tamaño al arreglo como para recorrerlo en los ciclos.

```
8      const int MAX = 4;
9
10     int numero[ MAX ], contador14, i;
11     int repetir;
```

El primer ciclo for, permite recorrer el vector a partir de la primera posición (posición cero) por medio de la variable i llegando hasta la última posición del vector (que sería una posición antes de lo estipulado en la constante MAX), mientras se van ingresando los números al arreglo.

Dentro de este primer ciclo `for` hay un ciclo `do-while`, que cumple la función de validar los números ingresados para que se encuentren en el intervalo 10 al 20. Si esto no se cumple, este ciclo se vuelve a repetir, solicitando nuevamente un número que sí cumpla con lo requerido por el programa.

```
15          do
16          {
17              printf( "Ingrese el número %i: ", i );
18              scanf( "%i", &numero[ i ] );
19
20              repetir = FALSO;
21              if( numero[ i ] < 10 || numero[ i ] > 20 )
22              {
23                  printf( "Este número no es válido\n\n" );
24                  repetir = VERDADERO;
25              }
26
27          } while( repetir == VERDADERO );
```

Analice el papel que cumple la instrucción `if`, que se usa para mostrar un mensaje si el número ingresado no es parte del intervalo especificado, y cambia el valor de la variable `repetir` de falso a verdadero. Esta variable bandera controla la salida del ciclo `do-while` o, por el contrario la ejecución de una nueva iteración.

Después de este primer ciclo `for`, la variable `contador14` se inicializa en cero, pues aún no se ha encontrado ningún número 14; a continuación, hay un segundo ciclo `for` que recorre nuevamente el arreglo de números y como parte del cuerpo de este ciclo, con una instrucción `if`, se indaga si el dato contenido en cada celda corresponde a un 14, para proceder a contarlo.

```
30      contador14 = 0;
31      for( i = 0 ; i < MAX ; i++ )
32      {
33          if ( numero[ i ] == 14 )
34          {
35              contador14++;
36          }
37      }
```

Finalmente, la instrucción `printf` muestra la cantidad de 14 que están almacenados en el arreglo.

```
39      printf( "El número 14 está %d  veces", contador14 );
```

.:Ejemplo 6.2. *Construya un programa en Lenguaje C que almacene en un arreglo los nombres de un grupo de* `n` *personas y que, a continuación, busque la posición del arreglo en la que se almacenó el nombre de una persona que el usuario ingresa. Si dicho nombre no está en el arreglo, el programa deberá informarlo.*

Para este ejercicio, suponga que todos los nombres ingresados son diferentes.

Aclaración:

Con la implementación de este ejercicio se pretenden explicar varias instrucciones que permiten el manejo de cadenas de caracteres en Lenguaje C y, por supuesto, el manejo de punteros.

Análisis del problema:

- **Resultados esperados:** se debe mostrar la posición del arreglo en la que está el nombre buscado o un mensaje que informe que el nombre no está en el arreglo.
- **Datos disponibles:** se conocen el tamaño del arreglo y el nombre de la persona que se va a buscar.
- **Proceso:** se solicita el tamaño del arreglo, preguntando cuántos nombres va a almacenar el arreglo y, utilizando un ciclo, estos nombres son almacenados en el arreglo. Enseguida, se lleva a cabo una búsqueda secuencial desde la primera hasta la última celda del arreglo de nombres. A medida que el programa se ubica en cada posición, se pregunta si el nombre buscado es el mismo que hay en la celda, si esto ocurre, se encontró lo buscado y se termina la búsqueda, mostrando la posición en la que se halló el nombre buscado; si la comparación es falsa, el ciclo pasa a la siguiente celda y se compara nuevamente hasta llegar al final del arreglo. Si el programa termina de recorrer el arreglo y no encuentra el nombre buscado, se informará al usuario.
- **Variables requeridas:**
  - `tamanio`[1]: variable que almacena el tamaño del arreglo.
  - `arregloNombres`: arreglo usado para almacenar los nombres.

[1]por norma, en los identificadores se evita el uso de caracteres como tildes y eñes.

- `i`: variable índice para controlar el ciclo y ubicar cada celda del arreglo.
- `nombreBuscar`: se utiliza para almacenar el nombre que se se va a buscar.
- `encontrado`: variable de tipo bandera que indica si se encontró o no el nombre buscado en el arreglo.

Programa 6.2: NombrePersonas

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define Verdadero 1
6  #define Falso     0
7
8  #define dimensionarC(n) (char *) malloc(sizeof(char)*n)
9
10 char **dimensionarS ( int n, int longitud );
11 void freeS ( char **vector, int n );
12
13 int main ()
14 {
15   int tamanio, i;
16   char **arregloNombres, nombreBuscar[ 20 ];
17   int encontrado;
18
19   printf( "Ingrese el tamaño del arreglo: ");
20   scanf( "%d", &tamanio );
21
22   arregloNombres = dimensionarS( tamanio, 20 );
23
24   for( i = 0 ; i < tamanio ;i++ )
25   {
26     printf( "Ingrese el nombre persona %d:" , i );
27     fgets( arregloNombres[ i ], 20, stdin );
28   }
29
30   printf( "Ingrese el nombre de la persona a buscar: " );
31   fgets( nombreBuscar, 20, stdin );
32
33   encontrado = Falso;
34   for( i = 0 ; i < tamanio ; i++ )
35   {
36     if( strcmp ( nombreBuscar, arregloNombres[ i ] )  == 0 )
37     {
38       printf( "El nombre está en la posición %d\n", i );
39       encontrado = Verdadero;
```

```
40      }
41    }
42
43    if( encontrado == Falso )
44    {
45      printf( "El nombre no está en el arreglo" );
46    }
47
48    freeS ( arregloNombres, tamanio );
49
50    return 0;
51  }
52
53  char **dimensionarS ( int n, int longitud )
54  {
55    char **vector;
56
57    vector =  (char **) malloc ( sizeof(char*) * n);
58
59    for ( int i = 0 ; i < n ; i++ )
60    {
61      vector[i] = dimensionarC ( longitud );
62    }
63
64    return vector;
65  }
66
67  void freeS ( char **vector, int n )
68  {
69    for ( int i = 0 ; i < n ; i++ )
70    {
71      free ( vector [ i ] );
72    }
73
74    free( vector );
75  }
```

Al ejecutar el programa:

Primera ejecución:

```
Ingrese el tamaño del arreglo: 3
Ingrese el nombre persona 0: Jacobo
Ingrese el nombre persona 1: Gabriela
Ingrese el nombre persona 2: Adriana
Ingrese el nombre de la persona a buscar: Adriana
El nombre está en la posición 2
```

Segunda ejecución:

```
Ingrese el tamaño del arreglo: 3
Ingrese el nombre persona 0: Jacobo
Ingrese el nombre persona 1: Gabriela
Ingrese el nombre persona 2: Adriana
Ingrese el nombre de la persona a buscar: Nidia
El nombre no está en el arreglo
```

Explicación del programa:

La primera parte del programa consiste en la declaración de variables, que ocupa las líneas 15 a 17.

Observe cómo se utiliza un puntero para almacenar el arreglo de nombres es de tipo `**char` para poder crear un arreglo de arreglo de `char` (para resumir un arreglo de cadenas). Luego de declarar las variables, se solicita al usuario el tamaño del arreglo y, con este dato, se dimensiona el mismo.

```
19  printf( "Ingrese el tamaño del arreglo: ");
20  scanf( "%d", &tamanio );
21
22  arregloNombres = dimensionarS( tamanio, 20 );
```

Mediante el uso del primer ciclo `for`, se hace el ingreso de los nombres que el usuario ingresa y se almacenan en el arreglo.

```
24  for( i = 0 ; i < tamanio ;i++ )
25  {
26    printf( "Ingrese el nombre persona %d:" , i );
27    gets( arregloNombres[ i ] );
28  }
```

Posteriormente, se pide el nombre que se va a buscar y la variable "centinela" denominada `encontrado` se inicializa en falso. Con esta variable centinela se informará si se encontró o no el nombre que se está buscando.

```
30  printf( "Ingrese el nombre de la persona a buscar: " );
31  gets( nombreBuscar );
```

Se usa un segundo ciclo `for` para recorrer nuevamente el arreglo buscando el nombre; dentro de este segundo ciclo hay una instrucción `if`, utilizada para comparar si el nombre buscado es el mismo que está almacenado en la celda actual del arreglo; de ser cierto, se muestra un mensaje con la posición del arreglo donde está el nombre y, el contenido de la variable `encontrado` cambia a verdadero, lo que quiere decir que el nombre buscado ya se encontró.

```
33   encontrado = Falso;
34   for( i = 0 ; i < tamanio ; i++ )
35   {
36     if( strcmp ( nombreBuscar, arregloNombres[ i ] )  == 0 )
37     {
38       printf( "El nombre está en la posición %d\n", i );
39       encontrado = Verdadero;
40     }
41   }
```

Finalmente, y luego del segundo ciclo, el algoritmo posee otra instrucción `if`, que evalúa si la variable encontrado tiene almacenado un valor de falso, lo cual indicaría que no se encontró el nombre buscado en el vector, por lo que se muestra este mensaje con la instrucción `int`. Luego de esto, el programa termina.

```
43   if( encontrado == Falso )
44   {
45     printf( "El nombre no está en el arreglo" );
46   }
```

.:Ejemplo 6.3. *Escriba un programa en Lenguaje C que almacene en un vector una cantidad `n` de números enteros positivos. El programa deberá almacenar en otro vector los elementos guardados en el arreglo inicial de forma invertida, esto es, el último elemento que está en el vector inicial, que ocupa la posición `n-1`, deberá quedar en la primera posición del otro vector y, el primer elemento almacenado en el vector inicial, quedará en la última posición del vector generado.*

Análisis del problema:

- **Resultados esperados:** mostrar un segundo vector con los elementos ubicados de forma invertida a como se ingresaron en un primer arreglo.
- **Datos disponibles:** se conoce el tamaño del vector inicial y los números que se van a ubicar dentro de él.
- **Proceso:** se pide el tamaño del vector inicial y los números a guardar dentro de él; a continuación, se recorre, con la ayuda de un ciclo el arreglo desde la primera hasta la última posición y se van ubicando los números en un segundo vector, que paralelamente, se va recorriendo de forma invertida, es decir, desde la última hasta la primera posición.

- **Variables requeridas:**
  - `n`: tamaño de los arreglos a utilizar.
  - `i`: variable de control que permitirá pasar por las posiciones del arreglo inicial.
  - `j`: variable de control que permitirá pasar por las posiciones del segundo arreglo.
  - `arregloInicial`: arreglo que almacena los números ingresados inicialmente.
  - `arregloFinal`: arreglo que almacenará los números que están en el primer arreglo pero en orden inverso.

De acuerdo con el análisis realizado, se propone el programa 6.3.

Programa 6.3: InversionArreglo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define dimensionarI(n) (int *) malloc(sizeof(int)*n)
5
6  int main ()
7  {
8    int *arregloInicial, *arregloFinal, n, i, j;
9
10   printf( "Ingrese el tamaño del Arreglo: " );
11   scanf( "%d", &n );
12
13   arregloInicial = dimensionarI( n );
14   arregloFinal   = dimensionarI( n );
15
16   for ( i = 0 ; i < n ; i++ )
17   {
18     do
19     {
20       printf( "Ingrese el número %d:" , i );
21       scanf( "%d", &arregloInicial[ i ] );
22
23       if( arregloInicial[ i ] < 0 )
24       {
25         printf( "Este número no es válido\n" );
26       }
27     }  while( arregloInicial[ i ] < 0 );
```

```
28    }
29
30    j = n - 1;
31    for ( i = 0 ; i < n ; i++ )
32    {
33      arregloFinal[ j ] = arregloInicial[ i ];
34      j = j - 1;
35    }
36
37    for ( i = 0 ; i < n ; i++ )
38    {
39      printf( "%d ", arregloFinal[ i ] );
40    }
41
42    free ( arregloInicial );
43    free ( arregloFinal );
44
45    return 0;
46  }
```

Al ejecutar el programa:

```
Ingrese el tamaño del Arreglo: 4
Ingrese el número 1: 3
Ingrese el número 2: 9
Ingrese el número 3: -8
Este número no es válido
Ingrese el número 3: 45
Ingrese el número 4: 92
92 45 9 3
```

Explicación del programa:

En la línea 8 se hace la declaración de las variables. Posteriormente, se ingresa el tamaño del arreglo, y se almacena en la variable `n`; esta variable dará tamaño al arreglo inicial y final.

```
10    printf( "Ingrese el tamaño del Arreglo: " );
11    scanf( "%d", &n );
12
13    arregloInicial = dimensionarI( n );
14    arregloFinal   = dimensionarI( n );
```

A continuación en el programa se encuentra un ciclo `for`, cuyo propósito es almacenar los números enteros positivos ingresado en las posiciones del `arregloInicial`. Analice cómo dentro de este primer ciclo `for`, existe

un ciclo `do-while` que permite validar el ingreso de números positivos, lo cual es un requisito propuesto en el enunciado.

En el ciclo `do-while`, se ubica una estructura de decisión `if` que imprime un mensaje en caso de que el número ingresado sea menor a cero, lo que indica que este número no es válido y no se guardará en el `arregloInicial`, ya que no es un número positivo.

```
16  for ( i = 0 ; i < n ; i++ )
17  {
18    do
19    {
20      printf( "Ingrese el número %d:" , i );
21      scanf( "%d", &arregloInicial[ i ] );
22
23      if( arregloInicial[ i ] < 0 )
24      {
25        printf( "Este número no es válido\n" );
26      }
27    }  while( arregloInicial[ i ] < 0 );
28  }
```

La segunda parte del programa empieza inicializando la variable `j` en `n-1`, es decir, en la última posición del vector. el segundo ciclo `for` que aparece allí, recorre el `arregloInicial` desde la primera hasta la última posición, pero, al mismo tiempo recorre el `arregloFinal` desde la última posición marcada con la variable `j` hasta la posición 0, ya que esta se va reduciendo dentro del ciclo; así se van pasando los elementos del `arregloInicial` a cada posición del `arregloFinal`.

```
30  j = n - 1;
31  for ( i = 0 ; i < n ; i++ )
32  {
33    arregloFinal[ j ] = arregloInicial[ i ];
34    j = j - 1;
35  }
```

Finalmente, un último ciclo `for` imprime los elementos del `arregloFinal`, los cuales están invertidos a como aparecen en el arreglo inicial.

```
37  for ( i = 0 ; i < n ; i++ )
38  {
39    printf( "%d ", arregloFinal[ i ] );
40  }
```

.:Ejemplo 6.4. *Escriba un programa usando Lenguaje C que almacene en un vector una cantidad n de números pares y en otro vector una cantidad*

m de números impares ingresados por el usuario. Genere un tercer vector con todos los números contenidos en los dos vectores iniciales y muestre este tercer vector como resultado.

Análisis del problema:

- **Resultados esperados:** mostrar un vector que contiene los números almacenados en los dos vectores iniciales.
- **Datos disponibles:** se conoce el tamaño de los dos vectores iniciales y los números a almacenar en ellos.
- **Proceso:** primero se solicita el tamaño de cada uno de los vectores iniciales; enseguida, se ingresan los números a almacenar en los dos vectores iniciales; obviamente, esto requiere el uso de ciclos. A continuación, los números del primer vector se van pasando al tercer vector; luego, se van pasando los números del segundo vector al tercer vector y, finalmente se imprime el tercer vector. Tenga en cuenta que deberá validarse que los números a ingresar en el primer vector sean pares y, al segundo vector sean impares.

- **Variables requeridas:**

  - `arregloPares`: vector que contendrá los números pares ingresados por el usuario.
  - `arregloImpares`: vector que contendrá los números impares ingresados por el usuario.
  - `arregloTotal`: vector que tendrá todos los elementos ingresados.
  - `tamanioPares`: tamaño del vector de números pares.
  - `tamanioImpares`: tamaño del vector de números impares.
  - `tamanioTotal`: tamaño del vector que contendrá todos los números.
  - `i`: variable de control para recorrer los vectores.
  - `j`: variable de control para recorrer los vectores.
  - `salida`: variable que almacena los datos del tercer vector para su posterior impresión.

Conforme al análisis realizado, se propone el programa 6.4.

Programa 6.4: ConcatenaArreglos

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

#define dimensionarI(n) (int *) malloc(sizeof(int)*n)

int main ()
{
  int *arregloPares, *arregloImpares, *arregloTotal;
  int tamanioPares, tamanioImpares, tamanioTotal, i, j;
  char salida[ 100 ];

  printf( "Ingrese la cantidad de pares: " );
  scanf( "%d", &tamanioPares );

  printf( "Ingrese la cantidad de impares: " );
  scanf( "%d", &tamanioImpares );

  tamanioTotal = tamanioPares + tamanioImpares;

  arregloPares   = dimensionarI( tamanioPares   );
  arregloImpares = dimensionarI( tamanioImpares );
  arregloTotal   = dimensionarI( tamanioTotal   );

  for ( i = 0 ; i < tamanioPares ; i++ )
  {
    do
    {
      printf( "Ingrese el par %d:" , i );
      scanf( "%d", &arregloPares[ i ] );

      if( arregloPares[ i ] % 2 != 0 )
      {
        printf( "Este número no es par\n" );
      }
    }
    while( arregloPares[ i ] % 2 != 0 );
  }

  for ( i = 0 ; i < tamanioImpares ; i++ )
  {
    do
    {
      printf( "Ingrese el impar %d: " , i );
      scanf( "%d", &arregloImpares[ i ] );

      if( arregloImpares[ i ] % 2 != 1 )
      {
        printf( "Este número no es impar\n" );
```

```
49        }
50      }
51      while( arregloImpares[ i ]  % 2 != 1 );
52    }
53
54    j = 0;
55    for ( i = 0 ; i < tamanioPares ; i++ )
56    {
57      arregloTotal[ j ] = arregloPares [ i ];
58      j = j + 1;
59    }
60
61    for ( i = 0 ; i < tamanioImpares ; i++ )
62    {
63      arregloTotal[ j ] = arregloImpares [ i ];
64      j = j + 1;
65    }
66
67    salida[ 0 ] = '\0';
68    for ( i = 0 ; i < tamanioTotal ; i++ )
69    {
70      sprintf ( salida, "%s %d", salida, arregloTotal [ i ] );
71    }
72
73    printf( "Arreglo concatenado: %s", salida );
74
75    free( arregloPares );
76    free( arregloImpares );
77    free( arregloTotal );
78
79    return 0;
80  }
```

Al ejecutar el programa:

```
Ingrese la cantidad de pares: 3
Ingrese la cantidad de impares: 2
Ingrese el par 1: 8
Ingrese el par 2: 9
Este número no es par
Ingrese el par 2: 34
Ingrese el par 3: 86
Ingrese el impar 1: 73
Ingrese el impar 2: 8
Este número no es impar
Ingrese el impar 2: 1
Arreglo concatenado 8 34 86 73 1
```

Explicación del programa:

Las líneas 8 a la 10 se usan para declarar todas las variables necesarias. A continuación, se solicitan la cantidad de números pares e impares y se determina, con los datos anteriores, la cantidad total de números a ingresar.

```
6   printf( "Ingrese la cantidad de pares: " );
7   scanf( "%d", &tamanioPares );
8
9   printf( "Ingrese la cantidad de impares: " );
10  scanf( "%d", &tamanioImpares );
11
12  tamanioTotal = tamanioPares + tamanioImpares;
```

Enseguida, se utiliza `dimensionarI` para dar tamaño a cada uno de los arreglos.

```
20  arregloPares   = dimensionarI( tamanioPares   );
21  arregloImpares = dimensionarI( tamanioImpares );
22  arregloTotal   = dimensionarI( tamanioTotal   );
```

Después en el programa, se procede a realizar la lectura de los números pares y almacenarlos en el respectivo vector:

```
24  for ( i = 0 ; i < tamanioPares ; i++ )
25  {
26    do
27    {
28      printf( "Ingrese el par %d:" , i );
29      scanf( "%d", &arregloPares[ i ] );
30
31      if( arregloPares[ i ] % 2 != 0 )
32      {
33        printf( "Este número no es par\n" );
34      }
35    }
36    while( arregloPares[ i ] % 2 != 0 );
37  }
```

Note cómo este ciclo `for` no solo solicita los números sino que los valida para recibir únicamente pares. La validación se lleva a cabo con un ciclo `do-while` y una estructura `if` que muestra al usuario, el intento de ingresar un dato no válido. Esto mismo se realiza en las líneas 39 a la 52 pero, esta vez con el ingreso de los números impares al vector respectivo.

Luego en el programa, la variable de control `j` toma el valor de 0, y se utilizará para recorrer el vector `arregloTotal`. Seguidamente se recorre con un ciclo `for` el vector `arregloPares` y se van pasando los números

ubicados en sus celdas a las celdas del vector `arregloTotal`. De la misma forma, se recorre con otro ciclo `for` el vector `arregloImpares` y se van pasando sus elementos al vector `arregloTotal`. Observe que la variable `j` no se inicializó nuevamente cuando se empezaron a pasar los números impares del segundo arreglo, de esta manera se empieza en la siguiente posición en donde quedó el último número par.

```
54   j = 0;
55   for ( i = 0 ; i < tamanioPares ; i++ )
56   {
57     arregloTotal[ j ] = arregloPares [ i ];
58     j = j + 1;
59   }
60
61   for ( i = 0 ; i < tamanioImpares ; i++ )
62   {
63     arregloTotal[ j ] = arregloImpares [ i ];
64     j = j + 1;
65   }
```

Al final del programa, se inicializa la variable `salida` con una cadena vacía (`'\0'`) (el carácter de fin de cadena en la primera posición), y se va recorriendo con otro ciclo el `arregloTotal` y se va concatenando en la variable `salida` el elemento ubicado en la celda actual del `arregloTotal`, más un espacio en blanco. Al terminar este ciclo, la variable `salida` contendrá todos los elementos que hay en el `arregloTotal`, por lo que se mostrará con el `printf`.

```
67   salida[ 0 ] = '\0';
68   for ( i = 0 ; i < tamanioTotal ; i++ )
69   {
70     sprintf ( salida, "%s %d", salida, arregloTotal [ i ] );
71   }
72
73   printf( "Arreglo concatenado: %s", salida );
```

.:Ejemplo 6.5. *Construya un programa al que se le ingresa un número entero entre cero y noveinta y nueve (0 y 99) y que, muestra como salida, ese mismo número, pero expresado con palabras.*

Análisis del problema:

- **Resultados esperados:** un número entre 0 y 99 expresado en palabras.

- **Datos disponibles:** el número que se desea convertir a palabras en el rango 0 a 99.
- **Proceso:** se pide el número a convertir en palabras; Enseguida, se realiza se lleva a cabo el proceso de conversión de este número a palabras, haciendo búsqueda de las palabras adecuadas en varios vectores; finalmente se imprime la palabra convertida.

  El proceso de conversión se lleva a cabo de la siguiente forma: si el número es menor a 11 se busca la palabra equivalente en un primer vector; sino, se determina si el número está entre 11 y 19, y se busca su respectiva palabra en otro vector; si el número es mayor a 19, se encuentran sus unidades y sus decenas y se buscan sus palabras equivalentes en los vectores respectivos.

  Este ejercicio requiere de tres vectores que almacenen las palabras de ciertos números claves, inicializados desde su declaración. El programa no solicita al usuario estas palabras, solo el número que se desea convertir.
- **Variables requeridas:**

  - `numero`: almacena el número a convertir a palabras.
  - `numeroPalabras`: almacena el número ya convertido en palabras.
  - `nomUnidades`: vector que almacena los nombres de los primeros once números (iniciando desde el cero).
  - `nomEspecial`: vector que almacena los nombres de los siguientes ocho números comenzando en once.
  - `nomDecenas`: vector que almacena los nombres de los números del veinte al noventa, solo de las decenas.
  - `decenas`: almacena la cantidad de decenas que tiene el número a convertir.
  - `unidades`: almacena la cantidad de unidades que tiene el número a convertir.

De acuerdo al análisis realizado, se propone el programa 6.5.

Programa 6.5: ConversionNumero

```
1  #include <stdio.h>
2
3  int main ()
4  {
5    int numero, decenas, unidades;
6    char numeroPalabras[ 20 ];
7    char *nomUnidades[ 11 ] = { "cero",   "uno",    "dos",
8                                "tres",  "cuatro", "cinco",
9                                "seis",  "siete",  "ocho",
10                               "nueve", "diez" };
11
12   char *nomEspecial[ 9 ]  = { "once",       "doce",
13                               "trece",      "catorce",
14                               "quince",     "dieciseis",
15                               "diecisiete", "dieciocho",
16                               "diecinueve" };
17
18   char *nomDecenas [ 8 ]  = { "veinte",   "treinta",
19                               "cuarenta", "cincuenta",
20                               "sesenta",  "setenta",
21                               "ochenta",  "noventa" };
22
23   do
24   {
25     printf ( "Ingrese el número a convertir: " );
26     scanf( "%d", &numero );
27
28     if ( numero < 0 || numero >  99 )
29     {
30       printf( "Este número no es válido" );
31     }
32   } while ( numero < 0 || numero > 99);
33
34   if( numero <= 10 )
35   {
36     sprintf ( numeroPalabras, "%s",  nomUnidades[numero] );
37   }
38   else
39   {
40     if( numero <= 19 )
41     {
42       sprintf( numeroPalabras,"%s",nomEspecial[numero-11] );
43     }
44     else
45     {
46       unidades = numero  % 10;
47       decenas  = numero / 10;
48
```

```
49        if( unidades == 0 )
50        {
51          sprintf(numeroPalabras," %s",nomDecenas[decenas-2]);
52        }
53        else
54        {
55           sprintf ( numeroPalabras, " %s y %s",
56                     nomDecenas[ decenas - 2 ],
57                     nomUnidades[ unidades ] );
58        }
59      }
60    }
61
62    printf( "El número es: %s", numeroPalabras );
63
64    return 0;
65  }
```

Al ejecutar el programa:

```
Ingrese el número a convertir: -4
Este número no es válido
Ingrese el número a convertir: 200
Este número no es válido
Ingrese el número a convertir: 87
El número es: ochenta y siete
```

Explicación del programa:

Se declaran las variables requeridas y se inicializan los arreglos con las palabras correspondientes a los números (líneas 5 a la 21); se solicita al usuario un número entero 0 y 99, y se valida su ingreso. La validación se implementa con un ciclo `do -while` y no permite ingresar un número menor a cero o mayor a noventa y nueve.

```
23  do
24    {
25      printf ( "Ingrese el número a convertir: " );
26      scanf( " %d", &numero );
27
28      if ( numero < 0 || numero >  99 )
29      {
30        printf( "Este número no es válido" );
31      }
32    } while ( numero < 0 || numero > 99);
```

Luego, usando la primera estructura `if`, se verifica si el número es menor o igual a 10. Si ser así, se busca la palabra que equivale al número en el

vector de unidades. Si el número es menor o igual a 19, el programa buscaría en el vector de especiales. Si el número es mayor a 19, se obtienen sus decenas y unidades, para así poder mostrar las palabras adecuadas. En caso de que el número no posea unidades (`unidades == 0`), se mostraría solo el nombre de las decenas, sino, se muestran decenas y unidades también.

```
34  if( numero <= 10 )
35  {
36    sprintf ( numeroPalabras, "%s",  nomUnidades[numero] );
37  }
38  else
39  {
40    if( numero <= 19 )
41    {
42      sprintf( numeroPalabras,"%s",nomEspecial[numero-11] );
43    }
44    else
45    {
46      unidades = numero  % 10;
47      decenas  = numero / 10;
48
49      if( unidades == 0 )
50      {
51        sprintf(numeroPalabras,"%s",nomDecenas[decenas-2]);
52      }
53      else
54      {
55         sprintf ( numeroPalabras, "%s y %s",
56                   nomDecenas[ decenas - 2 ],
57                   nomUnidades[ unidades ] );
58      }
59    }
60  }
```

Con el propósito de obtener la posición correcta en los arreglos, observe como se necesita sumar o restar ciertas cantidades a las índices. Por ejemplo, si el número es 8, se suma un uno para llegar a la novena posición del vector de unidades en la que está la palabra "ocho"; esto ocurre porque la palabra "cero" se encuentra en la posición cero. Algo parecido sucede en los otros dos vectores, por ejemplo, en caso de que el número a convertir fuera 17, el programa llega a la parte del `else` del primer `if`, allí se encuentra con un segundo `if` cuya condición daría verdadero, por lo que el programa almacenaría en la variable `numeroPalabras` lo que hay en el vector `arregloEspecial` celda `numero - 10`, que equivale a 17 - 10 = 7 y, en la posición 7 de este vector, está la palabra `diecisiete`.

Suponga ahora que el número ingresado es 60, el programa iría hasta la instrucción `if(numero<=19)` (Línea 40) que se evaluaría como falso, por lo que el programa obtendría las `unidades` = 0 y las `decenas` = 6; esto indicaría la posición 5 (`decenas - 2`) en el vector de `nomDecenas`, que tiene la plabra sesenta `sesenta`.

Observe cómo durante todo el programa se utiliza la función `sprintf`.

Por último, el programa imprime el número ingresado ya convertido a palabras.

```
62   printf( "El número es: %s", numeroPalabras );
```

Aclaración:

Los ejercicios que siguen a continuación se implementaron mediante el uso de funciones y procedimientos, pues estas estructuras permiten escribir un código más organizado, comprensible y modular. Además, con el empleo de funciones y procedimientos, la realización de pruebas y la corrección de errores, será mucho más fácil de llevar a cabo.

.:Ejemplo 6.6. *Construya un programa en Lenguaje C que guarde un grupo de* `n` *números enteros en un arreglo y que, posteriormente encuentre mediante funciones y procedimientos la cantidad de números pares y de impares que quedaron almacenados en el arreglo.*

Análisis del problema:

- **Resultados esperados:** mostrar la cantidad de números pares y de impares que quedaron almacenados en el arreglo de números.
- **Datos disponibles:** se conoce el tamaño del arreglo y los números enteros que se almacenarán en el mismo.
- **Proceso:** se pide al usuario el ingreso del tamaño del arreglo (`n`) y los números a almacenar. Posteriormente se recorre el arreglo con un ciclo y se determina con una estructura de decisión si el elemento que se encuentra en cada celda es par; y si es así, se cuenta, lo mismo ocurre si el número es impar. Por último, el programa muestra la cantidad de números pares y de impares contados.

- **Variables requeridas:**
  - En el programa principal
    - `n`: tamaño del arreglo o cantidad de números a almacenar.
    - `numeros`: vector que almacenará los los números ingresados por el usuario.
    - `cantidadPares`: almacena la cantidad de números pares contados en el vector.
    - `cantidadImpares`: almacena la cantidad de números impares contados en el vector.
  - En la función `leerDatos`
    - Parámetros:
      - `n`: cantidad de elementos a leer.
    - Variables locales:
      - `i`: variable para controlar el ciclo.
      - `arreglo`: vector que almacena los datos ingresados por el usuario.
  - En la función `obtenerCantidadPares`
    - Parámetros:
      - `arreglo`: vector que contiene los datos que el usuario ingresa.
      - `n`: tamaño del vector.
  - En la función `obtenerCantidadImpares`
    - Parámetros:
      - `arreglo`: vector que contiene los datos ingresados.
      - `n`: tamaño del vector.
- En la función `contar`
  - Parámetros:
    - `arreglo`: vector que contiene los datos a ser analizados.
    - `resto`: parámetro para especificar si lo que se va a contar son pares (`resto=0`) o impares (`resto=1`). `resto`; es el resultado del resto de la división entera en 2 (`arreglo[ i ] % 2 == resto`).
    - `n`: tamaño del vector.
  - Variables locales:
    - `i`: variable para controlar el ciclo.

- `cantidad`: variable para contar la cantidad de pares o impares (dependiendo del valor `resto`) ubicados en el arreglo.

- En el procedimiento `mostrarResultados`

  - Parámetros:
    - `pares`: almacena la cantidad de números pares encontrados en el vector.
    - `impares`: almacena la cantidad de números impares encontrados en el vector.

Conforme al análisis realizado, se propone el programa 6.6.

Programa 6.6: Numero

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

#define dimensionarI(n) (int *) malloc(sizeof(int)*n)

int *leerDatos             ( int n );
int obtenerCantidadPares   ( int arreglo[], int n );
int obtenerCantidadImpares( int arreglo[], int n );
int contar                 ( int arreglo[], int resto,int n);
void mostrarResultados     ( int pares,     int impares );

int main()
{
  int n, *numeros, cantidadPares, cantidadImpares;

  printf( "Ingrese el tamaño del Arreglo: " );
  scanf( "%d", &n );
  numeros = leerDatos( n );

  cantidadPares   = obtenerCantidadPares  ( numeros, n );
  cantidadImpares = obtenerCantidadImpares( numeros, n );

  mostrarResultados( cantidadPares, cantidadImpares );

  free ( numeros );

  return 0;
}

```

```
30 int *leerDatos( int n )
31 {
32   int i;
33   int *arreglo;
34
35   arreglo = dimensionarI ( n );
36
37   for ( i = 0 ; i < n ; i++)
38   {
39     printf( "Ingrese el número %d: " , i );
40     scanf( "%d", &arreglo[ i ] );
41   }
42   return arreglo;
43 }
44
45
46 int obtenerCantidadPares( int arreglo[], int n )
47 {
48   return contar ( arreglo, 0, n );
49 }
50
51 int obtenerCantidadImpares( int arreglo[], int n )
52 {
53   return contar ( arreglo, 1, n );
54 }
55
56 int contar( int arreglo[], int resto, int n )
57 {
58   int i, cantidad;
59
60   cantidad = 0;
61   for ( i = 0 ; i < n ; i++ )
62   {
63     if( arreglo[ i ] % 2 == resto )
64     {
65       cantidad = cantidad + 1;
66     }
67   }
68
69   return cantidad;
70 }
71
72 void mostrarResultados( int pares, int impares )
73 {
74    printf( "Cantidad de pares  : %d\n", pares );
75    printf( "Cantidad de impares: %d\n", impares );
76 }
```

Al ejecutar el programa:

```
Ingrese el tamaño del Arreglo: 5
Ingrese el número 1: 4
Ingrese el número 2: 71
Ingrese el número 3: 22
Ingrese el número 4: 15
Ingrese el número 5: 78
Cantidad de pares  : 3
Cantidad de impares: 2
```

Explicación del programa:

Este programa fue escrito de la siguiente manera:

Se creó un programa que en la función principal o `main` declara las variables que reciben los resultados, se solicita al usuario el tamaño del vector y se invocan las funciones: `leerDatos()`, `obtenerCantidadPares()`, `obtenerCantidadImpares()`, (las dos últimas a su vez, llaman a la función `contar()`). Así mismo, el procedimiento denominado `mostrarResultados()` muestra los resultados obtenidos por el programa. Al final se libera la memoria.

Cuando ya se tiene el tamaño del vector, se envía como parámetro a la función `leerDatos(n)`, que en su interior, crea un nuevo vector, lo llena con los datos ingresados por el usuario y lo retorna a la función principal. El valor de retorno es almacenado en la variable `numero`.

A continuación, se describen las funciones usadas y el procedimiento que muestra los resultados.

```
15  int n, *numeros, cantidadPares, cantidadImpares;
16
17  printf( "Ingrese el tamaño del Arreglo: " );
18  scanf( "%d", &n );
19  numeros = leerDatos( n );
20
21  cantidadPares   = obtenerCantidadPares  ( numeros, n );
22  cantidadImpares = obtenerCantidadImpares( numeros, n );
23
24  mostrarResultados( cantidadPares, cantidadImpares );
25
26  free ( numeros );
```

La función `leerDatos()`, captura los números ingresados y los guarda en cada posición del arreglo; esto requirió de un ciclo `for`, que inicia la variable `i` en 0 y va hasta una posición menos que el tamaño del vector

(n-1). Dentro del ciclo, con las instrucciones `printf` y `scanf` se solicitan los datos y se almacenan en cada celda del arreglo. Esta función recibe como parámetro el tamaño del vector con el que se inicializa el puntero al vector declarado localmente, lo llena y lo retorna al final de la función.

```
30 int* leerDatos( int n )
31 {
32   int i;
33   int *arreglo;
34
35   arreglo = dimensionarI ( n );
36
37   for ( i = 0 ; i < n ; i++)
38   {
39     printf( "Ingrese el número %d: " , i );
40     scanf( "%d", &arreglo[ i ] );
41   }
42   return arreglo;
43 }
```

La funciones para contar los pares `obtenerCantidadPares()` y los impares `obtenerCantidadImpares()` son casos especiales de una función más general llamada `contar()`. La diferencia que hay entre estas dos funciones es el valor del resto de la división entera entre dos. En el caso de los pares, el resto es cero, mientras que en el caso de los impares el resto es uno.

Los parámetros de la función, `obtenerCantidadPares(numeros, n)` son: la referencia al vector de números y el tamaño del mismo. Esta función invoca, a su vez, a la función `contar()` enviándole la referencia al vector de números, el cero y el tamaño del vector. El cero es el parámetro que le permite a la función `contar()` determinar la cantidad de números pares que hay en el vector.

De la misma forma que ya se mencionó, opera la función `obtenerCantidadImpares(numeros, n)` que posee los mismos parámetros que la anterior, y que también invoca a la función `contar()`, pero esta vez, se envía un uno en lugar de un cero.

```
46 int obtenerCantidadPares( int arreglo[], int n )
47 {
48   return contar ( arreglo, 0, n );
49 }
```

```
51 int obtenerCantidadImpares( int arreglo[], int n )
52 {
53   return contar ( arreglo, 1, n );
54 }
```

La función `contar()` recibe como parámetros el vector, un valor `resto` usado para determinar si el número almacenado es par(0) o impar(1) y, el tamaño del vector. En esta función se inicializa un contador, denominado `cantidad`, en cero, posteriormente se recorre el arreglo con un ciclo `for` analizando si cada número es par o impar; si la condición del `if`, que está dentro del ciclo, es verdadera, se cuenta uno más. Por último, la función retorna el valor de la variable `cantidad`.

```
56 int contar( int arreglo[], int resto, int n )
57 {
58   int i, cantidad;
59
60   cantidad = 0;
61   for ( i = 0 ; i < n ; i++ )
62   {
63     if( arreglo[ i ] % 2 == resto )
64     {
65       cantidad = cantidad + 1;
66     }
67   }
68
69   return cantidad;
70 }
```

Finalmente, el procedimiento `mostrarResultados()`, recibe como parámetros la cantidad de pares y de impares encontrados en el arreglo y los muestra mediante las instrucciones `printf` ubicadas dentro de él.

Aclaración:

Si se desea utilizar una función que cree un arreglo dinámicamente, es necesario usar la función `malloc` tal y como se presentó en el ejemplo anterior. Si por el contrario, se crea el arreglo de forma tradicional:

```
int *leerDatos( int n )
{
  int a[ n ];

  return a;
}
```

...

Aclaración:

...

El compilador mostrará una advertencia en donde indica que no debe retornar la dirección de memoria de una variable local.

```
static.c:16:10:
warning: address of stack memory associated with local
   variable 'a' returned [-Wreturn-stack-address]
return a;
       ^
1 warning generated.
```

Es precisamente por esta razón que es indispensable el uso de `malloc`.

.:Ejemplo 6.7. *Escribir un programa en Lenguaje C que almacene en un vector las notas finales obtenidas por cada uno de los 20 estudiantes de un grupo en una asignatura cualquiera y, que usando funciones, encuentre la nota promedio del grupo, la nota más alta y más baja obtenida en la asignatura.*

Análisis del problema:

- **Resultados esperados:** mostrar la nota promedio del grupo de estudiantes, la mayor y menor nota obtenidas entre las 20 que conforman el grupo.
- **Datos disponibles:** se conocen las 20 notas de los estudiantes.
- **Proceso:** Este problema se resuelve con varios recorridos al vector, luego de haberlo llenado con las notas de los estudiantes. Cada recorrido encontraría uno de los requisitos solicitados, es decir, el promedio, las notas más alta y la más baja del grupo. Por esto, se recomienda el uso de funciones, puesto que cada una resolverá un requisito particular. El ejercicio está propuesto para 20 estudiantes, pero perfectamente se puede adaptar a n estudiantes, tal como se ha hecho en ejercicios anteriores.

