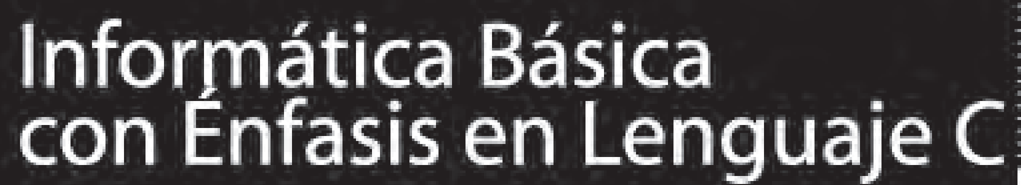


Universidad Nacional de Colombia
Sede Manizales

EDUARDO VILLEGAS JARAMILLO

INFORMÁTICA BÁSICA CON ÉNFASIS EN LENGUAJE C

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MANIZALES

I.S.B.N 958-9322-60-3

© 2000 UNIVERSIDAD NACIONAL
DE COLOMBIA SEDE MANIZALES

AUTOR:

EDUARDO VILLEGAS JARAMILLO
Ingeniero de Sistemas
Ms.Sc. Ingeniería de Sistemas y Computación
Profesor Asociado
Facultad de Ciencias y Administración
Universidad Nacional de Colombia
Sede Manizales

Trabajo presentado para optar a la categoría
de Profesor Asociado

IMPRESO POR:

Centro de Publicaciones
Universidad Nacional de Colombia
Sede Manizales

Noviembre de 2000
Primera Edición

CONTENIDO

INTRODUCCIÓN 7

PRIMERA PARTE: CONCEPTOS BÁSICOS DE INFORMÁTICA

1. INFORMÁTICA 9
1.1. Definición 9
1.2. Historia de la informática y la computación 9
1.3. Generaciones de computadores 11
1.4. Áreas de desempeño de la informática 12

2. CONCEPTOS BÁSICOS 13
2.1. Terminología 13
2.2. Operaciones realizables por un computador 16
2.3. Hardware 17
2.3.1 Dispositivos de Entrada 17
2.3.2 Dispositivos de Salida 19
2.3.3 Dispositivos de Procesamiento 21
2.3.4 Dispositivos de Almacenamiento 22
2.4. Software 24
2.4.1. Software del sistema 24
2.4.2. Software de aplicación 24

3. MANEJO DE COMPUTADORES 24
3.1. DOS 25
3.2. Windows 3.1 / Windows 3.11 27
3.3. Windows 95 / Windows 98 29
3.4. UNIX 30
3.5. Otros sistemas operacionales 31

SEGUNDA PARTE: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

4. SOLUCIÓN DE PROBLEMAS 33
4.1. Estrategias para alcanzar la solución de un problema 33
4.2. Particularización y generalización 34
4.3. Dividir y conquistar 43
4.4. Ensayo y error 48
4.5. Ejercicios propuestos 52

5. ALGORITMOS ... 54
5.1. Elementos de un algoritmo ... 55
5.1.1. Variables y Constantes ... 55
5.1.2. Operaciones y Expresiones ... 55
5.1.3. Instrucciones de Entrada y Salida ... 57
5.1.4. Instrucciones Condicionales ... 59
5.1.5. Instrucciones Repetitivas ... 65
5.2. Verificación de algoritmos ... 72
5.2.1. Pruebas de Escritorio ... 73
5.2.2. Verificación formal ... 76
5.3. Ejercicios propuestos ... 77

TERCERA PARTE: EL LENGUAJE C

6. CONCEPTOS BÁSICOS DEL LENGUAJE ... 79
6.1. Historia ... 79
6.2. Estructura del lenguaje ... 80
6.2.1. Declaraciones externas ... 80
6.2.2. Instrucciones para el preprocesador ... 82
6.2.3. Definición de constantes y macroinstrucciones ... 82
6.2.4. Definición de estructuras o registros ... 82
6.2.5. Declaración de prototipos para las funciones ... 83
6.2.6. Declaración de variables globales ... 83
6.2.7. Definición de funciones ... 83
6.3. Tipos de datos básicos ... 84
6.4. Constantes, macros y variables ... 86
6.4.1 Las constantes ... 86
6.4.2 Las macros ... 87
6.4.3 Las variables ... 87
6.5. Operaciones simples ... 88
6.5.1. Asignación ... 88
6.5.2. Incremento o decremento unitario ... 90
6.5.3. Operaciones en términos de la misma variable ... 91
6.6. Operaciones de salida ... 91
6.6.1. Función printf ... 92
6.6.2. Función puts ... 94
6.6.3. Macroinstrucción putchar ... 94
6.7. Operaciones de entrada ... 95
6.7.1. Función scanf ... 95
6.7.2. Función gets ... 96
6.7.3. Función getch ... 97
6.7.4. Función getche ... 97
6.8. Tipos de operadores ... 98
6.8.1. Operadores relacionales ... 98

6.8.2. Operadores de igualdad 98
6.8.3. Operadores lógicos 98
6.8.4. Operadores de bits 99
6.9. Bloques de instrucciones 100
6.10. Ejercicios propuestos 100