  En este programa, lo primero a realizar es solicitar las 20 notas al usuario y almacenarlas en el vector. Luego, para obtener la nota promedio del grupo, el vector se recorre en su totalidad sumando las notas que están en las celdas y dividiendo la suma en 20; esto se hace mediante un ciclo que recorra las diferentes posiciones del vector y

una variable que haga de acumulador en la que se suman las notas. Se recorre nuevamente el vector para encontrar la nota más alta y la más baja almacenadas en él; la nota más alta, se obtiene comparando la nota almacenada en la primera posición del arreglo con la segunda, la segunda con la tercera y así sucesivamente hasta llegar a la última celda; esto requiere del uso de una estructura de decisión. Lo mismo se hace para encontrar la nota más baja, la diferencia entre obtener la mayor y obtener la menor nota radica en la condición de la estructura `if` que se usará. Dado que el enunciado habla de 20 estudiantes, se usa una constante con este valor durante todo el programa.

- **Variables requeridas:**
  - En el programa principal
    - `MAX`: constante que almacena un 20, que corresponde al número de estudiantes.
    - `arrregloNotas`: vector que almacena las notas de los estudiantes.
    - `notaMayor`: almacena la menor nota entre las que están en el vector.
    - `notaMenor`: almacena la mayor nota entre las que están en el vector.
    - `notaPromedio`: almacena la nota promedio de los 20 estudiantes.
  - En la función `leerDatos`
    - Parámetros:
      - `tamanio`: cantidad de notas a almacenar.
    - Variables locales:
      - `i`: variable usada para controlar el ciclo.
      - `arreglo`: vector que almacena las 20 notas que el usuario ingresará.
  - En la función `calcularPromedio`
    - Parámetros:
      - `arregloNotas`: vector que almacena las notas de los 20 estudiantes.
      - `n`: tamaño del vector, equivalente a 20.
    - Variables locales:
      - `i`: variable usada para controlar el ciclo.
      - `suma`: variable de tipo acumulador que almacena la suma de las 20 notas.

        - ◇ promedio: variable que almacena el promedio de las notas.
    - En la función obtenerMayor
        - Parámetros:
            - ◇ arregloNotas: vector que contiene las notas de los estudiantes.
            - ◇ n: tamaño del vector.
        - Variables locales:
            - ◇ i: variable para controlar el ciclo.
            - ◇ mayor: variable que guarda la nota mayor contenida en el arreglo de notas.
    - En la función obtenerMenor
        - Parámetros:
            - ◇ arregloNotas: vector que contiene las notas de los estudiantes.
            - ◇ n: tamaño del vector.
        - Variables locales:
            - ◇ i: variable para controlar el ciclo.
            - ◇ menor: variable que guarda la nota menor contenida en el vector de notas.

- En el procedimiento mostrarResultados
    - Parámetros:
        - promedio: promedio de las notas de los estudiantes.
        - mayor: nota mayor entre las ingresadas.
        - menor: nota menor entre las ingresadas.

De acuerdo al análisis realizado, se propone el programa 6.7.

Programa 6.7: NotasMateria

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define dimensionarD(n) (double *) malloc(sizeof(double)*n)
5
6 double *leerDatos          ( int    tamanio );
7 double  calcularPromedio   ( double arregloNotas,   int n );
8 double  obtenerMayor       ( double arregloNotas[], int n );
9 double  obtenerMenor       ( double arregloNotas[], int n );
```

```
10 void    mostrarResultados ( double promedio,
11                             double mayor,    double menor );
12
13 int main()
14 {
15   const int MAX = 20;
16   double *arregloNotas, notaPromedio, notaMayor, notaMenor;
17
18   arregloNotas = leerDatos( MAX );
19   notaPromedio = calcularPromedio( arregloNotas, MAX );
20   notaMayor    = obtenerMayor( arregloNotas, MAX );
21   notaMenor    = obtenerMenor( arregloNotas, MAX );
22
23   mostrarResultados( notaPromedio, notaMayor, notaMenor );
24
25   free ( arregloNotas );
26   return 0;
27 }
28
29 double *leerDatos( int tamanio )
30 {
31   double *arreglo;
32   int i;
33
34   arreglo = dimensionarD ( tamanio );
35
36   for ( i = 0 ; i < tamanio ; i++ )
37   {
38     printf( "Ingrese el nota del estudiante %d: " , i );
39     scanf( "%lf", &arreglo[ i ] );
40   }
41   return arreglo;
42 }
43
44 double calcularPromedio( double arregloNotas[], int n )
45 {
46   int i;
47   double suma = 0.0;
48   double promedio;
49
50   for ( i = 0 ; i <  n ; i++ )
51   {
52     suma = suma + arregloNotas[ i ];
53   }
54
55   promedio = suma / n;
56   return promedio;
57 }
58
```

```
59 double obtenerMayor( double arregloNotas[ ], int n )
60 {
61   int i;
62   double mayor = arregloNotas[ 0 ];
63
64   for ( i = 1 ; i < n ; i++ )
65   {
66     if ( arregloNotas[ i ] > mayor )
67     {
68       mayor = arregloNotas[ i ];
69     }
70   }
71
72   return mayor;
73 }
74
75 double obtenerMenor( double arregloNotas[], int n )
76 {
77   int i;
78   double menor = arregloNotas[ 0 ];
79
80   for ( i = 1 ; i < n ; i++ )
81   {
82     if ( arregloNotas[ i ] < menor )
83     {
84       menor = arregloNotas[ i ];
85     }
86   }
87
88   return menor;
89 }
90
91 void mostrarResultados( double promedio,
92                         double mayor,    double menor )
93 {
94   printf( "Promedio del grupo: %.2lf\n", promedio  );
95   printf( "Nota más alta obtenida: %.2lf\n", mayor );
96   printf( "Nota más baja obtenida: %.2lf\n", menor );
97 }
```

Explicación del programa:

Este programa se escribió usando funciones y procedimientos que son llamados desde la función principal o `main`. En la función principal se declaran las variables que reciben los resultados de las funciones y posteriormente se invocan dichas funciones en el siguiente orden:

- La función `leerDatos()`, se usa para capturar las notas ingresadas por el usuario y guardarlas en el arreglo de notas. Esta función tiene un parámetro que le asigna el tamaño al arreglo; esta función opera de la misma manera en que se explicó en el ejercicio anterior.
- A continuación se invoca la función `calcularPromedio()`, cuyos parámetros son: el arreglo de notas y su tamaño. El arreglo se recorre de principio a fin con un ciclo `for`; la variable `suma` va acumulando la suma de las notas, por eso esta variable se inicializa en cero antes de ingresar al ciclo `for`. Finalizado el ciclo, se obtiene el promedio de notas del grupo al dividir la suma entre el número de estudiantes, es decir, 20, almacenado en la variable `n`.
- Luego se hace el llamado a la función `obtenerMayor()` cuyos parámetros son: el arreglo de notas y su tamaño. Esta función guarda en la variable `mayor` la primera posición del arreglo (índice 0) y luego lo recorre desde la posición 1 hasta la última posición. Para el recorrido se usa un ciclo `for`. Dentro del ciclo, se compara la celda actual del arreglo con la variable `mayor`. Si el contenido de la celda actual es mayor que la variable `mayor`, el dato de la celda se guarda en la variable `mayor` y el programa pasa a la iteración siguiente. En el caso de que la celda actual del arreglo no posea una nota mayor a la de la variable `mayor`, el ciclo ejecuta la siguiente iteración sin hacer nada, allí el programa se ubica en la siguiente posición del vector y vuelve a realizar la comparación. Finalizando el recorrido del vector, la función conocerá la mayor nota del arreglo y la retornará al programa principal.
- De igual manera trabaja la función `obtenerMenor()`, que es la siguiente instrucción invocada desde el programa principal; dentro de esta función la comparación que se hace con la estructura `if`, al interior del ciclo `for`, permite saber si lo que posee la posición actual del vector es menor a lo almacenado en la variable `menor`. Al final, esta función retornará la menor nota almacenada en el vector.
- Finalmente, el procedimiento `mostrarResultados()`, mostrará la nota promedio del grupo, la mayor nota y la menor nota almacenadas en el vector; estos son los parámetros con que opera este procedimiento y que muestra mediante la instrucción `printf`.

.:Ejemplo 6.8. *Construya un programa en Lenguaje C que guarde en vectores el nombre y el género de un grupo de* `n` *personas y, que por medio de funciones retorne la posición del vector de nombres en que está una persona así cómo su género. Utilice una 'M' para Masculino y una 'F' para femenino cuando se almacenen los géneros.*

Análisis del problema:

- **Resultados esperados:** mostrar la posición que ocupa en el vector de nombres una persona que el usuario quiere buscar y, el género que esta persona tiene.
- **Datos disponibles:** se conoce la longitud de los vectores, nombres y géneros de las `n` personas; luego de llenar los vectores, se conocerá el nombre de la persona a consultar.
- **Proceso:** Después del almacenamiento de los datos, el programa buscará la posición ocupada en el vector de nombres de un nombre que se consulte; esto se realizará implementando un ciclo que recorra el vector y una estructura de decisión que permita saber si existe coincidencia en la comparación que se haga. Si el nombre buscado está en el vector, el programa mostrará la posición donde se encuentra y su correspondiente género, el cuál está en la misma posición pero, en el vector de géneros. La solución propuesta a este ejercicio supone que cada nombre está almacenado en el vector una sola vez. Si el nombre buscado apareciera varias veces en el vector, el programa reportaría la posición del último de ellos.
- **Variables requeridas:**
  - En el programa principal
    - `tamanio`: tamaño de los arreglos
    - `posicion`: número de la celda del vector donde está el nombre buscado.
    - `nombreBuscar`: nombre de la persona a buscar en el vector de nombres.
    - `arregloNombres`: vector que almacena nombres de las personas.
    - `arregloGeneros`: vector que almacena los géneros de las personas.
  - En la función `leerNombres`
    - Parámetros:

        - ◇ `tamanio`: tamaño del arreglo.
    - Variables locales:
        - ◇ `i`: variable para controlar el ciclo.
        - ◇ `arregloNom`: vector para guardar los nombres ingresados.
- En la función `leerGeneros`
    - Parámetros:
        - ◇ `tamanio`: tamaño del arreglo.
    - Variables locales:
        - ◇ `i`: variable para controlar el ciclo.
        - ◇ `arregloG`: vector para guardar los géneros de las personas ingresadas.
- En la función `obtenerPosicion`
    - Parámetros:
        - ◇ `arregloNom`: arreglo que contiene los nombres de las personas.
        - ◇ `nombre`: nombre de la persona a buscar.
        - ◇ `tamanio`: tamaño del arreglo.
    - Variables locales:
        - ◇ `i`: variable para controlar el ciclo.
        - ◇ `pos`: almacena la posición del vector donde está el nombre buscado o, -1 si el nombre no aparece en el vector.

- En el procedimiento `mostrarResultados`
    - Parámetros:
        - ◇ `pos`: posición del vector donde está el nombre, o -1 de no encontrarse.
        - ◇ `arregloN`: vector que contiene los nombres de las personas.
        - ◇ `arregloG`: vector que contiene los genéros de las personas.

De acuerdo al análisis realizado, se propone el programa 6.8.

Programa 6.8: ArregloPersonas

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

#define dimensionarC(n) (char *) malloc(sizeof(char)*n)

char **dimensionarS     ( int n, int longitud  );
void   freeS            ( char **vector, int n );

char **leerNombres      ( int   tamanio );
char  *leerGeneros      ( int   tamanio );
int    obtenerPosicion  ( char *arregloNombres[],
                          char  nombre[], int   tamanio );
void   mostrarResultados( int   pos,      char *arregloN[],
                          char *arregloG );

int main()
{
    int    tamanio, posicion;
    char   nombreBuscar[ 20 ];
    char **arregloNombres, *arregloGeneros;

    printf ( "Ingrese la cantidad de personas: " );
    scanf  ( "%d", &tamanio );

    arregloNombres = leerNombres( tamanio );
    arregloGeneros = leerGeneros( tamanio );

    printf ( "Ingrese el nombre de la persona a buscar: " );
    fgets( nombreBuscar, 20, stdin );

    posicion = obtenerPosicion( arregloNombres,
                                nombreBuscar, tamanio );

    mostrarResultados ( posicion,
                        arregloNombres, arregloGeneros );

    freeS ( arregloNombres, tamanio );
    free  ( arregloGeneros );

    return 0;
}

char **leerNombres(int tamanio)
{
    int i;
    char **arregloNom;
```

```
49
50      arregloNom = dimensionarS ( tamanio, 20 );
51
52       for( i = 0 ; i < tamanio ; i++ )
53       {
54          printf ( "Ingrese el nombre de la persona %d: ", i);
55          fgets( arregloNom[ i ], 20, stdin );
56       }
57
58      return arregloNom;
59  }
60
61
62  char *leerGeneros(int tamanio)
63  {
64      int i;
65      char *arregloGen;
66
67      arregloGen = dimensionarC( tamanio );
68
69      for( i = 0 ; i < tamanio ; i++ )
70      {
71         do
72         {
73              printf ( "Ingrese el género %d: ", i );
74              scanf  ( " %c", &arregloGen[ i ] );
75
76              arregloGen[i] = toupper( arregloGen[i]  );
77
78              if ( arregloGen[i]!='F' && arregloGen[i]!='M' )
79              {
80                  printf("Ha ingresado un género erróneo\n");
81              }
82         } while ( arregloGen[i]!='F' && arregloGen[i]!='M' );
83      }
84
85      return arregloGen;
86  }
87
88  int obtenerPosicion( char *arregloNombres[],
89                       char  nombre[], int tamanio)
90  {
91      int i, pos;
92
93      pos = -1;
94      for( i = 0 ; i < tamanio ; i++ )
95      {
96         if ( strcmp( nombre, arregloNombres[ i ] ) == 0 )
97         {
```

```
98              pos = i;
99          }
100     }
101
102     return pos;
103 }
104
105 void mostrarResultados( int   pos,
106                         char *arregloN[], char *arregloG)
107 {
108     if (pos == -1)
109     {
110         printf( "El nombre no está en la lista" );
111     }
112     else
113     {
114         printf( "%s, está en la posición %i ",
115                 arregloN[ pos ], pos );
116         printf( "y su género es %c ", arregloG[ pos ] );
117     }
118 }
119
120 char **dimensionarS ( int n, int longitud )
121 {
122   char **vector;
123
124   vector =  (char **) malloc ( sizeof(char*) * n);
125
126   for ( int i = 0 ; i < n ; i++ )
127   {
128     vector[i] = dimensionarC ( longitud );
129   }
130
131   return vector;
132 }
133
134 void freeS ( char **vector, int n )
135 {
136   for ( int i = 0 ; i < n ; i++ )
137   {
138     free ( vector [ i ] );
139   }
140
141   free( vector );
142 }
```

Explicación del programa:

Desde la función `main` del programa, se llevan a cabo las siguientes tareas:

- Se hace la declaración de las variables requeridas para el funcionamiento general (líneas 20 a la 22).
- Luego, se solicita el tamaño de los vectores, pidiendo que se ingrese la cantidad de personas a procesar (líneas 24 y 25).
- Después, se hace el llamado a las funciones, `leerNombres()` y `leerGeneros()` (líneas 27 y 28), a través de las cuales se solicitan los datos de las personas y se almacenan en los arreglos respectivos.

  La función `leerGeneros()` contiene una validación que, por medio del ciclo `do-while`, hace que el usuario ingrese solo una 'F' de género Femenino o una 'M' de género Masculino.
- A continuación en el programa, se pide ingresar el nombre de la persona a buscar en el vector de nombres.
- Luego, con la función `obtenerPosicion()` se busca la posición en la que debe estar el nombre de la persona que se está buscando y que es enviada como parámetro. La función opera de la siguiente manera:

  - Utilizando un ciclo `for` se recorre el vector de nombres. Antes de inciciar el ciclo, se guardó un -1 en la variable `pos`, ya que aún no se tiene una posición válida del arreglo para retornar, lo que significa que, todavía no se ha encontrado nada. La estructura `if` que está al interior del ciclo `for` permite determinar si el nombre buscado es igual al que está ubicado en la posición actual del vector; de ser así, la variable `pos` que está dentro del `if` guarda la posición actual donde está el nombre.
  - Al final de la función `obtenerPosicion()` se retorna la variable `pos` que tiene la posición del nombre del estudiante encontrado en el arreglo. Si la función no obtuvo el nombre que se estaba buscando, la variable pos seguirá con -1 y será lo que retornará esta función.

- Por último, desde la función `main` se invoca el procedimiento `mostrarResultados()`, que recibe como parámetros: la posición encontrada en la función anterior, el vector de nombres y el vector de géneros. A partir de estos parámetros, el procedimiento imprime

el nombre, la posición y el género de la persona que se buscó; si la la variable `pos` es igual a -1, significa que no se encontró el nombre, por eso se imprime un mensaje informando este hecho.

.:Ejemplo 6.9. *Construya un programa en Lenguaje C utilizando funciones, procedimientos y vectores que perita almacenar las estaturas y los géneros de un conjunto de* `n` *personas y que determine el promedio de estaturas de las mujeres y el porcentaje de hombres ingresados.*

Análisis del problema:

- **Resultados esperados:** mostrar el promedio de las estaturas de las mujeres y el porcentaje de hombres que se encuentran en el arreglo.
- **Datos disponibles:** se conocen, la cantidad de personas, la estatura y el género de estas.
- **Proceso:** el programa se organizará con funciones; las primeras serán las funciones para ingresar los datos de las `n` personas y almacenarlos en los vectores. Luego, habrá una función que calcule el promedio de las estaturas de las mujeres sumando las estaturas de ellas y dividiendo entre el total de mujeres. Después, se tendrá una función que determine el porcentaje de hombres ingresados en el vector; esto implica contar el número de personas con género masculino y dividirlo entre el total de personas ingresadas. Por último, el programa mostrará mediante un procedimiento los resultados obtenidos.
- **Variables requeridas:**
  - En el programa principal
    - `numeroPersonas`: servirá para dar tamaño a los arreglos.
    - `arregloEstatura`: arreglo que almacena las estaturas de las personas.
    - `arregloGenero`: arreglo que almacena los géneros de las personas.
    - `promEstMujeres`: variable que almacena el promedio de las estaturas de las mujeres.
    - `porcHombres`: variable que almacena el porcentaje de hombres ingresados al vector.

- En la función `leerEstaturas`
  - Parámetros:
    - ◇ `tamanio`: tamaño del vector.
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo.
    - ◇ `arregloE`: vector para almacenar las estaturas de las personas.
- En la función `leerGeneros`
  - Parámetros:
    - ◇ `tamanio`: tamaño del vector.
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo.
    - ◇ `arregloG`: vector para almacenar los géneros de las personas ingresadas.
- En la función `obtenerPromedio`
  - Parámetros:
    - ◇ `arregloE`: vector que contiene las estaturas de las personas.
    - ◇ `arregloG`: vector que contiene los géneros de las personas.
    - ◇ `tamanio`: tamaño de los vectores.
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo.
    - ◇ `suma`: almacena la suma de las estaturas de las mujeres.
    - ◇ `promedio`: almacena el promedio de las estaturas de las mujeres.
    - ◇ `contadorMujeres`: variable para almacenar la cantidad de mujeres en el arreglo.
- En la función `obtenerPorcentajeHombres`
  - Parámetros:
    - ◇ `arregloG`: vector con los géneros de las personas.
    - ◇ `tamanio`: tamaño del vector.
  - Variables locales:
    - ◇ `i`: variable de control del ciclo.
    - ◇ `hombres`: cantidad de hombres que hay en el arreglo.
    - ◇ `porcentaje`: variable que almacena el porcentaje de hombres encontrados en el vector.

- En el procedimiento `mostrarResultados`
  - Parámetros:
    - ◇ `promEstatura`: almacena el promedio de las estaturas de las mujeres.
    - ◇ `porcHombres`: almacena el porcentaje de hombres que hay en el vector.

Conforme al análisis realizado, se propone el programa 6.9.

Programa 6.9: ArregloEstaturas

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4
5  #define dimensionarC(n) (char *)  malloc(sizeof(char)*n)
6  #define dimensionarD(n) (double*) malloc(sizeof(double)*n)
7
8  double *leerEstaturas           ( int    tamanio         );
9  char   *leerGeneros             ( int    tamanio         );
10 double  obtenerPromedio         ( double arregloEst[],
11                                   char   arregloGen[],
12                                   int    tamanio         );
13 double  obtenerPorcentajeHombres( char   arregloGen[],
14                                   int    tamanio         );
15 void    mostrarResultados       ( double promEstMujeres,
16                                   double porcHombres     );
17
18 int main()
19 {
20     int    numeroPersonas;
21     double *arregloEstaturas;
22     char   *arregloGeneros;
23     double promEstMujeres, porcHombres;
24
25     printf ( "Ingrese la cantidad de personas: " );
26     scanf  ( "%d", &numeroPersonas );
27
28     arregloEstaturas = leerEstaturas( numeroPersonas );
29     arregloGeneros   = leerGeneros  ( numeroPersonas );
30
31     promEstMujeres = obtenerPromedio ( arregloEstaturas,
32                                        arregloGeneros,
33                                        numeroPersonas);
34
35     porcHombres = obtenerPorcentajeHombres (arregloGeneros,
36                                             numeroPersonas);
37
```

```
38      mostrarResultados ( promEstMujeres, porcHombres );
39
40      free ( arregloEstaturas );
41      free ( arregloGeneros );
42
43      return 0;
44  }
45
46  double *leerEstaturas(int tamanio)
47  {
48      int i;
49      double *arregloEst;
50
51      arregloEst = dimensionarD( tamanio );
52
53      for( i = 0 ; i < tamanio ; i++ )
54      {
55         printf ( "Ingrese estatura de la persona %d: ", i );
56         scanf  ( "%lf", &arregloEst[ i ] );
57      }
58      return arregloEst;
59  }
60
61  char *leerGeneros( int tamanio )
62  {
63      int i;
64      char *arregloGen;
65
66      arregloGen = dimensionarC( tamanio );
67
68      for( i = 0 ; i < tamanio ; i++ )
69      {
70         printf ( "Ingrese el género %d: ", i );
71         scanf  ( " %c", &arregloGen[ i ] );
72
73         arregloGen[ i ] = toupper( arregloGen[ i ] );
74      }
75
76      return arregloGen;
77  }
78
79  double obtenerPromedio( double arregloEst[],
80                          char   arregloGen[],
81                          int    tamanio )
82  {
83      int    i, contadorMujeres;
84      double suma, promedio;
85
86      suma = 0.0;
```

```
87      contadorMujeres = 0;
88
89      for( i = 0 ; i < tamanio ; i++ )
90      {
91         if ( arregloGen[i]=='F' )
92         {
93             suma = suma + arregloEst[i];
94             contadorMujeres = contadorMujeres + 1;
95         }
96      }
97
98      promedio = suma / contadorMujeres;
99
100     return promedio;
101 }
102
103 double obtenerPorcentajeHombres( char arregloGen[],
104                                  int  tamanio)
105 {
106     int    i;
107     int    porcentaje;
108     double contadorHombres = 0;
109
110     for( i = 0 ; i < tamanio ; i++ )
111     {
112        if ( arregloGen[i]=='M' )
113        {
114            contadorHombres = contadorHombres + 1;
115        }
116     }
117
118     porcentaje = contadorHombres / tamanio * 100.0;
119
120     return porcentaje;
121 }
122
123 void mostrarResultados( double promEstMujeres,
124                         double porcHombres )
125 {
126     printf( "Estatura promedio de las mujeres %.2f\n",
127             promEstMujeres );
128     printf( "Porcentaje de Hombres %.2f%%", porcHombres );
129 }
```

Explicación del programa:

Al inicio de la función `main`, se declaran las variables y se pide la cantidad de personas; con este dato (almacenado en la variable `numeroPersonas`), se inicializan los vectores que almacenarán las

estaturas y los géneros de las personas.

Posteriomente, se hace el llamado a las funciones `leerEstaturas()` y `leerGeneros()` con las que se leen los datos de estaturas y géneros y se almacenan en las diversas posiciones de los vectores destinados a este fín.

Luego, se invoca la función `obtenerPromedio()`, que se encarga de calcular el promedio de estaturas de las mujeres; esto lo hace recorriendo el vector de estaturas por medio de un ciclo `for`; al interior de este ciclo, se utiliza una estructura `if` para determinar si el género ubicado en la posición `i` del vector de géneros es femenino; en este caso, se acumula la estatura que está en la posición `i` del vector de estaturas en la variable `suma` y se incrementa el contador de mujeres. Al terminar el ciclo, se obtiene el promedio de estaturas al dividir la variable `suma` entre la variable `contadorMujeres` y, este resultado se retorna a la función principal.

Continuando con la ejecución del programa, se llama a la función `obtenerPorcentajeHombres()`, que utiliza un ciclo `for` para recorrer el vector de géneros y contar la cantidad de hombres que hay en el arreglo, utilizando para ello una estructura `if`; cuando se obtiene esta cantidad, se divide entre el total de personas y se multiplica por 100.0, lo cual permite obtener el porcentaje de hombres. este dato será el retornado por la función.

Por último, con el procedimiento `mostrarResultados()`, se muestran: promedio de estaturas de las mujeres y el porcentaje de hombres.

.:Ejemplo 6.10. *Construya un programa en Lenguaje C que, a través de funciones y procedimientos guarde en un vector 4 números enteros y luego ordene estos números de menor a mayor aplicando el método de ordenamiento llamado "**Ordenamiento por selección**". El programa debe mostrar el vector con los datos ingresados inicialmente (en desorden) y el vector con los datos ya ordenados.*

Análisis del problema:

- **Resultados esperados:** mostrar un arreglo de cuatro números enteros ordenados de menor a mayor.
- **Datos disponibles:** se conocen los 4 números enteros que se van a almacenar en el arreglo.

- **Proceso:** se ingresan los 4 números al vector; a continuación se muestra este vector original con los números en desorden; posteriormente, se aplica el método de ordenamiento por selección al vector para que lo ordene y, se vuelve a mostrar el vector.

  Antes de implementar el programa en Lenguaje C, es fundamental comprender los siguientes tres conceptos:

  Primero, el propósito de un algoritmo de ordenamiento es recibir un conjunto de datos y entregarlos ordenados en forma ascendente o descendente según se haya especificado.

  Segundo, existen diversas formas de realizar el ordenamiento, conocidas como algoritmos de ordenamiento, uno de estos algoritmos es el conocido como "ordenamiento por selección".

  Tercero, el algoritmo de ordenamiento por selección tiene varias formas de implementación. Enseguida se muestra una de estas formas. Un ejemplo permitirá ilustrar su funcionamiento; el ejemplo consiste en ordenar un arreglo con los números enteros: 7 2 8 1.

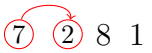
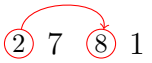
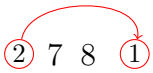
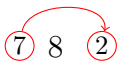
  En esta versión del "ordenamiento por selección" se utilizan dos ciclos[2]. El primer ciclo va indicando la posición del primer elemento a ser comparado (círculo de donde sale la flecha); el segundo ciclo va mostrando la posición del segundo elemento con el que se debe comparar el primero (círculo donde llega la flecha); por cada comparación, se determina si el primer número es mayor al segundo, de ser así, los elementos se intercambian de posición, si no es así, se pasa al siguiente número. Cuando se haya comparado el elemento que indica el primer ciclo con todas las posiciones, el primer ciclo pasa a la siguiente posición y realiza nuevamente las comparaciones con el segundo ciclo y, así hasta llegar a la penúltima posición[3].

[2]Comúnmente se usa el ciclo `for`

[3]Es la última comparación posible, la penúltima posición con la última.

Ejemplo de ordenamiento

`arreglo` $->$ **7 2 8 1**

$Ciclo_1$	$Ciclo_2$	**Comparación**
1	2	
	3	
	4	
2	3	
	4	
3	4	

`arreglo` $->$ **1 2 7 8**

- **Variables requeridas:**
  - En el programa principal
    - Constante requerida:
      - `MAX`: constante que toma el valor de 4.
    - Variable:
      - `numero`: vector que contiene los números a ordenar.
  - En la función `leerArreglo`
    - Parámetros:
      - `tamanio`: tamaño del arreglo.
    - Variables locales:
      - `i`: variable para controlar el ciclo.
      - `arregloE`: vector que almacena los números enteros ingresados por el usuario.
  - En la función `imprimirArreglo`

- Parámetros:
  - ◇ `titulo`: texto que almacena los datos del vector y es mostrado al usuario.
  - ◇ `arreglo`: vector que contiene los números ingresados.
  - ◇ `n`: tamaño del arreglo.

- Variables locales:
  - ◇ `i`: variable que controla el primer ciclo.
  - ◇ `cadena`: variable que almacena los elementos del arreglo como una cadena de caracteres, concatenando los elementos.

- En la función `ordenar`

- Parámetros:
  - ◇ `arreglo`: arreglo que contiene los números a ordenar.
  - ◇ `n`: tamaño del vector.

- Variables locales:
  - ◇ `i`: variable para controlar el primer ciclo.
  - ◇ `j`: variable para controlar el segundo ciclo.
  - ◇ `auxiliar`: variable auxiliar para intercambiar dos elementos del vector.

De acuerdo al análisis planteado, se propone el programa 6.10.

Programa 6.10: OrdenamientoSeleccion

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define dimensionarI(n) (int *)  malloc(sizeof(int)*n)
5
6  int *leerArreglo    ( int n );
7  void imprimirArreglo( char cadena[20], int numero[], int n);
8  void ordenar        ( int numero[], int n );
9
10 int main()
11 {
12     const int MAX=4;
13     int       *numero;
```

```
14
15      numero = leerArreglo ( MAX );
16
17      imprimirArreglo ( "Arreglo original: ", numero, MAX );
18
19      ordenar(numero, MAX);
20
21      imprimirArreglo ( "Arreglo Ordenado: ", numero, MAX );
22
23      free ( numero );
24
25      return 0;
26  }
27
28  int *leerArreglo(int n)
29  {
30      int i;
31      int *arreglo;
32
33      arreglo = dimensionarI ( n );
34
35      for( i = 0 ; i < n ; i++ )
36      {
37         printf ( "\n Ingrese el número %d: ", i );
38         scanf  ( "%d", &arreglo[ i ] );
39      }
40
41      return arreglo;
42  }
43
44  void ordenar( int numero[], int n )
45  {
46      int i, j;
47      int auxiliar;
48
49      for( i = 0 ; i < n - 1 ; i++ )
50      {
51          for( j = i + 1 ; j < n ; j++ )
52          {
53              if( numero[i] > numero[j] )
54              {
55                  auxiliar  = numero[i];
56                  numero[i] = numero[j];
57                  numero[j] = auxiliar;
58              }
59          }
60      }
61  }
62
```

```
63 void imprimirArreglo( char cadena[20], int numero[], int n)
64 {
65     int i;
66
67     printf("\n\n %s", cadena);
68
69     for( i = 0 ; i < n ; i++ )
70     {
71       printf(" %i", numero[i]);
72     }
73 }
```

Explicación del programa:

Inicialmente se hizo la declaración de las variables. Luego, invoca a la función `leerArreglo()`, que se usa para realizar la captura de los `MAX` números enteros (4) y almacenarlos en el vector.

```
12     const int MAX=4;
13     int      *numero;
14
15     numero = leerArreglo ( MAX );
```

A continuación, se imprime el vector a través del procedimiento `imprimirArreglo()`; luego se llama a la función `ordenar()`, encargada de hacer el ordenamiento de los elementos del vector; finalmente se imprime otra vez el vector ya ordenado.

```
17     imprimirArreglo ( "Arreglo original: ", numero, MAX );
18
19     ordenar(numero, MAX);
20
21     imprimirArreglo ( "Arreglo Ordenado: ", numero, MAX );
```

Si se analiza el código que conforma la función `ordenar()` se podrá concluir que este coincide con la explicación hecha previamente durante el análisis del problema.

```
44 void ordenar( int numero[], int n )
45 {
46     int i, j;
47     int auxiliar;
48 
49     for( i = 0 ; i < n - 1 ; i++ )
50     {
51         for( j = i + 1 ; j < n ; j++ )
52         {
53             if( numero[i] > numero[j] )
54             {
55                 auxiliar  = numero[i];
56                 numero[i] = numero[j];
57                 numero[j] = auxiliar;
58             }
59         }
60     }
61 }
```

En la función `ordenar()`, se emplean dos ciclos; el primero recorre las posiciones del vector, desde la primera hasta la penúltima y el segundo empieza en la segunda posición; de esta manera se van comparando, al principio, la primera con la segunda posición del vector, luego la primera con la tercera y así sucesivamente. Cuando el ciclo externo lleva a cabo la segunda iteración, el programa compara la segunda posición del vector con la tercera, cuarta, hasta `n` posición, intercambiando los elementos, si cumplen con la condición de la estructura `if` que se encuentra al interior del segundo ciclo.

El intercambio del contenido de dos posiciones del vector (líneas de la 53 a 55) requiere del uso de una variable (`auxiliar`) que almacene temporalmente el contenido de una posición (cualquiera de la dos a intercambiar); luego, en esa posición se guarda el contenido de la segunda posición y, en esta segunda posición se ubica lo que tiene la variable `auxiliar`.

.:Ejemplo 6.11. *Escriba un programa que guarde un conjunto de `n` números enteros en un vector, los ordene ascendentemente a través del método de ordenamiento por selección y, que lleve a cabo la búsqueda de un número cualquiera utilizando el denominado método de* ***"Búsqueda binaria"*** *indicando la posición que ocupa el número en el vector; si el número no se encuentra en el vector, se debe informar al usuario. Use funciones para escribir el programa.*

Análisis del problema:

- **Resultados esperados:** mostrar la posición en la que se encuentra un número que se busca en el vector.
- **Datos disponibles:** se conocen los números a almacenar en el vector y el número que se va a buscar posteriormente.
- **Proceso:** el algoritmo de "Búsqueda binaria" es una estrategia eficiente de buscar un elemento en un vector, ya que en la mayoría de las ocasiones, encuentra el elemento realizando menos iteraciones que con la búsqueda secuencial. El algoritmo inicia ubicándose en la posición del medio y, por medio de una comparación, define si ha encontrado el elemento o, si debe realizar la búsqueda a la derecha del vector (elementos mayores a él) o en los elementos del lado izquierdo (elementos menores a él); en cada iteración se excluyen la mitad de los elementos del vector; a través de este procedimiento, se busca rápidamente el elemento o se concluye que no está en el vector.

  Luego de ingresar los datos en el vector, el programa ordena este arreglo, usando el método de ordenamiento por selección explicado en el ejercicio anterior. Enseguida, el programa solicita un número a buscar en el vector y, mediante una función que implemente el método de búsqueda binaria, realiza la búsqueda del número. Al final, el programa muestra la posición del vector donde se encuentra el número o, un mensaje informando que el número no está en el arreglo. Para este ejercicio, el tamaño del vector será de 13 posiciones.

  - En la función principal
    - Constante requerida:
      - `MAX`: constante que contiene el tamaño del vector, es decir, 13.
    - Variables requeridas:
      - `numero`: vector que almacena los MAX números a ordenar.
      - `numeroBuscar`: número que se va a buscar.
      - `posicion`: posición del vector donde está el número buscado o, -1 de no estar almacenado en el vector.

- En la función `leerArreglo`
  - Parámetros:
    - ◇ `tamanio`: tamaño del vector.
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo.
    - ◇ `arregloE`: arreglo que contiene los números ingresados por el usuario.

- En la función `ordenar`
  - Parámetros:
    - ◇ `arreglo`: vector que contiene los números ingresados por el usuario.
    - ◇ `n`: tamaño del vector.
  - Variables locales:
    - ◇ `i`: variable para controlar el primer ciclo.
    - ◇ `j`: variable para controlar el segundo ciclo.
    - ◇ `auxiliar`: variable auxiliar para intercambiar dos posiciones del vector.

- En la función `buscarElemento`
  - Parámetros:
    - ◇ `arreglo`: vector que contiene los números ingresados por el usuario.
    - ◇ `n`: tamaño del vector.
    - ◇ `numeroBuscar`: número que se va a buscar.
  - Variables locales:
    - ◇ `primero`: punto inicial del rango de búsqueda.
    - ◇ `ultimo`: punto final del rango de búsqueda.
    - ◇ `centro`: punto medio del rango de búsqueda.
    - ◇ `posicion`: posición en donde se ubica el número buscado o, -1 de no encontrarlo.

El siguiente ejemplo explica el método de búsqueda binaria:

Ejemplo de búsqueda binaria

numero -> **1 3 5 9 14 21 45 63 73 87 89 90 97**
elemento = **73**

Ciclo	**Comparación**
1	1 3 5 9 14 21 45 **63** **73** 87 **89** **90** **97** dado que **45** es menor que **73**
2	1 3 5 9 14 21 45 **63** **73** 87 89 90 97 dado que **87** es mayor que **73**
3	1 3 5 9 14 21 45 63 **73** 87 89 90 97 Dato encontrado!!!

De acuerdo al análisis planteado, se propone el programa 6.11.