7. ESTRUCTURAS DE CONTROL 101
7.1. Estructuras condicionales 101
7.1.1. Condicional Simple 101
7.1.2. Condicional Múltiple 103
7.1.3. Operador Condicional 105
7.2. Estructuras repetitivas 105
7.2.1. Estructura while 105
7.2.2. Estructura for 106
7.2.3. Estructura do ... while 108
7.3. Ejercicios propuestos 110

8. ARREGLOS : VECTORES Y MATRICES 110
8.1. Arreglos unidimensionales 111
8.2. Cadenas de caracteres 116
8.3. Arreglos bidimensionales 122
8.4. Ejercicios resueltos 124
8.5. Ejercicios propuestos 126

9. MANEJO DE REGISTROS 127
9.1. Declaración 127
9.2. Inicialización de estructuras 128
9.3. Acceso a los campos de las estructuras 128
9.4. Ejercicios resueltos 130
9.5. Ejercicios propuestos 132

10. FUNCIONES 133
10.1. Declaración de funciones 133
10.2. Tipos de parámetros 134
10.3. Obtención de resultados 136
10.4. Recursión 137
10.5. Ejercicios propuestos 140

11. MANEJO DE ARCHIVOS 142
11.1. Declaración 142
11.2. Apertura y cierre de archivos 142
11.3. Escritura en archivos 144
11.3.1. Escritura en archivos texto 144
11.3.2. Escritura en archivos binarios 145
11.4. Lectura de archivos 146
11.4.1. Lectura en archivos texto 146

11.4.2. Lectura en archivos binarios 146
11.5. BORRADO FÍSICO DE ARCHIVOS 147
11.6. CONTROL DE LOS ARCHIVOS 148
11.6.1. Movimiento del apuntador del archivo 148
11.6.2. Determinación de la posición actual del apuntador del archivo 149
11.6.3. Detección del fin de archivo 149
11.6.4. Grabación de buffers 150
11.7. Ejercicios resueltos 151
11.8. Eejercicios propuestos 153

12. APUNTADORES 154
12.1. Definiciones y declaración de los apuntadores 154
12.2. Operadores con apuntadores 155
12.3. Apuntadores en los parámetros por referencia 158
12.4. Uso de apuntadores en vectores y matrices 163
12.5. Ejercicios propuestos 165

13. MEMORIA DINÁMICA / ESTRUCTURAS DE DATOS DINÁMICAS 166
13.1. Listas con encadenamiento simple 167
13.1.1. Representación de una lista encadenada simple 169
13.1.2. Insertar un elemento 170
13.1.3. Eliminar un elemento 175
13.1.4. Recorrer una lista 178
13.2. Listas con encadenamiento doble 182
13.2.1. Representación de una lista encadenada doble 182
13.2.2. Insertar un elemento 183
13.2.3. Eliminar un elemento 187
13.2.4. Recorrer una lista 190
13.3. Estructuras de datos avanzadas 191
13.4. Ejercicios propuestos 193

BIBLIOGRAFÍA 194

ANEXOS

ANEXO 1: TABLA DE CARACTERES ASCII 195
ANEXO 2: FUNCIONES MATEMÁTICAS BÁSICAS 197
ANEXO 3: ERRORES DE EJECUCION FRECUENTES 199

INTRODUCCIÓN

El mundo de la informática rápidamente ha transformado el trabajo cotidiano del hombre y su entorno. Cada vez encontramos una mayor necesidad de soluciones informáticas para el normal desenvolvimiento en el trabajo, el hogar, las comunicaciones e inclusive en el entretenimiento.

La informática y su principal herramienta, el computador comienza a ser un accesorio indispensable para el desarrollo del hombre. Por esta razón elementos de uso común como el dinero tiende a desaparecer, ya que las transacciones monetarias se convierten en simples operaciones aritméticas realizadas desde una terminal de un computador, un cajero electrónico o incluso por medio del teléfono. Es posible encontrar hoy en día que elementos como la máquina de escribir, la fotocopiadora, el fax, el correo, los equipos de sonido e inclusive el teléfono se ha ido desplazando paulatinamente ante el avance de los computadores, razón por la cual el desconocimiento de su importancia y necesidad de manejo es cada vez más evidente.

La toma de decisiones en las empresas depende cada vez más de lo oportuna, veraz, fiable y completa que sea la información suministrada sobre el estado de la empresa, razón por la cual los sistemas de información ganan cada vez mayor importancia en el desarrollo de las mismas siendo las empresas a todo nivel uno de los principales usuarios masivos de la informática en todas sus áreas. En el campo productivo, la automatización de procesos ha llegado a niveles inesperados siendo completamente dependiente de los microprocesadores (motor de los computadores), sin los cuales sería imposible alcanzar este nivel de eficiencia y confiabilidad en mercados cada vez más competitivos.

La informática surge como una ayuda para que el hombre pueda realizar trabajos rutinarios y repetitivos en su vida diaria de una forma más fácil y eficiente, razón más que suficiente para que en las universidades la enseñanza de la informática sea obligatoria en todas y cada una de las disciplinas, eso sí teniendo en cuenta el tipo de disciplina para que dicha enseñanza se realice según las necesidades de cada profesión.