Programa 6.11: BusquedaBinaria

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  #define dimensionarI(n) (int *)  malloc(sizeof(int)*n)
6
7  int *leerArreglo    ( int n );
8  int buscarElemento  ( int numero[], int n, int numeroBuscar);
9  void ordenar        ( int numero[], int n );
10
11 int main()
12 {
13     const int MAX = 13;
14     int       *numero;
15     int        numeroBuscar, posicion;
16
17     numero = leerArreglo( MAX );
18
19     printf ( "Ingrese el número a buscar: " );
20     scanf  ( "%d", &numeroBuscar );
21
22     ordenar ( numero, MAX );
```

```
23
24      posicion = buscarElemento( numero, MAX, numeroBuscar );
25
26      if (posicion >= 0)
27      {
28          printf ( "El número buscado se encontró en: %d ",
                posicion);
29      }
30      else
31      {
32          printf ( "El número no se encuentra en el arreglo");
33      }
34
35      free ( numero );
36
37      return 0;
38  }
39
40  int *leerArreglo( int n )
41  {
42      int i;
43      int *arreglo;
44
45      arreglo = dimensionarI( n );
46
47      for( i = 0 ; i < n ; i++ )
48      {
49         printf ( "Ingrese el número %d: ", i );
50         scanf  ( "%d", &arreglo[ i ] );
51      }
52      return arreglo;
53  }
54
55  void ordenar( int numero[], int n )
56  {
57      int i, j;
58      int auxiliar;
59
60      for( i = 0 ; i < n - 1 ; i++ )
61      {
62          for( j = i+1 ; j < n ; j++ )
63          {
64              if( numero[i] > numero[j] )
65              {
66                  auxiliar  = numero[i];
67                  numero[i] = numero[j];
68                  numero[j] = auxiliar;
69              }
70
```

```
71          }
72      }
73  }
74
75
76  int buscarElemento( int numero[], int n, int numeroBuscar )
77  {
78      int primero, ultimo, centro;
79      int posicion = -1;
80
81      primero = 0;
82      ultimo  = n-1;
83
84      while( posicion == -1 && primero <= ultimo )
85      {
86          centro = (primero + ultimo) / 2;
87
88          if( numeroBuscar == numero[centro] )
89          {
90              posicion = centro;
91          }
92          else
93          {
94              if( numeroBuscar < numero[centro] )
95              {
96                  ultimo = centro - 1;
97              }
98              else
99              {
100                 primero = centro + 1;
101             }
102         }
103     }
104
105     return posicion;
106 }
```

Explicación del programa:

Después de declarar las variables, se leen los 13 elementos (constante `MAX`) a través de la función `leerArreglo()` (línea 17), y que retorna los números leídos a la variable `numero`; esta función opera como en los ejercicios anteriores; Posteriormente, se pide ingresar el número a buscar empleando con las instrucciones `printf` y `scanf` (líneas 19 y 20); A continuación, el vector es ordenado con la función `ordenar()` (línea 22) descrita ya en el anterior ejemplo; luego, se realiza la búsqueda del elemento por medio de la función `buscarElemento()` (línea 24), dependiendo de

lo que retorne la función, se imprime la posición del vector donde está el elemento buscado o, se imprime que el número solicitado no está en el arreglo (líneas de la 26 a la 33).

La función `buscarElemento()` contiene el código en Lenguaje C de la búsqueda binaria. La función trabaja así: Se tiene una estructura cíclica `while` cuya condición es que la variable posición sea -1 y que la variable `primero` sea menor o igual a la variable `ultimo`; si esta condición es false, indicaría que se realizó la búsqueda en el arreglo y se encnotró o no el elemento.

Al interior del ciclo `while` está la instrucción que obtiene el centro para cada iteración y una instrucción `if` para determinar si se encontró el elemento buscado; de ser asi, se ejecuta la instrucción `posicion=centro`; si no se encontró, se reubica la primera o última posición ya sea a la izquierda o derecha del centro (dependiendo si el número buscado es menor o mayor) del vector; de esta forma se descarta la búsqueda en alguno de los dos lados. Por esta razón el vector debe estar ordenado.

La instrucción:

```
58      centro = (primero + ultimo) / 2;
```

Parte o divide el vector en dos secciones y, reubica el centro del vector, lo cual le permite al algoritmo determinar si la búsqueda del elemento se hace del lado izquierdo o derecho del mismo. Por cada vuelta del ciclo se recalcula el centro entre aquellos elementos más cercanos al número buscado.

Esta instrucción contiene tres variables de tipo `int`, por tanto, el resultado será entero, esto es, si la variable `primero` es igual a 1 y la variable `ultimo` es igual a 6 el resultado es 3 y no 3.5.

```
        (1 + 6) / 2
        7 / 2
        3
```

Actividad 6.1

Para los siguientes ejercicios propuestos, construya programas en Lenguaje C, usando funciones y procedimientos en su solución.

1. Haga un programa que almacene en diferentes vectores los nombres, géneros y edades de un grupo de n personas. El programa deberá obtener:

   a) El número de personas de género masculino.

   b) El número de personas de género femenino que son mayores de edad

   c) El promedio de edad de los hombres.

   d) El nombre de la mujer más joven.

2. Escriba un programa que almacene nombres y estaturas de un grupo de n personas y que luego ordene y muestre los nombres y las estaturas de las persona:

   a) De manera ascendente, es decir, de la persona más baja a la más alta.

   b) De forma descendente, esto es, de la persona más alta a la más baja

3. Desarrolle un programa que almacene n número enteros en un vector y que luego determine si un número cualquiera ingresado por el usuario hace parte del arreglo. Si el número aparece en el vector, el programa deberá informar la posición que ocupa; si no está, el programa mostrará un mensaje informando la situación.

4. Haga un programa que almacene en un vector n números enteros entre 40 y 80. El programa tendrá que hallar los números pares y los impares que quedaron almacenados en el vector y guardarlos en otros vectores. Al final, el programa mostrará estos vectores.

5. El profesor de la materia de “Lenguaje de Programación” necesita un programa en el que pueda almacenar los nombres de los 30 estudiantes del curso y, las 5 notas obtenidas por cada estudiante durante el semestre. El programa deberá hacer lo siguiente:

a) obtener la nota definitiva del alumno, que se calcula como la media aritmética de las 5 notas del estudiante. Estas definitivas se almacenarán en un vector.

b) Hallar al estudiante (nombre) con la mayor nota definitiva.

c) Determinar los estudiantes (nombres) y, almacenarlos en un arreglo que perdieron la materia y la tendrán que repetir. Un estudiante pierde la materia si su nota definitiva es inferior a 2.0. (Dos punto cero).

d) Determinar el nombre de los alumnos (y almacenarlos en un arreglo) que se quedaron habilitando la asignatura. Un estudiante habilita la materia si su nota definitiva es inferior a 3.0. pero superior a 2.0, en otras palabras, si la nota definitiva está entre 2.0 y 2.99.

e) Encontrar el porcentaje de alumnos que ganaron la materia.

6. Una clínica de control al sobrepeso requiere de un programa en el que puedan almacenar los nombres y los pesos tomados durante un periodo de tiempo a un grupo de n pacientes. en el periodo, cada paciente es pesado 3 veces (una pesada inicial, una intermedia y una pesada final), esto para observar su evolución durante ese periodo. el programa debe mostrar:

   a) El peso ganado o perdido por un paciente durante el periodo.

   b) El número de pacientes que perdieron peso entre la pesada inicial y la pesada intermedia.

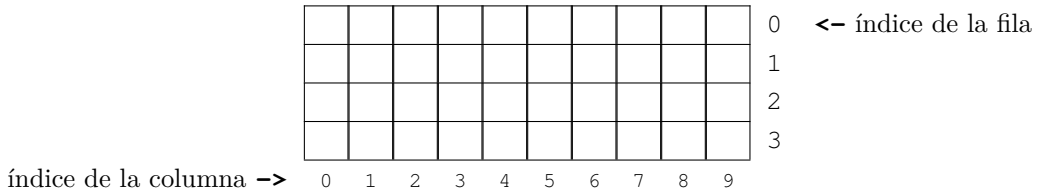
   c) Imagine que se ha guardado en otro vector el objetivo de cada paciente: ganar o perder peso durante el periodo. Encuentre el porcentaje de pacientes que lograron el objetivo.

6.2. Matrices

Matríz es un segundo nombre que se da a los arreglos bidimensionales. Las matrices, como los vectores, son conjuntos de celdas de memoria que guardan temporalmente datos del mismo tipo, pero dentro de filas y columnas que conforman tablas; es por esto que se denominan arreglos bidimensionales. Las matrices son variables suscritas que deben declararse asignándoles un nombre con un identificador válido y un tipo de datos de los que maneja Lenguaje C.

Las matrices son apropiadas para solucionar aquellos problemas en los que es necesario almacenar varios datos de varios objetos, por ejemplo, imagine que hay que almacenar las 10 notas parciales de cada uno de los 4 estudiantes de un grupo, en total se necesitan 40 espacios de memoria, que requerirían de la declaración de 40 variables diferentes. Sin embargo, con una matriz o arreglo bidimensional también sería posible almacenar esta cantidad de datos.

La imagen a continuación, ilustra cómo se vería una matriz de 4 filas y de 10 columnas.



Cada "cajón" es una celda o posición que requiere de dos coordenadas `(fila, columna)` para identificarlo. En Lenguaje C, siempre se menciona primero la fila y luego la columna. Por ejemplo, imagine una matriz de m filas y n columnas, el primer indice especifica la fila y el segundo la columna.

$$matriz = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}$$

6.2.1 Declaración de una matriz

Como toda variable, las matrices deben declararse antes de poderse utilizar.Una variable suscrita de una matriz se diferencia de la de un vector por la cantidad de asteriscos en su declaración.

```
int    **matrizEdades;
double **matrizSalarios;
char   **matrizNombres;
```

Los dos asteriscos se usan para declarar una variable que apunta a otra variable que apunta a datos (vector de vector de datos, es decir, una

matriz). Obviamente, luego de declarar la matriz, deberá especificarse su tamaño, es decir, la cantidad de filas y columnas que tendrá. Esto se lleva a cabo de la misma manera en que se hizo con los arreglos unidimensionales, o sea, a través de la función `dimensionarXX` como aparece en los siguientes ejemplos:

```
int    **matrizEdades;
double **matrizSalarios;
char   **matrizNombres;

matrizEdades   = dimensionarMI(  4,  3 );
matrizSalarios = dimensionarMD( 10, 12 );
matrizNombres  = dimensionarS( 10, 10 );
```

Analice que `dimensionarS` ya se había trabajado en apartados anteriores. Ahora bien, la utilización de `dimensionarMI` y `dimensionarMD` se puede observar en el segmento de código siguiente:

```
int **dimensionarMI ( int filas, int columnas )
{
  int **vector;

  vector =  (int **) malloc ( sizeof(int*) * filas );

  for ( int i = 0 ; i < filas ; i++ )
  {
    vector[i] = dimensionarI ( columnas );
  }

  return vector;
}
```

```
double **dimensionarMD ( int filas, int columnas )
{
  double **vector;

  vector =  (int **) malloc ( sizeof(double*) * filas );

  for ( int i = 0 ; i < filas ; i++ )
  {
    vector[i] = dimensionarD ( columnas );
  }

  return vector;
}
```

De manera análoga, siempre que se cree un vector o una matriz con `dimensionarD` se hace necesario liberar la memoria con su respectivo comando, tal y como sucede con `free`;

```
void freeMI ( int **vector, int columnas )
{
  for ( int i = 0 ; i < columnas ; i++ )
  {
    free ( vector [ i ] );
  }

  free( vector );
}
```

```
void freeMD ( double **vector, int columnas )
{
  for ( int i = 0 ; i < columnas  ; i++ )
  {
    free ( vector [ i ] );
  }

  free( vector );
}
```

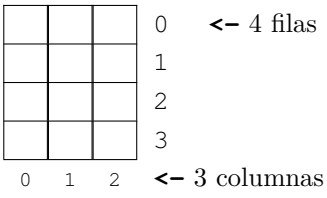
O también, Lenguaje C permite especificar el tamaño cuando se está declarando la matriz, como se ve a continuación:

```
int    matrizEdades  [ 3 ][ 10 ];
double matrizSalarios[ 15 ][ 5 ];
```

Recuerde que, en todos los casos, el número que va en el primer juego de corchetes indica el número de filas de la matriz, mientras que el número en el segundo juego de corchetes especifica la cantidad de columnas.

```
int matrizEdades [ 4 ][ 3 ];
```

Gráficamente, la matriz que se acaba de declarar, `matrizEdades` se vería de la siguiente forma:



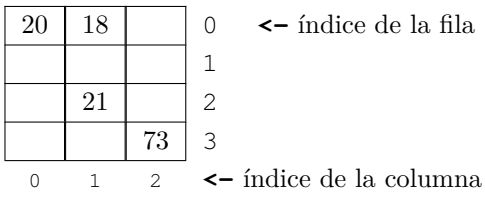
6.2.2 Almacenamiento de datos en una matriz

En una matriz los datos se guardan en cada celda, por lo que es necesario indicar la celda en la que se va a guardar el dato; esto se logra a través de

números índice para la fila y la columna en la que está ubicada la celda, como se observa enseguida:

```
matrizEdades [0][0] = 20
matrizEdades [0][1] = 18
matrizEdades [2][1] = 21
matrizEdades [3][2] = 73
```

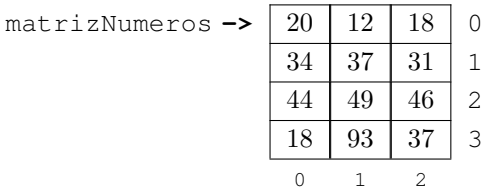
Con estas asignaciones, se guardaría un 20 en la celda que se está en la primera fila y primera columna , un 18 en la celda ubicada en la primera fila segunda columna, un 21 en la celda en la tercera fila segunda columna y un 73 en la celda que ubicada en la cuarta fila tercera columna. Así, la matriz de edades se vería de la siguiente forma:



Lenguaje C permite almacenar los datos directamente. Por ejemplo, imagine una matriz de 3 filas por 3 columnas, directamente, los datos se guardarían así:

```
int matrizNumeros[4][3] = { { 20, 12, 18},
                            { 34, 37, 31},
                            { 44, 49, 46},
                            { 18, 93, 37} };
```

Donde cada juego de llaves interiores especifica una fila de la matriz.



6.2.3 Recorrido de una matriz

Recorrer una matriz significa visitar cada una de las celdas que la componen, esto se hace para ingresar datos o para consultar los ya almacenados. El recorrido de una matriz puede hacerse de diversas formas,

a saber: lo más común, es recorrer primero todas las celdas de la primera fila y, al llegar a la última celda de esta fila, pasar a la primera celda de la segunda fila y así sucesivamente hasta llegar a la última celda ubicada en la última fila y última columna.

Otra manera de recorrer una matriz sería pasar primero por las celdas que conforman la primera columna desde la primera fila, luego pasar a la primera celda de la segunda columna y recorrer toda la columna y así hasta llegar a la última celda de la matriz en la última fila y última columna. El programador podría idearse muchas otras formas de recorrer una matriz.

En los ejercicios de este libro, el recorrido se hará primero visitando todas las celdas de cada fila. Este tipo de recorrido requiere de dos ciclos anidados, uno externo para avanzar de una fila a otra y, uno interno que pase de las celdas de una columna a la siguiente. A continuación, se encuentra un segmento de código para recorrer una matriz y llenarla con datos ingresados por el usuario:

```
const int NUMERO_FILAS    = 4;
const int NUMERO_COLUMNAS = 5;
int       i, j;

int matriz[ NUMERO_FILAS ][ NUMERO_COLUMNAS ];

for ( i = 0 ; i < NUMERO_FILAS ; i++ )
{
   for ( j = 0 ; j < NUMERO_COLUMNAS ; j++ )
   {
     printf( "Ingrese la posición ( %d, %d ): ", i, j );
     scanf( "%d", &matriz[ i ][ j ] );
   }
}
```

En este segmento de código, se destaca lo siguiente:

- El ciclo externo recorre las filas, mientras que el ciclo interno recorre las columnas.
- El ciclo interno se ejecuta más rápidamente. Cuando el ciclo interno finalice su ejecución, el algoritmo pasará al ciclo externo que lo llevará a una fila siguiente.
- La constante `NUMERO_FILAS` guarda el número de filas de la matríz. Con la condición del ciclo externo, el recorrido se limita al número de filas que tiene la matriz.

- La constante NUMERO_COLUMNAS guarda la cantidad de columnas de la matríz; con la condición del ciclo interno se recorren las columnas en cada fila.

En las siguientes páginas se exponen varios ejemplos que ilustran el manejo de matrices. Los primeros ejemplos se escribieron en forma secuencial y los siguientes, empleando funciones y procedimientos.

.:Ejemplo 6.12. *Una matriz transpuesta es aquella en que se invierten sus filas y columnas, es decir, que la transpuesta de una matriz tiene como columnas las filas de la matriz original y, las filas corresponden a las columnas de la matriz original. Construya un programa en Lenguaje C que genere la matriz transpuesta de una matriz inicial y, muestre al usuario ambas matrices.*

En Álgebra lineal se estudian las matrices y sus propiedades; la transpuesta de una matriz es estudiada también por el álgebra lineal y puede analizarse tal como se observa a continuación:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}$$

La transpuesta de la matriz A, es simbolizada como A^T.

$$A^T = \begin{bmatrix} a_{00} & a_{10} & a_{20} & \cdots & a_{(n-1)0} \\ a_{01} & a_{11} & a_{21} & \cdots & a_{(n-1)1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{0(m-1)} & a_{1(m-1)} & a_{2(m-1)} & \cdots & a_{(n-1)(m-1)} \end{bmatrix}$$

Análisis del problema:

- **Resultados esperados:** mostrar una matriz original y su transpuesta.
- **Datos disponibles:** se conocen la matriz original, con su tamaño y los elementos que contiene.
- **Proceso:** Después de tener la matriz original con sus respectivos datos almacenados, el programa deberá recorrer esta matriz fila a fila

e ir pasando los elementos a cada columna de la matriz transpuesta. Cuando se hayan pasado todos los datos, se deberán mostrar las dos matrices.

- **Variables requeridas:**
  - `matrizInicial`: matriz con los datos iniciales ingresados por el usuario.
  - `matrizTranspuesta`: matriz que transpone los elementos de la matriz inicial.
  - `filas`: cantidad de filas que tendrá la matriz inicial.
  - `columnas`: cantidad de columnas que tendrá la matriz inicial.
  - `i`: variable para controlar el primer ciclo.
  - `j`: variable para controlar el segundo ciclo.

De acuerdo al análisis realizado, se propone el programa 6.12

Programa 6.12: TranspuestaMatriz

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int filas, columnas, i, j;
6
7      printf( "Número de filas para la matriz: " );
8      scanf( "%d", &filas );
9
10     printf("\n Número de columnas para la matriz: " );
11     scanf( "%i", &columnas );
12
13     int matrizInicial    [ filas ][ columnas ];
14     int matrizTranspuesta[ columnas ][ filas ];
15
16     for( i = 0 ; i < filas ; i++ )
17     {
18         for( j = 0 ; j < columnas ; j++ )
19         {
20             printf( "Número para posición %d %d: ", i, j);
21             scanf( "%d", &matrizInicial[ i ][ j ] );
22         }
23     }
24
```

```
25      for( i = 0 ; i < filas ; i++ )
26      {
27          for( j = 0 ; j < columnas ; j++ )
28          {
29              matrizTranspuesta[j][i] = matrizInicial[i][j];
30          }
31      }
32
33      printf( "\n MATRIZ INICIAL \n \n" );
34      for( i = 0 ; i < filas ; i++ )
35      {
36          for( j = 0 ; j < columnas ; j++ )
37          {
38              printf( "%10d ", matrizInicial[ i ][ j ] );
39          }
40          printf( "\n" );
41      }
42
43      printf( "\n MATRIZ TRANSPUESTA \n \n" );
44      for( i = 0 ; i < columnas ; i++ )
45      {
46          for( j = 0 ; j < filas ; j++ )
47          {
48              printf( "%10d ", matrizTranspuesta[ i ][ j ] );
49          }
50          printf( "\n" );
51      }
52
53      return 0;
54  }
```

Explicación del programa:

Inicialmente, se declaran las variables necesarias (línea 5 y 13 y 14). Luego, se piden la cantidad de `filas` y `columnas` para dar tamaño a la matriz inicial, que equivale al número de `columnas` y de `filas` de la matriz transpuesta.

```
7       printf( "Número de filas para la matriz: " );
8       scanf( "%d", &filas );
9
10      printf("\n Número de columnas para la matriz: " );
11      scanf( "%i", &columnas );
12
13      int matrizInicial    [ filas ][ columnas ];
14      int matrizTranspuesta[ columnas ][ filas ];
```

Una alternativa para dar tamaño a la matriz, sería utilizando la función `dimensionarMI`, como se ve a continuacíon:

```
1     int **matrizInicial, *matrizTranspuesta;
2
3     printf( "Número de filas para la matriz: " );
4     scanf( "%d", &filas );
5
6     printf("\n Número de columnas para la matriz: " );
7     scanf( "%i", &columnas );
8
9     matrizInicial     = dimensionarMI( filas, columnas )
10    matrizTranspuesta = dimensionarMI( columnas, filas )
```

Esta alternativa, implicaría declarar las matrices como punteros y liberar la memoria solicitada mediante `freeMI`.

Después en el programa, se ingresan los datos a la matriz inicial, a través de dos ciclos `for` anidados.

```
16    for( i = 0 ; i < filas ; i++ )
17    {
18        for( j = 0 ; j < columnas ; j++ )
19        {
20            printf( "Número para posición %d %d: ", i, j);
21            scanf( "%d", &matrizInicial[ i ][ j ] );
22        }
23    }
```

Posteriormente, otra vez utilizando dos ciclos `for` anidados, se van pasando los elementos de la matriz inicial a la transpuesta.

```
25    for( i = 0 ; i < filas ; i++ )
26    {
27        for( j = 0 ; j < columnas ; j++ )
28        {
29            matrizTranspuesta[j][i] = matrizInicial[i][j];
30        }
31    }
```

Observe cómo, en la anterior porción de código con los índices `i` y `j` se recorren filas y columnas de la matriz inicial y columnas y filas de la matriz transpuesta. Por esta razón, los índices están invertidos en la instrucción:

```
29    matrizTranspuesta[j][i] = matrizInicial[i][j];
```

De esta manera, a medida que se recorren las filas de la matriz inicial, se recorren las columnas de la transpuesta y se van pasando de la una a la otra, los respectivos elementos.

A continuación, se recorre la matriz inicial y se imprimen los valores:

```
33      printf( "\n MATRIZ INICIAL \n \n" );
34      for( i = 0 ; i < filas ; i++ )
35      {
36          for( j = 0 ; j < columnas ; j++ )
37          {
38              printf( "%10d ", matrizInicial[ i ][ j ] );
39          }
40          printf( "\n" );
41      }
```

De la misma forma, se recorre e imprime la matriz transpuesta, salvo que ahora las variables `i` y `j` representan las columnas y las filas respectivamente.

```
43      printf( "\n MATRIZ TRANSPUESTA \n \n" );
44      for( i = 0 ; i < columnas ; i++ )
45      {
46          for( j = 0 ; j < filas ; j++ )
47          {
48              printf( "%10d ", matrizTranspuesta[ i ][ j ] );
49          }
50          printf( "\n" );
51      }
```

Note que, en ambos casos se utiliza `"%10d"` para reservar 10 espacios por cada número y, la impresión de la matriz sea más elegante.

Al ejecutar el programa, se visualizaría:

```
Matriz Inicial:    20   12   18
                   34   37   31
                   44   49   46

Matriz transpuesta: 20   34   44
                    12   37   49
                    18   31   46
```

.:Ejemplo 6.13. *Escriba un programa en Lenguaje C que guarde números enteros entre 50 y 100 en una matriz de* `n` *filas y* `m` *columnas y posteriormente, permita buscar un número cualquiera determinando la posición (*`fila` *y* `columna`*) donde se encuentra, si es que está almacenado en la matriz, en caso contrario, el programa deberá informar que el número buscado no hace parte de la matriz. Si el número se encuentra varias veces, el programa deberá retornar la posición de la primera ocurrencia.*

En este ejercicio es necesario validar el ingreso de los números para que estén en el rango 50 - 100. Las filas y columnas serán ingresadas por

el usuario. Posteriormente, el programa tendrá que realizar una búsqueda recorriendo la matriz como se explicó anteriormente.

Análisis del problema:

- **Resultados esperados:** mostrar la posición (fila y columna) donde está el número buscado en la matriz. Si el número no se encuentra dentro de la matriz, se mostrará un mensaje.
- **Datos disponibles:** el número de filas y de columnas de la matriz y los números que se guardarán en ella.
- **Proceso:** Posterior al ingreso de datos, el programa deberá hacer una búsqueda secuencial, recorriendo las celdas de la primera fila para ir después a la siguiente fila y continuar buscando, hasta hallar el número ingresado por el usuario. Si se recorrió toda la matriz y no se encontró el número, se informará al usuario. Si el número aparece varias veces en la matriz, se informa la posición de la primer ocurrencia.

- **Variables requeridas:**
  - `matriz`: variable que almacena los datos ingresados por el usuario.
  - `filas`: cantidad de filas que tendrá la matriz.
  - `columnas`: cantidad de columnas que tendrá la matriz.
  - `numero`: número que se quiere buscar.
  - `encontrado`: variable de tipo bandera que indica si el número si está o no en la matriz.
  - `i`: variable para controlar el primer ciclo.
  - `j`: variable para controlar el segundo ciclo.

Conforme al análisis planteado, se propone el programa 6.13.

Programa 6.13: BusquedaElemento

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define Verdadero 1
5 #define Falso     0
6
```

```
7  #define dimensionarI(n) (int *) malloc(sizeof(int)*n)
8
9  int  **dimensionarMI ( int filas,    int columnas );
10 void   freeMI        ( int **vector, int columnas );
11
12 int main()
13 {
14     int filas, columnas, i, j, numero;
15     int encontrado, **matriz;
16
17     printf( "\n Número de filas para la matriz: " );
18     scanf( "%d", &filas);
19
20     printf( "\n Número de columnas para la matriz: " );
21     scanf( "%d", &columnas );
22
23     matriz = dimensionarMI( filas, columnas );
24
25     for( i = 0 ; i < filas ; i++ )
26     {
27         for( j = 0 ; j < columnas ; j++ )
28         {
29           do
30           {
31             printf( "Número para posición %i %i : ", i, j );
32             scanf( "%d", &matriz[ i ][ j ] );
33
34             if( matriz[i][j] < 50 || matriz[i][j] > 100 )
35             {
36                 printf( "\n Número fuera de rango  \n" );
37             }
38           } while(matriz[i][j] < 50 || matriz[i][j] > 100 );
39         }
40         printf( "\n" );
41     }
42
43     printf( "Ingrese número a buscar en la matriz : " );
44     scanf( "%d", &numero );
45
46     encontrado = Falso;
47     for( i = 0 ; i < filas ; i++ )
48     {
49         for( j = 0 ; j < columnas ; j++ )
50         {
51             if( matriz[i][j] == numero &&
52                   encontrado == Falso )
53             {
54                 printf( "Encontrado en: (%i, %i)", i, j );
55                 encontrado = Verdadero;
```

```
56                 }
57             }
58         }
59
60         if ( encontrado == Falso )
61         {
62             printf( "El número no está en la matriz " );
63         }
64
65
66         freeMI ( matriz, columnas );
67
68         return 0;
69 }
70
71
72 int **dimensionarMI ( int filas, int columnas )
73 {
74   int **vector;
75
76   vector =  (int **) malloc ( sizeof(int*) * filas );
77
78   for ( int i = 0 ; i < filas ; i++ )
79   {
80     vector[i] = dimensionarI ( columnas );
81   }
82
83   return vector;
84 }
85
86 void freeMI ( int **vector, int columnas )
87 {
88   for ( int i = 0 ; i < columnas ; i++ )
89   {
90     free ( vector [ i ] );
91   }
92
93   free( vector );
94 }
```

Explicación del programa:

Luego de la declaración de variables, el programa pide la cantidad de `filas` y `columnas` que tendrá la matriz y captura estos datos. Se usa la función `dimensionarMI()`, para dar tamaño a la matriz.

```
17     printf( "\n Número de filas para la matriz: " );
18     scanf( "%d", &filas);
19
```

```
20      printf( "\n Número de columnas para la matriz: " );
21      scanf( "%d", &columnas );
22
23      matriz = dimensionarMI( filas, columnas );
```

Después en el programa hay dos ciclos `for` anidados,con los que se piden los datos al usuario y se almacenan en las celdas de la matriz. Note que, como cuerpo del segundo ciclo, hay un ciclo `do-while`, cuyo propósito es validar los números que está ingresando el usuario para que hagan parte del rango entre 50 y 100, según lo requiere el enunciado.

```
25      for( i = 0 ; i < filas ; i++ )
26      {
27          for( j = 0 ; j < columnas ; j++ )
28          {
29            do
30            {
31              printf( "Número para posición %i %i : ", i, j );
32              scanf( "%d", &matriz[ i ][ j ] );
33
34              if( matriz[i][j] < 50 || matriz[i][j] > 100 )
35              {
36                  printf( "\n Número fuera de rango  \n" );
37              }
38            } while(matriz[i][j] < 50 || matriz[i][j] > 100 );
39          }
40          printf( "\n" );
41      }
```

Posterior al ingreso de los datos, es decir, al terminar los dos ciclos `for` anidados, están las instrucciones que solicitan el número a buscar en la matriz.

```
43      printf( "Ingrese número a buscar en la matriz : " );
44      scanf( "%d", &numero );
```

Ahora el programa va a realizar la búsqueda. La variable `encontrado` arranca en falso, ya que aún no se ha encontrado el número. Nuevamente se hace un recorrido a la matriz con dos ciclos `for` anidados, como se describió iniciando el tema de matrices o arreglos bidimensionales. Al interior de estos ciclos hay una estructura de decisión `if` que permite determinar si lo que hay almacenado en la posición `i`, `j` de la matriz, es igual a lo guardado por la variable `numero`; de ser así, se ha hallado el elemento y se muestra al usuario la posición donde este se encuentra y se cambia el valor de la variable `encontrado` a verdadero, para no hacer mas comparaciones.

```
46      encontrado = Falso;
47      for( i = 0 ; i < filas ; i++ )
48      {
49          for( j = 0 ; j < columnas ; j++ )
50          {
51              if( matriz[i][j] == numero &&
52                    encontrado == Falso )
53              {
54                  printf( "Encontrado en: (%i, %i)", i, j );
55                  encontrado = Verdadero;
56              }
57          }
58      }
```

Luego de los dos ciclos `for` anidados, hay una estructura de decisión `if` que solo muestra el mensaje si la variable `encontrado` es falso, esto es, si el número buscado no se halló en la matriz.

```
60      if ( encontrado == Falso )
61      {
62          printf( "El número no está en la matriz " );
63      }
```

Los siguientes ejercicios de matrices, implementarán funciones y procedimientos, pués como se mencionó antes, estas estructuras optimizan la escritura del código.

.:Ejemplo 6.14. *Construya un programa en Lenguaje C que guarde en dos matrices de tamaño 3 x 3, es decir, 3 filas y 3 columnas, números enteros y que, por medio de una función calcule y almacene la suma de los números almacenados en las celdas equivalentes de las dos matrices en una tercera matriz. El programa debe mostrar las tres matrices.*

Analice la ilustración que sigue:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{00} & b_{01} & b_{02} & \cdots & b_{0(n-1)} \\ b_{10} & b_{11} & b_{12} & \cdots & b_{1(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ b_{(m-1)0} & b_{(m-1)1} & b_{(m-1)2} & \cdots & b_{(m-1)(n-1)} \end{bmatrix}$$

$$C = A + B = \begin{bmatrix} a_{00} + b_{00} & \cdots & a_{0(n-1)} + b_{0(n-1)} \\ a_{10} + b_{10} & \cdots & a_{1(n-1)} + b_{1(n-1)} \\ \vdots & \cdots & \vdots \\ a_{(m-1)0} + b_{(m-1)1} & \cdots & a_{(m-1)(n-1)} + b_{(m-1)(n-1)} \end{bmatrix}$$

Análisis del problema:

- **Resultados esperados:** mostrar las tres matrices, las dos ingresadas por el usuario más la matriz que contiene la suma de las dos anteriores.
- **Datos disponibles:** se conocen los números enteros a guardar en las dos primeras matrices.
- **Proceso:** este programa deberá calcular la suma, celda por celda, de los números que se encuentran en la misma posición de dos matrices y almacenar estas sumas en las celdas de una tercera matriz. Para lograr esto y, dado que las matrices tiene el mismo tamaño, basta con recorrer las matrices con dos ciclos anidados e ir realizando las sumas. Finalmente, mediante un procedimiento, se deben mostrar las tres matrices.
- **Variables requeridas:**
  - En el programa principal
    - Constante requerida:
      - ◇ `MAX`: constante para dar tamaño a las matrices (para el ejercicio es 3).
    - Variables:
      - ◇ `matrizA`: corresponde a la primera matriz a sumar.
      - ◇ `matrizB`: corresponde a la segunda matriz a sumar.
      - ◇ `matrizC`: corresponde a la matriz que almacena la suma las dos primeras.
  - En la función `leerDatos`
    - Parámetros:
      - ◇ `tamanio`: contiene el tamaño de las matrices, filas y columnas.
    - Variables locales:
      - ◇ `i`: variable de control del ciclo externo.

- ◇ `j`: variable de control xel ciclo interno.
- ◇ `matriz`: matriz que contiene los datos captuarados y que será retornada.

- En la función `calcularSuma`
  - Parámetros:
    - ◇ `m1`: corresponde a la primera matriz a sumar.
    - ◇ `m2`: corresponde a la segunda matriz a sumar.
    - ◇ `tamanio`: Tamaño de las matrices (igual número de filas y columnas)
  - Variables locales:
    - ◇ `i`: variable de control del ciclo externo.
    - ◇ `j`: variable de control del ciclo interno.
    - ◇ `matriz`: variable que almacena la suma de las matrices.
- En la función `mostrarResultados`
  - Parámetros:
    - ◇ `m1`: contiene la primera matriz a imprimir.
    - ◇ `m2`: contiene la segunda matriz a imprimir.
    - ◇ `m3`: contiene la tercera matriz a imprimir.
    - ◇ `tamanio`: Tamaño de las matrices (igual número de filas y columnas)
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo externo.
    - ◇ `j`: variable para controlar el ciclo interno.
    - ◇ `cadenaA`: almacena los números de la matriz 1.
    - ◇ `cadenaB`: almacena los números de la matriz 2.
    - ◇ `cadenaC`: almacena los números de la matriz 3.

De acuerdo al análisis planteado, se propone el programa 6.14.

Programa 6.14: SumaMatrices

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int **leerDatos( int n );
5  int **calcularSuma( int **mA, int **mB, int n );
6  void mostrarResultados(int **mA, int **mB, int **mC, int n);
7  void imprimir ( char titulo[], int **m, int n );
8
9  #define dimensionarI(n) (int *) malloc(sizeof(int)*n)
10
```

```
11 int  **dimensionarMI ( int filas,    int columnas );
12 void   freeMI        ( int **vector, int columnas );
13
14 int main()
15 {
16     const int MAX = 3;
17     int **matrizA;
18     int **matrizB;
19     int **matrizC;
20
21     matrizA = leerDatos( MAX );
22     matrizB = leerDatos( MAX );
23
24     matrizC = calcularSuma( matrizA, matrizB, MAX );
25
26     mostrarResultados(matrizA, matrizB, matrizC, MAX );
27
28     freeMI ( matrizA, MAX );
29     freeMI ( matrizB, MAX );
30     freeMI ( matrizC, MAX );
31
32     return 0;
33 }
34
35 int **leerDatos( int n )
36 {
37   int i, j;
38   int **matriz;
39
40   matriz = dimensionarMI ( n, n );
41
42   for( i = 0 ; i < n ; i++ )
43   {
44      for( j = 0 ; j < n ; j++ )
45      {
46         printf ( "Ingrese el número %d %d: ", i, j );
47         scanf  ( "%d", &matriz[ i ][ j ] );
48      }
49      printf ( "\n" );
50   }
51   return matriz;
52 }
53
54 int **calcularSuma( int **mA, int **mB, int n )
55 {
56     int i, j;
57     int **mC;
58
59     mC = dimensionarMI ( n, n );
```

```
60
61      for( i = 0 ; i < n ; i++ )
62      {
63          for( j = 0 ; j < n ; j++ )
64          {
65              mC[i][j] = mA[i][j] + mB[i][j];
66          }
67      }
68
69      return mC;
70  }
71
72
73  void mostrarResultados(int **mA, int **mB, int **mC, int n)
74  {
75    imprimir("\n MATRIZ A \n \n",     mA, n );
76    imprimir("\n MATRIZ B \n \n",     mB, n );
77    imprimir("\n MATRIZ SUMA \n \n", mC, n );
78  }
79
80
81  void imprimir ( char titulo[], int **m, int n )
82  {
83      int i, j;
84
85      printf( "%s", titulo );
86      for(i=0;i<n;i++)
87      {
88         for(j=0;j<n;j++)
89         {
90             printf(" %10d", m[i][j]);
91         }
92         printf("\n");
93      }
94  }
95
96
97  int **dimensionarMI ( int filas, int columnas )
98  {
99    int **vector;
100
101   vector =  (int **) malloc ( sizeof(int*) * filas );
102
103   for ( int i = 0 ; i < filas ; i++ )
104   {
105     vector[i] = dimensionarI ( columnas );
106   }
107
108   return vector;
```

```
109 }
110
111 void freeMI ( int **vector, int columnas )
112 {
113   for ( int i = 0 ; i < columnas ; i++ )
114   {
115     free ( vector [ i ] );
116   }
117
118   free( vector );
119 }
```

Explicación del programa:

Como siempre, se declaran las variables necesarias. Con la constante `MAX` de tipo `int` que almacena el número 3 se dá tamaño a las matrices, mismo número para las filas y las columnas.

```
16      const int MAX = 3;
17      int **matrizA;
18      int **matrizB;
19      int **matrizC;
```

La función `leerDatos()` permite capturar los números ingresados y almacenarlos en las dos primeras matrices; a ella se envía como argumento el tamaño de las mismas (número de filas igual al número de columnas), con el fín de dimensionarlas internamente;

```
21      matrizA = leerDatos( MAX );
22      matrizB = leerDatos( MAX );
```

Esta función consta de dos ciclos `for` anidados que permiten recorrer las filas y las columnas e ir ubicándose en cada celda de la matriz, para almacenar allí cada número que el usuario va ingresando.

Una vez ingresados los datos, se invoca a la función `calcularSuma()` la cuál tiene como parámetros las dos matrices y el tamaño de las mismas; este último se emplea para dar tamaño a una matriz dentro de la función y que se usa para almacenar los resultados de la suma; el retorno de esta función es la matriz que tiene los resultados de las sumas de las celdas de las dos matrices que llegaron como parámetros. La instrucción que invoca a la matriz es:

```
24      matrizC = calcularSuma( matrizA, matrizB, MAX );
```

El cuerpo de la función `calcularSuma()` es el siguiente:

```
54 int **calcularSuma( int **mA, int **mB, int n )
55 {
56     int i, j;
57     int **mC;
58
59     mC = dimensionarMI ( n, n );
60
61     for( i = 0 ; i < n ; i++ )
62     {
63         for( j = 0 ; j < n ; j++ )
64         {
65             mC[i][j] = mA[i][j] + mB[i][j];
66         }
67     }
68
69     return mC;
70 }
```

Como ya se ha enciondado antes, la función `calcularSuma()` utiliza dos ciclos `for` para hacer el recorrido de las celdas de las dos matrices con datos y, realizar al mismo tiempo la suma de los valores que están allí, guardándolos en las celdas de la matriz de resultados.

```
61     for( i = 0 ; i < n ; i++ )
62     {
63         for( j = 0 ; j < n ; j++ )
64         {
65             mC[i][j] = mA[i][j] + mB[i][j];
66         }
67     }
```

Finalmente, las tres matrices y su tamaño se envían como argumentos al procedimiento `mostrarResultados()`, el cual tiene como propósito imprimir las tres matrices.

```
26     mostrarResultados(matrizA, matrizB, matrizC, MAX );
```

Esta función, a su vez, invoca tres veces una única función denominada `imprimir()` que imprime las tres matrices. Observe cómo cambian los argumentos de los tres llamados.

```
73 void mostrarResultados(int **mA, int **mB, int **mC, int n)
74 {
75   imprimir("\n MATRIZ A \n \n",     mA, n );
76   imprimir("\n MATRIZ B \n \n",     mB, n );
77   imprimir("\n MATRIZ SUMA \n \n", mC, n );
78 }
```

```
81 void imprimir ( char titulo[], int **m, int n )
82 {
83     int i, j;
84
85     printf("%s", titulo );
86     for(i=0;i<n;i++)
87     {
88        for(j=0;j<n;j++)
89        {
90            printf(" %10d", m[i][j]);
91        }
92        printf("\n");
93     }
94 }
```

La función `imprimir()` recorre con los dos ciclos `for` anidados cada una de las celdas de la matriz y va imprimiendo su contenido.

.:Ejemplo 6.15. *Escribir un programa en Lenguaje C que permita almacenar en una matriz cada uno de los doce sueldos pagados durante un año a un conjunto de 10 empleados. El programa debe obtener el valor total de la nómina pagada durante un mes cualquiera del año y el total pagado durante el año a un empleado cualquiera. Construya el programa a partir de funciones y procedimientos.*

$$matrizSueldos = \begin{bmatrix} s_{00} & s_{01} & s_{02} & \cdots & s_{0\,11} \\ s_{10} & s_{11} & s_{12} & \cdots & s_{1\,11} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ s_{9\,0} & s_{9\,1} & s_{9\,2} & \cdots & s_{10\,11} \end{bmatrix}$$

Este ejercicio puede interpretarse como una matriz de 12 columnas, una pra almacenar los salarios de cada mes y 10 filas, una para cada empleado. De este modo, s_{23} corresponde al sueldo del segundo empleado en el tercer mes (Marzo).

Análisis del problema:

- **Resultados esperados:** mostrar el total de la nómina para un mes en particular, así como el sueldo pagado durante el año a un empleado específico.
- **Datos disponibles:** para cada empleado se conocen los 12 sueldos obtenidos en cada uno de los 12 meses del año.

- **Proceso:** luego del ingreso de los sueldos, el programa obtendrá, realizando una suma, el valor total pagado a todos los empleados durante el mes `m` (`m` corresponde a un mes ingresado por el usuario); para lograr esto, hay que recorrer la columna `m` de la matriz y realizar la sumar de todos sus valores, retornando la suma total. Así mismo, se tendrán que sumar los sueldos de la fila `e` (que representa al empleado ingresado por el usuario) y retornar el esta suma total. esto también se hace recorriendo la fila correspondiente con un ciclo.
- **Variables requeridas:**
  - En el programa principal
    - Constantes requeridas:
      - `MAX_EMPLEADOS`: constante que contiene la cantidad de empleados.
      - `MAX_MESES`: constante que contiene la cantidad de meses del año.
    - Variables:
      - `matrizSueldos`: matriz que guarda los sueldos de los 10 empleados durante los 12 meses.
      - `nominaMesM`: variable que almacena el total de la nómina del mes `m`.
      - `nominaEmpleadoE`: variable que almacena el total de la nómina del empleado (`e`).
      - `numeroMes`: almacena el número del mes (`m`) ingresdo por el usuario y del que se desea calcular la nómina total.
      - `numeroEmpleado`: almacena el número del empleado (`e`) ingresado por el usuario y del que se quiere obtender su nómina total.
  - En la función `leerDatos`
    - Parámetros:
      - `filas`: contiene el número de filas de la matriz.
      - `columnas`: contiene el número de columnas de la matriz.
    - Variables locales:
      - `i`: variable para controlar el ciclo externo.
      - `j`: variable para controlar el ciclo interno.
      - `matriz`: matriz que almacena los datos de los sueldos ingresados por el usuario.

- En la función `calcularNominaMesM`
  - Parámetros:
    - ◇ `matriz`: matriz que contiene los sueldos de los empleados.
    - ◇ `numeroMes`: contiene el número del mes que al que se le va a calcular la nómina.
    - ◇ `maximoEmpleados`: almacena el número total de empleados.
  - Variables locales:
    - ◇ `i`: variable para controlar el ciclo.
    - ◇ `nominaMesM`: variable que acumula la nómina del mes (`m`).
- En la función `calcularNominaEmpleadoE`
  - Parámetros:
    - ◇ `matriz`: matriz que contiene los sueldos de los empleados.
    - ◇ `numeroEmp`: almacena el número del empleado al que se le va a calcular la nómina.
    - ◇ `maximoMeses`: cantidad total de meses.
  - Variables locales:
    - ◇ `i`: variable que controla el ciclo.
    - ◇ `nominaEmpleadoE`: variable que acumula la nómina del empleado (`e`).
- En la función `mostrarResultados`
  - Parámetros:
    - ◇ `nominaMesM`: variable con el total de la nómina del mes al que se le calculó.
    - ◇ `nominaEmpleadoE`: variable con la nómina del empleado al que se le calculó.

Conforme al análisis realizado, se propone el Programa 6.15.

Programa 6.15: SueldosEmpleados

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

#define dimensionarD(n) (double *) malloc(sizeof(double)*n)

double **dimensionarMD ( int filas,       int columnas );
void     freeMD        ( double **vector, int columnas );

double **leerDatos               ( int f, int c );
double   calcularNominaMesM      ( double **matrizSueldo,
                                   int numeroMe,
                                   int maxEmp );
double   calcularNominaEmpleadoE( double **matrizSueldo,
                                   int numeroEmpleado,
                                   int maxMes );
void     mostrarResultados       ( double nominaMesM,
                                   double nominaEmpleadoE );

int main()
{
    const int MAX_EMPLEADOS = 10;
    const int MAX_MESES     = 12;

    double **matrizSueldos, nominaMesM, nominaEmpleadoE;
    int      numeroMes, numeroEmpleado;

    matrizSueldos = leerDatos( MAX_EMPLEADOS, MAX_MESES );

    printf ( "Número del mes que desea totalizar: ");
    scanf  ( "%d", &numeroMes );
    numeroMes--;

    printf ( "Número del empleado que desea totalizar: ");
    scanf  ( "%d", &numeroEmpleado);
    numeroEmpleado--;

    nominaMesM = calcularNominaMesM( matrizSueldos,
                                     numeroMes,
                                     MAX_EMPLEADOS );

    nominaEmpleadoE = calcularNominaEmpleadoE(matrizSueldos,
                                               numeroEmpleado
                                                  ,
                                               MAX_MESES );

    mostrarResultados( nominaMesM, nominaEmpleadoE );

    return 0;
```

```
48 }
49
50 double **leerDatos( int f, int c )
51 {
52     int      i, j;
53     double **matriz;
54
55     matriz = dimensionarMD ( f, c );
56
57     for( i = 0 ; i < f ; i++ )
58     {
59       for( j = 0 ; j < c ; j ++)
60       {
61         printf("Salario empleado %i (mes %i):",(i+1),(j+1));
62         scanf ( "%lf", &matriz[ i ][ j ] );
63       }
64     }
65     return matriz;
66 }
67
68 double calcularNominaMesM ( double **matrizSueldo,
69                             int numeroMe, int maxEmp )
70 {
71     int    i;
72     double nominaMesM;
73
74     nominaMesM = 0.0;
75
76     for( i = 0 ; i < maxEmp ; i++ )
77     {
78         nominaMesM = nominaMesM + matrizSueldo[i][numeroMe];
79     }
80
81     return nominaMesM;
82 }
83
84 double calcularNominaEmpleadoE( double **matrizSueldo,
85                                 int      numeroEmpleado,
86                                 int      maxMes )
87 {
88     int i;
89
90     double nominaEmpleadoE;
91
92     nominaEmpleadoE = 0.0;
93
94     for( i = 0 ; i < maxMes ; i++ )
95     {
96         nominaEmpleadoE = nominaEmpleadoE +
```

```
97                              matrizSueldo[numeroEmpleado][i];
98      }
99
100     return nominaEmpleadoE;
101 }
102
103 void mostrarResultados ( double nominaMesM,
104                          double nominaEmpleadoE )
105 {
106
107   printf( "Total del mes: %.2f\n", nominaMesM );
108   printf( "Total del empleado: %.2f ", nominaEmpleadoE );
109 }
110
111 double **dimensionarMD ( int filas, int columnas )
112 {
113   double **vector;
114
115   vector =  (double **) malloc ( sizeof(double*) * filas );
116
117   for ( int i = 0 ; i < filas ; i++ )
118   {
119     vector[i] = dimensionarD ( columnas );
120   }
121
122   return vector;
123 }
124
125 void freeMI ( int **vector, int columnas )
126 {
127   for ( int i = 0 ; i < columnas ; i++ )
128   {
129     free ( vector [ i ] );
130   }
131
132   free( vector );
133 }
```

Explicación del programa:

Al principio del programa se declaran las variables requeridas, entre ellas, la matriz de sueldos, la nomina del mes que se desea y la nómina del empleado requerido. Posteriormente se invoca la función `leerDatos()`, cuyos parámetros son: el número máximo de empleados y el máximo de meses que tiene la matriz de sueldos.

```
27      matrizSueldos = leerDatos( MAX_EMPLEADOS, MAX_MESES );
```

Con el empleo de esta función se guardan los 12 sueldos de los 10 empleados en la matriz. La función posee dos ciclos `for` anidados y permite retornar una matriz con los sueldos ingresados, que será almacenada en la variable de tipo puntero `matrizSueldos`. Es importante enfatizar que las filas de la matriz corresponden a los empleados y las columnas a los meses del año.

Luego en el programa, se solicitan el número del mes y del empleado a los que se les va a calcular la nómina total y se reducen en uno las variables `numeroMes` y `numeroEmpleado` (Líneas 29 a 35). Se invoca después a la función `calcularNoinaMes()`; la referencia a la matriz de sueldos se envía como argumento, junto con el número del mes del que se desea conocer el valor total de la nómina y el máximo de empleados a dicha función, la cual emplea un ciclo `for` con el propósito de recorrer las todas filas de la matriz en la columna que corresponde al mes analizado (el que el usuario ingresó) y va sumando los sueldos de ese mes; después de hacer el recorrido por todas las celdas y sumar sus valores, la función retorna el resultado de esta suma, que corresponde a lo pagado durante ese mes a los 10 empleados.

```
68 double calcularNominaMesM ( double **matrizSueldo,
69                             int numeroMe, int maxEmp )
70 {
71     int    i;
72     double nominaMesM;
73
74     nominaMesM = 0.0;
75
76     for( i = 0 ; i < maxEmp ; i++ )
77     {
78         nominaMesM = nominaMesM + matrizSueldo[i][numeroMe];
79     }
80
81     return nominaMesM;
82 }
```

D forma similar opera la función `calcularNominaEmpleadoE()`, que recibe como argumentos la referencia a la matriz de sueldos, el número de fila que representa al empleado y el máximo de meses, y por medio de un ciclo `for` recorre las columnas de la matriz en la fila que se envió como argumento y que corresponde al empleado del que se va a conocer su nómina del año, retornando al final la suma de los 12 sueldos del empleado.

```
84 double calcularNominaEmpleadoE( double **matrizSueldo,
85                                 int      numeroEmpleado,
86                                 int      maxMes )
87 {
88     int i;
89
90     double nominaEmpleadoE;
91
92     nominaEmpleadoE = 0.0;
93
94     for( i = 0 ; i < maxMes ; i++ )
95     {
96         nominaEmpleadoE = nominaEmpleadoE +
97                           matrizSueldo[numeroEmpleado][i];
98     }
99
100    return nominaEmpleadoE;
101 }
```

Por último, el programa muestra, a través del procedimiento `mostrarResultados()` el total de la nómina pagado en el mes `m` del año y lo pagado durante el año al empleado `e`.

.:Ejemplo 6.16. *Escriba un programa en Lenguaje C que almacene caracteres en una matriz cuadrada de `n` filas y columnas, y que, a continuación, le permita al usuario intercambiar los caracteres que se encuentran en las celdas de dos filas cualquiera. El programa mostrará la matriz inicial y la matriz resultante con los elementos de las filas ya intercambiados. Utilice funciones y procedimientos para realizar los procesos.*

Análisis del problema:

- **Resultados esperados:** mostrar la matriz inicial con los caracteres como los ingresó el usuario en un principio y la matriz con las filas intercambiadas.
- **Datos disponibles:** se conocen los caracteres que se guardarán en la matriz inicial.
- **Proceso:** luego de ingresar los caracteres a la matriz de n filas y columnas (matriz cuadrada), el programa deberá preguntar qué filas se van a intercambiar, posteriormente, con el uso de un ciclo se recorre una de las filas y se pasan los elementos contenidos en las celdas de esta a una variable temporal para, después pasar los de la otra fila

a intercambiar a la fila inicial y, por último, los elementos que están en la variable temporal se van pasando a la segunda fila a cambiar. Finalmente, el programa deberá mostrar la matriz inicial y la que tiene las fials intercambiadas.

- **Variables requeridas:**
  - En el programa principal
    - `matrizCaracteres`: variable que almacena los caracteres que el usuario ingresa.
    - `matrizIntercambiada`: variable que guarda los caracteres con las filas ya intercambiadas.
    - `fila1`: corresponde a la primera fila a intercambiar.
    - `fila2`: representa la segunda fila a intercambiar.
    - `tamanio`: es el tamaño de la matriz o `n` del enunciado (igual número de filas y columnas).
  - En la función `leerDatos`
    - Parámetros:
      - `tamanio`: representa el tamaño de la matriz.
    - Variables locales:
      - `i`: variable para controlar el ciclo externo.
      - `j`: variable para controlar el ciclo interno.
      - `matriz`: variable que representa la matriz de caracteres.
  - En la función `intercambiarF`
    - Parámetros:
      - `matriz`: matriz que contiene los caracteres.
      - `tamanio`: representa al tamaño de la matriz.
      - `f1`: representa la primera fila a intercambiar.
      - `f2`: corresponde a la segunda fila a intercambiar.
    - Variables locales:
      - `i`: variable para controlar el ciclo.
      - `aux`: variable auxiliar que almacena los caracteres a intercambiar temporalmente.
      - `mat`: matriz resultante en la que se hacen los cambios de filas (`m`).

- En la función `mostrarResultados`
  - Parámetros:
    - ◇ `m1`: matriz inicial que tiene los caracteres ingresados por el usuario.
    - ◇ `m2`: matriz resultante con las dos filas ya intercambiadas.
    - ◇ `t`: tamaño de las matrices, corresponde a la cantidad de filas y columnas, que es igual.

De acuerdo al análisis realizado, se propone el programa 6.16.

Programa 6.16: MatrizCaracteres

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

#define dimensionarC(n)   (char *) malloc(sizeof(char)*n)

char **dimensionarMC      ( int filas,     int columnas );
void freeMC               ( char **vector, int columnas );

char **leerDatos          ( int n);
char **intercambiarF      ( char **matriz,
                            int tmanio, int f1, int f2 );
void   mostrarResultados  ( char **mA, char **mB, int n );

int main()
{
    char **matrizCaracteres;
    char **matrizIntercambiada;
    int    fila1, fila2, tamanio;

    printf ( "Ingrese el tamaño de la matriz: ");
    scanf  ( "%d", &tamanio);

    matrizCaracteres = leerDatos(tamanio);

    printf ( "Ingrese fila 1 a intercambiar: ");
    scanf  ( "%d", &fila1);

    printf ( "Ingrese fila 2 a intercambiar: ");
    scanf  ( "%d", &fila2);

    matrizIntercambiada = intercambiarF( matrizCaracteres,
                                         tamanio,
                                         fila1, fila2 );

    mostrarResultados( matrizCaracteres,
                       matrizIntercambiada, tamanio );
```

```
37
38      freeMC ( matrizCaracteres );
39
40      return 0;
41  }
42
43  char **leerDatos(int n)
44  {
45      int i, j;
46      char **matriz;
47
48      matriz = dimensionarMC ( n, n );
49
50      for( i = 0 ; i < n ; i++ )
51      {
52         for(j = 0 ; j < n ; j ++)
53         {
54              printf ( "Carácter para posición %i %i: ",i,j );
55              scanf  ( " %c", &matriz[ i ][ j ] );
56         }
57         printf( "\n" );
58      }
59      return matriz;
60  }
61
62  char **intercambiarF( char **matriz,
63                        int tamanio, int f1, int f2 )
64  {
65      int i, j;
66      char **mat;
67      char aux;
68
69      mat = dimensionarMC ( tamanio, tamanio );
70      for( i = 0 ; i < tamanio ; i++ )
71      {
72          for(j = 0 ; j < tamanio ; j ++)
73          {
74              mat[i][j] = matriz[i][j];
75          }
76      }
77
78      for( i = 0 ; i < tamanio ; i++ )
79      {
80          aux = mat[f1][i];
81          mat[f1][i] = mat[f2][i];
82          mat[f2][i] = aux;
83      }
84      return mat;
85  }
```

```
86
87 void mostrarResultados(char **mA, char **mB, int n)
88 {
89   int i, j;
90
91   printf("\n MATRIZ ORIGINAL \n \n");
92   for( i = 0 ; i < n ; i++ )
93   {
94     for( j = 0 ; j < n ; j++ )
95     {
96        printf("%c ", mA[ i ][ j ] );
97     }
98     printf("\n");
99   }
100
101  printf("\n MATRIZ INTERCAMBIADA \n \n");
102  for( i = 0 ; i < n ; i++ )
103  {
104    for( j = 0 ; j < n ; j++ )
105    {
106      printf("%c ", mB[ i ][ j ] );
107    }
108    printf("\n");
109  }
110 }
111
112 char **dimensionarMC ( int filas, int columnas )
113 {
114   char **vector;
115
116   vector =  (char **) malloc ( sizeof(char*) * filas );
117
118   for ( int i = 0 ; i < filas ; i++ )
119   {
120     vector[i] = dimensionarC ( columnas );
121   }
122
123   return vector;
124 }
125
126 void freeMC ( char **vector, int columnas )
127 {
128   for ( int i = 0 ; i < columnas ; i++ )
129   {
130     free ( vector [ i ] );
131   }
132
133   free( vector );
134 }
```

Explicación del programa:

El programa inicia con la declaración de variables y pidiéndole al usuario el tamaño de la matriz. Ya que la matriz es cuadrada, el número de filas será igual al de las columnas (líneas 2 a 7); el tamaño se enviará como parámetro a la función `leerDatos()` para que allí se inicialice la matriz. Esto se puede ver en la línea 9 del programa.

La función `leerDatos()`, permite ingresar los caracteres a la matriz de caracteres (la inicial); esto se lleva a cabo por medio de los dos ciclos `for` anidados, como ya se hecho ya en ejercicios anteriores (líneas de la 25 a la 37).

Posteriormente, el programa solicita al usuario las filas a intercambiar y, a partir de estos datos, se invoca a la función `intercambiarF()` (líneas de la 11 a la 19) que tiene como argumentos la matriz inicial, su tamaño y las filas a intercambiar.

Al interior de esta función, se declara una matriz llamada `mat` a la que se pasan los caracteres de la matriz inicial y, así no modificar esta última; también se declara la variable `aux` que guardará temporalmente los caracteres ubicados en la primera fila a intercambiar; de esta forma no se perderán; a continuación, los caracteres de la segunda fila a intercambiar se ubican en las celdas de la primera fila y, los que estaban en la variable `auxiliar` (es decir, los que se copiaron inicialmente) se ubican en la cedla actual de la segunda fila de la matriz que se está intercambiando. la realización de estos cambios se lleva a cabo con un ciclo `for` que recorre las columnas de las filas a intercambiar. En este caso, la variable `i`, identifica la columna actual en que el programa se va ubicando para leer el carácter. Este procedimiento intercambia los caracteres que estaban en las dos filas indicadas por el usuario. Al término de la función `intercambiarF()`, se retorna la variable `mat`.

```
60 char **intercambiarF( char **matriz,
61                       int tamanio, int f1, int f2 )
62 {
63     int i, j;
64     char **mat;
65     char aux;
66
67     mat = dimensionarMC ( tamanio, tamanio );
68     for( i = 0 ; i < tamanio ; i++ )
69     {
70         for(j = 0 ; j < tamanio ; j ++)
71         {
72             mat[i][j] = matriz[i][j];
73         }
74     }
75
76     for( i = 0 ; i < tamanio ; i++ )
77     {
78         aux = mat[f1][i];
79         mat[f1][i] = mat[f2][i];
80         mat[f2][i] = aux;
81     }
82     return mat;
83 }
```

El procedimiento `mostrarResultados()` permite mostrar la matriz inicial con los caracteres como los ingresó el usuario y la matriz con sus filas ya intercambiadas. En este procedimiento se usan dos ciclos `for` que recorren todas las celdas de las matrices y muestran los caracteres con la instrucción `printf()`, de tal modo que se aprecian los datos de las matrices como tablas.

.:Ejemplo 6.17. *Escriba un programa enLenguaje C que permita guardar números enteros en una matriz cuadrada de orden n (igual número de filas y columnas) y que, a través de funciones y procedimientos determine si la suma de los elementos de la diagonal principal es igual, mayor o menor a la suma de los elementos que se encuentran en la diagonal secundaria de la matriz.*

La diagonal principal de una matriz, es aquella que inicia en la primera celda de la matriz (fila 0, columna 0) y termina en la última celda (fila n-1, columna n-1) de la misma.

$$diagonalPrincipal = \begin{bmatrix} a_{00} & & & \\ & a_{11} & & \\ & & \dots & \\ & & & a_{(n-1)(n-1)} \end{bmatrix}$$

De esta forma, la suma de los elementos de la diagonal principal de una matriz, se define como:

$$sumaDiagonalPrincipal = \sum_{i=0}^{n-1} a_{ii}$$

Ahora bien, la diagonal secundaria de una matriz inicia en la última celda de la primera fila (fila 0, columna n-1) y termina en la última fila de la primera columna (fila n-1, columna 0).

$$diagonalSecundaria = \begin{bmatrix} & & a_{0(n-1)} \\ & a_{1(n-2)} & \\ \cdots & & \\ a_{(n-1)0} & & \end{bmatrix}$$

La suma de los elementos de la diagonal secundaria es:

$$sumaDiagonalSecundaria = \sum_{i=0}^{n-1} a_{i(n-i)}$$

Si bien se pueden hallar diagonales en matrices no cuadradas, este ejercicio propone emplear una matriz cuadrada cuyo número de filas es igual al número de columnas.

Análisis del problema:

- **Resultados esperados:** mostrar un mensaje que indique si la suma de los elementos ubicados en la diagonal principal de una matriz es igual, mayor o menor a la suma de los elementos de la diagonal secundaria.
- **Datos disponibles:** se conocen el tamaño de la matriz y los números enteros que se almacenarán en ella.
- **Proceso:** primero se ingresarán los números enteros a la matriz, luego, el programa deberá recorrer, mediante funciones, la diagonal principal y la diagonal secundaria y sumar los elementos que hacen parte de estas diagonales, posteriormente, las sumas obtenidas se compararán y se indicará si las sumas son iguales o alguna de las dos es mayor. El recorrido de las diagonales se hará haciendo uso de una estructura cíclica, mientras que la comparación de las sumas requerirá del uso de una estructura de decisión.

- **Variables requeridas:**
  - En el programa principal
    - `n`: almacena el tamaño de la matriz.
    - `matriz`: matriz que guardará los números ingresados por el usuario.
    - `sumaDiagonalPrincipal`: guarda la suma de los elelemtnso presentes en la diagonal principal de la matriz.
    - `sumaDiagonalSecundaria`: guarda la suma de los elementos presentes en la diagonal secundaria de la matriz.
  - En la función `leerDatos`
    - Parámetros:
      - `tamanio`: contiene el tamaño de la matriz.
    - Variables locales:
      - `i`: variable para controlar el ciclo externo.
      - `j`: variable para controlar el ciclo interno.
      - `matriz`: matriz que contiene los números y que se retornará al programa principal.
  - En la función `sumarDiagPrincipal`
    - Parámetros:
      - `matriz`: matriz que contiene los números.
      - `tamanio`: contiene el tamaño de la matriz.
    - Variables locales:
      - `i`: variable para controlar el ciclo.
      - `suma`: variable de tipo acumulador para sumar los elementos de la diagonal principal.
  - En la función `sumarDiagSecundaria`
    - Parámetros:
      - `matriz`: matriz que contien los números ingresados.
      - `tamanio`: contiene el tamaño de la matriz.
    - Variables locales:
      - `columna`: variable que indica la columna actual.
      - `i`: variable para controlar el ciclo.
      - `suma`: variable de tipo acumulador que suma los elementos de la diagonal secundaria.

- En el procedimiento `mostrarResultados`
  - Parámetros:
    - `sumaDiagonalP`: variable con la suma de los elementos de la diagonal principal.
    - `sumaDiagonalS`: variable con la suma de los elementos de la diagonal secundaria.

Conforme al análisis realizado, se propone el programa 6.17.

Programa 6.17: DiagonalesMatriz

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

#define dimensionarI(n)  (int *) malloc(sizeof(int)*n)

int  **dimensionarMI      ( int filas,    int columnas );
void   freeMI             ( int **vector, int columnas );

int  **leerDatos          ( int n );
int    sumarDiagPrincipal ( int **matriz, int tamanio );
int    sumarDiagSecundaria( int **matriz, int tamanio );
void   mostrarResultados  ( int sumaDiagonalP,
                            int sumaDiagonalS );
int main()
{
  int **matriz;
  int sumaDiagonalPrincipal, sumaDiagonalSecundaria, n;

  printf ( "Ingrese el tamaño de la matriz: ");
  scanf  ( "%d", &n );

  matriz = leerDatos( n );

  sumaDiagonalPrincipal  = sumarDiagPrincipal ( matriz, n );
  sumaDiagonalSecundaria = sumarDiagSecundaria( matriz, n );

  mostrarResultados( sumaDiagonalPrincipal,
                     sumaDiagonalSecundaria );

  freeMI ( matriz, n );
}


int **leerDatos( int n )
{
    int i, j;
    int **matriz;
```

```
38
39      matriz = dimensionarMI ( n, n );
40
41      for( i = 0 ; i < n ; i++ )
42      {
43         for(j = 0 ; j < n ; j ++)
44         {
45              printf ( "Carácter para posición %i %i: ",i,j );
46              scanf  ( " %d", &matriz[ i ][ j ] );
47         }
48         printf( "\n" );
49      }
50      return matriz;
51  }
52
53  int sumarDiagPrincipal( int **matriz, int tamanio )
54  {
55    int suma, i;
56
57    suma = 0;
58    for ( i = 0 ;i < tamanio ; i++ )
59    {
60      suma = suma + matriz[ i ][ i ];
61    }
62    return suma;
63  }
64
65  int sumarDiagSecundaria( int **matriz, int tamanio )
66  {
67    int suma, i, columna;
68
69    columna = tamanio - 1;
70    suma = 0;
71    for ( i = 0 ; i < tamanio ; i++ )
72    {
73      suma = suma + matriz[ i ][ columna ];
74      columna = columna - 1;
75    }
76
77    return suma;
78  }
79
80  void mostrarResultados( int sumaDiagonalP,
81                          int sumaDiagonalS )
82  {
83    if ( sumaDiagonalP == sumaDiagonalS )
84    {
85      printf( "Las sumas de las diagonales son iguales" );
86    }
```

```
87    else
88    {
89      if( sumaDiagonalP > sumaDiagonalS )
90      {
91        printf( "La suma de la diagonal principal es mayor");
92      }
93      else
94      {
95        printf( "La suma de la diagonal principal es menor");
96      }
97    }
98  }
99
100 int **dimensionarMI ( int filas, int columnas )
101 {
102   int **vector;
103
104   vector =  (int **) malloc ( sizeof(int*) * filas );
105
106   for ( int i = 0 ; i < filas ; i++ )
107   {
108     vector[i] = dimensionarI ( columnas );
109   }
110
111   return vector;
112 }
113
114 void freeMI ( int **vector, int columnas )
115 {
116   for ( int i = 0 ; i < columnas ; i++ )
117   {
118     free ( vector [ i ] );
119   }
120
121   free( vector );
122 }
```

Explicación del programa:

Después de la declaración de variables, en el programa principal, se solicita al usuario el ingreso del tamaño de la matriz, lo que equivale al número de filas como de columnas, pués el ejercicio indica la declaración de una matriz cuadrada (líneas de la 16 a la 20).

La lectura de los datos se lleva a cabo al invocar la función `leerDatos()` (línea 22); a su vez, esta función se implementó con dos ciclos `for` que recorren filas y columnas (Líneas 41 a 49), tal como se ha trabajado en otros ejercicios.

La función `sumarDiagPrincipal()` recibe como parámetro a la matriz ya cargad con los números y, a través de un ciclo `for` va pasando por las celdas de la diagonal principal, donde el número de la fila es igual al número de la columna, es por esto que, dentro del ciclo, la variable `i` indica fila y columna al mismo tiempo. Los números ubicados en estas celdas se van sumando en la variable `suma`, la cual es retornada al programa principal finalizando la función.

```
53 int sumarDiagPrincipal( int **matriz, int tamanio )
54 {
55   int suma, i;
56
57   suma = 0;
58   for ( i = 0 ;i < tamanio ; i++ )
59   {
60     suma = suma + matriz[ i ][ i ];
61   }
62   return suma;
63 }
```

La función `sumarDiagSecundaria()` opera de forma similar a la función `sumarDiagPrincipal()`, la diferencia radica en que el recorrido (que también se hace con un ciclo `for`) va desde la primera fila, indicada con la variable `i` y la última columna con la variable `columna`, que se inicializó antes del ingreso al ciclo con el valor del parámetro `tamanio-1`. Al interior del ciclo, la variable `i` se va incrementando, lo que permite avanzar por filas en cada iteración, a la vez que las columnas se van decrementando con la instrucción `columna=columna - 1`,lo que permite al programa ubicarse en la columna anterior. De esta forma, el programa va recorriendo las celdas de la diagonal secundaria; los valores guardados en estas celdas se acumulan en la variable suma, retornada al final de la función.

```
65 int sumarDiagSecundaria( int **matriz, int tamanio )
66 {
67   int suma, i, columna;
68
69   columna = tamanio - 1;
70   suma = 0;
71   for ( i = 0 ; i < tamanio ; i++ )
72   {
73     suma = suma + matriz[ i ][ columna ];
74     columna = columna - 1;
75   }
76
```

```
77    return suma;
78 }
```

Desde el programa principal, se envían como argumentos los resultados producidos por las dos funciones anteriores al procedimiento `mostrarResultados();` allí, mediante una estructura de decisión `if` se determina si las sumas son iguales o alguna es mayor (líneas 13 y 14, así como las líneas 56 a la 69) y se muestra el mensaje acorde con el resultado de la comparación.

Actividad 6.2

Para los siguientes ejercicios propuestos, construya programas en Lenguaje C, usando funciones y procedimientos en su solución.

1. El profesor de la materia de "Lenguaje de Programación" necesita un programa en el que pueda almacenar los nombres de los 30 estudiantes del curso y, las 5 notas obtenidas por cada estudiante durante el semestre. El programa deberá hacer lo siguiente:

   a) obtener la nota definitiva de estudiante, que se calcula como la media aritmética de las 5 notas del estudiante. Estas definitivas se almacenarán en un vector.

   b) Hallar al estudiante (nombre) con la mayor nota definitiva.

   c) Determinar los estudiantes (nombres) y, almacenarlos en un arreglo que perdieron la materia y la tendrán que repetir. Un estudiante pierde la materia si su nota definitiva es inferior a 2.0. (Dos punto cero).

   d) Determinar el nombre de los estudiantes (y almacenarlos en un arreglo) que se quedaron habilitando la asignatura. Un estudiante habilita la materia si su nota definitiva es inferior a 3.0. pero superior a 2.0, en otras palabras, si la nota definitiva está entre 2.0 y 2.99.

   e) Encontrar el porcentaje de estudiantes que ganaron la materia.

2. Haga un programa que almacene caracteres en una matriz de `n` filas por `m` columnas y que posteriormente, obtenga y muestre la matriz transpuesta de la matriz originalmente ingresada. La matriz transpuesta es una matriz que tiene como filas, las columnas de otra matriz.

3. Una clínica de control al sobrepeso requiere de un programa en el que puedan almacenar los nombres y los pesos tomados durante un periodo de tiempo a un grupo de `n` pacientes. en el periodo, cada paciente es pesado 3 veces (una pesada inicial, una intermedia y una pesada final), para determinar su evolución durante ese periodo. el programa debe mostrar:

   a) El peso ganado o perdido por un paciente durante el periodo.

   b) El número de pacientes que perdieron peso entre la pesada inicial y la pesada intermedia.

   c) Imagine que se ha guardado en otro vector el objetivo de cada paciente: ganar o perder peso durante el periodo. Determine el porcentaje de pacientes que lograron el objetivo.

4. Construya un programa que almacene números enteros en una matriz de 5 filas por 5 columnas y, que luego la recorra (mostrando los números almacenados) desde la última celda (fila 5 columna 5) hasta la primera celda (fila1 columna 1) recorriendo primero las celdas de cada columna.

5. Desarrolle un programa que almacene unos (1) en la primera y última fila y en la primera y última columna de una matriz cuadrada de tamaño 5, esto es, en las filas y columnas exteriores de la matriz. El programa almacenará ceros (0) en las demás celdas de la matriz, es decir, en las celdas ubicadas al interior de la matriz.

6. Elabore un programa que permita multiplicar dos matrices. Dos matrices se pueden multiplicar si, en primer lugar, estas almacenan números y, en segundo lugar, si el número de columnas de la primera matriz es igual al número de filas de la segunda matriz. La matriz resultante tendrá el mismo número de filas que la primera matriz e igual número de columnas de la segunda matriz.

7. Haga un programa que llene una matriz cuadrada de orden `n` con números enteros positivos. Luego, el programa debe determinar si la matriz es simétrica.

8. Elabore un programa que recorra una matriz cuadrada de orden `n`, que ya está cargada con caracteres, en forma de espiral iniciando en la celda 0,0.

CAPÍTULO 7

ARCHIVOS

El ordenador nació para resolver problemas que antes no existían.

Bill Gates

Objetivos del capítulo:

- Estudiar las generalidades de archivos.
- Conocer las funciones de biblioteca para la gestión de archivos.
- Comprender la diferencia entre archivos de texto y archivos binarios.
- Realizar operaciones básicas sobre archivos.

7.1. Generalidades

Todos los ejemplos que se han trabajado hasta el momento, solicitan datos del usuario, los procesan y luego, al terminar la ejecución del programa se pierden. Esto sucede porque esos datos se trabajan en la memoria principal (RAM) del equipo de procesamiento y no se almacenan de forma permanente. Para guardar esos datos y tenerlos para futuras consultas, se deben llevar a la memoria secundaria mediante el uso de archivos.

Una lista de amigos con datos personales y de contacto, una nómina de una empresa en donde se guardan datos de los empleados, un documento creado en un procesador de textos o una nota creada con el editor de notas de un dispositivo móvil, son apenas unos de los tantos ejemplos que se pueden tener de lo que es un archivo. Los archivos son colecciones de registros, relacionados entre sí, que se guardan en diferentes medios de almacenamiento: discos duros, memorias USB, memorias SD, discos compactos, alojamiento en la nube, entre otros.

Cada archivo tiene sus propias características y un nombre que lo identifica de los demás. Para trabajar archivos en programación, este ha de tener un nombre interno que es usado por el programa para realizar sus procesos y un nombre externo con el cual se guarda en un medio de almacenamiento.

Los archivos tienen diferentes clasificaciones, para el propósito de este texto, se analizarán conforme a los datos que almacenan: archivos de texto y archivos binarios.

Aclaración:

1. La memoria RAM (Random Access Memory) de un equipo es temporal o volátil, esto significa, que los datos que sean almacenados en ella se perderán en el momento que termine de ejecutarse el programa o se apague el dispositivo de procesamiento.

2. Los archivos no procesan los datos, solamente cumplen la función de almacenarlos para que sean procesados desde los programas.

La implementación de un archivo, de forma general, se resume en el siguiente diagrama:

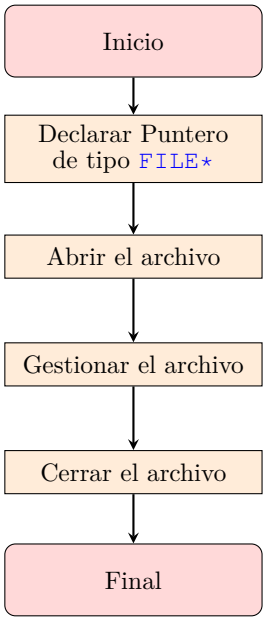


Figura 7.1: Implementación archivos

Como se observa, el primer paso que se presenta, es la declaración de un puntero de tipo `FILE` (fichero – archivo), la cual debe hacerse de la siguiente manera:

```
FILE* puntero_archivo;
```

`FILE`: estructura de datos que controla los archivos, su definición se encuentra dentro del archivo de cabecera `stdio.h`.

La variable `puntero_archivo`: apunta a la información de un archivo en particular: nombre, posición, estado. Con esta variable el programa reconocerá, de manera interna, el nombre del archivo.

Continuando con el análisis del diagrama de la Figura 7.1, los siguientes pasos en la implementación de un archivo: apertura, gestión y cierre, requieren del uso de funciones del Lenguaje; todas ellas están incluidas dentro de la biblioteca `stdio.h`.

En lo concerniente a la apertura y al cierre, sea de texto o binario, en Lenguaje C, se usan las funciones `fopen` y `fclose`, respectivamente.

Función `fopen`: permite la apertura de un archivo. Tiene la siguiente forma general o sintaxis:

```
fopen (nombre, modo);
```

`nombre`: es una cadena de caracteres que representa el nombre externo del archivo, es decir, el nombre con el que será grabado en un dispositivo de almacenamiento. Esta cadena de caracteres podrá tener una ruta válida de directorios o carpetas. Por defecto busca o crea el archivo en la misma carpeta en donde se encuentra el programa. Puede declararse una variable o se puede escribir directamente el nombre del archivo, en cuyo caso, se hace entre comillas dobles.

`modo`: indica la forma en que se trabajará la información, es decir, para qué se abrirá el archivo. El modo está representado por una cadena de caracteres, en la cual encontrará algunos de los siguientes caracteres: `w` (`write`, escribir), `r` (`read`, leer), `a` (`append`, adicionar), `t` (`text`, texto), `b` (`binary` - binario) y `+` (lectura/escritura):

Texto	Binario	Aplicación
“r”, “rt”	“rb”	Abre un archivo existente para lectura.
“w”, “wt”	“wb”	Crea un archivo para escritura. Si ya existe, lo trunca a longitud cero, es decir lo reemplaza por uno nuevo.
“a”, “at”	“ab”	Permite adicionar al final un archivo existente, si no existe lo crea.
“a+”, “a+t”, “at+”	“a+b”, “ab+”	Abre un archivo para lectura y escritura. Permite adicionar al final del archivo. Si el archivo no existe, lo crea.
“r+”, “r+t”, “rt+”	“r+b”, “rb+”	Abre un archivo existente para lectura y escritura. En escritura, si el archivo existe no lo borra, adiciona el nuevo texto desde el principio del archivo, sobrescribiendo su contenido. Si el archivo no existe, genera un error.
“w+”, “w+t”, “wt+”	“w+b”, “wb+”	Abre el archivo en modo lectura y escritura. Si el archivo existe lo trunca a longitud cero, en caso contrario lo crea.

Tabla 7.1: Modos de apertura para archivos [Sznajdleder, 2017]

Para la apertura de un archivo, se procede de la siguiente manera:

```
1  FILE* puntero_archivo;
2
3  puntero_archivo = fopen (nombre, modo);
```

Con esta expresión, se está asignando el nombre externo del archivo (`nombre`) al nombre interno (`puntero_archivo`), a la vez que se especifica el modo en que será abierto y el tipo de archivo: para lectura, escritura o adición y si es de texto o binario.

`puntero_archivo`: puntero declarado por el programador, con el propósito de apuntar a la información del archivo. Se declara de tipo `FILE`, es decir, de tipo archivo.

La función `fopen`, devuelve un `NULL`[1] si se presenta algún error en la apertura del archivo, o un puntero a un dato de tipo `FILE`, si la apertura fue exitosa. Dicho esto, se puede validar la apertura del archivo y así evitar resultados inesperados en la ejecución de los programas:

```
1  FILE* puntero_archivo;
2
3  puntero_archivo = fopen (nombre, modo);
4
5  if ( puntero_archivo == NULL )
6  {
7     perror( "Error en la apertura del archivo \a" );
8     return 1;
9  }
10 else
11 {
12     //Realizar las operaciones de gestión con el archivo;
13 }
```

Se debe tener presente, que es igualmente válido formular la condición del `if` de la siguiente manera:

```
    if ( puntero_archivo = fopen (nombre, modo) == NULL )
```

En las líneas de código presentadas para la validación, se puede observar el uso de una nueva función, denominada `perror`; con ella se identifica e imprime el error ocurrido.