Este libro está dirigido a aquellas personas que deseen involucrarse con la informática, específicamente para aquellos que estén interesados en aprender sus conceptos básicos y algunas técnicas de programación orientadas a uno de los principales lenguajes de desarrollo de programas como lo es el Lenguaje C.

El desarrollo de este libro se basó en las notas de clase generadas para la orientación de los cursos Informática I e Informática II de la carrera de Ingeniería Electrónica, debido a la actualización y modernización que se realizó al interior de la carrera, pasando del aprendizaje del lenguaje Pascal al lenguaje C y a los conceptos necesarios para el curso de Informática y Lógica Computacional de la carrera de Administración de Sistemas Informáticos.

Este libro está dividido en tres partes principalmente, donde en la primera se introducen los conceptos básicos sobre la informática, conociendo parte de la terminología, la historia, los componentes de su principal herramienta como lo es el computador y los diferentes tipos de programas diseñados para su manejo. La segunda parte se enfoca en los conceptos básicos de la programación, siguiendo los diferentes pasos desde el planteamiento de un problema, su entendimiento, búsqueda de una solución y posterior formalización de la misma mediante un lenguaje algorítmico, que más adelante servirá como base para la escritura de programas en un lenguaje determinado y la tercera parte, en la cual se especifican los fundamentos del lenguaje de programación C, presentando sus principales características y posibilidades, mediante la explicación de los diferentes tipos de elementos e instrucciones, desarrollando ejemplos y ejercicios que permitan al interesado realizar las debidas prácticas que le aseguren el entendimiento y utilización del lenguaje.

PRIMERA PARTE

CONCEPTOS BÁSICOS DE INFORMÁTICA

1. INFORMÁTICA

1.1. Definición

La informática se define como una parte de la ciencia que estudia el tratamiento automático y racional de la información, presentando entre otras las siguientes funciones básicas:

- desarrollo de nuevas máquinas
- desarrollo de nuevos métodos de trabajo
- construcción de aplicaciones informáticas
- mejoramiento de métodos y aplicaciones existentes

Etimológicamente, el término informática surge en Francia en la década de los años sesenta como *informatique* el cual se compone de la contracción de las palabras INFORmation autoMATIQUE.

1.2. Historia de la informática y la computación

Las necesidades de cálculo por parte del hombre datan desde hace miles de años, cuando este requería contabilizar sus pertenencias que básicamente se componían de árboles, ovejas, aves y otros animales, los cuales representaba mediante piedras o palos que correspondían a cada una de sus pertenencias.

Dadas las crecientes necesidades del hombre en el cálculo y almacenamiento de información, surgieron diferentes dispositivos que le facilitaron esta tarea. El ábaco construido en el medio oriente, que data del año mil doscientos a.C. se toma como elemento precursor de los sistemas actuales de cálculo.

Posteriormente se desarrollaron diferentes elementos para la realización de cálculos destacándose la regla de cálculo, la sumadora de Pascal y la máquina analítica de Babbage entre otras.

El primer computador electrónico se construyó entre 1943 y 1946 y se denominó ENIAC (Electronic Numerical Integrator and Calculator), basaba su funcionamiento en los tubos de vacío, y su peso llegaba a las 30 toneladas, con una capacidad de proceso que le permitía realizar 5,000 sumas o 300

multiplicaciones por segundo. Desarrollos posteriores de ENIAC condujeron a la construcción de UNIVAC (Universal Automatic Computer), que fue el primer computador digital producido comercialmente. Su principal característica era que utilizaba diodos de cristal en lugar de los tubos de vacío, dándole una mayor confiabilidad.

En el campo de los computadores personales, IBM en el año de 1975 introdujo el modelo 5100 que tenía una memoria de 16 KBytes e incluía un intérprete del lenguaje BASIC, el cual evolucionó hasta llegar a los computadores personales que tenemos hoy en día. Otras líneas de cómputo se han desarrollado en forma paralela destacándose Apple, DEC, Hewlett Packard, Compaq y Texas Instruments principalmente.

Paralelamente con el avance de los computadores se desarrollaron los sistemas operacionales y los lenguajes de programación, siendo importante destacar la aparición en 1,954 del lenguaje FORTRAN (Formula Translating System) como el primer lenguaje de programación de propósito general y relativamente fácil de utilizar. Otros lenguajes han seguido los pasos del FORTRAN como son el ALGOL (Algoritmic Language) en 1,958, el COBOL (Common Business Oriented Language) en 1959, el BASIC(Beginner's All purpose Symbolic Instruction Code) en 1965, el PASCAL, el C, Delphi y Java.