```
7  perror( "Error en la apertura del archivo \a" );
```

[1] `NULL`: macro definida en `stdio.h`. Especifica que no se pudo ejecutar alguna operación sobre el archivo. [Fernánadez et al., 2014]

En este caso, se muestra un mensaje similar al siguiente: `Error en la apertura del archivo: No such file or directory`. La secuencia de escape "\a", acompaña el mensaje con un sonido de alerta.

Este mismo resultado, también puede lograrse usando la función `fprintf`:

```
fprintf( stderr,"Error en la apertura del archivo: %s \a",
    strerror(errno) );
```

Con esta nueva condición, el diagrama de flujo 7.1, puede configurarse de la siguiente manera:

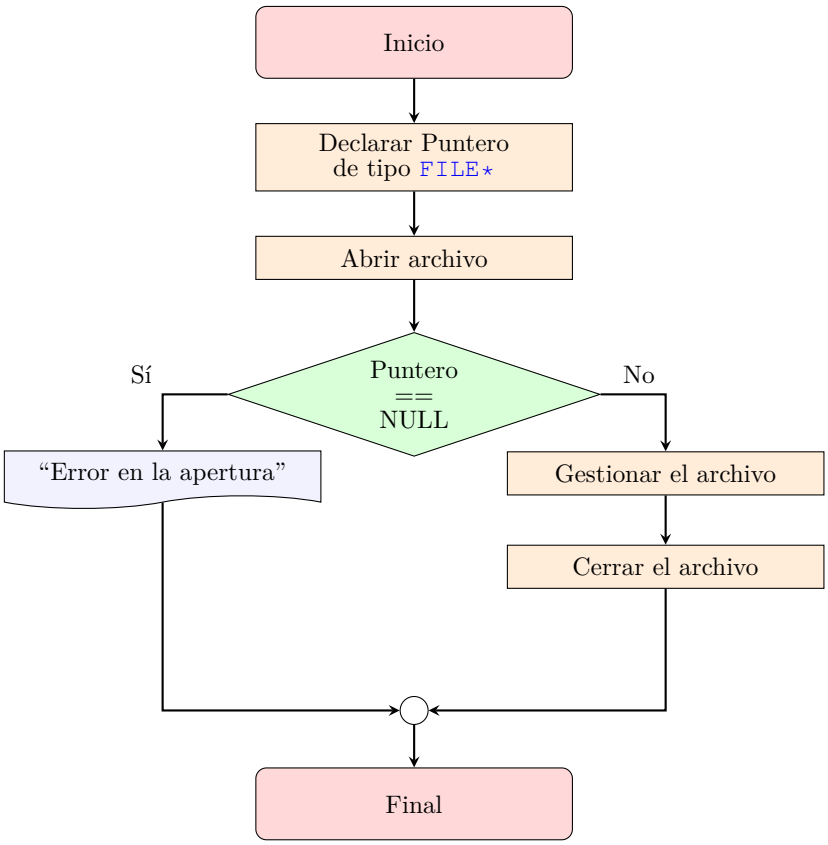


Figura 7.2: Implementación de archivos con validación en la apertura.

Aclaración:

Para algunos compiladores, el tipo de archivo a trabajar, por defecto es el de texto, por lo tanto, es suficiente con escribir "`w`", "`r`" o "`a`".

Hay que tener precaución en el modo que se use para la apertura. Por ejemplo, si se quiere adicionar información nueva al archivo existente y se abre con "`r+`" o "`w+`", se puede perder la información que había sido almacenada previamente.

Función `fclose`: cumple la función de cerrar el archivo. Es la última instrucción que se debe dar sobre el manejo del archivo.

Al ejecutar la función `fclose`, se cierra el archivo y se libera su puntero, al igual que el buffer de memoria. Si se presentara algún error en el cierre, la función devuelve el valor `EOF`[2], en caso contrario, el valor devuelto es `0`. La forma general de esta función es la siguiente:

```
fclose (puntero_archivo);
```

Actividad 7.1

Observe las siguientes líneas de código y explique qué es lo que ellas realizan:

```
1  FILE* interno;
2
3  interno = fopen ("Lista.txt","rt");
4
5  if ( interno = NULL )
6  {
7    perror ("No se pudo abrir el archivo \a");
8    interno = fopen ("Lista.txt", "wt");
9    fclose (interno);
10   interno = fopen ("Lista.txt", "rt");
11 }
```

[2] `EOF`: End Of File. Expresión que se retorna para indicar el final de un archivo.

Hasta este momento, hay que tener claridad en que todo archivo se debe abrir y cerrar; pero no tendría mucho sentido hacer esto sin haber realizado un proceso de creación, adición o de consulta de los textos o registros almacenados. Para ello existen otras funciones, las cuales se analizarán a continuación. Inicialmente, se estudiarán las funciones para la gestión de archivos de texto.

7.2. Archivos de texto

En esta clasificación se encuentran los archivos cuya información almacenada está conformada por palabras, oraciones o párrafos; es decir, por cadenas de texto, producto de la combinación de códigos ASCII. Cualquiera de los programas de este libro, son ejemplos de archivos de texto. Pueden ser creados y leídos por cualquier editor de texto, por ejemplo, en el Bloc de notas de Windows o en TextEdit de Mac[3].

Funciones para almacenar información

La razón de ser de los archivos, es la de almacenar datos que posteriormente podrán ser consultados o modificados. Las funciones que se estudiarán en este libro y que permiten guardar datos en un archivo de texto, son las siguientes: `fputs`, `fprintf`, `fputc`.

Función `fputs`: permite escribir el contenido de una cadena en un archivo abierto por el puntero. El carácter nulo no se escribe. Cuando se produce un error, la función retorna `EOF`; en caso contrario retorna un valor no negativo. Su forma general es la siguiente:

```
fputs( cadena, puntero_archivo );
```

`cadena`: es la cadena de caracteres que se almacenará en el archivo.

`puntero_archivo`: puntero de archivo que se retorna al ejecutar la función `fopen`, representa el nombre interno.

Función `fprintf`: esta función es similar a la función `printf`, que se ha venido trabajando para mostrar datos en la pantalla de su dispositivo; la diferencia se encuentra en que los datos se pueden transferir a un archivo que se guardará en un dispositivo de almacenamiento. Su forma general es:

```
fprintf( puntero_archivo, "cadena de control", argumentos );
```

[3]Cuando se graba como texto sin formato.

`puntero_archivo`: puntero de archivo que se retorna al ejecutar la función `fopen`, representa el nombre interno.

"`cadena de control`": indica el tipo de dato a imprimir.

`argumentos`: variables de las cuales se almacenará su contenido.

Función `fputc`: permite almacenar un carácter en un archivo abierto por el puntero. Si ocurre algún error en la escritura, la función retorna `EOF`, en caso contrario retorna el carácter escrito. Su forma general es la siguiente:

```
fputc( caracter, puntero_archivo );
```

`caracter`: variable de tipo carácter que captura el carácter a almacenar en el archivo.

`puntero_archivo`: puntero de archivo que se retorna al ejecutar la función `fopen`, representa el nombre interno.

A continuación, se estudiará el uso de estas funciones.

.:Ejemplo 7.1. *El siguiente programa crea un archivo de texto, que almacena una frase proporcionada por el usuario. Usa la función `fputs`.*

Análisis del problema:

- **Resultados esperados:** un archivo guardado en un dispositivo de almacenamiento, que contenga un texto digitado por el usuario.
- **Datos disponibles:** el usuario digitará un texto. Según el enunciado del ejemplo, se debe usar la función `fputs` para almacenar la información en el archivo.
- **Proceso:** seguir los pasos presentados en la Figura 7.1. La apertura del archivo se debe hacer en modo "`wt`" ya que se va a escribir en un archivo de texto. Se procede con la lectura de la frase, la cual se almacena en el archivo y luego se hace el cierre del mismo. Finalmente, termina el proceso.
- **Variables requeridas:**
  - `interno`: puntero que hará referencia al nombre interno del archivo.
  - `frase`: cadena de caracteres, donde se leerá el texto a almacenar en el archivo.

De acuerdo al anterior análisis, se propone el Programa 7.1

Programa 7.1: CrearTextofputs

```
1  #include <stdio.h>
2
3  int main()
4  {
5    FILE* interno;
6    char frase [250];
7
8    interno = fopen ( "Archivo.txt", "wt" );
9
10   printf ( "Digite un texto para ser archivado.
11             Termine con ENTER: \n\n " );
12
13   fgets (frase, sizeof ( frase ) ,stdin);
14
15   fputs ( frase, interno );
16
17   fclose ( interno );
18
19   return 0;
20 }
```

Explicación del programa:

Se declara un puntero llamado `interno`, de tipo `FILE`:

```
5  FILE* interno;
```

Con él se apuntará hacia el nombre interno del archivo que se va a crear.

Luego se declara un arreglo de caracteres donde se almacenará en la memoria del dispositivo, el texto que digite el usuario:

```
6  char frase [250];
```

Se realiza la apertura del archivo:

```
8  interno = fopen ( "Archivo.txt", "wt" );
```

El archivo externo, tendrá el nombre de `Archivo.txt`. Recuerde que se denomina nombre externo al identificador del archivo que se tendrá grabado en el dispositivo de almacenamiento. Como no se le especificó una ruta de directorios o carpetas, el archivo se grabará en la misma carpeta donde se encuentre este programa. Para el modo de apertura se empleó la cadena "`wt`", especificando así, que el archivo de texto será abierto en

modo de escritura. Si no se presenta un error en la apertura del archivo, la función relaciona el nombre externo con el nombre interno.

Luego, con la función `fgets` se lee el texto a almacenar y se guarda temporalmente en la variable `frase` (que está en la memoria primaria del dispositivo - RAM). Seguidamente, con la función `fputs` se realiza la grabación del contenido de la variable `frase` en `Archivo.txt`. En vista a que no hay que hacer más operaciones con el archivo, se procede a su cierre con la función `fclose`:

```
10  printf ("Digite un texto para ser archivado.
11           Termine con ENTER: \n\n ");
12
13  fgets (frase, sizeof(frase),stdin);
14
15  fputs (frase, interno);
16
17  fclose (interno);
```

Una vez realizado todo este proceso, se termina el programa.

Aclaración:

En todo programa que se abra un archivo con la función `fopen`, debe hacerse el correspondiente cierre con la función `fclose`, una vez se terminen las operaciones con él.

.:Ejemplo 7.2. *El siguiente programa crea un archivo de texto, que almacena una frase proporcionada por el usuario. Usa la función `fputs`. Valida la apertura del archivo.*

Como puede observar, es casi el mismo ejemplo anterior, la diferencia radica en que se va a validar la apertura del archivo.

Análisis del problema:

- **Resultados esperados:** un archivo guardado en un dispositivo de almacenamiento, que contenga un texto digitado por el usuario. Se debe informar si ocurre un error en la creación del archivo.
- **Datos disponibles:** el usuario digitará un texto.
- **Proceso:** se siguen los pasos presentados en la Figura 7.2. La apertura del archivo se debe hacer con "`wt`" ya que es la creación

de un archivo que no existe. Para evitar que el programa termine de manera inesperada, se valida la apertura del archivo: en caso de que se produzca un error, se informa; si la apertura es exitosa se procede con la lectura de la frase, se almacena en el archivo y luego se hace el cierre del mismo. Finalmente, termina el proceso.

- **Variables requeridas:**
  - `interno:` puntero que hará referencia al nombre interno del archivo.
  - `frase:` cadena de caracteres, donde se leerá el texto a almacenar en el archivo.

De acuerdo al anterior análisis, se propone el Programa 7.2

Programa 7.2: CrearTexto2fputs

```
1  #include <stdio.h>
2
3  int main()
4  {
5    FILE* interno;
6    char frase [250];
7
8    interno = fopen ( "Archivo.txt", "wt" );
9
10   if ( interno == NULL )
11   {
12     perror ( "Error en la creación del archivo \a" );
13     return 1;
14   }
15   else
16   {
17     printf ( "Digite un texto para ser archivado. Termine
           con ENTER: \n\n " );
18     fgets ( frase, sizeof(frase),stdin );
19
20     fputs ( frase, interno );
21
22     fclose ( interno );
23   }
24
25   return 0;
26 }
```

Explicación del programa:

En este programa se incorporó una decisión, con el fin de validar la apertura del archivo.

La asignación que hace la función `fopen` al puntero interno, relaciona el nombre externo del archivo (`Archivo.txt`) con el nombre interno, a la vez que le indica que el modo de apertura es de escritura en un archivo de texto ("`wt`"). Si ocurre algún error en la apertura del archivo, el puntero interno tomará el valor de `NULL`.

```
8   interno = fopen ( "Archivo.txt", "wt" );
```

Teniendo en cuenta estos resultados, se hace la validación para que el programa no vaya a tener una terminación inesperada por algún error en la apertura del archivo; para ello se emplea la siguiente estructura de decisión `if`, comparando el valor del puntero interno:

```
10  if ( interno == NULL )
11  {
12    perror ( "Error en la creación del archivo \a" );
13    return 1;
14  }
```

Consecuente a la decisión que se plantea, se imprime un mensaje informando que hubo un error en la creación del archivo. La secuencia de escape "\a", se usa para emitir un beep al mismo tiempo que se muestra el mensaje.

En este caso, se imprime el mensaje "`Error en la creación del archivo`", acompañado del mensaje correspondiente al número de error que detecte el compilador. Por ejemplo:

```
Error en la creación del archivo: No such file or directory
```

Cuando la condición es falsa, es decir, no se produce ningún error en la creación del archivo, se procede a solicitar el texto al usuario y el flujo continúa, tal como se explicó en el ejemplo anterior.

Es importante resaltar que, en el caso de este programa no es indispensable escribir la palabra reservada `else` para indicar las acciones a ejecutar cuando la condición sea falsa; esto debido a que si la condición es verdadera se informará que se produjo un error en la creación del archivo y la instrucción `return 1`, terminará con la ejecución del programa de forma inmediata.

Ahora bien, si el código se estructura sin la parte falsa de la decisión, así:

```
10  if ( interno == NULL )
11  {
12    perror ( "Error en la creación del archivo \a" );
13    return 1;
14  }
15
16  printf ( "Digite un texto para ser archivado.
17            Termine con ENTER: \n\n " );
18
19  fgets ( frase, sizeof(frase),stdin );
20
21  fputs ( frase, interno );
22
23  fclose ( interno );
24
25  return 0;
```

y la condición del `if`, resulta falsa, se ejecutan las instrucciones que hay por debajo de la llave "}" que cierra la decisión. Aunque la palabra reservada `else` no es indispensable, en el Programa 7.2 se codificó porque se considera una buena práctica establecer este tipo de alcances que dependen de una condición; de esta manera, si el software tiene algún cambio durante un proceso de mantenimiento, su lógica no se verá afectada por algún error involuntario del desarrollador.

.:Ejemplo 7.3. *En este ejemplo se crea un archivo de texto con el nombre que el usuario proporcione. Debe leer una frase que terminará en el momento que se presione la tecla Enter. Usa la función* `fputc`.

Análisis del problema:

- **Resultados esperados:** un archivo, con el nombre que el usuario le asigne, guardado en un dispositivo de almacenamiento, el cual contendrá un texto digitado a través del teclado.
- **Datos disponibles:** el nombre de un archivo y un texto para almacenar en él, ambos proporcionados por el usuario.
- **Proceso:** se le debe solicitar al usuario el nombre del archivo. Se hace la apertura de él, validándola para evitar que el programa termine de manera inesperada.

  Para leer el texto se usará la función `fputc`. Como esta función captura un carácter a la vez, es indispensable que se codifique dentro

de un proceso repetitivo, condicionado a terminar en el momento que se presione la tecla Enter, que indicará un salto de línea.

Una vez se termine de ejecutar el ciclo, finaliza el proceso.

- **Variables requeridas:**
  - `interno:` puntero que hará referencia al nombre interno del archivo.
  - `nombreArchivo:` captura el nombre externo del archivo que el usuario le asigne, con el cual será grabado en un medio de almacenamiento secundario. Es válido que el usuario digite también una ruta de carpetas o directorios.
  - `caracter:` almacena uno a uno los caracteres que conformarán el texto que se va a almacenar.

El Programa 7.3, proporciona una solución al ejercicio planteado.

Programa 7.3: CrearTextofputc

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6    FILE* interno;
7    char nombreArchivo[40];
8    char caracter;
9
10   printf ( "Digite el nombre del archivo a crear.
11             Incluya la extensión: .TXT: " );
12
13   fgets ( nombreArchivo, sizeof(nombreArchivo),stdin );
14   strtok ( nombreArchivo,"\n" );
15
16   interno = fopen ( nombreArchivo,"wt" );
17
18   if ( interno == NULL )
19   {
20     perror ( "Error en la creación del archivo. \a" );
21     return 1;
22   }
23   else
24   {
25     printf ( "Digite un texto para ser archivado en %s.
26               Termine con ENTER: \n\n ", nombreArchivo );
27
```

```
28        while ( (caracter = getchar()) != '\n' )
29        {
30          fputc ( caracter, interno );
31        }
32
33        fclose ( interno );
34      }
35
36      return 0;
37    }
```

Explicación del programa:

Luego de declarar las variables, se le solicita al usuario que digite el nombre del archivo que va a crear.

```
10    printf ("Digite el nombre del archivo a crear.
11             Incluya la extensión: .TXT: ");
12
13    fgets (nombreArchivo, sizeof(nombreArchivo),stdin);
14    strtok (nombreArchivo,"\n");
```

Con la función `fgets` se hace la lectura, que se almacena en la variable `nombreArchivo`. Con la función `strtok` se rompe una cadena en tókens, evitando así el final de línea, esto con el propósito de que no sea tomada como parte del nombre del archivo.

Al igual que en el anterior ejemplo, luego de la validación de la apertura, se solicita el texto que se va a almacenar en el archivo; es en este punto en donde se encuentra la diferencia en el modo de lectura:

```
28    while ( (caracter = getchar()) != '\n' )
29    {
30      fputc ( caracter, interno );
31    }
```

La condición del ciclo `while`, indica que se leerá un carácter mientras no sea el final de línea ("\n"), con lo cual se logra que la función `fputc`, se ejecute de forma repetitiva. Esto obedece a que `fputc`, hace la lectura de un carácter a la vez, pero como en este caso se trata de una frase o párrafo que se leerá, se deben realizar múltiples lecturas.

Luego de la lectura, se hace el cierre del archivo y finaliza el programa.

Aclaración:

En la escritura de la información en los archivos, también pudo emplearse la función `fprintf`. Si desea hacerlo, solamente haga el cambio de la línea donde está la función `fputs`, en el Programa 7.2:

```
20      fputs ( frase, interno );
```

debe cambiarla por:

```
20      printf ( interno,"%s",frase );
```

Luego de la creación de los archivos de texto, se debe proceder a la lectura de sus contenidos, aunque esta se puede hacer con cualquier editor de textos sin mayor dificultad, también, existen funciones que permiten realizar esta tarea. Entre ellas están: `fgets` y `fgetc`.

Función `fgets`: esta función permite leer un número (`n`) de caracteres desde un archivo y los almacena en un arreglo de caracteres (cadena). Su forma general es la siguiente:

```
    fgets (cadena, n, puntero_archivo)
```

La función retorna un puntero `NULL` si se presenta algún problema en la lectura, uno de estos problemas es llegar al final del archivo. La lectura se realiza hasta encontrar un final o salto de línea ("\n"), el final del archivo (`EOF`) o hasta alcanzar la cantidad de caracteres especificados en la función, valor que está dado por la variable `n` en la anterior forma general.

Función `fgetc`: esta función se usa para leer carácter a carácter el contenido de un archivo de texto. Su forma general es la siguiente:

```
    fgetc (puntero_archivo)
```

`fgetc` retorna el carácter leído si no hay errores en la operación, en caso contrario o al encontrar el final del archivo retorna `EOF`.

Dadas las explicaciones que anteceden, se procederá con los siguientes ejemplos.

.:Ejemplo 7.4. *Con el siguiente programa se podrá leer el contenido almacenado en el archivo denominado Archivo.txt, creado con los programas de los ejemplos 7.1 y 7.2.*

Análisis del problema:

- **Resultados esperados:** la lectura del contenido del archivo Archivo.txt.
- **Datos disponibles:** un archivo de texto, denominado `Archivo.txt`.
- **Proceso:** seguir los pasos presentados en la Figura 7.2. La apertura del archivo se debe hacer en modo "`rt`", que permite la lectura de un archivo de texto. Se debe realizar la validación de la apertura del archivo, igual que en los programas anteriores. Se procede a la lectura del archivo y luego al cierre.
- **Variables requeridas:**
  - `interno:` puntero que hará referencia al nombre interno del archivo.
  - `frase:` arreglo de caracteres (cadena), donde se almacenará el texto recuperado del archivo.

Se propone el Programa 7.4. como solución a este ejemplo:

Programa 7.4: LeerTextofgets

```
1  #include <stdio.h>
2
3  int main()
4  {
5    FILE* interno;
6    char frase [250];
7
8    interno = fopen ( "Archivo.txt", "rt" );
9
10   if ( interno == NULL )
11   {
12     perror ( "Error en la apertura del archivo \a" );
13     return 1;
14   }
15   else
16   {
17     printf ( "Consultando el contenido de Archivo.txt:
18               \n\n" );
```

```
19
20        fgets ( frase, 250, interno );
21        printf ( "%s", frase );
22
23        fclose ( interno );
24      }
25      return 0;
26    }
```

Explicación del programa:

Al igual que en los ejemplos anteriores, se declaran las variables necesarias y se hace la apertura del archivo con su correspondiente validación.

En la apertura se tiene en cuenta que el nombre del archivo externo es `Archivo.txt` y que, al tratarse de un proceso de lectura de un archivo de texto, el modo de apertura debe ser "`rt`":

```
8    interno = fopen ( "Archivo.txt", "rt" );
```

Tenga en cuenta que para que el programa funcione correctamente, `Archivo.txt` debe estar en la misma carpeta donde lo ejecutará.

Cuando la apertura del archivo se hace de forma correcta, se ejecuta la función `fgets`, que lee desde el archivo apuntado por interno una cantidad de 250 caracteres y los almacena en la variable `frase`. Recuerde que la función termina su lectura por cualquiera de las siguientes tres razones: que encuentre el final del archivo, el salto de línea o la cantidad de caracteres especificada (250):

```
20    fgets ( frase, 250, interno );
21    printf ( "%s", frase );
22
23    fclose ( interno );
```

Cuando ejecute este Programa notará que solo muestra la primera línea del archivo. Si `Archivo.txt`, tiene más de una línea que contenga el salto de línea correspondiente, debe realizar varias lecturas. El siguiente ejemplo, ilustra esta situación.

.:Ejemplo 7.5. *Con este programa se puede leer un archivo de texto que contenga varios saltos de línea.*

Análisis del Problema:

Teniendo en cuenta que este ejemplo es una variante del ejemplo 7.4, se explicará únicamente que el proceso de lectura se realizará de forma repetitiva y que para ello se usará un ciclo. Los demás elementos del análisis permanecen iguales.

Programa 7.5: LeerTexto2gets

```
1   #include <stdio.h>
2
3   int main()
4   {
5     FILE* interno;
6     char frase [250];
7
8     interno = fopen ( "Archivo.txt", "rt" );
9
10    if ( interno == NULL )
11    {
12      perror ( "Error en la apertura del archivo \a" );
13      return 1;
14    }
15    else
16    {
17      printf ( "Consultando el contenido de Archivo.txt:
18                \n\n" );
19
20      while ( fgets(frase, 250, interno) != NULL )
21      {
22        printf ( "%s", frase );
23      }
24      fclose ( interno );
25    }
26    return 0;
27  }
```

Explicación del programa:

La única diferencia entre este ejemplo y el 7.4 está en la lectura del archivo. Para ello se empleó un ciclo `while`, buscando así que la función `fgets` se ejecute en repetidas ocasiones:

```
20   while ( fgets (frase, 250, interno) != NULL )
21   {
22     printf ("%s", frase);
23   }
```

La condición que se codificó establece que `fgets` leerá desde el archivo, cadenas de 250 caracteres y las copiará en la variable `frase`. Mientras no encuentre el valor de `NULL`, imprimirá su contenido. La función `fgets`, devuelve `NULL` cuando llega al final del archivo.

Actividad 7.2

Tomando como base el ejemplo 7.5, diseñe un programa que cuente la cantidad de líneas que posee un archivo de texto.

.:Ejemplo 7.6. *Se dispone de un archivo llamado Archivo.txt y se desea leer todo su contenido con la función `fgetc`.*

Análisis del problema:

- **Resultados esperados:** la lectura del contenido del archivo `Archivo.txt`.
- **Datos disponibles:** un archivo de texto, denominado `Archivo.txt`.
- **Proceso:** se abre el archivo en modo "`rt`", que permite su lectura. Se debe realizar la validación de la apertura del archivo, igual que en los programas anteriores. Se procede a la lectura del archivo y luego al cierre. Para la lectura se usará la función `fgetc`, la cual lee uno a uno los caracteres del archivo, por lo tanto, es indispensable realizar múltiples lecturas hasta encontrar la marca de fin de archivo (`EOF`).
- **Variables requeridas:**
  - `interno:` puntero que hará referencia al nombre interno del archivo.
  - `caracter:` en esta variable se almacenará el carácter que sea leído desde el archivo.

Se propone el Programa 7.6. como solución a este ejemplo:

Programa 7.6: LeerTextofgetc

```
#include <stdio.h>

int main()
{
  FILE* interno;
  char caracter;

  interno = fopen( "Archivo.txt", "rt" );

  if ( interno == NULL )
  {
    perror ( "Error en la apertura del archivo \a" );
    return 1;
  }
  else
  {
    printf ( "Consultando el contenido de Archivo.txt:
             \n\n" );

    while ( (caracter = fgetc (interno)) != EOF )
    {
      printf ( "%c", caracter );
    }
    fclose( interno );
  }

  return 0;
}
```

Explicación del programa:

Teniendo en cuenta que el inicio del programa es igual a los estudiados hasta ahora, se procederá a explicar solamente la parte donde se realiza la lectura:

```
while ( (caracter = fgetc (interno)) != EOF )
{
  printf ("%c", caracter);
}
```

En vista de que la función `fgetc`, lee carácter a carácter, se codificó un ciclo que hiciera dicha lectura mientras el carácter leído no fuera `EOF` (End Of File). En cada lectura, el carácter extraído es almacenado en la variable `caracter` y, mientras la condición del `while`, sea verdadera se realiza la impresión en pantalla: `printf` (“ `%c`”, `caracter`).

Aclaración:

Para adicionar caracteres, líneas o párrafos a los archivos de texto, se usan las mismas funciones empleadas para crearlos, solamente se debe cambiar el modo de apertura de "`wt`", a "`at`".

Actividad 7.3

1. Diseñe un programa, que lea desde el teclado un carácter cualquiera y determine cuántas veces está dicho carácter dentro de un archivo llamado `Texto.txt`.

   Nota: el archivo `Texto.txt`, lo puede crear con un editor de textos.

2. Mediante un programa en Lenguaje C, lea el contenido del archivo creado para el punto 1 de esta actividad (`Texto.txt`) y genere otro archivo llamado `TextoCopia.txt`, con el mismo contenido.

3. Implemente un programa que permita adicionar líneas de texto o párrafos a uno de los archivos creados en los ejemplos anteriores.

7.3. Archivos Binarios

Antes de entrar en detalle con el manejo de archivos binarios, es importante familiarizarse con algunos conceptos, tales como: **dato**, **campo**, **registro**, **archivo** y **estructura**. Cuando se desea guardar los datos de alguien en la agenda de contactos del teléfono celular, se están manejando todos estos conceptos de manera transparente para el usuario. Por ejemplo, la aplicación de contactos de un teléfono móvil, le permite guardar el nombre, el número telefónico, el correo electrónico, el año de nacimiento y muchos otros datos más.

La tabla 7.2, contiene los datos de 3 contactos. Cada columna tiene un encabezado: nombre, número telefónico, correo y año nacimiento; con el cual se identifican los campos, es decir, la casilla en donde se escribe cada dato.

Nombre	Número Telefónico	Correo	Año Nacimiento
Luna Guzmán P	5343206918534	luna@tucorreo.com	2001
Benji Giraldo L	4563118734353	benji@gogmail.com	1998
Bruno Botero V	7634972736421	bruno@maxmail.com	1999

Tabla 7.2: Lista de contactos

Por ejemplo, en la columna que identifica el campo llamado nombre, se tienen tres datos registrados de forma individual: Luna Guzmán P, Benji Giraldo L y Bruno Botero V; cada fila de la tabla representa un registro, el cual almacena la información de un contacto.

En el primer registro de la tabla se posee la siguiente información:

Nombre:	Luna Guzmán P
Número telefónico:	5343206918534
Correo:	luna@tucorreo.com
Año de nacimiento:	2001

En atención a lo expuesto, se pueden definir los siguientes conceptos:

- **Dato:** es la mínima unidad de información y se registra en un campo.
- **Campo:** es el lugar donde se almacena un dato.
- **Registro:** es una agrupación de campos que guardan una relación entre sí.
- **Archivo:** es una agrupación de registros que guardan una relación entre sí, en el ejemplo anterior, el archivo está representado por toda la tabla.

Por otra parte, una estructura, en su forma más simple, se puede definir como: un conjunto de variables que se referencian bajo el mismo nombre. De acuerdo al tema que le concierne a este capítulo, el de archivos, un registro en Lenguaje C, debe ser tratado como una estructura.

Para declarar una estructura se usa la siguiente forma general:

```
struct nombreRegistro
{

  campos;

};
```

`struct`: es parte del conjunto de las palabras reservadas del lenguaje C. Se usa para diseñar la plantilla de una estructura.

`nombreRegistro`: es un identificador que asigna el programador a la estructura o registro.

`campos`: son los miembros o elementos que conforman la estructura y están representados por una variable con su respectivo tipo.

De este modo, la estructura o registro que representa la Tabla 7.2 del ejemplo que se viene considerando, quedaría así:

```
struct contacto
{
  char nombre [30];
  char numTelefonico [20];
  char correo [40];
  int anioNacimiento;

};
```

En el caso anterior, el registro tiene el nombre de `contacto` y está compuesto por 4 elementos o campos, declarados mediante las variables: `nombre`, `numTelefonico`, `correo` y `anioNacimiento`, cada una con su respectivo tipo de dato. Estas variables, son los campos donde se almacenarán los datos que proporcione el usuario y que en conjunto formarán uno a uno los registros de los contactos.

Para poder trabajar con los elementos (campos) de una estructura, debe declararse una variable con ese registro, usando la siguiente forma general:

```
struct nombreRegistro variables;
```

`nombreRegistro`: identificador con el que se nombró la estructura o registro.

`variables`: identificador para la variable, de tipo `nombreRegistro`, con la cual se hará referencia a los elementos de la estructura. Puede declararse una o varias variables.

Consecuente con el ejemplo que se viene exponiendo, se declarará una variable de tipo `contacto`, denominada `campoContacto`:

```
struct contacto campoContacto;
```

De igual forma, la declaración del registro y de la variable de su tipo, puede realizarse de la siguiente forma:

```
1  struct nombreRegistro
2  {
3
4    campos;
5
6  } variables;
```

Ejemplo:

```
1  struct contacto
2  {
3    char nombre [30];
4    char numTelefonico [20];
5    char correo [40];
6    int anioNacimiento;
7
8  } campoContacto;
```

Para hacer referencia a los elementos o campos de la estructura, se hace uso del operador de punto (.). Para ello, se usa la siguiente forma general:

Variable del tipo registro	.	Elemento o campo

Por ejemplo, para hacer referencia al campo `numTelefonico`, se debe escribir la siguiente sentencia:

```
campoContacto.numTelefonico;
```

En este mismo orden, resulta oportuno analizar la utilidad de los tipos de datos definidos por el programador. Con la palabra reservada `typedef`, Lenguaje C, le permite definir un nuevo tipo de dato que resulta equivalente a alguno de los propios del Lenguaje, así [Girón, 2015]:

```
typedef tipo tipoProgramador;
```

`tipo` es uno de los tipos propios del Lenguaje y `tipoProgramador` es la forma como se va a identificar el nuevo tipo definido por el programador.

De acuerdo al anterior contexto, el registro que se viene trabajando como ejemplo, se puede redefinir de la siguiente forma:

```
typedef struct
{
  char nombre [30];
  char numTelefonico [20];
  char correo [40];
  int anioNacimiento;
} contacto;
```

Con referencia a todo lo anterior y teniendo claridad sobre el concepto de registro, se puede definir un archivo binario como una secuencia de bytes que tienen una correspondencia uno a uno con un dispositivo externo. A diferencia de los archivos de texto, estudiados al inicio del capítulo, su contenido no se puede visualizar desde un editor de texto; para poder hacerlo, se requiere la construcción de programas en Lenguaje C diseñados para tal fin.

Actividad 7.4

Analice su documento de identificación (Cédula o Tarjeta de Identidad) y diseñe la estructura o registro para almacenar los datos que se presentan en él.

Aclaración:

Para visualizar el contenido de un archivo de texto se puede usar un editor de texto; mientras que para un archivo binario se requiere de programas en Lenguaje C, diseñados para tal fin.

Manejo de archivos binarios.

Para la gestión de los registros de un archivo binario, también existe un protocolo similar al que se estudió en los archivos de texto y que puede apreciarse en la Figura 7.2; sin embargo, se presenta una discreta diferencia que se puede ver en la Figura 7.3.

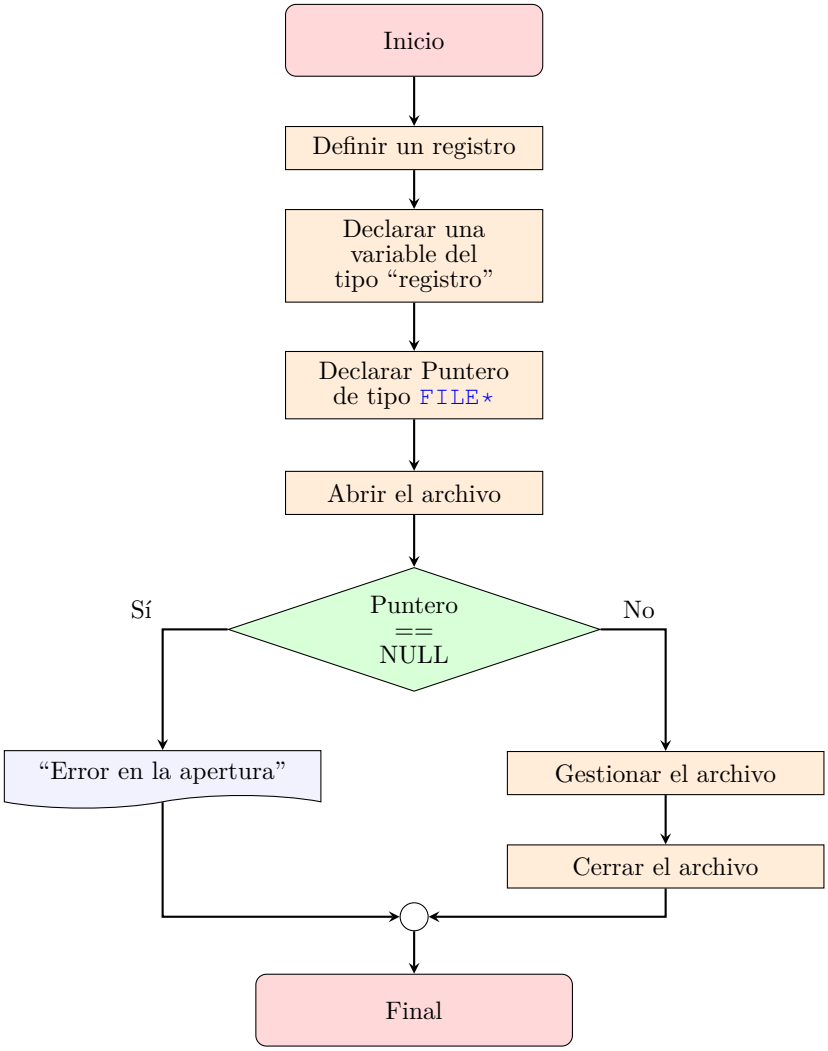


Figura 7.3: Implementación archivos Binarios

Observe que al inicio de los pasos que se ejecutan, hay dos que no se tenian en cuenta con los archivos de texto: definir un registro y declarar una variable del tipo “registro”.

En cuanto a los demás elementos presentes en el diagrama, existen diferentes funciones para el manejo de los archivos binarios. Varias de ellas ya fueron usadas, por ejemplo: `fopen` y `fclose`. Es conveniente repasar su teoría, para entender de manera más fácil los siguientes programas.

Funciones para archivos binarios.

Función `fwrite`: permite grabar los datos de uno o varios registros, de longitud constante en un archivo. Puede ser usada, tanto en archivos binarios, como en archivos de texto. Su forma general es la siguiente:

```
fwrite (&variable_registro, tamaño, n, puntero_archivo);
```

`&variable_registro`: variable que apunta a la zona de memoria que contiene los datos que se almacenarán en el archivo.

`tamaño`: longitud del registro. Para determinar este tamaño se hace uso del operador `sizeof`, el cual se analizará en el próximo ejemplo.

`n`: cantidad de registros que se grabarán cuando se ejecute la función. Recuerde, que la función puede grabar uno o varios registros a la vez.

`puntero_archivo`: apunta a la información de un archivo en particular donde se almacenará el registro. Con esta variable el programa reconoce, de manera interna, el nombre del archivo.

La función `fwrite` devuelve el número de registros grabados; si el valor devuelto es menor que el del parámetro `n` es porque se presentó un error de escritura.

Función `fread`: esta función lee registros desde un archivo binario. Su forma general es la siguiente:

```
fread (&variable_registro, tamaño, n, puntero_archivo);
```

`&variable_registro`: este primer parámetro apunta a una posición de memoria que almacenará los datos leídos desde el archivo apuntado por `puntero_archivo`. Cómo se puede apreciar, el resto de parámetros son iguales a los de la función `fwrite` y trabajan de manera similar.

Cuando la lectura del registro haya presentado algún error, la función retorna un valor diferente a la cantidad de registros que debe leer (`n`). Cuando alcanza el fin del archivo (`EOF`) la función devuelve un valor de 0.

Función `fseek`: permite ubicar el puntero de archivo en la posición que se especifique mediante un valor de tipo `long`. También puede ser usada con archivos de texto. Su forma general se expresa así:

```
fseek (puntero_archivo, desplazamiento, origen);
```

`puntero_archivo`: apunta a la información de un archivo en particular. Con esta variable el programa reconoce, de manera interna, el nombre del archivo.

`desplazamiento`: variable de tipo `long`, que indica el número de posiciones que avanzará desde el origen que se especifique en el tercer parámetro de la función.

`origen`: este tercer parámetro, de tipo entero, establece la posición desde donde comenzará la búsqueda del registro. Los valores que puede tomar están referenciados en la Tabla 7.3.

Origen	Valor	Constante
Comienzo de archivo	0	`SEEK_SET`
Posición actual	1	`SEEK_CUR`
Fin de archivo	2	`SEEK_END`

Tabla 7.3: Valores de referencia para el parámetro origen

Tal como se aprecia en la Tabla 7.3, es igualmente válido usar las macros o identificadores: `SEEK_SET`, `SEEK_CUR` y `SEEK_END`. La función retorna un valor diferente de cero (0) si se llegara a presentar alguna inconsistencia.

Función `feof`: identifica el final del archivo, el cual es reconocido por la marca `EOF` (End Of File). Su forma general es la siguiente:

```
feof (puntero_archivo);
```

La función retorna un valor diferente de cero (verdadero) cuando encuentra el final del archivo y 0 en caso contrario.

Con el análisis de las anteriores funciones, ya se tienen los elementos necesarios para crear algunos programas para el manejo de archivos binarios.

Inicialmente se va a plantear un enunciado general y a partir de él, se propondrán los ejemplos que ilustrarán el uso de varias funciones.

.:Enunciado general: Nómina empleados

Para los ejemplos 7.7, 7.8, 7.9, 7.10, 7.11 y 7.12 se trabajará el pago del sueldo de un empleado que labora por horas. El sueldo depende del número de horas que labore, del valor que devengue por hora y de los descuentos que tenga. Es un ejemplo básico y, es precisamente por este motivo que se ha elegido, por lo didáctico que resulta; además, se desea demostrar cómo se hacen cálculos con variables que hacen parte de una estructura o registro.

Los datos de cada empleado se almacenarán en un archivo, con el cual se deben realizar las siguientes acciones:

- *Crear y adicionar registros*
- *Consultar de forma individual y general los datos allí almacenados.*
- *Modificar los datos de un registro.*

Adicionalmente a los datos ya mencionados, de cada empleado se conoce el número de la cédula y el nombre completo.

Análisis general del Problema

Se debe crear un archivo, el cual se grabará en un dispositivo de almacenamiento externo, para este caso se le dará el nombre de `Nomina.dat` (la extensión `.dat` hace referencia a que es un archivo de datos, puede tener cualquier otra extensión).

`Nomina.dat` almacenará registros, cada uno con los siguientes datos de los empleados: cédula de ciudadanía para tener una identificación única de la persona, nombre completo, cantidad de horas laboradas y el valor de la hora; el sueldo, a pesar de que se va a calcular, no se almacenará dentro del archivo. En este ejemplo el cálculo del sueldo es una operación muy elemental y no requiere de mucho procesamiento de la máquina, por lo cual resulta más eficiente hacer el cálculo en memoria y no consumir espacio en el dispositivo de almacenamiento[4].

Basados en este enunciado, se construirán los programas para gestionar los registros del archivo.

[4]Dependiendo de la complejidad y la cantidad de los cálculos, se debe hacer una evaluación si es mejor sacrificar tiempo de procesamiento o espacio en el dispositivo de almacenamiento.

.:Ejemplo 7.7. *Diseñar un programa para crear los primeros registros del archivo* `Nomina.dat`.

Análisis del Problema:

- **Resultados esperados:** una vez ejecutado el programa, se espera obtener un archivo binario, en él deben quedar almacenados los registros que el usuario digite.
- **Datos disponibles:** el nombre del archivo: `Nomina.dat`. De cada empleado se tienen los siguientes datos: cédula, nombre completo, cantidad de horas laboradas, valor de la hora y el valor de los descuentos. Adicionalmente, se debe calcular el sueldo de acuerdo a la cantidad de horas laboradas y el valor que recibe por cada hora.
- **Proceso:** se deben seguir los pasos establecidos en la Figura 7.3. Para definir el registro se usará la palabra reservada `struct`; luego de la declaración de la variable del tipo del registro, se procede a declarar el puntero y seguidamente se hace la apertura del archivo con la función `fopen`. Se solicitan los datos de cada empleado, se calcula el sueldo y se realiza la grabación en el archivo, para ello se usa la función `fwrite`. Una vez finalizado el proceso con los registros, se hace el cierre con la función `fclose`.

  La solicitud de los datos de cada registro de los empleados, se hará repetitivamente, involucrando así el uso de una estructura de repetición.
- **Variables requeridas:** en la teoría estudiada, se estableció que todo registro está formado por campos, esos campos deben representarse mediante variables y así formar una estructura. Las variables que representarán los campos en este ejemplo son:

  - `cedula`: documento de identificación del empleado.
  - `nombre`: nombre completo del empleado.
  - `horasLaboradas`: horas que el empleado laboró.
  - `valorHora`: compensación monetaria que recibirá el empleado por cada hora laborada.
  - `descuentos`: cantidad de dinero que será descontada de su salario.

  Para el registro o estructura, se debe trabajar una variable, esta agrupará todos los campos anteriores:

- `empleado`: registro del empleado.

De acuerdo a los pasos que se establecieron para trabajar con archivos binarios, es necesario declarar una variable del tipo del registro, en este caso, una variable de tipo `empleado`:

- `campoEmpleado`: con esta variable se podrá tener acceso a la información de cada uno de los campos del registro.

Adicionalmente, es necesario declarar otras variables para el manejo del archivo:

- `interno`: puntero de archivo y nombre interno.
- `siga`: controlará la repetición del ciclo.
- `sueldo`: para realizar el cálculo del sueldo que devengará el empleado.

Para la solución del problema, se propone el siguiente programa:

Programa 7.7: CrearBinario1

```
1  #include <stdio.h>
2  #include <string.h>    // Para strtok()
3  #include <ctype.h>     // Para toupper()
4
5  struct empleado
6  {
7    char cedula[13];
8    char nombre[30];
9    int horasLaboradas;
10   float valorHora, descuentos;
11 };
12
13 struct empleado campoEmpleado;
14
15 int main()
16 {
17   FILE* interno;
18   char siga;
19   float sueldo;
20
21   if ((interno=fopen ("Nomina.dat","wb")) == NULL)
22   {
23     perror ("Error en la creación del archivo \a");
24     return 1;
25   }
```

```
26    else
27    {
28      do
29      {
30        printf("Ingrese los siguientes datos: \n\n");
31
32        printf("         Cédula: ");
33        fgets (campoEmpleado.cedula,
34             sizeof(campoEmpleado.cedula),stdin);
35        strtok(campoEmpleado.cedula,"\n");
36
37        printf("         Nombre: ");
38        fgets (campoEmpleado.nombre,
39               sizeof(campoEmpleado.nombre),stdin);
40        strtok(campoEmpleado.nombre,"\n");
41
42        printf("Horas Laboradas: ");
43        scanf("%d", &campoEmpleado.horasLaboradas);
44
45        printf("    Valor hora : ");
46        scanf("%f", &campoEmpleado.valorHora);
47
48        printf("     Descuentos: ");
49        scanf("%f", &campoEmpleado.descuentos);
50
51        sueldo = (campoEmpleado.horasLaboradas
52                  * campoEmpleado.valorHora)
53                  - campoEmpleado.descuentos;
54
55        printf("\n   El sueldo es: %.2f \n\n", sueldo);
56
57        fwrite(&campoEmpleado,sizeof(struct empleado),
58            1,interno);
59
60        printf( "\nLos datos fueron adicionados al archivo.
61             \n\n" );
62
63        printf (" Adicionar otro empleado S/N?: ");
64        scanf ( " %c", &siga );
65        siga = toupper(siga);
66        getchar();
67        printf("\n\n");
68      } while(siga == 'S');
69
70      fclose(interno);
71      return 0;
72    }
73  }
```

Explicación del Programa:

Inicialmente se incluyó la biblioteca `stdio.h` para poder trabajar con las funciones de archivos, de igual forma, se relacionaron las bibliotecas `string.h` y `ctype.h`, las cuales contienen los prototipos de las funciones `strtok` y `toupper`, respectivamente. Recuerde que `strtok` permite romper una cadena en tókens y `toupper` convierte una letra (carácter) en su equivalente mayúscula.

Seguidamente, se declaró una variable de estructura denominada `empleado`, es decir, se diseñó una plantilla con los datos que contendrá el registro de cada empleado.

```
5   struct empleado
6   {
7     char cedula[13];
8     char nombre[30];
9     int horasLaboradas;
10    float valorHora, descuentos;
11  };
```

Del mismo modo, se hizo la declaración de la variable de tipo del registro (`empleado`), con la cual se hace referencia a cada uno de los campos de la estructura:

```
13  struct empleado campoEmpleado;
```

Hay que tener presente que el registro y la variable del tipo del registro, también se pueden declarar así:

```
    struct empleado
    {
      char cedula[13];
      char nombre[30];
      int horasLaboradas;
      float valorHora, descuentos;
    } campoEmpleado;
```

Siguiendo con la explicación del programa, dentro de la función `main`, se encuentra la declaración del puntero del registro y otras variables: `FILE* interno`.

La siguiente instrucción a ejecutar es la apertura del archivo, con su correspondiente validación:

```
21  if ((interno=fopen("Nomina.dat","wb")) == NULL)
```

Se observa que, el nombre externo del archivo a crear es: `Nomina.dat`. y su modo de apertura indica que se abre un archivo binario para escritura (“`wb`”). Si se presenta algún error en la apertura, se informa y termina la ejecución del programa con `return 1`; esta parte es igual a las ya analizadas en los ejemplos de los programas para archivos de texto. En el caso de no presentarse ningún error, se procede a solicitar los datos que conforman el registro de cada empleado:

```
30   printf ("Ingrese los siguientes datos: \n\n");
31
32   printf ("         Cédula: ");
33   fgets (campoEmpleado.cedula,
34        sizeof (campoEmpleado.cedula), stdin);
35   strtok (campoEmpleado.cedula,"\n");
```

Lo especial a resaltar en estas instrucciones, es la forma de referirse a los campos del registro; se usa la variable del tipo de registro, separada del campo mediante un punto:

```
     campoEmpleado.cedula
```

Luego de solicitar los demás datos, se procede a realizar el cálculo y la impresión del sueldo:

```
51   sueldo = (campoEmpleado.horasLaboradas
52           * campoEmpleado.valorHora)
53             - campoEmpleado.descuentos;
54
55   printf("\n   El sueldo es: %.2f \n\n", sueldo);
```

Se observa que los campos del registro: `horasLaboradas`, `valorHora` y `descuentos`, están precedidos de un punto y de la variable de tipo del registro, lo cual no es el caso de la variable `sueldo`, esto debido a que ella no es parte de la estructura.

Seguidamente, con la función `fwrite`, se procede a la grabación del registro:

```
57   fwrite (&campoEmpleado, sizeof (struct empleado),
58          1, interno);
```

Con el parámetro `&campoEmpleado`, se apunta a la dirección de memoria en donde están almacenados los datos que el usuario digite. La expresión `sizeof (struct empleado)` indica que se debe grabar el número de bytes de acuerdo al tamaño del registro, el número `1` señala que se grabará un solo registro en el archivo señalado por el puntero `interno`.

El siguiente paso es informar que se hizo la grabación de los datos en el archivo. Luego, se pregunta si se desea continuar con la grabación de registros; en caso afirmativo, se repite el proceso, de lo contrario termina la solicitud de datos, se cierra el archivo y termina el programa.

.:Ejemplo 7.8. *Solucionar el Ejemplo 7.7, pero a través del uso de funciones.*

Análisis del problema

Este análisis se concentrará en el proceso, ya que los resultados esperados y los datos disponibles son los mismos que se estudiaron en el Ejemplo 7.7.

- **Proceso:** el programa se estructurará en 3 funciones, más la función principal (`main`). Desde `main` se invocará la apertura del archivo, la lectura de los datos y la pregunta si se desea continuar grabando registros; estas acciones estarán ubicadas dentro de las funciones a implementar.

Con este corto análisis, se planteará el Programa 7.8 como una posible solución. Hay que hacer énfasis que es una alternativa entre muchas posibles codificaciones que se pueden realizar.

Programa 7.8: CrearBinario2

```
1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4
5  #define ERROR      1
6  #define OK         0
7
8  #define NOMBRE_ARCHIVO "Nomina.dat"
9
10 typedef struct
11 {
12   char  cedula[13];
13   char  nombre[30];
14   int   horasLaboradas;
15   float valorHora, descuentos;
16 } empleado;
17
18 void leerEmpleado ( empleado *e );
19 int  abrirArchivo ( char *nombreArchivo, FILE **archivo );
20 char continuar();
```

```
21
22  int main ()
23  {
24    FILE* interno;
25    empleado campoEmpleado;
26
27    if (abrirArchivo ( NOMBRE_ARCHIVO, &interno ) == ERROR)
28    {
29      perror( "Error en la apertura del archivo \a" );
30      return ERROR;
31    }
32    else
33    {
34      do
35      {
36        leerEmpleado( &campoEmpleado );
37
38        fwrite( &campoEmpleado, sizeof( empleado ),
39              1, interno );
40
41        printf( "\nLos datos fueron adicionados al archivo.
42              \n\n" );
43
44      } while ( continuar() == 'S' );
45
46      fclose(interno);
47
48      return OK;
49    }
50  }
51
52  int  abrirArchivo ( char *nombreArchivo, FILE **archivo )
53  {
54    if (( *archivo = fopen( nombreArchivo, "wb" )) == NULL )
55    {
56      return ERROR;
57    }
58    else
59    {
60      return OK;
61    }
62  }
63
64  void leerEmpleado ( empleado *e )
65  {
66    float sueldo;
67
68    printf("Ingrese los siguientes datos: \n\n");
69
```

```
70      printf ( "          Cédula: " );
71      fgets (e -> cedula, sizeof(e -> cedula),stdin);
72      strtok(e->cedula,"\n");
73
74      printf ( "          Nombre: " );
75      fgets (e->nombre, sizeof(e->nombre),stdin);
76      strtok(e->nombre,"\n");
77
78      printf ( "Horas Laboradas: " );
79      scanf  ( "%d", &e->horasLaboradas );
80
81      printf ( "     Valor hora: " );
82      scanf  ( "%f", &e->valorHora );
83
84      printf ( "     Descuentos: " );
85      scanf  ( "%f", &e->descuentos );
86
87      sueldo = e->horasLaboradas *
88            e->valorHora - e->descuentos;
89
90      printf ( "\n   El sueldo es: %.2f \n", sueldo);
91    }
92
93    char continuar()
94    {
95      char siga;
96
97      printf ( "Adicionar otro empleado S/N?: " );
98      scanf ( " %c", &siga );
99      siga = toupper( siga );
100
101     getchar();
102     printf("\n\n");
103
104     return (siga);
105   }
```

Explicación del programa

Este programa, tiene la misma función que la del ejemplo anterior, se encarga de abrir un archivo binario y escribir en él los registros que el usuario digite. La diferencia entre ambos, radica en la forma en que se estructuraron. Este está codificado haciendo uso de funciones propias del programador.

Al inicio, se incluyeron las bibliotecas (`stdio.h`, `ctype.h` y `string.h`) para que sean reconocidas las funciones propias del Lenguaje, tal y como se explicó con el ejemplo anterior.

Luego, con la directiva `#define` se declaran 3 constantes simbólicas.

```
5   #define ERROR           1
6   #define OK              0
7
8   #define NOMBRE_ARCHIVO "Nomina.dat"
```

Las constantes simbólicas `ERROR` y `OK` se usan, respectivamente, para indicar si se presenta o no un error en la apertura del archivo. El valor de una de estas constantes es retornado por la función `abrirArchivo`, que se explicará más adelante.

La constante `NOMBRE_ARCHIVO` permite establecer el nombre externo del archivo, es una forma diferente de hacerlo con relación al programa del Ejemplo 7.7.

La siguiente declaración crea la plantilla para el registro:

```
10  typedef struct
11  {
12    char  cedula[13];
13    char  nombre[30];
14    int   horasLaboradas;
15    float valorHora, descuentos;
16  } empleado;
```

Tiene una pequeña variación con relación al ejemplo anterior; en este caso, se usó la palabra reservada `typedef` para definir un tipo de dato del usuario (programador). El nombre de este tipo de dato es `empleado` que, en este programa, es equivalente a la estructura `empleado`.

Teniendo en cuenta que el enunciado solicita que se usen funciones para la solución, se codificaron los siguientes prototipos:

- `void leerEmpleado ( empleado *e )`: lee los campos del registro, no devuelve ningún valor.
- `int abrirArchivo ( char *nombreArchivo,FILE ** archivo )`: abre el archivo y retorna un valor entero de acuerdo al valor que devuelva la función `fopen`.
- `char continuar()`: pregunta si desea continuar, devuelve un valor de tipo `char` con la respuesta.

Dentro de la función `main` se declaró el puntero del archivo, también la variable `campoEmpleado` del tipo del registro (`empleado`), con ella se hace referencia a los campos de la estructura.

Antes de continuar con la ejecución de las instrucciones que se encuentran dentro de la función `main`, se explicarán en detalle las funciones creadas.

La primera de estas funciones de programador es `abrirArchivo`:

```
52  int  abrirArchivo ( char *nombreArchivo, FILE **archivo )
53  {
54    if (( *archivo = fopen( nombreArchivo, "wb" ) ) == NULL)
55    {
56      return ERROR;
57    }
58    else
59    {
60      return OK;
61    }
62  }
```

Esta función presenta dos parámetros: el primero `*nombreArchivo`, es un puntero que recibe la dirección del nombre externo del archivo (`NOMBRE_ARCHIVO`); el segundo es un puntero de un puntero, este recibe como valor, la dirección del puntero al nombre interno del archivo (`&interno`).

Dentro de esta función, se hace la apertura del archivo, con su respectiva validación:

```
54  if (( *archivo = fopen( nombreArchivo, "wb" ) ) == NULL)
```

Si ocurre algún error en la apertura del archivo, la función `fopen` devuelve `NULL`, haciendo la condición del `if` verdadera, por lo tanto, se informa la situación y la función `abrirArchivo` retorna el valor de la constante `ERROR`:

```
29  perror( "Error en la apertura del archivo \a" );
30  return ERROR;
```

Por el contrario, si la condición es falsa, la función `abrirArchivo` retorna el valor de la constante simbólica `OK` (`return OK`).

Tenga en cuenta que el llamado a `abrirArchivo` se hizo desde la función `main`, por lo tanto, el retorno se hará al sitio donde fue invocada.

La siguiente función que se codificó, se denomina `leerEmpleado`. Como su nombre lo indica se usará para leer los datos del empleado.

```
64   void leerEmpleado ( empleado *e )
```

Es una función que no retorna valores (`void`), su único parámetro es un puntero de tipo `empleado`, que recibe la dirección de memoria de la variable `campoEmpleado`; de esta forma los datos que sean manipulados con el puntero `*e`, serán almacenados en dicha posición de memoria.

Localmente, dentro de la función, se declaró la variable `sueldo`, allí se almacena el cálculo correspondiente. Seguidamente, se solicitan los datos del empleado, se calcula el sueldo y se imprime. La función devuelve el control al `main`, de donde fue invocada.

Cabe resaltar el hecho de que, para trabajar los campos del registro, se hace de forma diferente a la expresada en el programa del ejemplo anterior. En ese programa se hizo a través del operador de punto (.). Por ejemplo, para leer la cédula y las horas laboradas, se codificaron las siguientes instrucciones:

```
    fgets (campoEmpleado.cedula, sizeof(campoEmpleado.cedula)
        ,stdin)
    //para leer la cédula.

    scanf ("%d", &campoEmpleado.horasLaboradas)
    //para leer las horas laboradas.
```

En esta nueva versión, se hizo así:

```
    fgets (e -> cedula, sizeof(e -> cedula),stdin)
    //para leer la cédula.

    scanf  ( "%d", &e-> horasLaboradas )
    //para leer el nombre.
```

El operador `->` se usa cuando se señala al campo de una estructura mediante un puntero.

La última función que se escribió, se denominó `continuar`. Es usada para preguntar si se desea continuar con la grabación de registros.

Esta función devuelve un valor de tipo `char` y no tiene parámetros. Dentro de ella se define la variable `siga`, que servirá para leer el valor que digiten a la pregunta: "`Adicionar otro empleado S/N?:`". La función retorna el valor que se digite.

El siguiente punto a analizar, es el orden en que se ejecutan estas funciones y las demás instrucciones del programa; para ello, hay que dirigirse a la función principal (`main`) donde se invoca la función para la apertura del archivo. El llamado a la función `abrirArchivo`, hace parte de la expresión relacional de una instrucción `if`:

```
27  if (abrirArchivo ( NOMBRE_ARCHIVO, &interno ) == ERROR)
28  {
29    perror ( "Error en la apertura del archivo \a" );
30    return ERROR;
31  }
```

Al hacer el llamado a la función `abrirArchivo`, se pasa como parámetros el nombre del archivo, que para este caso es `Nomina.dat`, declarado al inicio del programa con la constante: `NOMBRE_ARCHIVO`. Del mismo modo, `&interno` hace referencia a la posición de memoria que apunta al archivo interno.

La función `abrirArchivo` devuelve un valor, que es comparado en la condición que se planteó en el `if`, si ese valor es `ERROR` (recuerde que esta constante simbólica tiene el valor de 1), quiere decir que se produjo un error en la apertura del archivo y en consecuencia se termina la ejecución del programa (`return ERROR`).

Si la anterior condición es falsa, se inicia un ciclo `do-while`:

```
32  else
33  {
34    do
35    {
36      leerEmpleado( &campoEmpleado );
37
38      fwrite( &campoEmpleado, sizeof( empleado ),
39            1, interno );
40
41      printf( "\nLos datos fueron adicionados al archivo.
42            \n\n" );
43
44    } while ( continuar() == 'S' );
45
46    fclose(interno);
47
48    return OK;
49  }
```

La primera instrucción que se ejecuta dentro del `do-while`, es el llamado a la función: `leerEmpleado (&campoEmpleado);` el operador `&`, indica que se está haciendo un paso por referencia, es decir se está enviando la dirección de memoria de la variable del tipo del registro, llamada `campoEmpleado;` de esta manera, todos los datos que digiten en la función `leerEmpleado` quedan almacenados en esta dirección. Una vez se ejecute esta función, el control retorna a la función `main` y se ejecuta la siguiente instrucción:

```
38   fwrite( &campoEmpleado, sizeof( empleado ),
39         1, interno );
```

Con ella se indica que se debe grabar: `&campoEmpleado` hace referencia a la dirección de memoria de esta variable donde están los datos que se deben grabar, `sizeof (empleado)` determina el tamaño de registro, el número `1` indica que se almacenará un (1) grupo de datos, `interno` es el puntero del archivo donde se guardará el registro.

Luego de realizar la grabación, se visualiza un mensaje informando sobre esta operación:

```
41   printf( "\nLos datos fueron adicionados al archivo.
42         \n\n" );
```

El siguiente paso, en evaluar la condición del `do-while`:

```
44   while ( continuar() == 'S' );
```

Esta condición, se construyó usando la función `continuar`, la cual debe retornar el valor de la variable `siga`, que almacena la respuesta que proporcione el usuario a la pregunta "`Adicionar otro empleado S/N?:`". Esta respuesta es comparada con la letra "`S`", si la evaluación es verdadera, se repite el proceso: solicitar más datos, grabar la información, informar sobre esa grabación y volver a preguntar.

Cuando la respuesta sea diferente a una "`S`", se cerrará el archivo y terminará la ejecución del programa:

```
46   fclose (interno);
47
48   return OK;
```

Aclaración:

La referencia a los elementos o campos del registro o estructura, se hace de la siguiente manera:

1. Con el operador `.` (punto) cuando se usa la variable de estructura o registro:

```
variableEstructura.campoEstructura;
```

2. Con el operador `->` cuando se usa un puntero:

```
punteroEstructura ->campoEstructura;
```

.:Ejemplo 7.9. *Hasta el momento, solo se han guardado datos en los registros del archivo* `Nomina.dat`*. Otra de las operaciones que se pueden realizar con los archivos, es la consulta de sus registros. Hay varias maneras de hacerlo, una es usando la posición que ocupa el registro dentro del archivo, otra utilizando alguno de los campos como elemento identificador, por ejemplo, la cédula.*

En este primer ejemplo de consulta, se usará la posición del registro como criterio de búsqueda.

Análisis de problema

- **Resultados esperados:** ver en pantalla los datos del registro solicitado, de acuerdo a la posición que ocupe.

  Aunque el enunciado no lo menciona, es responsabilidad del diseñador del programa, que el software le proporcione la mayor información y ayuda al usuario; por eso, para este ejemplo se mostrará un mensaje que informe si el registro no se encuentra dentro del archivo.

- **Datos disponibles:** nombre del archivo `Nomina.dat` y número de la posición del registro.

- **Proceso:** se deben incluir las bibliotecas necesarias, de acuerdo a las funciones que se utilicen. De igual manera, se definirá la estructura

del archivo: es importante que la estructura conserve los mismos campos, el orden y el mismo tipo de dato para cada uno, en todos los programas donde se vaya a utilizar.

Se proponen las siguientes funciones para la solución del problema: una para buscar el registro, otra para imprimir los datos de los campos y una para preguntar si se desea continuar con nuevas búsquedas.

Desde la función `main`, se abrirá el archivo binario, con su correspondiente validación. El modo de apertura debe ser de lectura ("`rb`").

Luego se solicitará el dato conocido (posición del registro) y se invocará la función para buscar el registro, una vez encontrado se mostrarán los datos y se preguntará si se quiere continuar con una nueva búsqueda. En caso de que el registro no exista, también se procede a realizar el informe dentro de la función `main`.

- **Variables requeridas:** Adicionalmente, a las variables que ya se han analizado en los ejemplos anteriores (la estructura, el puntero del archivo y la variable de estructura), se propone declarar las que se listan enseguida:

  - `busqueda:` bandera para determinar los resultados de la búsqueda.
  - `numeroRegistro:` para almacenar la posición del registro a buscar.
  - `encontro:` bandera para retornar el valor de la operación de búsqueda.
  - `sueldo:` sueldo del empleado.
  - `siga:` almacena el resultado a la pregunta, de continuar con las búsquedas.

Con base al anterior análisis, una posible solución se aprecia en el Programa 7.9.

Programa 7.9: LeerBinario1

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

typedef struct
{
  char cedula[13];
  char nombre[30];
  int horasLaboradas;
  float valorHora, descuentos;
} empleado;

int  buscarRegistro ( empleado *e, FILE **interno,
            long numRegistro );
void imprimirEmpleado ( empleado e );
char continuar();

int main()
{
  FILE* interno;
  empleado campoEmpleado;

  int busqueda;
  long numeroRegistro;

  if ((interno = fopen ("Nomina.dat","rb")) == NULL)
  {
    perror ("Error abriendo el archivo \a");
    return 1;
  }
  else
  {
    do
    {

      printf( "Número del registro a buscar(iniciando
            en 0): " );
      scanf( "%ld", &numeroRegistro);

      busqueda = buscarRegistro( &campoEmpleado, &interno,
                  numeroRegistro );

      if (busqueda)
      {
        imprimirEmpleado (campoEmpleado);
      }
      else
      {
        printf("\nRegistro no encontrado \n\n \a");
```

```
49          }
50
51        } while(continuar() == 'S');
52
53        fclose(interno);
54      }
55      return 0;
56    }
57
58    int  buscarRegistro ( empleado *e, FILE **interno,
59                    long numRegistro )
60    {
61
62      int  encontro;
63
64      encontro = fseek( *interno, sizeof( empleado ) *
65                     numRegistro, 0 );
66
67      if (encontro == 0)
68      {
69        encontro = fread( e, sizeof(empleado), 1, *interno );
70      }
71
72      return encontro;
73    }
74
75    void imprimirEmpleado ( empleado e )
76    {
77
78      float sueldo;
79
80      printf("\n Datos del empleado que se está consultando
81          \n\n");
82      printf( "          Cédula: %s \n", e.cedula );
83      printf( "          Nombre: %s \n", e.nombre );
84      printf( "Horas trabajadas: %d \n", e.horasLaboradas);
85      printf( "  Valor por hora: %.2f \n", e.valorHora );
86      printf( "      Descuentos: %.2f \n\n", e.descuentos );
87
88      sueldo = (e.horasLaboradas * e.valorHora) -
89                      e.descuentos;
90
91      printf( "          Sueldo: %.2f\n\n", sueldo );
92    }
93
94    char continuar()
95    {
96
97      char siga;
```

```
98
99      printf ( "Consultar otro registro S/N?: " );
100     scanf ( " %c", &siga );
101     siga = toupper( siga );
102
103     getchar();
104     printf("\n\n");
105
106     return (siga);
107   }
```

Explicación del programa:

La codificación de este programa, es una combinación de los dos ejemplos anteriores. Se busca mostrar las diferentes formas de dar solución a un problema.

En el inicio se incluyen las bibliotecas, se define la estructura y se declaran los prototipos de las funciones de programador. En este caso, se trabajarán las siguientes:

```
int  buscarRegistro ( empleado *e, FILE **interno,
                long numRegistro )

//busca la posición donde se encuentra el registro
//y hace su lectura.
```

```
void imprimirEmpleado ( empleado e )

//imprime los datos del registro que ocupe la
//posición encontrada.
```

```
char continuar()

//determina si se continúa con nuevas búsquedas.
```

Para mejor comprensión del funcionamiento de este programa, se iniciará con la explicación de las funciones de programador. La primera es `buscarRegistro`:

```
58  int  buscarRegistro ( empleado *e, FILE **interno,
59                  long numRegistro )
60  {
61    int  encontro;
62
63    encontro = fseek( *interno, sizeof( empleado ) *
64                  numRegistro, 0 );
65
```

```
66      if (encontro == 0)
67      {
68        encontro = fread( e, sizeof(empleado), 1, *interno );
69      }
70
71      return encontro;
72    }
```

La función retorna un valor de tipo `int`, que indicará si encontró o no la posición del registro buscado. Recibe tres parámetros: `empleado *e`, `FILE **interno`, `long numRegistro`.

- `empleado *e`: este puntero recibirá la dirección de la variable donde se almacenarán los campos del registro.
- `FILE **interno`: es un puntero de un puntero que señala hacia el nombre interno del archivo.
- `long numRegistro`: recibe el número de la posición que se desea buscar.

Se declara la variable local `encontro`: servirá como una variable temporal o bandera, cuyo valor establecerá si se encontró o no la posición del registro buscado:

```
62    int  encontro;
```

La primera función de biblioteca que ejecuta, es `fseek`:

```
64    encontro = fseek ( *interno, sizeof( empleado ) *
65                       numRegistro, 0 );
```

Ella se encarga de ubicar el indicador de posición del archivo. En este caso se posiciona en el registro que indique la operación `sizeof(empleado) * numRegistro`, es decir, multiplica el tamaño en bytes del registro llamado `empleado` por la posición que el usuario digite, la cual es recibida como parámetro mediante `numRegistro;` recuerde que esta ubicación la busca desde el inicio del archivo, de acuerdo a lo indicado con el 0 que está como último parámetro (el 0 significa comienzo de archivo - `SEEK_SET`) .

`fseek` retornará un valor que almacenará en la variable `encontro`. El valor será distinto de cero (0) en caso de presentarse alguna inconsistencia.

Con el valor que tome la variable `encontro`, se evalúa la siguiente decisión:

```
67   if (encontro == 0)
68   {
69     encontro = fread( e, sizeof(empleado), 1, *interno );
70   }
```

En el caso de ser 0, indica que `fseek` pudo encontrar la posición buscada, entonces se procede a la lectura del registro de esa posición, para ello se usa la función `fread`.

`fread` lee el registro desde el archivo `(*interno). sizeof (empleado)`, determina que se debe leer la cantidad de bytes indicada por este operador, el `1` señala que es un solo bloque de información (1 registro); todos los datos leídos son almacenados en el puntero `e`, declarado como el primer parámetro de la función `buscarRegistro` y que también es usado en la función `fread`.

La función `fread`, devuelve el número de elementos leídos. Para este caso, debe devolver un valor de 1; el cual es el número que se escribió como tercer parámetro en la función para indicar que se debe leer 1 registro `(línea 69)`. Si el valor retornado es diferente a 1, denota entonces que se presentó alguna inconsistencia.

El valor que tome la variable `encontro`, es retornado a la función principal `(línea 39)`.

```
72   return encontro;
```

Continuando con las funciones de programador implementadas en este Programa, se encuentra `imprimirEmpleado`: imprime el registro del empleado que ocupe la posición encontrada. Está compuesta por un solo parámetro y no retorna valores:

```
75   void imprimirEmpleado ( empleado e )
76   {
77     float sueldo;
78
79     printf("\n Datos del empleado que se está consultando
80           \n\n");
81     printf( "          Cédula: %s \n", e.cedula );
82     printf( "          Nombre: %s \n", e.nombre );
83     printf( "Horas trabajadas: %d \n", e.horasLaboradas);
84     printf( "  Valor por hora: %.2f \n", e.valorHora );
85     printf( "      Descuentos: %.2f \n\n", e.descuentos );
86
87     sueldo = ( e.horasLaboradas * e.valorHora ) -
88                 e.descuentos;
```

```
89
90      printf( "           Sueldo: %.2f\n\n", sueldo );
91    }
```

La función tiene un parámetro del tipo del registro denominado `empleado`. Dentro de ella, se declaró una variable local para calcular el sueldo.

En el cuerpo de la función se encuentran varias funciones `printf`, que se encargan de mostrar en pantalla, cada uno de los campos del registro. Adicionalmente, se calcula y se imprime el sueldo.

La última función de programador diseñada, tiene el nombre de `continuar`. Igual que en el programa anterior, se usa para preguntar si se desea continuar con otro registro; la respuesta del usuario es retornada a la función `main`.

Considerando las anteriores explicaciones, ahora se analizará como es el engranaje de estas funciones para la solución del problema. Toda la ejecución se controla desde la función principal o `main`. Después de la declaración de las variables locales a `main`, se controla la apertura correcta del archivo, en caso de algún error, se termina la ejecución del programa.

```
25    if ((interno = fopen ("Nomina.dat","rb")) == NULL)
26    {
27      perror ("Error abriendo el archivo \a");
28      return 1;
29    }
```

Estas instrucciones son similares a las del Programa 7.7. La apertura en este nuevo ejemplo se hace en modo "`rb`", con el fin de abrir el archivo binario en modo de lectura. La salida del programa se hace retornando el valor de 1 (`return 1`), equivalente a la instrucción `return ERROR`, del Programa 7.8.

En el caso de no presentarse un error, se inicia un ciclo `do-while` donde se le solicita al usuario el número del registro que se va a consultar[5]. El valor digitado por el usuario, se almacena en la variable `numeroRegistro`. Luego, con la siguiente instrucción se invoca a la función que realizará la búsqueda del registro solicitado:

```
39    busqueda = buscarRegistro( &campoEmpleado, &interno,
40                 numeroRegistro );
```

[5]Se debe tener en cuenta que el primer registro se almacena en la posición 0.

De esta forma, la función `main`, le cede el control a la función `buscarRegistro`, enviándole tres parámetros:

- `&campoEmpleado`: envía la dirección en memoria de esta variable. Se pretende que esta variable almacene los datos que se lean en la función `buscarRegistro`.
- `&interno`: la dirección del puntero que señala al archivo.
- `numeroRegistro`: el valor de la posición del registro que se debe buscar dentro del archivo.

Una vez ejecutada la función `buscarRegistro`, se retornará un valor que es asignado a la variable `busqueda` (`línea 39`).

El siguiente paso, es evaluar la condición:

```
42   if (busqueda)
43   {
44     imprimirEmpleado (campoEmpleado);
45   }
46   else
47   {
48     printf("\nRegistro no encontrado \n\n \a");
49   }
```

Con ella se quiere verificar que no hubo ninguna inconsistencia en la lectura del registro. Consecuente a este resultado, se procede a invocar la función `imprimirEmpleado`, enviando como parámetro la variable de estructura `campoEmpleado`, la cual tiene almacenados los datos de cada campo del registro. Tenga presente, que esto se logró cuando se hizo el llamado a la función `buscarRegistro` y se envió como parámetro su dirección (`línea 39`).

Acto seguido de que los datos sean mostrados por la función `imprimirEmpleado`, el control regresa a la función `main`, para proceder con la evaluación de la condición del `do-while`, igual que se hizo en el programa del ejemplo anterior:

```
51   while (continuar() == 'S');
```

Cuando la respuesta sea diferente a "`S`", se cierra el archivo y se termina con la ejecución del programa.

Aclaración:

Todos los programas que vayan a procesar el mismo archivo, deben declarar la estructura o registro de manera idéntica. Se debe conservar el orden de los campos y los tipos de dato. No hay problema si se cambia el nombre de los campos.

.:Ejemplo 7.10. *Este programa permite consultar los datos de un empleado en particular, del archivo* `Nomina.dat`*, para ello se usará el número de cédula.*

Análisis de problema

El análisis de este ejemplo, está influenciado en gran medida por el que se hizo para el Ejemplo 7.9, teniendo en cuenta que solamente cambia el criterio de búsqueda. En el ejemplo anterior, la búsqueda se realizó mediante la posición del registro en el archivo, en este enunciado se solicita realizarla a través del número de cédula del empleado.

- **Resultados esperados:** ver en pantalla los datos del registro solicitado, de acuerdo al número de cédula. En caso de no encontrarlo, se informará la situación.
- **Datos disponibles:** nombre del archivo `Nomina.dat` y número de la cédula del empleado.
- **Proceso:** igual que en el Ejemplo 7.9 se incluyen las bibliotecas necesarias y se define la estructura del archivo.

  Se trabajarán las mismas funciones de programador para la solución del problema: una para buscar el registro, otra para imprimir los datos de los campos y una para preguntar si se desea continuar con nuevas búsquedas. Aunque en esta última se hará un pequeño ajuste, habrá notado que la función `continuar` se ha empleado en los dos últimos Programas (7.8 y 7.9) y que en lo único que cambian es en el mensaje:

  - `printf ( " Adicionar otro empleado S/N?:");` mensaje del Programa 7.8, acorde a la función que realiza, en él se están adicionando registros al archivo.

- `printf ( " Consultar otro registro S/N?:")`: mensaje del Programa 7.9, en él se están consultando registros de acuerdo a la posición que ocupan en el archivo.

  En este nuevo ejemplo, lo más indicado sería codificar el siguiente mensaje:

  ```
  printf ( "Consultar otro empleado S/N?: " );
  ```

Hecho este análisis, se puede concluir que esta es una función muy común a todos los programas y que su diferencia solo radica en el mensaje que se muestra, ya que el resto de instrucciones permanecen iguales. Es por esto, que esta es una función candidata a tener esa diferencia como un parámetro, lo cual se podrá apreciar más adelante en la implementación.

El resto del funcionamiento del programa, será igual al del ejemplo anterior, solo que en lugar de solicitar la posición del registro, se solicitará la cédula.

- **Variables requeridas:** Casi todas las variables del ejemplo anterior se conservan, excepto `numeroRegistro` que se cambia por `cedulaConsulta`, debido a que este es el criterio de búsqueda en este programa.

Una vez realizado este análisis, se plantea el Programa 7.10 como una posible solución:

Programa 7.10: LeerBinario2

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4
5  typedef struct
6  {
7    char cedula[13];
8    char nombre[30];
9    int horasLaboradas;
10   float valorHora, descuentos;
11 } empleado;
12
13 int  buscarRegistro ( empleado *e, FILE **interno,
14              char *cedula );
15 void imprimirEmpleado ( empleado e );
16 char continuar(char mensaje[80]);
17
```

```
18  int main()
19  {
20    FILE * interno;
21    empleado campoEmpleado;
22
23    char cedulaConsulta[13];
24    int busqueda;
25
26    if ((interno = fopen ("Nomina.dat","rb")) == NULL)
27    {
28      perror ("Error abriendo el archivo \a");
29      return 1;
30    }
31    else
32    {
33      do
34      {
35        printf("\nIngrese cédula del empleado a consultar:
36                              ");
37        fgets(cedulaConsulta, sizeof(cedulaConsulta),stdin);
38        strtok(cedulaConsulta,"\n");
39
40        busqueda = buscarRegistro( &campoEmpleado, &interno,
41                  cedulaConsulta );
42
43        if (busqueda)
44        {
45          imprimirEmpleado (campoEmpleado);
46        }
47        else
48        {
49          printf("\nEmpleado no encontrado \n\n \a");
50        }
51
52      }while(continuar("Consultar otro empleado S/N?: ") ==
53                'S');
54
55      fclose(interno);
56    }
57    return 0;
58  }
59
60  int buscarRegistro ( empleado *e, FILE **interno,
61              char *cedula )
62  {
63    long posicion = 0;
64    int  encontro = 0;
65
66    fseek(*interno, sizeof(empleado)* 0,0);
```

```
67
68      while(!feof(*interno) && !encontro)
69      {
70        fseek(*interno, sizeof(empleado)* posicion,0);
71
72        fread(e, sizeof(empleado), 1, *interno);
73
74        if (strcmp(e->cedula, cedula)==0)
75        {
76          encontro = 1;
77        }
78
79        posicion ++;
80      }
81
82      return encontro;
83    }
84
85    void imprimirEmpleado ( empleado e )
86    {
87      float sueldo;
88
89      printf("\n Datos del empleado que se está consultando
90              \n\n");
91      printf( "          Cédula: %s \n", e.cedula );
92      printf( "          Nombre: %s \n", e.nombre );
93      printf( " Horas Laboradas: %d \n", e.horasLaboradas );
94      printf( "  Valor por hora: %.2f \n", e.valorHora );
95      printf( "      Descuentos: %.2f \n\n", e.descuentos );
96
97      sueldo = ( e.horasLaboradas * e.valorHora ) -
98               e.descuentos;
99
100     printf( "          Sueldo: %.2f\n\n", sueldo );
101   }
102
103   char continuar (char mensaje[80])
104   {
105     char siga;
106
107     printf ( "%s", mensaje );
108     scanf ( " %c", &siga );
109     siga = toupper( siga );
110
111     getchar();
112     printf("\n\n");
113
114     return (siga);
115   }
```

Explicación del programa:

Para hacer más ágil esta sección, solamente se explicarán los elementos que sean diferentes a los del Programa 7.10, empezando por el prototipo de la función continuar:

```
103  char continuar (char mensaje[80]);
```

Se adicionó un parámetro llamado `mensaje`, es un arreglo de 80 caracteres (cadena). Su funcionamiento se detallará unos párrafos más adelante. Otro aspecto diferente entre los dos programas, es la función `buscarRegistro`, la cual tiene la siguiente codificación:

```
60  int buscarRegistro ( empleado *e, FILE **interno,
61                       char *cedula )
62  {
63    long posicion = 0;
64    int  encontro = 0;
65
66    fseek(*interno, sizeof(empleado)* 0,0);
67
68    while(!feof(*interno) && !encontro)
69    {
70      fseek(*interno, sizeof(empleado)* posicion,0);
71
72      fread(e, sizeof(empleado), 1, *interno);
73
74      if (strcmp(e->cedula, cedula)==0)
75      {
76        encontro = 1;
77      }
78
79      posicion ++;
80    }
81
82    return encontro;
83  }
```

La función tiene tres parámetros, los dos primeros permanecen iguales a los del Ejemplo 7.8, el tercer parámetro: `char *cedula`, se encarga de recibir la cédula del empleado que se va a consultar:

```
60  int buscarRegistro (empleado *e, FILE **interno,
61                                   char *cedula)
```

Posee dos variables locales, una de ellas es nueva: `posicion`. Con ella se irá indicando la posición del puntero dentro del archivo. Inicia en 0, debido a que el primer registro se ubica en este lugar, tiene incrementos de 1 para poder ir avanzando.