Cronología de los equipos de cómputo:

- 500 a.C. Un artefacto similar al ábaco es utilizado en Egipto
- 200 a.C. El ábaco que conocemos hoy es desarrollado en China, en Japón se denominó sorobán
- 1620 El Inglés John Napier, crea las tablas de multiplicar
- 1653 El Francés Blaise Pascal, desarrolla una máquina para calcular denominada la pascalina
- 1673 El Alemán Gottfried Wilhelm von Leibniz, desarrolla máquinas superiores a la pascalina de propósito general
- 1823 El Inglés Charles Babbage, diseña una máquina para el cálculo analítico, la cual nunca llegó a ser operativa
- 1930 El Americano George Stibitz, de los laboratorios Bell construye una calculadora numérica utilizada para la interconexión de circuitos en telefonía
- 1834 El Inglés Charles Babbage, formula, diseña y construye lo que se denominó máquina analítica
- 1936 El matemático Inglés Alan Turing, establece los principios teóricos del computador mediante la construcción de la máquina de Turing
- 1946 Se desarrolla en Estados Unidos el primer computador basado en válvulas de vacío denominado ENIAC
- 1947 William Sockley, Jhon Bardeen y Walter Brattin inventan el transistor
- 1949 Se desarrolló el primer computador con manejo de datos en paralelo, al cual se le denominó BINAC
- 1952 Se realizó la primer actualización de ENIAC, a la cual se le denominó EDVAC
- 1957 Se construye el primer computador IBM
- 1959 La Texas Instruments inventa y desarrolla el circuito integrado
- 1971 Aparece un circuito integrado denominado microprocesador
- 1971 Intel produce el microprocesador, que se denominó 8008 como precursor de los actuales

- 1975 CRAY Research lanza su primer super computador
- 1979 Motorola lanza el MC-68000
- 1981 IBM produce el primer computador personal (PC), utilizando el DOS como sistema operativo
- 1984 APPLE presenta el Macintosh como la revolución del mercado
- 1990 Microsoft presenta el Windows 3.0 como su nuevo sistema operacional
- 1994 Intel produce el Pentium como su último desarrollo en microprocesadores
- 1995 Microsoft lanza el Windows 95 como la nueva generación de los sistemas operacionales
- 1999 Surge el microprocesador Intel Pentium III
- 2000 Lanzamiento del sistema operación Windows 2000

1.3. Generaciones de computadores

Los diferentes y continuos cambios tecnológicos han originado que los computadores se clasifiquen por generaciones, las cuales hoy no son muy tenidas en cuenta debido a la velocidad con que cambian dichas tecnologías, pero en el transcurso del tiempo desde su aparición ha sido la principal forma de clasificación.

Las principales generaciones de computadores se han definido como:

Primera generación: (1940-1952) La constituyen aquellos computadores cuyo diseño se basó en las válvulas de vacío como principal elemento de control y cuyo uso principal fue el campo científico y militar. Utilizaban el lenguaje de máquina como forma de programación, y como memorias o unidades de almacenamiento las tarjetas y cintas perforadas.

Segunda generación: (1952-1964) comienza con la substitución de las válvulas de vacío por los poderosos transistores (cumplían la misma tarea de una manera mas simple, barata y confiable), dando como resultado máquinas más potentes y confiables; Adicionalmente se perdió tamaño, consumo y disminuyó el precio. Su uso se generalizó a empresas de altos recursos económicos que se dedicaban a los negocios en los cuales el manejo de la información era la clave del éxito.

Las unidades de almacenamiento se basaban en núcleos de ferrita, tambores magnéticos y cintas magnéticas. Adicionalmente, comenzaron a utilizarse los llamados lenguajes de programación incluyendo FORTRAN, COBOL y ALGOL.

Tercera Generación: (1964-1971) El elemento más significativo es la aparición del circuito integrado, el cual consistía en el encapsulamiento de gran cantidad de componentes (transistores) sobre una pastilla de silicona o plástico. La miniaturización se extendió a todos los circuitos de los computadores, apareciendo así los minicomputadores. Paralelamente, comenzó el desarrollo de sistemas operacionales en los que se incluyó la multiprogramación, el tiempo real y el modo de proceso interactivo. Los semiconductores comienzan a ser parte de las memorias y surgen los discos magnéticos.

Cuarta generación: (1971-1981) El surgimiento del microprocesador en 1971 permite la integración de todos los componentes de la UCP (Unidad Central de Proceso) de un computador en un

solo circuito, dando paso al surgimiento de los microcomputadores y computadores personales. El disquete hace su aparición como elemento de almacenamiento externo. Surgen gran cantidad de sistemas operacionales y lenguajes de programación.

Quinta generación: (1981-) Las nuevas tecnologías incluyen la muy alta escala de integración de componentes (VLSI), inteligencia artificial, lenguaje natural, interconexión con todo tipo de computadoras, integración de imágenes, sonido, datos y voz (Multimedia), la realidad virtual, discos ópticos (CD-DVD), Internet, etc. revolucionando completamente las tendencias de la informática y por ende del hombre.

Adicionalmente se habla de nuevas generaciones que dividen a la quinta generación, teniendo en cuenta elementos tan importantes como: los multimedios, la realidad virtual y las redes internacionales.

1.4. Áreas de desempeño de la informática

La informática hoy en día se desempeña en todas las áreas realizando tareas que van desde su uso como máquina de escribir, utilización como archivador, máquina de entretenimiento, hasta la realización de sofisticados cálculos para el control de dispositivos de diferente índole.