La instrucción: `fseek (*interno,sizeof(empleado)*0 0)`, posiciona el puntero en el registro 0 del archivo.

La ejecución del programa continúa con un ciclo `while`:

```
68   while (!feof (*interno) && !encontro)
```

Su condición establece que debe ejecutar su cuerpo de ciclo, mientras no encuentre el fin de archivo y la variable `encontro` tenga un valor igual a 0 (en este ejemplo, la expresión `!encontro` es equivalente a `encontro == 0`). Visto de otra forma: debe terminar de iterar cuando llegue al fin del archivo o cuando `encontro` tenga un valor de 1. El valor de 1 en la variable `encontro`, significa que se ha localizado el registro cuya cédula coincide con la que se está buscando.

Una vez se evalúe la condición y de un resultado verdadero, se ejecutan las instrucciones que están dentro del `while`, en primera instancia ejecuta la función `fseek`:

```
70   fseek (*interno, sizeof(empleado)* posicion,0);
```

Esta instrucción se encarga de ir posicionando el puntero del archivo, registro tras registro[6] hasta que el `while` no vuelva a iterar.

La siguiente línea presenta la función `fread`:

```
72   fread (e, sizeof(empleado), 1, *interno);
```

Se encarga de leer el registro donde está posicionado el puntero. Hay que tener presente que los datos se almacenan en la variable de tipo puntero `empleado`, llamada `e`. Ahora, se procede a realizar una comparación de cadenas:

```
74   if (strcmp(e->cedula, cedula) == 0)
75   {
76     encontro = 1;
77   }
```

Interpretándose así: si el valor del campo `cedula (e -> cedula)` es igual al dato que hay en `cedula` (valor que digitó el usuario como criterio de búsqueda), entonces la variable `encontro` toma el valor de 1.

En la línea 79, que está por fuera de la condición, la variable `posicion` se incrementa en 1: `posicion ++`.

[6] Se está realizando un recorrido secuencial.

La última instrucción de la función es un retorno. La función `buscarRegistro`, tiene dos criterios de terminación: primero cuando se alcance el fin del archivo, en cuyo caso la variable `encontro` permanecerá con el valor de 0, con el cual fue inicializada, significando con ello que el registro del empleado no fue encontrado y el segundo criterio se presenta cuando se encuentre el registro, con lo cual la variable `encontro` tomará el valor de 1.

De acuerdo a las circunstancias anteriores, se hace el retorno desde la función y termina su ejecución.

Siguiendo con el análisis de los aspectos diferenciadores entre este programa y el 7.9, está la función `continuar`:

```
103   char continuar (char mensaje[80])
104   {
105     char siga;
106
107     printf ( "%s", mensaje );
108     scanf ( " %c", &siga );
109     ...
110     ...
111   }
```

El cambio se encuentra solo en dos partes de la función:

```
103   char continuar (char mensaje[80])
```

Primero, se adicionó un parámetro denominado `mensaje` que recibirá una cadena de caracteres para ser mostrada en pantalla.

El segundo cambio está en la función `printf`, en lugar de mostrar un mensaje fijo, mostrará un mensaje variable y que llega a la función como parámetro.

```
107   printf ( "%s", mensaje );
```

Con estos ajustes el programador no tendrá que escribir el mensaje dentro de la función, sino que lo hará en el momento que la invoque, que es precisamente lo que se presenta en la línea 52 y que se detallará a continuación.

Para finalizar con el análisis del programa, la función `main`, solamente presenta un cambio en la condición del `do-while`:

```
52   while (continuar ("Consultar otro empleado S/N?: ") ==
53             'S');
```

La condición sigue siendo la misma; el ciclo se repetirá mientras se responda con una "S" a la pregunta: "Consultar otro empleado S/N?:". Lo que cambia aquí, es que el mensaje de la pregunta fue enviado como parámetro al invocar la función continuar.

Cuando la respuesta anterior, sea diferente a "S", terminará la ejecución del programa.

Al resto de instrucciones del programa, no se le hizo ninguna observación, ya que permanecen y funcionan exactamente igual a las del Programa 7.9.

.:Ejemplo 7.11. *En este ejemplo se ilustrará la forma de realizar una consulta general del archivo Nomina.dat, es decir, que se desplieguen en pantalla todos los registros del archivo. Se hará en forma de lista, un registro debajo del otro.*

Análisis de problema

- **Resultados esperados:** un listado con todos los registros que hay en el archivo Nomina.dat.
- **Datos disponibles:** un archivo binario identificado como Nomina.dat.
- **Proceso:** luego de realizar todas las operaciones comunes al manejo de archivos, se hará un recorrido secuencial, desde el inicio del archivo hasta encontrar la marca de fin de archivo. A medida que se avance dentro del archivo, se leerán los datos y se irán mostrando en pantalla.
- **Variables requeridas:**
  - interno: nombre interno del archivo.
  - sueldo: almacena el sueldo del empleado.

Programa 7.11: ListarBinario

```
1  #include <stdio.h>
2  #define ERROR  1
3  #define OK     0
4
5  #define NOMBRE_ARCHIVO  "Nomina.dat"
6
7  typedef struct
8  {
9    char  cedula[13];
```

```
10      char  nombre[30];
11      int   horasLaboradas;
12      float valorHora, descuentos;
13    } empleado;
14
15    int  abrirArchivo ( char *nombreArchivo, FILE **archivo );
16    void listarEmpleados (char *nombreArchivo, FILE *archivo);
17
18    int main ()
19    {
20      FILE * interno;
21
22      if ( abrirArchivo (NOMBRE_ARCHIVO, &interno) == ERROR )
23      {
24        perror( "Error en la apertura del archivo \a" );
25        return ERROR;
26      }
27      else
28      {
29        listarEmpleados ( NOMBRE_ARCHIVO, interno );
30
31        fclose(interno);
32
33        return OK;
34      }
35    }
36
37    void listarEmpleados (char *nombreArchivo, FILE *archivo)
38    {
39      empleado campoEmpleado;
40      float sueldo;
41
42      printf ( "\nLISTANDO EMPLEADOS\n\n" );
43      printf( "   Cédula \t\t" );
44      printf( "Nombre \t\t\t" );
45      printf( "Horas Laboradas \t" );
46      printf( "Valor H.L\t" );
47      printf( "Descuentos \t" );
48      printf( "Sueldo\n\n");
49
50      while ( fread( &campoEmpleado, sizeof( empleado ),
51                1, archivo ) != 0)
52      {
53        sueldo = ( campoEmpleado.horasLaboradas *
54              campoEmpleado.valorHora )
55              -campoEmpleado.descuentos;
56
57        printf( "%13s  %-30s %10d %25.2f %15.2f %13.2f \n",
58        campoEmpleado.cedula,campoEmpleado.nombre,
```

```
59        campoEmpleado.horasLaboradas, campoEmpleado.valorHora,
60        campoEmpleado.descuentos, sueldo );
61      }
62    }
63
64    int  abrirArchivo ( char *nombreArchivo, FILE **archivo )
65    {
66      if ( ( *archivo = fopen(nombreArchivo, "rb") ) == NULL )
67      {
68        return ERROR;
69      }
70      else
71      {
72        return OK;
73      }
74    }
```

Explicación del programa:

Esta solución toma elementos de programas anteriores, por lo tanto, se invita al lector a que repase las explicaciones dadas en su momento. Por ejemplo, se vuelven a trabajar las constantes simbólicas del Programa 7.8.

```
2   #define ERROR  1
3   #define OK     0
4
5   #define NOMBRE_ARCHIVO  "Nomina.dat"
```

En cuanto a la estructura del registro, se define la misma para todos los programas que manejan el archivo `Nomina.dat`.

Se incluye nuevamente la función `abrirArchivo`, que se codificó y explicó en el Programa 7.8:

```
15 int  abrirArchivo ( char *nombreArchivo, FILE **archivo );
```

La única diferencia es que en el Programa 7.8 se usó el modo de escritura "`wb`" y en este se hizo en modo de lectura "`rb`".

Se ingresa una nueva función de programador, la cual tendrá la misión de imprimir todos los registros del archivo en forma de listado, unos párrafos más adelante se describe en detalle:

```
16   void listarEmpleados (char *nombreArchivo, FILE *archivo);
```

Al inicio de la función `main`, se comprueba que no haya problema en la apertura del archivo asociado al puntero `interno`. Para la apertura se invoca la función `abrirArchivo` desde una decisión, igual que en el Programa 7.8.

```
22   if ( abrirArchivo (NOMBRE_ARCHIVO, &interno) == ERROR )
```

Si no hay inconvenientes con la apertura, se invoca la función `listarEmpleados`:

```
29   listarEmpleados ( NOMBRE_ARCHIVO, interno );
```

Luego de su ejecución, el control es devuelto a la función `main`, donde se cierra el archivo y se termina el programa:

```
31   fclose (interno);
32
33   return OK;
```

La función `listarEmpleado`, es la primera vez que se usa dentro de estos programas; declarada de tipo `void`, lo cual indica que no retorna ningún valor. Consta de dos parámetros de tipo puntero, que reciben el nombre externo e interno del archivo:

```
37   void listarEmpleados (char *nombreArchivo, FILE *archivo)
38   {
39     empleado campoEmpleado;
40     float sueldo;
41
42     printf ( "\nLISTANDO EMPLEADOS\n\n" );
43     printf( "   Cédula \t\t" );
44     printf( "Nombre \t\t\t" );
45     printf( "Horas Laboradas \t" );
46     printf( "Valor H.L\t" );
47     printf( "Descuentos \t" );
48     printf( "Sueldo\n\n");
49
50     while ( fread( &campoEmpleado, sizeof( empleado ),
51               1, archivo ) != 0)
52     {
53       sueldo = ( campoEmpleado.horasLaboradas *
54             campoEmpleado.valorHora )
55             -campoEmpleado.descuentos;
56
57       printf( "%13s  %-30s %10d %25.2f %15.2f %13.2f \n",
58       campoEmpleado.cedula,campoEmpleado.nombre,
59       campoEmpleado.horasLaboradas, campoEmpleado.valorHora,
60       campoEmpleado.descuentos, sueldo );
61     }
62   }
```

El cuerpo de la función inicia con la declararación de dos variables locales: la primera denominada `campoEmpleado` del tipo del

registro `empleado (empleado campoEmpleado)`, usada para hacer referencia a los campos del registro y la segunda de tipo `float`, llamada `sueldo`, en la cual se calculará el sueldo de cada empleado.

A continuación, se encuentra un conjunto de funciones `printf`, usadas para mostrar en pantalla los mensajes que allí se especifican. Por ejemplo: `printf( " Cédula \t \t" )`, muestra la palabra Cédula y con las secuencias de escape "`\t`" se insertan tabulaciones, logrando así que cada cadena impresa por la función `printf`, quede separada una de la otra; todas se imprimen sobre el mismo renglón.

Se procede a ejecutar el ciclo `while`, cuya condición planteada con una función `fread`, tiene dos propósitos:

```
50   while ( fread( &campoEmpleado, sizeof( empleado ),
51           1, archivo ) != 0)
```

1. Leer el registro: cada que se ejecuta, almacena en `campoEmpleado` un registro extraído del archivo.
2. Comprobar que no se haya alcanzado el fin de archivo: la función devuelve un valor de 0 cuando alcanza el fin de archivo.

Mientras no se haya alcanzado el fin de archivo, el cuerpo del ciclo, calcula el sueldo y lo imprime.

Actividad 7.5

Teniendo como punto de partida el programa anterior, construya uno nuevo que informe el nombre de los empleados que tienen el sueldo más alto y el más bajo. Así mismo, el total de los descuentos y el promedio del sueldo.

.:Ejemplo 7.12. *Dentro de la gestión de archivos, hay una actividad que consiste en la modificación de los datos que hay almacenados. Con este programa se mostrará cómo se puede realizar dicha modificación.*

El programa permitirá modificar cualquier campo de los registros de `Nomina.dat`, menos la cédula, que será el campo que servirá para obtener el registro que se quiere gestionar.

Análisis de problema

- **Resultados esperados:** una vez se ingresen los nuevos datos del registro a modificar, estos deben quedar grabados en el archivo `Nomina.dat`.
- **Datos disponibles:** se posee el archivo `Nomina.dat` y la cédula del empleado cuyo registro será modificado; adicionalmente, se cuenta con el nombre, las horas laboradas, el valor de la hora y los descuentos, todos ellos susceptibles a cualquier modificación.
- **Proceso:** una vez realizada la declaración de las respectivas variables y demás elementos requeridos en cualquier archivo, se procede a la apertura en modo de lectura/escritura. Se tendrá una combinación de las funciones de creación y de consulta de registros, analizadas en programas previos.

  Para el usuario, el proceso inicia en el momento que el programa le solicite el número de la cédula del empleado cuyo registro será modificado. Una vez se muestren los datos que están grabados en el archivo, el programa solicitará los nuevos para que sean digitados. Aunque el enunciado no lo expresa, se permitirá presionar la tecla Enter, cuando solicite el nuevo nombre y no se requiera su modificación.

  Después de las consideraciones anteriores, se pueden proponer tres funciones para la modificación del registro: la primera para realizar la búsqueda y lectura del registro en el archivo, la segunda será encargada de imprimir los datos en pantalla y la última efectuará la lectura de los nuevos datos. Adicionalmente, se incluirá la función `continuar`, con la que se preguntará si se desean hacer modificaciones de otros registros.
- **Variables requeridas:** todas las que ya se conocen para poder gestionar un archivo, más las propias del proceso de modificación, por ejemplo:

  - `cedulaConsulta:` para almacenar la cédula del empleado cuyo registro se va a modificar.
  - `lugar:` indica el lugar o posición que ocupa el registro dentro del archivo.

  A nivel de variables locales a las funciones de programador, son necesarias, las siguientes:

- `posicion:` va posicionando el puntero del archivo.
- `encontrado`: variable bandera para determinar si se encuentra o no le registro.
- `sueldo:` almacena el cálculo del sueldo.

Así mismo, son necesarias unas variables de memoria para almacenar los nuevos datos, antes de ser reemplazados dentro de los campos en el archivo:

- `memNombre`, `memHorasLaboradas`, `memValorHora` y `memDescuentos`.

Una vez terminado este análisis, se propone el Programa 7.12 como una posible solución.

Es importante resaltar, que este programa será usado como elemento didáctico para ilustrar el uso de la biblioteca `Uniquindio.h`, de la cual se habló en el Capítulo 5 Procedimientos y Funciones.

`Uniquindio.h`, tiene algunas funciones que no son del estándar de Lenguaje C, pero que si son usadas por algunos compiladores y que están incluidas en la biblioteca `conio.h`. En el caso del ambiente de desarrollo que se recomendó para trabajar estos programas, `conio.h` no está disponible.

Aclaración:

Como valor agregado a este texto, los autores han creado la biblioteca `Uniquindio.h`, adaptando algunas funciones presentes en `conio.h`.

En el desarrollo de este ejemplo, se darán las indicaciones de cuáles funciones pertenecen a esta biblioteca.

Para poder usar `Uniquindio.h`, consulte el Anexo A, en donde encontrará toda la información.

Programa 7.12: ModificarBinario

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4
5  #include "uniquindio.h"
6
7  typedef struct
8  {
9    char cedula[13];
10   char nombre[30];
11   int horasLaboradas;
12   float valorHora, descuentos;
13 } empleado;
14
15 long buscarRegistro (empleado *e, FILE *interno,
16             char *cedula);
17 void imprimirEmpleado (empleado e);
18 void leerEmpleado (empleado *e);
19 char continuar (char *mensaje, int fila);
20
21 int main()
22 {
23   FILE* interno;
24   empleado campoEmpleado;
25
26   char cedulaConsulta[13];
27   long lugar;
28
29   if ((interno = fopen ("Nomina.dat","rb+")) == NULL)
30   {
31     perror ("Error abriendo el archivo \a");
32     return 1;
33   }
34   else
35   {
36     do
37     {
38       clrscr();    // Se encuentra en Uniquindio.h
39
40       gotoxy(20,2); // Se encuentra en Uniquindio.h
41       printf("Ingrese cédula del empleado a modificar: ");
42
43       getString (cedulaConsulta, sizeof(cedulaConsulta) );
44       // getString se encuentra en Uniquindio.h
45
46       lugar = buscarRegistro(&campoEmpleado, interno,
47                   cedulaConsulta);
48
```

```
49          if (lugar >= 0)
50          {
51            imprimirEmpleado (campoEmpleado);
52
53            leerEmpleado( &campoEmpleado );
54
55            fseek (interno, sizeof (empleado) * lugar, 0);
56
57            fwrite(&campoEmpleado, sizeof( empleado ),
58                          1, interno);
59
60            gotoxy (25, 19);
61            printf( "Los datos fueron modificados." );
62          }
63          else
64          {
65            gotoxy (25,13);
66            printf("Empleado no encontrado \a");
67          }
68        }while (continuar("Modificar otro registro S/N?: ",
69                     22) == 'S');
70
71        fclose (interno);
72
73        return 0;
74      }
75    }
76
77    long buscarRegistro ( empleado *e, FILE *interno,
78                char *cedula )
79    {
80      long posicion = 0;
81      long encontrado = -1;
82
83      fseek(interno, sizeof(empleado)* 0,0);
84
85      while(!feof(interno) && encontrado < 0)
86      {
87        fseek(interno, sizeof(empleado)* posicion,0);
88
89        fread(e, sizeof(empleado), 1, interno);
90
91        if (strcmp(e->cedula, cedula)!=0)
92        {
93          posicion ++;
94        }
95        else
96        {
97          encontrado = posicion;
```

```
98        }
99      }
100
101     return encontrado;
102   }
103
104   void imprimirEmpleado ( empleado e )
105   {
106     float sueldo;
107
108     gotoxy(40,5);
109     printf("Datos del empleado");
110     gotoxy (20,7);
111     printf( "Cédula: %s", e.cedula );
112     gotoxy(20,9);
113     printf( "Nombre: %s", e.nombre );
114     gotoxy(11,11);
115     printf("Horas laboradas: %d", e.horasLaboradas );
116     gotoxy(12,13);
117     printf( "Valor por hora: %.2f", e.valorHora );
118     gotoxy(16,15);
119     printf( "Descuentos: %.2f", e.descuentos );
120
121     sueldo = (e.horasLaboradas * e.valorHora)- e.descuentos;
122
123     gotoxy(20,17);
124     printf( "Sueldo: %.2f", sueldo );
125   }
126
127   void leerEmpleado ( empleado *e )
128   {
129     char memNombre[30];
130     int memHorasLaboradas;
131     float memValorHora,memDescuentos,sueldo;
132
133     gotoxy (55, 7);
134     printf("Ingrese los siguientes datos...");
135
136     gotoxy(54,9);
137     printf ( "Nombre: " );
138     getString ( memNombre, sizeof(memNombre) );
139
140     gotoxy(45,11);
141     printf ( "Horas Laboradas: " );
142     scanf  ( "%d", &memHorasLaboradas );
143
144     gotoxy(50,13);
145     printf ( "Valor hora: " );
146     scanf  ( "%f", &memValorHora );
```

```
147
148     gotoxy(50,15);
149     printf ( "Descuentos: " );
150     scanf  ( "%f", &memDescuentos );
151
152     sueldo = memHorasLaboradas * memValorHora -
153              memDescuentos;
154
155     gotoxy (45,17);
156     printf ( "El nuevo sueldo es: %.2f", sueldo);
157
158     if (strcmp (memNombre, e->nombre) != 0 && strlen
159             (memNombre) != 0)
160     {
161       strcpy (e->nombre,memNombre);
162     }
163
164     e->horasLaboradas = memHorasLaboradas;
165     e->valorHora = memValorHora;
166     e->descuentos =  memDescuentos;
167
168   }
169
170   char continuar(char *mensaje,int fila)
171   {
172     char siga;
173     int centrar;
174
175     centrar = (80 - strlen(mensaje)) / 2;
176
177     gotoxy(centrar, fila);
178     printf ( "%s", mensaje );
179     do
180     {
181       siga = getch();   // Se encuentra en Uniquindio.h
182       siga = toupper( siga );
183     } while ( siga != 'S' && siga != 'N' );
184
185     return (siga);
186   }
```

Explicación del programa:

Aparte de las bibliotecas propias del lenguaje, se incluyó la creada por los autores del libro:

```
5   #include "uniquindio.h"
```

Se observa que, al tratarse de una biblioteca de programador, no se

usan los corchetes de ángulo < > para incluirla, en su lugar, el nombre se encierra entre comillas dobles.

A continuación, se definieron los prototipos de las funciones:

```
15  long buscarRegistro (empleado *e, FILE *interno,
16                       char *cedula);
17  void imprimirEmpleado (empleado e);
18  void leerEmpleado (empleado *e);
19  char continuar (char *mensaje, int fila);
```

Todas ya han sido trabajadas en programas anteriores. La función `continuar`, tiene un cambio con relación a la usada en el Programa 7.10, allá el parámetro `mensaje`, fue declarado como una cadena de 80 caracteres (`char mensaje [80]`), ahora se trabajará como un puntero de tipo `char (char *mensaje)`.

Luego de declarar las variables locales a la función `main`, se hace la validación para la apertura del archivo. En el caso de no presentarse ningún inconveniente, se inicia un proceso repetitivo a través de un ciclo `do-while`. La primera instrucción que se ejecuta en el cuerpo del ciclo es una función contenida dentro de `Uniquindio.h`:

`clrscr:` (Clear Screen – Limpiar pantalla) con ella se borra el contenido de la pantalla.

Enseguida se encuentra otra función contenida en `Uniquindio.h`:

`gotoxy`: (Vaya a X, Y) dirige el cursor a una coordenada de los ejes `X` y `Y` de la pantalla. Esta función tiene la siguiente forma general:

`gotoxy(X,Y)`, donde `X` es un valor entero que indica el número de la columna y `Y` indica el número de la fila en pantalla.

En esta primera instrucción `gotoxy` que se encuentra en el programa: `gotoxy(20,2)`, el cursor se ubica en la columna 20, fila 2. Luego, con la función `printf`, se imprime el mensaje a partir de esa coordenada:

```
41  printf("Ingrese cédula del empleado a modificar: ");
```

A esta petición, el usuario responderá digitando un número de una cédula. La lectura del dato se hace mediante otra nueva función, también perteneciente a la biblioteca `Uniquindio.h`:

```
43  getString (cedulaConsulta, sizeof(cedulaConsulta) );
```

Con esta función se pueden leer cadenas. A pesar que el Lenguaje tiene sus propias funciones para realizar estas lecturas, los autores del libro,

decidieron crear una propia que controle algunos aspectos en el manejo de ellas, como por ejemplo: la lectura de únicamente la cantidad de caracteres de acuerdo al tamamaño en la declaración de la variable y además, que no se almacene el salto de línea como parte de la cadena leída.

Una vez obtenido el número de cédula a buscar, se ejecuta la siguiente expresión de asignación:

```
46   lugar = buscarRegistro(&campoEmpleado, interno,
47                cedulaConsulta);
```

La variable `lugar`, almacenará el valor que retorne la función `buscarRegistro`, que es invocada con tres parámetros.

La función `buscarRegistro`, retornará la posición donde encuentre el registro en el archivo, o un valor de `-1` en el caso de no hallarlo. Este valor fue definido por el programador, atendiendo a que la ubicación de los registros inicia en la posición 0 y va en forma ascendente.

Con el valor que se almacene en la variable `lugar`, se toma la siguiente decisión y que, en el caso de resultar verdadera, indica que se encontró el registro:

```
49   if (lugar >= 0)
50   {
51     imprimirEmpleado (campoEmpleado);
52
53     leerEmpleado( &campoEmpleado );
54
55     fseek (interno, sizeof (empleado) * lugar, 0);
56
57     fwrite(&campoEmpleado, sizeof( empleado ), 1, interno);
58
59     gotoxy (25, 19);
60     printf( "Los datos fueron modificados." );
61   }
```

La línea 51, invoca la función `imprimirEmpleado`, donde se muestran los datos del registro.

Se llama a la función `leerEmpleado` en la línea 53, allí se leen los nuevos datos del registro.

La instrucción `fseek (interno, sizeof (empleado) * lugar, 0)`, de la línea 55, ubica el puntero del archivo en la posición que se haya encontrado el registro, la cual está determinada por el valor que tenga la variable `lugar`.

Seguidamente, se procede a la grabación de los nuevos datos del registro:

```
57   fwrite(&campoEmpleado, sizeof( empleado ), 1, interno).
```

Con las instrucciones:

```
59   gotoxy (25, 19);
60   printf( "Los datos fueron modificados." );
```

Se imprime el mensaje "Los datos fueron modificados", a partir de la columna 25, fila 19 de la pantalla.

Cuando la condición del `if` resulte falsa, se informará que el empleado no fue encontrado.

Al terminar de ejecutarse la instrucción `if`, se evalúa la condición del `do-while`:

```
68   while (continuar ("Modificar otro registro S/N?: ",
69              22) == 'S');
```

Esta condición es similar a la planteada en el Programa 7.10, solo que acá se le adicionó un parámetro más, con el cual se indica la fila en la que debe imprimirse el mensaje, para este caso, en la línea 22 de la pantalla. Cuando el ciclo deje de iterar, se cerrará el archivo y terminará el programa.

```
71   fclose (interno);
72
73   return 0;
```

Con esto termina el análisis de las instrucciones direccionadas por la función `main`, ahora, se deben estudiar las funciones creadas por el programador. La primera en entrar en escena, es la responsable de buscar el registro a modificar:

```
77   long buscarRegistro ( empleado *e, FILE *interno,
78                  char *cedula )
```

Esta función ya fue usada con anterioridad, los detalles de su funcionamiento, están descritos en la explicación del Programa 7.10.

Otra función presente es `imprimirEmpleado ( empleado e )`, que también fue usada en el Programa 7.10. Con relación a la presentada en este programa, se puede apreciar que en esta versión se está usando la función `gotoxy`, con el fin de darle una mejor presentación a la interfaz, ya que se logra ubicar de forma más ordenada los datos en las coordenadas que se especifiquen de la pantalla. El resto de instrucciones, son exactamente las mismas del Programa 7.10.

Para la lectura de los nuevos datos, se implementó la función `leerEmpleado`:

```
127   void leerEmpleado ( empleado *e )
```

Dentro de esta función se declaran variables locales, en las cuales se almacenarán los valores de los datos que el usuario digite y el sueldo que se calcule:

```
129   char memNombre[30];
130   int memHorasLaboradas;
131   float memValorHora,memDescuentos,sueldo;
```

Para ubicar los mensajes en la pantalla, se usaron varias funciones `gotoxy`. Adicionalmente, la lectura del nombre, que es una cadena de caracteres, se hizo con la función `getString`, una función que se encuentra dentro de `Uniquindio.h`:

```
136   gotoxy(54,9);
137   printf ( "Nombre: " );
138   getString ( memNombre, sizeof(memNombre) );
```

Los demás datos fueron capturados con la función `scanf` y almacenados en las variables locales, previamente declaradas.

Se planteó una decisión para determinar si el nombre que se digitó y se almacenó en la variable `memNombre` es diferente al almacenado en el archivo `(e->nombre)` y adicionalmente, si la longitud de lo leído tiene por lo menos un carácter. En el caso que esta condición sea verdadera, se cambia el valor del nombre que hay en el archivo, por el nuevo nombre:

```
158   if (strcmp (memNombre, e->nombre) != 0 && strlen
159                   (memNombre) != 0)
160   {
161     strcpy (e->nombre,memNombre);
162   }
```

Las últimas líneas de la función, cambian las horas laboradas, el valor de la hora y los descuentos que hay en el archivo, por los nuevos digitados:

```
164   e->horasLaboradas = memHorasLaboradas;
165   e->valorHora = memValorHora;
166   e->descuentos =  memDescuentos;
```

Finalmente, se encuentra la función de programador `continuar`, la cual presenta cambios significativos con relación a las que ya se han trabajado en ejemplos anteriores. En primer lugar, se declararon dos

parámetros: el primero `char *mensaje`, que recibirá el mensaje que se quiere mostrar y `int fila`, para indicar la fila de la pantalla donde se desplegará el mensaje:

```
170  char continuar (char *mensaje, int fila)
171  {
172    char siga;
173    int centrar;
174
175    centrar = (80 - strlen(mensaje)) / 2;
176
177    gotoxy(centrar, fila);
178    printf ( "%s", mensaje );
179    do
180    {
181      siga = getch();   // Se encuentra en Uniquindio.h
182      siga = toupper( siga );
183    } while ( siga != 'S' && siga != 'N' );
184
185    return (siga);
186  }
```

La función `continuar`, cuenta con dos variables locales:

- `siga:` para almacenar la respuesta que digite el usuario.
- `centrar:` se usará para calcular la columna desde donde empezará a imprimirse el mensaje.

Para calcular esa columna, se usa la siguiente expresión:

```
175  centrar = (80 - strlen(mensaje)) / 2;
```

80, es el número total de columnas que se toma como referencia en una consola (pantalla), a este valor se le resta la longitud en caracteres que tenga el mensaje a desplegar, ese total se divide entre 2, para que a ambos lados del mensaje quede la misma cantidad de espacios en blanco y este se vea centrado.

Con estas dos instrucciones se muestra el mensaje, en las coordenadas que se especifiquen:

```
177  gotoxy (centrar, fila);
178  printf ( "%s", mensaje );
```

Tenga presente que los valores de `fila` y `mensaje` son enviados a esta función como parámetros.

Continuando con las diferencias de esta versión de la función `continuar`, con las usadas en otros programas, se hace notorio el cambio en la forma de leer la respuesta:

```
179   do
180   {
181     siga = getch();    // Se encuentra en Uniquindio.h
182     siga = toupper ( siga );
183   } while ( siga != 'S' && siga != 'N' );
```

La lectura se encuentra validada, de tal forma que solamente reciba la letra 'N' o la letra 'S' como respuesta, se toman valores en mayúscula ya que la función `toupper` se encarga de convertir cualquier letra en su equivalente en mayúscula. Para la validación se contó con la ayuda de un ciclo `do-while`, donde la condición establece que, si se digita un valor diferente a las letras mencionadas, el ciclo vuelve a repetirse y por consiguiente se vuelve a leer el dato. Se observa, que para la lectura de la respuesta se usó `getch`, que también hace parte de la biblioteca `Uniquindio.h`.

La función `getch` permite la lectura de un carácter, no tiene eco ni retorno de carro, es decir, no se ve en pantalla el carácter digitado y no es necesario presionar Enter para que ejecute la lectura.

Por último, la función retorna el valor digitado:

```
185   return (siga);
```

Con este programa, se terminan los ejemplos de la gestión básica que se le puede hacer a los registros de un archivo.

Actividad 7.6

En el ejemplo que se acabó de analizar, se observó que en el nombre del empleado, el Programa recibe la tecla Enter como dato de entrada. Cuando esto sucede se conserva el nombre anterior y dicho campo no sufre modificación alguna.

De acuerdo a este contexto y retomando el ejemplo en mención, construya un nuevo programa que permita recibir la tecla Enter como dato de entrada, en cualquiera de los otros campos para indicar que se desea conservar el valor que hay en el archivo; se exeptúa la cédula por ser el campo elegido para búsqueda del registro a modificar.

Adicionalmente a la gestión de registros, existen algunas funciones útiles en la gestión de archivos. A continuación, se estudiarán dos de ellas con las cuáles podrá borrar un archivo o cambiarle su nombre. Ambas funciones, pertenecen a la biblioteca `stdio.h`.

Función `remove`: permite eliminar un archivo del dispositivo de almacenamiento. Su forma general es:

```
remove (nombreArchivo);
```

Donde `nombreArchivo`, es una cadena de caracteres que indica el nombre del archivo a borrar, se puede incluir una ruta de carpetas o directorios.

La función retorna el valor de 0 si la operación de borrado fue exitosa, en caso contrario, devuelve cualquier valor diferente a 0.

Función `rename`: con esta función se puede cambiar el nombre a un archivo que repose en un dispositivo de almacenamiento. A continuación, se presenta su forma general:

```
rename (nombreArchivo, nuevoNombre);
```

Donde `nombreArchivo`, es el nombre del archivo al que se le va a cambiar el nombre y `nuevoNombre` es el nombre con que será reconocido el archivo después de aplicar el cambio. Ambos parámetros son cadenas de caracteres que pueden incluir la ruta de carpetas o directorios.

La función devuelve 0 si la operación se pudo efectuar, en caso contrario devuelve un valor diferente. Si el `nuevoNombre` ya existe en el lugar donde se guardará el archivo renombrado, no se permitirá el cambio y el archivo conservará el nombre original (`nombreArchivo`).

.:Ejemplo 7.13. *Con el siguiente programa, el usuario podrá borrar el archivo que especifique.*

Análisis de problema

- **Resultados esperados:** luego de que el usuario digite el nombre del archivo a borrar, este debe desaparecer del medio de almacenamiento. En caso de no existir, se mostrará un mensaje informando la situación.
- **Datos disponibles:** el nombre del archivo a borrar.
- **Proceso:** el proceso a realizar se mostrará de forma gráfica con el siguiente diagrama:

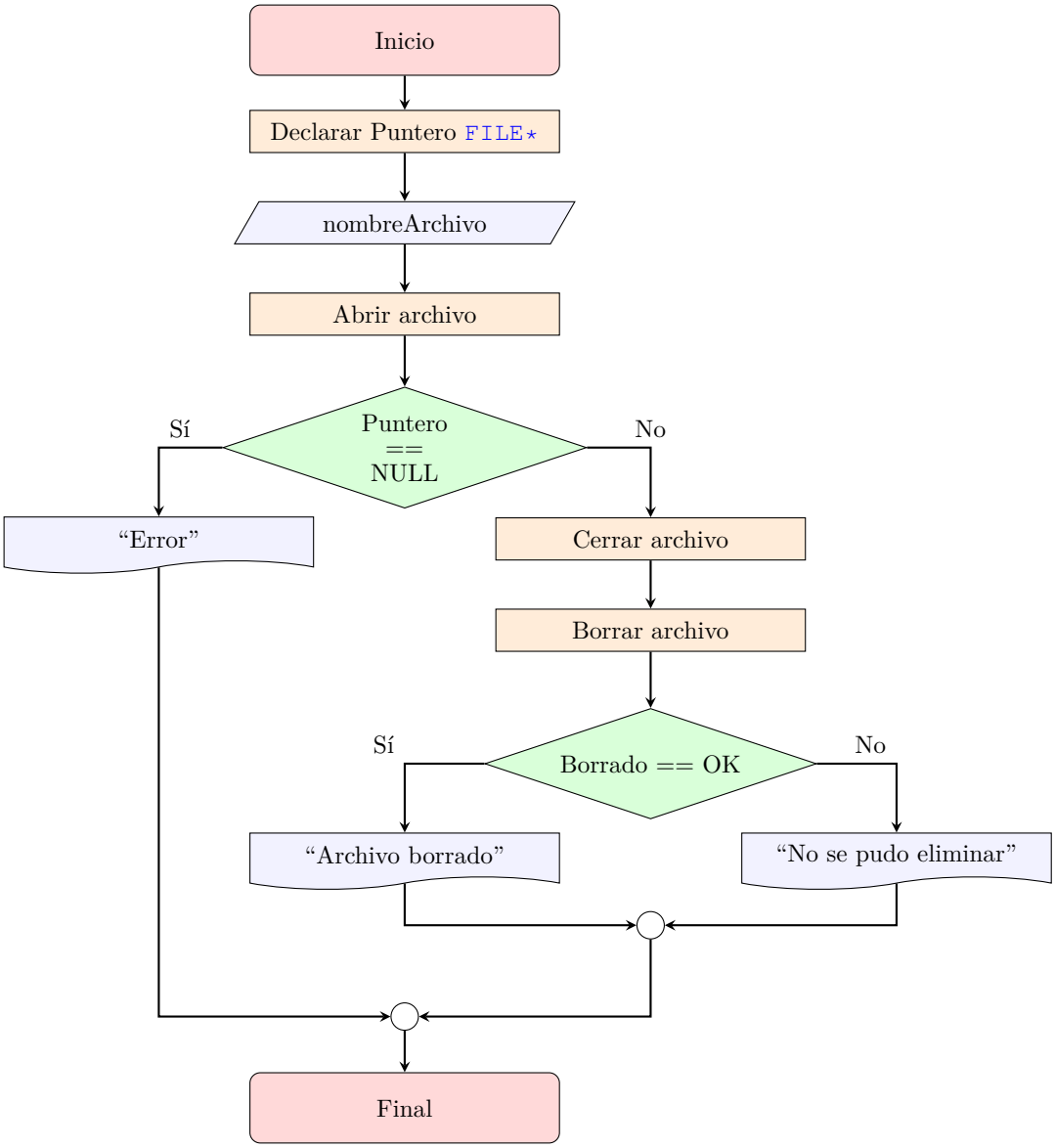


Figura 7.4: Proceso para eliminar un archivo.

- **Variables requeridas:**
  - `interno`: nombre interno del archivo. Igual que en todos los ejemplos anteriores.
  - `nombreArchivo`: nombre externo del archivo, es decir, con el que está grabado en el medio de almacenamiento.

De acuerdo al análisis realizado, una posible solución se presenta en el Programa 7.13.

Programa 7.13: BorrarArchivo

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6    FILE* interno;
7    char nombreArchivo[40];
8
9    printf("Digite el nombre del archivo a borrar.
10             Incluya extensión: ");
11   fgets (nombreArchivo, sizeof(nombreArchivo),stdin);
12   strtok(nombreArchivo,"\n");
13
14   interno = fopen(nombreArchivo,"r");
15
16   if (interno == NULL)
17   {
18     perror("\n\nEl archivo no existe!. \a");
19     return 1;
20   }
21   else
22   {
23     fclose(interno);
24
25     if ( remove(nombreArchivo) == 0 )
26     {
27       printf( "\n\n%s: fue eliminado.\n", nombreArchivo );
28     }
29     else
30     {
31       printf( "\n\n%s: no se puede eliminar.\n",
32             nombreArchivo );
33     }
34   }
35   return 0;
36 }
```

Explicación del programa:

Luego de solicitar el nombre del archivo a borrar, se realiza la apertura en modo de lectura. Si se presenta algún error en esta operación, se informa la situación y se termina la ejecución del programa. En caso contrario, se cierra el archivo (`fclose(interno)`), ya que algunos compiladores no soportan una operación de borrado con un archivo abierto.