Entre las principales áreas de desempeño de la informática es importante tener en cuenta:

- **Administrativa / Financiera:** Las empresas hoy día requieren de una mayor disponibilidad de la información con el fin de tomar decisiones sobre sus inversiones, desarrollos, y otras actividades. En esta área vale la pena destacar programas tales como: manejo de inventarios, contabilidad, nómina, facturación, cartera, cuentas por pagar y diferentes cuadros comparativos como la base para el manejo y control de las empresas.

- **Diseño / Manufactura / Control asistidos por computador (CAD, CAM, CAP):** En el área de producción de las empresas, los sistemas computarizados van desde el diseño del elemento que se desea construir, la simulación de su comportamiento, la programación de las diferentes máquinas que conforman el proceso, el control de calidad, su transporte y posterior mantenimiento.

- **Simulación / Gráficas:** Las características gráficas y de simulación que poseen los computadores permiten que cada día se ahorren cientos de millones de dólares con el fin de determinar la viabilidad y funcionamiento de un determinado producto sin realizar físicamente su construcción. Los simuladores de robots, aviones, carros e incluso de maquinaria son cada vez más comunes siendo útiles para el adiestramiento y capacitación de los operarios.

- **Enseñanza:** Cada día los computadores son mas utilizados en el aprendizaje de las diferentes disciplinas, ya sea mediante los cursos virtuales que ofrecen innumerables instituciones, enciclopedias multimedia, o con el uso de los bancos de datos a través de las redes internacionales como Internet.

- **Oficina:** Los procesadores de palabras, las hojas electrónicas, las agendas y las bases de datos son elementos que no pueden faltar en una oficina moderna, simplificando en gran medida el trabajo que allí se realiza.

▪ **Entretenimiento:** En el campo del entretenimiento los computadores se han puesto a la vanguardia, presentando innumerables juegos que van desde el tradicional triqui, juegos de cartas, la ciencia del ajedrez, hasta llegar a los poderosos simuladores de vuelo con efectos de verdadero realismo.

▪ **Comunicaciones:** El campo de mayor desarrollo en las últimas décadas tiene que ver con las comunicaciones, el auge de las redes de datos como Internet han cambiado físicamente la forma de trabajo de muchas personas y empresas en el ámbito mundial, permitiendo que todas las personas puedan comunicarse de una forma sencilla y económica desde cualquier lugar del mundo, mediante servicios tales como: correo, chats, videofono, etc.

▪ **Hogar:** La posición del computador en el hogar es cada vez de mayor jerarquía, inicialmente su uso se limitaba únicamente para realizar algunos cálculos, pequeños y aburridos juegos, o como máquinas de escribir. Hoy en día su uso va mas allá llegando a reemplazar elementos tales como: juegos de video, máquina de escribir, fax, contestador telefónico, servicio de correo, cajero automático, equipo de sonido, entre otros.

▪ **Mercadeo:** Con el surgimiento de las redes mundiales como Internet, se ha abierto una nueva posibilidad para las empresas, siendo viable realizar publicidad y ventas mediante el uso de las páginas web, de tal manera que les permita mercadear todo tipo de productos sin la necesidad de tener puntos de venta o almacenes abiertos al público, realizando dichas labores en forma virtual, desde cualquier lugar en que se tenga acceso a dicha red.

2. CONCEPTOS BÁSICOS

2.1. Terminología

Con el fin de entender mas fácilmente la teoría asociada con la informática, es importante conocer un conjunto de términos de uso frecuente, entre los cuales vale la pena mencionar:

ALGORITMO: Conjunto de pasos o instrucciones descritas en un lenguaje sencillo que permiten llegar a la solución sistemática de un problema.

ARCHIVO: Conjunto de bytes almacenados en un dispositivo de memoria secundaria.

ASCII: (American Standard Code for Information Interchange), Medio de codificación estándar representado por un Byte y compuesto por 256 caracteres, diseñado para el intercambio de información.

BAUDIO: Unidad de medida para el intercambio de información equivalente a un bit por segundo.

BINARIO: Sistema de numeración cuya base está formada por dos símbolos el cero y el uno.

BIT: Dígito Binario (Binary Digit); es la unidad mínima de información almacenable y procesable por un computador; es equivalente a la presencia o ausencia de una señal (ON - OFF) y se asocia con un dígito de numeración binaria.

BUFFER: Dispositivo para el almacenamiento temporal de información.

BUS: Conducto que proporciona la comunicación entre dos o más dispositivos.

BYTE: Unidad de información que permite almacenar un caracter o símbolo (asociado con el código ASCII); se compone de 8 bits y numéricamente representa valores entre 0 y 255.

C: Lenguaje de programación apto para el desarrollo de aplicaciones técnico-científicas.

C++: Evolución natural del lenguaje C, incluyendo el manejo de programación orientada a objetos con sus principales características como son el manejo de clases, abstracción, herencia y polimorfismo.

CASE: Herramientas de software para la construcción de nuevo software.

CHIP: Envoltura de silicio que permite el encapsulamiento de un circuito integrado.

COMPILADOR: Programa que permite la verificación de los errores del código fuente de un programa y posteriormente lo traduce para generar el código objeto o intermedio.

COMPUTADOR: Es un dispositivo que acepta datos, realiza operaciones lógicas y/o matemáticas que manipulan o modifican la información y finalmente producen nuevos resultados a partir de esos datos. Adicionalmente, un computador puede ser visto como un dispositivo compuesto de hardware, software y firmware.

DATOS: Es la representación de valores que se almacenan en los diferentes dispositivos de hardware destinados para este fin y que son manipulados mediante los componentes del software que requieren de la información allí almacenada.

DISQUETE: Disco magnético flexible y removible utilizado para el almacenamiento de información.

EDITOR: Programa que permite la escritura, modificación y almacenamiento de archivos de texto.

ENCADENADOR: Programa que toma como entrada el código objeto o intermedio de un programa y con las librerías del lenguaje produce un código ejecutable.

FIRMWARE: Es una combinación de hardware y software que se utiliza para el almacenamiento de instrucciones. Se encuentra en las memorias ROM de los computadores y además contiene información de la configuración que se necesita para el normal funcionamiento de los computadores.

GIGABYTE: Conjunto de 1024 MegaBytes.

HARDWARE: Es la parte física de los sistemas computarizados, la cual consta de elementos electrónicos y mecánicos. El hardware permite almacenar y manipular la información que recibe

mediante sus respectivos dispositivos de entrada y salida y ejecuta las órdenes que le provee el componente de software.

HEXADECIMAL: Sistema de numeración utilizado en los computadores, cuya base esta formada por 16 símbolos.

INFORMÁTICA: Es la parte de la ciencia que se encarga del manejo organizado de la información. También se define como la ciencia que estudia el tratamiento automático y racional de la información.

INTERNET: Red mundial de información que permite el intercambio de datos, mensajes, correos, voz, video y adicionalmente brinda servicios de publicidad y mercadeo.

JAVA: Lenguaje de programación basado en C++, diseñado para trabajar en diferentes plataformas y orientado al desarrollo de programas para Internet.

KILOBYTE: Conjunto de 1024 Bytes.

LENGUAJE: Conjunto de símbolos y/o sonidos que permite el intercambio de información.

MEGABYTE: Conjunto de 1024 KiloBytes.

MEMORIA: Dispositivo para el almacenamiento de información.

MODEM: Periférico cuyo fin es el de convertir señales análogas a digitales o viceversa, con el objetivo de comunicar dos o más equipos mediante el uso de las líneas telefónicas.

NÚMEROS BINARIOS: Representación numérica basada en los dígitos 0 y 1 sobre la cual opera la aritmética y la lógica de los computadores.

PASCAL: Lenguaje de programación de alto nivel desarrollado por Nicklaus Wirth en 1970.

PERIFÉRICOS: Diferentes dispositivos que se pueden conectar a los computadores con el fin de proporcionarle una mayor capacidad en alguna de sus funciones. (Impresoras, modems, unidades de almacenamiento, micrófonos, parlantes, etc.).

PROCESADOR: Dispositivo de tipo electrónico cuyo fin es el de ejecutar las instrucciones de un programa.

PROGRAMA: Conjunto de instrucciones ejecutables por una persona o máquina con el fin de realizar una determinada tarea o labor.

RAM: Memoria de Acceso Aleatorio (Random Access Memory), conocida como la memoria principal del computador, en la cual se almacenan los diferentes datos y programas en ejecución; es de carácter volátil ya que depende del suministro de energía eléctrica para mantener la información.

RED DE COMPUTADORES: Interconexión de computadores y demás periféricos con el fin de compartir recursos e intercambiar información.

ROM: Memoria de solo Lectura (Read Only Memory), conocida como la memoria de configuración en la cual se almacenan microprogramas y datos que permiten la configuración del computador.

SISTEMA: Conjunto de elementos que interactuan para cumplir un determinado objetivo,

SISTEMA OPERACIONAL: Conjunto de programas y datos que controlan el funcionamiento del computador.

SOFTWARE: Es el conjunto de instrucciones que le indican al computador que debe hacer. El software se encuentra en forma de programas y datos, donde los primeros contienen las instrucciones para el procesamiento de los segundos.

- Software del Sistema: Está compuesto por aquellos programas que hacen que el computador funcione, interactuando con los distintos elementos que lo componen. Se les denomina sistemas operacionales y entre los mas conocidos se encuentran el DOS, UNIX y WINDOWS.

- Software de Aplicación: Es aquel dirigido para que el computador realice una tarea específica para el usuario. Los procesadores de palabra, hojas electrónicas, navegadores de red y bases de datos son los programas más comunes en esta clasificación.

TERABYTE: Conjunto de 1024 GigaBytes.

VIDEO RAM: Memoria destinada para el almacenamiento de la información desplegada en las pantallas de los computadores.

2.2. Operaciones realizables por un computador

En la práctica los computadores realizan tareas de diferente índole y complejidad, donde algunas veces se les califica con la posibilidad de realizar cualquier tipo de tarea. Debe tenerse en cuenta que su capacidad no va mas allá de los diferentes dispositivos, periféricos, programas e información a los cuales tiene acceso.