Posteriormente, se usa la función `remove`, con el propósito de eliminar el archivo. Note que la función se emplea como parte de una expresión relacional de un `if`, donde se comprueba que no haya inconveniente con la operación a realizar. La función `remove`, devuelve un 0 cuando tiene éxito en su ejecución. Si la expresión es verdadera se imprime un mensaje informando que el archivo fue eliminado. Si, por el contrario, no se puede llevar a cabo el borrado, se informará dicha inconsistencia.

Finalmente, el programa termina su ejecución al encontrar la instrucción `return 0`.

.:Ejemplo 7.14. *Con este ejemplo, se ilustrará el uso de la función `rename`, la cual permite renombrar un archivo dentro de un medio de almacenamiento.*

Análisis de problema

- **Resultados esperados:** luego de que el usuario digite el nombre del archivo a renombrar y el nuevo nombre, se debe reflejar el cambio dentro del medio de almacenamiento. Si el cambio no se puede realizar, el programa deberá informar esta situación.
- **Datos disponibles:** el nombre del archivo a renombrar y su nuevo nombre.
- **Proceso:** el proceso a realizar se mostrará de forma gráfica con el siguiente diagrama:
- **Variables requeridas:** para este programa es necesario tener las variables que se listan a continuación:
  - `interno`: nombre interno del archivo. Igual que en todos los ejemplos anteriores.
  - `nombreArchivo`: nombre externo del archivo a renombrar, es decir, con el que está grabado en el medio de almacenamiento.
  - `nuevoNombre`: nombre externo con el que quedará el archivo a renombrar.

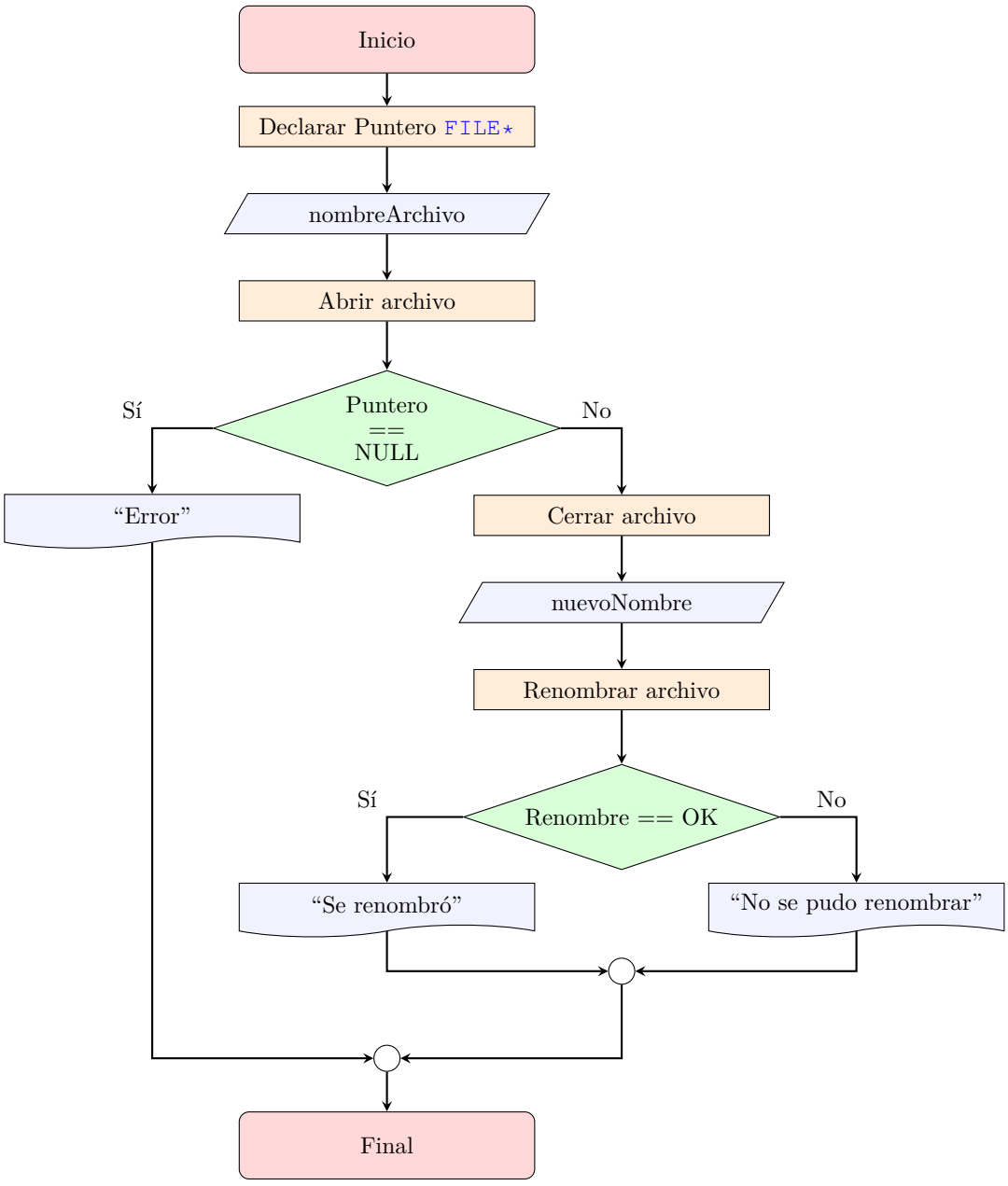


Figura 7.5: Proceso para renombrar un archivo.

Una posible solución se propone en el Programa 7.14.

Programa 7.14: RenombrarArchivo

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6    FILE* interno;
7    char nombreArchivo[40], nuevoNombre[40];
8
9    printf("Digite el nombre del archivo a renombrar.
10         Incluya extensión: ");
11   fgets (nombreArchivo, sizeof(nombreArchivo),stdin);
12   strtok(nombreArchivo,"\n");
13
14   interno = fopen(nombreArchivo,"r");
15
16   if (interno == NULL)
17   {
18     perror("\n\nEl archivo no existe!. \a");
19     return 1;
20   }
21   else
22   {
23     fclose(interno);
24
25     printf("\n\nDigite el nuevo nombre del archivo.
26           Incluya extensión: ");
27     fgets (nuevoNombre, sizeof(nuevoNombre),stdin);
28     strtok(nuevoNombre,"\n");
29
30     if( rename(nombreArchivo, nuevoNombre) == 0 )
31     {
32       printf( "\n\n%s: ahora se llama: %s\n",
33               nombreArchivo, nuevoNombre );
34     }
35     else
36     {
37       printf( "\n\n%s: no se le pudo cambiar el nombre\n",
38                 nombreArchivo );
39     }
40   }
41   return 0;
42 }
```

Explicación del programa:

Luego de que se le proporcione al programa, el nombre del archivo a renombrar, se hace la apertura y su correspondiente validación. Si se presenta algún error, se informa y se da por terminada la ejecución del programa.

Cuando la apertura se hace de forma correcta, se procede a cerrar el archivo, puesto que algunos compiladores no pueden ejecutar cambio de nombre en archivos abiertos. Seguidamente, se solicita el nuevo nombre del archivo; una vez se tenga este dato se procede a realizar el cambio, validando a la vez, que no se presente alguna inconsistencia en la operación, ello se logra con la siguiente decisión:

```
30   if ( rename (nombreArchivo, nuevoNombre) == 0 )
```

La función `rename` retorna 0 si la operación se puede ejecutar. Si la expresión del `if`, es verdadera, se informa del nuevo nombre del archivo. Si el resultado que arroja la función es un valor diferente a 0, se despliega un mensaje comunicando que no se pudo realizar el cambio del nombre.

Una vez terminada la ejecución de la estructura `if`, se da por terminado el programa (`return 0`).

Actividad 7.7

En esta última actividad de este capítulo, usted deberá analizar que tipo de archivo debe de usar y que funciones va a emplear para llevar a feliz término los siguientes ejercicios.

1. En el Ejemplo 4.20, se generaron e imprimieron las letras del abecedario de una determinada manera. Ahora usted deberá construir un programa que genere el mismo abecedario y que lo almacene en un archivo.

2. Construya un programa que lea desde el archivo, creado en el punto anterior, ese abecedario y lo muestre en pantalla, tal como se pidió en el Ejemplo 4.20.

3. Con base al Ejemplo 4.23, de la Universidad, construya los Programas necesarios que permitan almacenar la información que allí se procesa y generar los reportes que solicita el enunciado.

4. Genere un solo programa que gestione los datos almacenados en el archivo creado en el punto anterior. Es decir, que se puedan hacer consultas de tipo individual (usando el código del estudiante) o modificar cualquiera de los datos.

5. A un amigo que vive en el norte de la ciudad, su terapeuta le recomendó caminar mínimo 3 días a la semana desde su apartamento hasta el centro, lo cual él hace sin falta alguna; el recorrido tiene aproximadamente 22 cuadras y debe hacerlo con ropa cómoda. Este ejercicio lo debe realizar durante 4 meses.

Al momento de volver a consulta, el amigo debe informarle a su terapeuta lo siguiente:

Promedio de tiempo por semana, por mes y por los 4 meses. Adicionalmente, cuál fue el menor y el mayor tiempo empleado en el recorrido.

Para esta tarea, el amigo lleva un registro del tiempo que invierte en cada caminata.

Este amigo, requiere de su ayuda para poder llevar todos estos registros mediante archivos en su computador, además, quiere que cuando vaya a la consulta mostrar en la pantalla toda la información que el terapeuta le exige.

NOTA: En la implementación de estos programas, puede hacer uso de todas las estructuras estudiadas a lo largo de los capítulos del libro. [Aldea et al., 2017]

ANEXO

INSTALACIÓN DEL AMBIENTE

El ambiente de trabajo para el lenguaje de programación C, así como para muchos otros lenguajes, consta de dos elementos:

- El compilador del lenguaje.
- El Entorno Integrado de Desarrollo (IDE[7]).

El primero permite crear una aplicación a partir de un código fuente del programa, en este caso escrito en Lenguaje C; el segundo provee un entorno de trabajo para la escritura del código fuente, similar a un editor de textos, pero con herramientas relacionadas con el desarrollo de aplicaciones.

A. Compilador de C de GCC/GNU

Con relación al compilador del Lenguaje C, se usará el que provee la colección de compiladores del proyecto GNU llamado GCC[8]. Esta colección es ampliamente utilizada en una gran cantidad de sistemas y arquitecturas; incluso, es un estándar de desarrollo en sistemas operativos Unix.

GCC es distribuido por la Fundación de Software libre llamada FSF[9]. Inicialmente GCC era un compilador exclusivo de Lenguaje C (*GNU C Compiler*), pero con el paso del tiempo, se extendió a otros lenguajes, de ahí el cambio en el significado de su nombre.

El primer paso antes de instalar el compilador del Lenguaje C, es definir el sistema y arquitectura en el que se desea instalar, para así descargar de Internet la versión apropiada. A continuación, se presentan unas generalidades para su instalación en los sistemas operativos: Windows, MacOs y Linux en arquitectura Intel de 64 bits (x64).

[7] *Integrated Development Environment*
[8] *GNU Compiler Collection*
[9] Free Software Foundation

A.1. Instalación en Windows

Como la colección de GCC no es nativa de Windows, es necesario usar un proyecto, realizado por terceros, para portar a este sistema GCC. Dos de estos proyectos son: MinGW (*Minimalist GNU for Windows*)[10]; y Cywin[11].

Para facilitar la instalación, se optó por usar un único instalador provisto por TDM-GCC MinGW Compiler[12], la cual se puede obtener directamente de la siguiente dirección de Internet.

```
https://sourceforge.net/projects/tdm-gcc/files/TDM-GCC%20Installer/tdm64-gcc-5.1.0-2.exe/download
```

Esta versión seleccionada, a pesar de no proveer la última versión del GCC, es más que suficiente para el desarrollo de los primeros pasos en Lenguaje C, e incluso para el desarrollo de proyectos de mayor escala.

El instalador `tdm64-gcc-5.1.0-2.exe` permite instalar GCC en Windows de arquitectura de 64 bits (x64) como en arquitectura de 32 bits (x32) (Ver Figura 7.6).

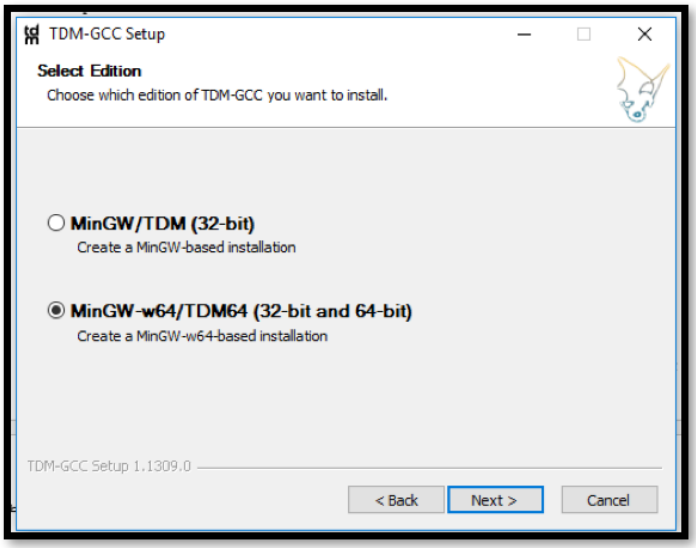


Figura 7.6: TDM-GCC MinGW Compiler para Windows

Una vez hecha la instalación por medio del asistente, se debe proceder a instalar el respectivo Entorno Integrado de Desarrollo y realizar una prueba de compilación que deje saber que todo está correctamente instalado.

[10] `http://www.mingw.org`
[11] `http://www.cygwin.com`
[12] `http://tdm-gcc.tdragon.net/`

A.2. Instalación en MacOs

En MacOs, basta con abrir una terminal y ejecutar el comando `gcc`. Aquí pueden suceder dos cosas: Que el sistema no cuente con el GCC instalado, o que ya esté disponible para su uso desde el Entorno de Desarrollo Integrado.

GCC aún sin instalar

Si GCC no se encuentra instalado, al intentar ejecutar el comando se obtiene algo como:

```
$ gcc
xcode-select: note: no developer tools were found at '/
   Application/Xcode.app', requesting install. Choose an
   option in the dialog to download the command line
   developper tools.
```

Si este fuera el caso, MacOs solicitará automáticamente instalar GCC (Ver Figura 7.7).

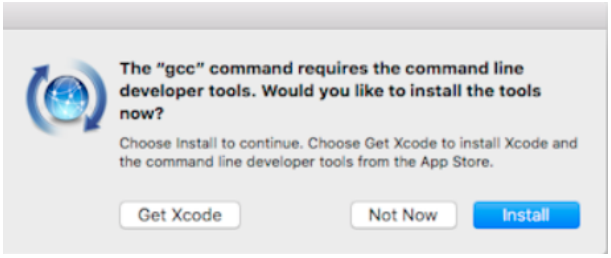


Figura 7.7: Instalación gcc en MacOS

Esta instalación contiene todo lo necesario para GCC, sin necesidad de instalar el ambiente completo de desarrollo para MacOS llamado XCode. XCode es el ambiente obligatorio para todo programador para aplicaciones nativas para el sistema MacOS y iOS.

Si por alguna razón, MacOS no solicitara la instalación automática, se puede usar el siguiente comando para realizar la instalación manualmente.

```
$ xcode-select --install
```

Una vez realizada la instalación, se puede proceder a verificar su correcto funcionamiento o a verificar la versión que ha sido instalada.

GCC diponible

Si el ejecutar el comando GCC en la terminal de MacOS se obtiene algo como:

```
$ gcc
clang: error: no input files
```

Se puede asegurar que él se encuentra instalado en el sistema, si este fuera el caso, se puede proceder a consultar la versión instalada mediante el siguiente comando:

```
$ gcc --version

Configured with: --prefix=/Library/Developer/
   CommandLineTools/usr --with-gxx-include-dir=/usr/include
   /c++/4.2.1
Apple LLVM version 10.0.0 (clang-1000.10.44.4)
Target: x86_64-apple-darwin17.7.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

A.3. Instalación en Linux

Si se desea instalar el GCC en un sistema operativo Linux, basta con ejecutar el comando GCC en una terminal, tal y como se ha explicado en MacOS. De nuevo, GCC podría no estar instalado, en cuyo caso se obtiene algo como:

```
$ gcc
-bash: gcc: command not found
```

Si GCC ya está disponible, se obtendrá una salida como:

```
$ gcc
gcc: fatal error: no input files
compilation terminated.
```

GCC aún sin instalar

Si GCC no está instalado, se debe proceder a realizar el respectivo proceso de instalación, según el gestor de paquetes que la distribución de Linux que se encuentre usando, por ejemplo:

- Basados en Linux Ubuntu

```
$ sudo apt-get install gcc
```

- Basados en Linux CentOS

```
$ sudo yum -y install gcc
```

- Basados en Linux Arch

```
$ sudo pacman -Syu gcc
```

GCC diponible

Una vez esté instalado el GCC, es posible consultar la versión que quedó instalada, tal y como se presenta a continuación:

```
$ gcc --version

gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions
    .  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.
```

A.4. Instalación en Android

Es posible programar en el Lenguaje C en un dispositivo móvil, como por ejemplo, en el Sistema Operativo Android. Para ello basta con instalar una de tantas aplicaciones que permite realizar esta tarea. Dos de estas aplicaciones son:

- **Cxxdroid - C++ compiler IDE for mobile development**
  https://play.google.com/store/apps/details?id=ru.iiec.cxxdroid
- **CPP N-IDE - C/C++ Compiler & Programming - Offline**
  https://play.google.com/store/apps/details?id=com.duy.c.cpp.compiler

A.5. Compilando un programa de prueba

Para probar que el compilador está instalado correctamente, se puede crear un pequeño programa, tal y como se muestra a continuación, `hola.c`.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf( "Hola Mundo\n" );
6
7     return 0;
8 }
```

Una vez creado el programa en algún editor de textos, se procede a utilizar el compilador de C para crear la respectiva aplicación:

```
$ gcc hola.c -o hola
$ ./hola
Hola Mundo
```

El resultado una vez ejecutado el programa debe corresponder a mensaje impreso `"Hola Mundo"`, si por el contrario, se obtienen otros mensajes, como por ejemplo:

```
$ gcc hola.c -o hola
-bash: gcc: command not found
```

Indicaría que el compilador no está correctamente instalado. No obstante si el mensaje fuera algo como:

```
$ gcc hola.c -o hola
hola.c:5:29: error: expected ';' after expression
    printf( "Hola Mundo\n" )
                            ^
                            ;
1 error generated.
```

Indicaría que el compilador está correctamente instalado, pero el programa digitado tiene errores. Para este ejemplo, se omitió el terminador de punto y coma al final de la línea 5 (`printf`).

Independiente del sistema operativo, una vez se ha instalado GCC, se debe proceder a instalar uno de muchos Entornos de Desarrollo Integrado.

B. Instalación del Entorno Integrado de Desarrollo (IDE)

Existe un conjunto grande de Entornos Integrados de Desarrollo (IDE), entre los más conocidos están: Eclipse, NetBeans, XCode, Dev-C++, Code::Blocks, Qt Creator, KDevelop, Geany.

Dependiendo de la experiencia, los requisitos y alcances de los proyectos a desarrollar, se selecciona uno u otro. Como la idea es disponer de un IDE que sea lo más simple posible para dar los primeros pasos en la programación en Lenguaje C, se seleccionó el IDE llamado Geany[13].

Geany provee versiones para varios sistemas operativos, entre ellos, Windows, MacOs y Linux. Lo único a tener en cuenta, es seleccionar la versión requerida de la página oficial (Ver Figura 7.8) y seguir el asistente de instalación.

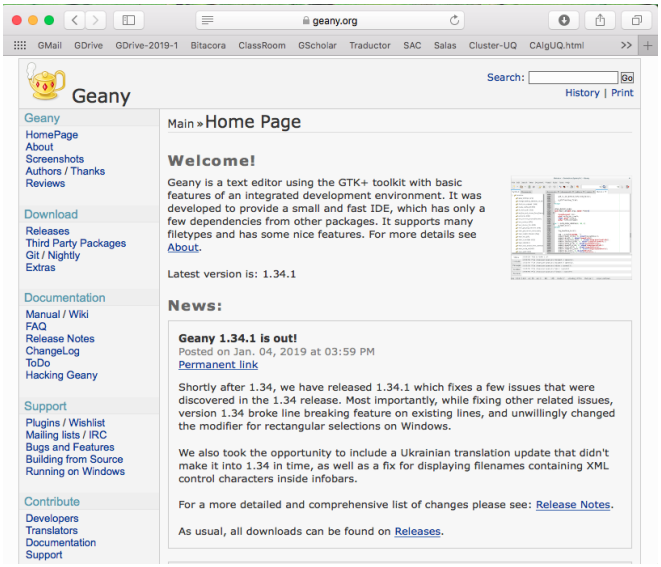


Figura 7.8: Página oficial del IDE Geany

Una vez se instale correctamente el IDE, se recomienda realizar un pequeño programa para verificar su correcto funcionamiento (Ver Figura 7.9).

Una vez digitado el programa, se debe proceder a construir la aplicación haciendo clic en el icono para dicha tarea (Ver Figura 7.9).

[13] `https://www.geany.org/Download/Releases`

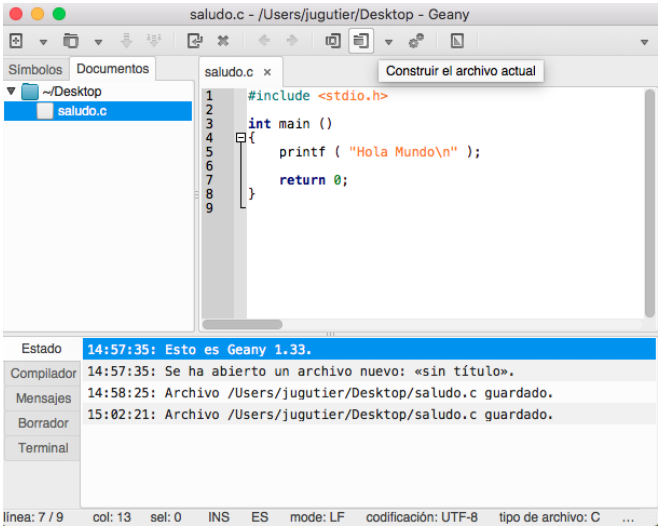


Figura 7.9: Creando un ejemplo en Geany

Una vez realizada la construcción (compilación) correctamente (sin errores), se debe proceder a ejecutar la aplicación construida haciendo clic en el icono destinado para esta tarea (Ver Figura 7.10).

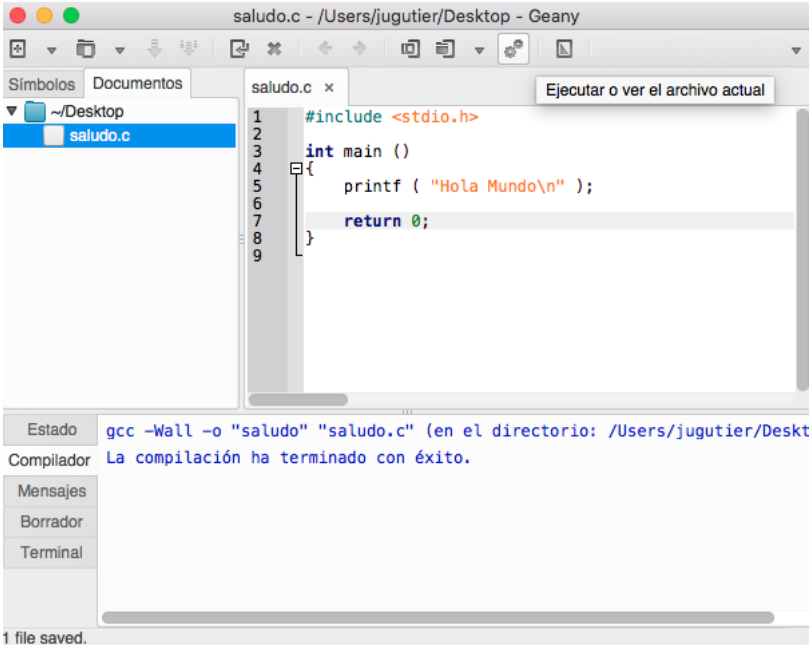


Figura 7.10: Ejecutando un ejemplo en Geany

El resultado obtenido se puede observar en la Figura 7.11.

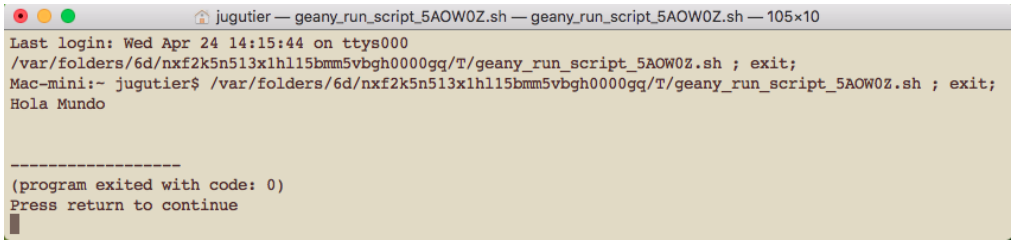


Figura 7.11: Salida del ejemplo

C. Compilando un programa con la biblioteca `uniquindio`

Una vez el compilador se encuentre correctamente instalado, puede experimentar con la compilación de un programa que emplee la biblioteca `uniquindio.h`.

`http://sara.uniquindio.edu.co/LenguajeC/BibliotecaUQ/`

Allí encontrará tres archivos:

- **`main.c`**: ejemplo que ilustra el uso de la biblioteca `uniquindio.h`.
- **`uniquindio.h`**: archivo de cabecera de la biblioteca de la Universidad. En el momento de la publicación del libro, la biblioteca solo dispone de una funcióon y tres procedimientos:
  - `void clrscr();`
    Procedimiento para borrar la pantalla.
  - `void gotoxy( int x, int y);`
    Procedimiento para ubicar el cursor en cierta posición de la pantalla.

- `char getch();`
  Función para conocer cual tecla fue presionada por el usuario.

- `void getString( char *cadena, int longitud);`
  Procedimiento para leer una cadena de una longitud especificada, impidiendo que se ingresen más caracteres de los indicados y reservando siempre la última posición para el fin de cadena `\0`.

- **`uniquindio.c`**: archivo con una de las posibles formas de implementar la biblioteca de la Universidad (este archivo puede ser escrito de "mil maneras", siempre y cuando respete las interfaces declaradas en el archivo de cabecera).

Una vez ha descargados los tres archivos en una carpeta del sistema, puede proceder a compilar el ejemplo y verificar su correcto funcionamiento:

```
1  #include <stdio.h>
2  #include "uniquindio.h"
3
4  int main()
5  {
6      char nombre[ 5 ], apellido[ 5 ];
7      int  edad, hijos;
8
9      clrscr();
10
11     printf ( "       Ingrese la edad: " );
12     scanf  ( "%d", &edad );
13
14     printf ( "     Ingrese el nombre: " );
15     getString ( nombre, 5 );
16
17     printf ( "   Ingrese el apellido: " );
18     getString ( apellido, 5 );
19
20     printf ( "Ingrese cantidad hijos: " );
21     scanf  ( "%d", &hijos );
22
23     printf ( "Edad    : (%d)\n", edad     );
24     printf ( "Nombre  : (%s)\n", nombre   );
25     printf ( "Apellido: (%s)\n", apellido );
26     printf ( "Hijos   : (%d)\n", hijos    );
27
28     return 0;
29 }
```

Este ejemplo, ilustra el uso de la biblioteca `uniquindio.h`, y observe cómo el archivo de cabecera es incluido en el programa, empleando comillas en lugar de < ... >. Esto permite, entre otras cosas, el poder identificar rápidamente si la biblioteca es propia, o es del Lenguaje C.

Al observar el contenido del archivo de cabecera `uniquindio.h`, puede evidenciar que allí están declarados (interfaces) los elementos antes descritos (una función y tres procedimientos).

```
1  #ifndef UNIQUINDIO
2
3    #define UNIQUINDIO
4
5    void clrscr ( );
6    void gotoxy ( int x, int y );
7    char getch();
8    void getString ( char *cadena, int longitud );
9
10 #endif
```

Lo único desconocido en el archivo de cabecera, es el uso de `#ifndef` y `#endif`, que evita que estas declaraciones sean incluidas, por error, más de una vez. La primera línea, le dice al compilador que continué, solo si no está definido `UNIQUINDIO`, de ser así, la define y luego declara los elementos de la biblioteca, la próxima vez que se intente incluir el archivo de cabecera, el compilador hará caso omiso de las declaraciones, debido a que ahora sí, está definido `UNIQUINDIO`.

En el libro, no se presenta ni explica el contenido del archivo `uniquindio.c`, ya que hace uso, necesariamente, de elementos que están por fuera del alcance del libro, no obstante su uso es importante para los últimos capítulos y para los nuevos proyectos que el lector desee desarrollar.

Una vez el lector adquiera cierto dominio del lenguaje, podrá crear sus propias funciones y procedimientos en su biblioteca personal.

La forma correcta de compilar un programa que haga uso de la biblioteca `uniquindio.c`, es la siguiente:

```
$ gcc main.c uniquindio.c -o main
$ ./main
        Ingrese la edad: 14
      Ingrese el nombre: Juan
    Ingrese el apellido: Guti
     Ingrese cant hijos: 0
Edad     : (14)
Nombre   : (Juan)
Apellido: (Guti)
Hijos    : (0)
```

El ejemplo, limita de forma intencional y arbitraria, la longitud máxima a cinco caracteres, para así ilustrar como el procedimiento consigue impedir que el usuario ingrese más elementos de los especificados.

En algunos ambientes de programación, por ejemplo en Android, en donde el acceso a la terminal es más restringido, se puede hacer lo siguiente: (esto para evitar hacerlo a nivel del IDE o del compilador; no obstante, aunque funciona en todos los sistemas, esto no es una buena práctica).

```c
#include <stdio.h>
#include "uniquindio.h"

#include "uniquindio.c"
```

El lugar correcto para realizar la configuración del IDE, es propio de cada uno, por ejemplo en Geany, si se intenta compilar el programa de `main.c`, sin incluir manualmente `uniquindio.c`, se obtiene un mensaje de error, Ver Figura 7.12.

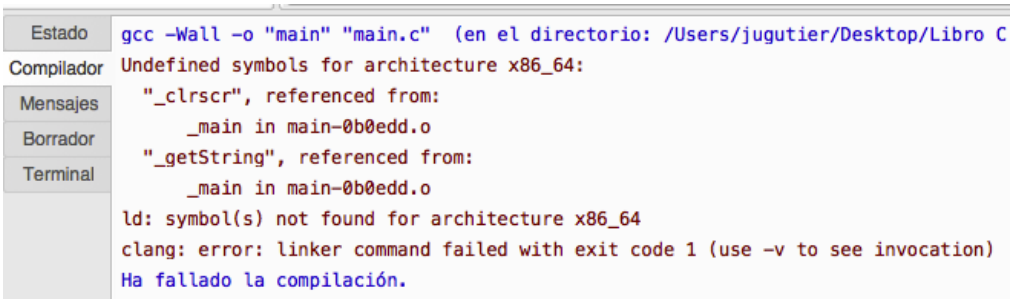


Figura 7.12: Compilación fallida en el Geany

Este error es provocado, porque el compilador no encuentra una implementación de la biblioteca (`uniquindio.c`). Por lo tanto, el siguiente paso consiste en buscar la opción para configurar el Geany (Ver Figura 7.13).

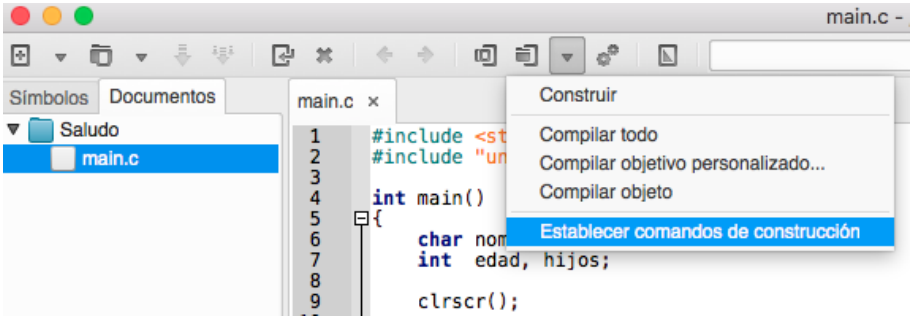


Figura 7.13: Opción de configuración en el Geany

Una vez allí, se le indica al Geany que cuando compile el programa, incluya la implementación de la biblioteca (Ver Figura 7.14).

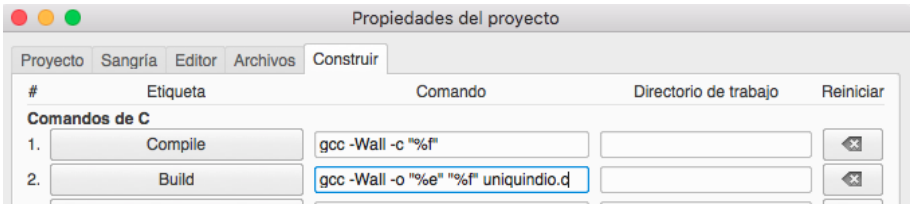


Figura 7.14: Configuración en el Geany para la biblioteca `uniquindio`

Ahora sí, cuando se compile de nuevo el programa, el resultado deberá ser exitoso (Ver Figura 7.15) y podrá ejecutar el programa (Ver 7.16).

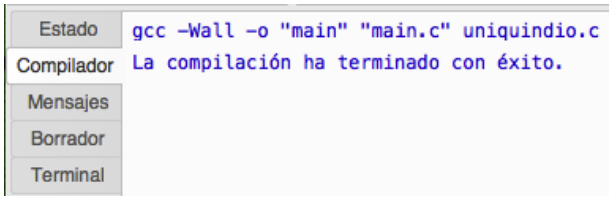


Figura 7.15: Compilación exitosa en el Geany

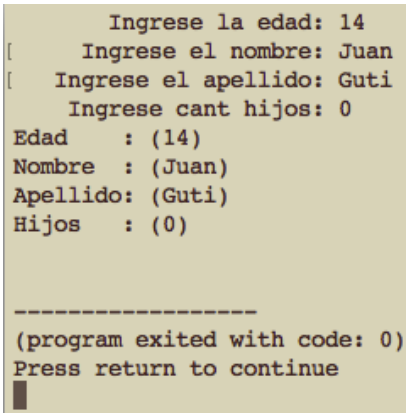


Figura 7.16: Resultado de la ejecución del ejemplo `demo.c`

BIBLIOGRAFÍA

[Aldea et al., 2017] Aldea, I. A., Torreblanca, J. M. M., Venegas, C. V., and Cabeza, A. Z. (2017). *100 Problemas resueltos de programación en Lenguaje C para ingeniería*. Paraninfo Universidad.

[Corona and Ancona, 2011] Corona, M. A. N. and Ancona, M. V. (2011). *Diseño de algoritmos y su codificación en lenguaje C*. McGraw-Hill Interamericana.

[Fernánadez et al., 2014] Fernánadez, R. M., y Beltrán, . G., Fernández, S. T., Gallego, J. J., and Álamo Lobo, F. (2014). *Programación en C. Ejercicios*. IDextra Editorial.

[Girón, 2015] Girón, A. M. M. (2015). *Diseño de algoritmos y su programación en C*. Alfaomega.

[Jiménez et al., 2016] Jiménez, J. A. M., Jiménez, E. M. H., and Alvarado, L. N. Z. (2016). *Fundamentos de Programación, Diagramas de flujo, Diagramas N-S, Pseudocódigo y Java*. Alfaomega.

[Mancilla et al., 2016] Mancilla, A. H., Ebratt, R. G., and Capacho, J. P. (2016). *Diseño y construcción de algoritmos*. Universidad del Norte – Editorial.

[Pimiento, 2009] Pimiento, W. M. C. (2009). *Fundamentos de Lógica para Programación de Computadores*. Universidad Piloto de Colombia.

[Pixabay.com, 2018a] Pixabay.com (2018a). Imagen 1: ok. `https://pixabay.com/es/vectors/enganche-hacer-autostop-154664/`.

[Pixabay.com, 2018b] Pixabay.com (2018b). Imagen 2: libro en las manos. https://pixabay.com/es/vectors/biblia-libro-manos-otros-2026336/.

[Pixabay.com, 2018c] Pixabay.com (2018c). Imagen 3: computador libros. https://pixabay.com/es/vectors/laptop-conocimiento-informacion-1723059/.

[Sznajdleder, 2017] Sznajdleder, P. A. (2017). *Programación Estructurada a Fondo - Implementación de Algoritmos en Lenguaje C*. Alfaomega.

[Trejos, 2017] Trejos, O. I. B. (2017). *Programación Imperativa con Lenguaje C*. Ecoe Ediciones.

[Villalobos, 2014] Villalobos, R. M. (2014). *Fundamentos de programación C++*. Editorial Macro.

ÍNDICE ALFABÉTICO

INTRODUCCIÓN A LA PROGRAMACIÓN EN C.

Elizcom s.a.s

Sin duda alguna, el Lenguaje de Programación C, es uno de los lenguajes de programación de sistema que más se ha utilizado desde los inicios de la computación, no solo por el tiempo que tiene desde su creación, sino porque está disponible, básicamente, en todos los sistemas y arquitecturas de computadores existentes, ya sea a escala muy pequeña (sistemas embebidos o dedicados), hasta súper-computadoras de todo tipo; esto incluye, por supuesto, todos los sistemas operativos Unix y similares, así como Windows.

Algunas de las razones de su gran portabilidad, es que posee un número reducido de instrucciones y bibliotecas estándar, que facilitan su migración, además de ser un lenguaje muy eficiente que emplea pocos recursos computacionales; por esto, es uno de los principales lenguajes en las competencias de programación mundial y es utilizado con frecuencia en la computación de alto desempeño a nivel de clusters y supercomputación.

Por todo esto, se creó esta obra como un primer libro de consulta para docentes, estudiantes y todas aquellas personas autodidactas, interesadas en aprender programación en Lenguaje C. Dirigido a cualquier lector con conocimientos básicos o nulos sobre programación y que desee ampliarlos o adquirirlos. La obra comienza desde algunos elementos históricos y hace un recorrido por los temas fundamentales de la programación hasta llegar al trabajo con archivos y la creación de nuevas bibliotecas. Todo esto mediante el empleo de ejemplos que son analizados, resueltos y explicados para darle al lector los elementos necesarios para realizar un conjunto de problemas adicionales.

El código fuente
de los ejemplos
está disponible en:

http://sara.uniquindio.edu.co/LenguajeC/

9 789588 801711