Las operaciones realizables por los computadores se reducen básicamente a dos tipos:

- Operaciones Matemáticas: suma, resta, multiplicación y división.

- Operaciones Lógicas: comparan dos valores para determinar pruebas lógicas (combinaciones de mayor, menor o igual) y tomar decisiones.

Otro tipo de operaciones que realizan los computadores es el de mover información de un lado a otro, permitiéndole almacenarla en su memoria, en discos, disquetes o enviarla a través de uno de sus

puertos para su almacenamiento remoto (redes), impresión, o para el control de un determinado dispositivo electrónico.

2.3. Hardware

Por su arquitectura un computador se divide básicamente en:

- dispositivos de entrada
- dispositivos de salida
- dispositivos de procesamiento
- dispositivos de almacenamiento

2.3.1. Dispositivos de Entrada

Son aquellos dispositivos encargados de realizar dos de las tareas computacionales básicas tales como entrada de datos y recepción de comandos o instrucciones.

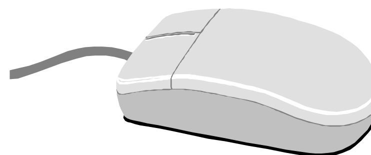
Los principales dispositivos de entrada son:

a. Teclado: Es el dispositivo de entrada mas comúnmente utilizado, el cual fue adoptado de las máquinas de escribir. Los teclados de computador se basan en el estándar de distribución de teclas QWERTY adoptado universalmente con algunas variaciones dependiendo del país. Esta familiar y conocida distribución permite a los usuarios ingresar rápidamente información o comandos al sistema.

b. Dispositivos de Señalamiento: Existen diferentes dispositivos de este tipo que se utilizan para mover el cursor o elemento apuntador sobre la pantalla.

El dispositivo más común es el Mouse o ratón, llamado así porque se desliza sobre el escritorio con su cable o cola conectada al computador. El ratón es utilizado para mover el cursor sobre la pantalla e indicar o seleccionar una porción de la pantalla mediante los botones destinados para tal fin, lo que se traduce en realizar una determinada acción o ejecutar un conjunto de operaciones. Existen dos tipos de ratones; electromecánicos y ópticos. Los electromecánicos poseen una esfera la cual al deslizarse sobre la superficie acciona un dispositivo que convierte el movimiento en una señal eléctrica, mientras que los ópticos proyectan un haz de luz hacia abajo, el cual se refleja sobre una superficie metálica que le sirve de base y determina la dirección de los movimientos.

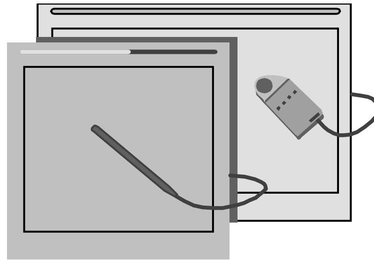
El joystick o palanca de juegos es otro dispositivo de señalamiento el cual es mas comúnmente utilizado en simulación, control y entretenimiento.



c. Dispositivos de Escritura y Dibujo: Existen varios dispositivos similares a un lápiz para el ingreso de información. El lápiz óptico es utilizado para dibujar, escribir o invocar comandos cuando se posiciona sobre una pantalla de características especiales.

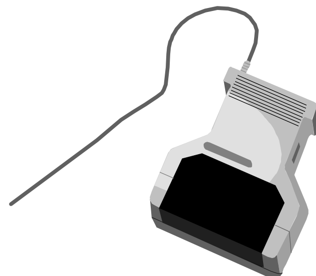
Las pantallas sensitivas (touch screen) se accionan con el dedo y son utilizadas como mecanismos de selección en menús.

Las tabletas digitalizadoras son utilizadas como ratones de desplazamiento relativo sobre una tableta, dedicadas especialmente para la realización de dibujos o planos.

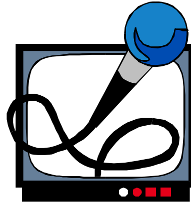


d. Entradas de Vídeo: Imágenes provenientes de cámaras de vídeo, cámaras fotográficas u otros dispositivos pueden ser ingresadas al computador, las cuales se almacenan para su visualización o procesamiento.

e. Entradas de Texto / Gráficas: Una de las tareas más desagradables en el trabajo con computadores es la de digitar información previamente procesada por otros programas u otros medios. Para agilizar este proceso se tienen los *scanners*, dispositivos mediante los cuales se transfiere la información de un medio escrito a la memoria del computador.



f. Entrada de Voz: Aparentemente la forma más fácil y natural de ingresar información al computador es por medio del habla; sin embargo cada persona pronuncia de forma diferente, existen distintos acentos y se realiza a diferente velocidad, lo cual dificulta la interpretación de dicha información, la cual normalmente proviene de micrófonos.



g. Elementos de Realidad Virtual: Se conforma por todo tipo de dispositivos que emulan la realidad para permitir el ingreso de información con base en el entorno que los rodea; sus principales elementos son los cascos, trajes y guantes.

h. Puertos: Son dispositivos electrónicos utilizados para la interconexión del computador con otros equipos, de tal manera que permitan intercambiar información (serial, paralelo, USB).

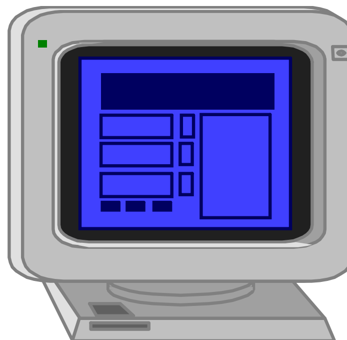
2.3.2. Dispositivos de Salida

Son aquellos encargados de presentar la información o los resultados de la realización de las diferentes tareas que el computador hace.

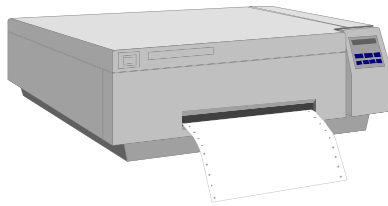
Los principales dispositivos de salida son:

a. Pantallas de Vídeo: Similares a las pantallas de televisión, utilizando tubos de rayos catódicos (CRT), cristal líquido, gas plasma u otros componentes para proyectar las imágenes compuestas de información.

Dependiendo de las características, configuraciones y capacidades existen diferentes tipos de pantallas y tarjetas controladoras: CGA, Hercules, EGA, VGA, SVGA.



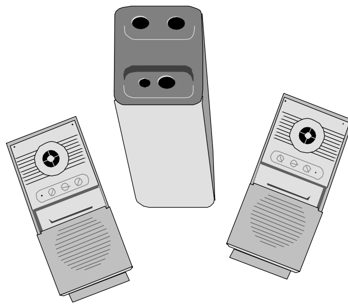
b. Impresoras: Proveen una copia en papel u otro medio, de los resultados procesados en el computador.



Existen impresoras de diferentes tipos:

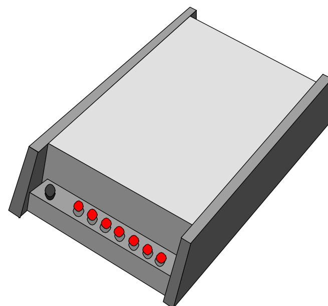
- Matriz de punto
- Láser
- Térmicas
- Ploters o trazadores
- Margarita
- Inyección de tinta
- LED

c. Salidas de Voz: Conocidos como parlantes, permiten la presentación de toda clase de sonidos almacenados o generados en el computador.



d. Elementos de Realidad Virtual: Todo tipo de dispositivos que emulan la realidad para permitir la presentación de información con base en el entorno que los rodea; se compone básicamente de cascos y anteojos especiales.

e. Modems: Dispositivos encargados de modular y demodular las señales digitales que se manejan en los computadores con el fin de permitir su difusión mediante la línea telefónica.



f. Puertos: Son dispositivos electrónicos utilizados para la interconexión del computador con otros equipos, de tal manera que permitan intercambiar información (serial, paralelo, USB).

2.3.3. Dispositivos de Procesamiento

Es la parte del computador encargada del proceso y manipulación de la información; está conformada por:

a. Microprocesador (UCP): La unidad central de proceso es el principal componente del computador, el cual contiene los circuitos que controlan la interpretación y ejecución de las instrucciones que conforman los programas.

Tiene tres componentes básicos:

•Unidad de control: parte encargada de la supervisión y control de las operaciones que se realizan en el microprocesador. Carga los programas, analiza las diferentes operaciones y decide que acción debe realizar con cada una de ellas, apoyándose en los demás dispositivos que conforman el computador.

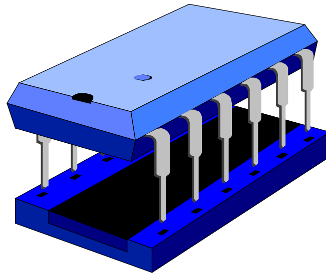
•Unidad aritmético-lógica (ALU): es la encargada de realizar las operaciones lógicas y matemáticas a nivel del procesador.

•Memoria de configuración: se le conoce como memoria de solo lectura y es la encargada de almacenar la información requerida para el funcionamiento del computador, así como las microinstrucciones necesarias para el proceso de configuración.

Existen tres familias de procesadores que operan en la mayoría de micro computadores hoy en día; Intel, Motorola y Power PC; los Intel se encuentran en las líneas de computadores IBM y compatibles que basan su funcionamiento en los sistemas operacionales DOS, Windows, Windows 95, Windows 98 y UNIX, basándose en la serie 80x86, la cual ha tenido numerosas variantes y evoluciones, siendo los mas conocidos: 8086, 8088, 80286, 80386, 80486, Pentium, Pentium MMX, Pentium pro, Pentium II y Pentium III. Mientras que los Motorola equipan las líneas de computadores personales Apple Macintosh, Commodore Amiga, entre otros, con sus propios sistemas operacionales, siendo la serie 68000 la más representativa con sus modelos 68000, 68010, 68020, 68030 y 68040. Por último la familia Power PC con sus serie G3, creada como una alianza entre IBM y Apple ha generado

una nueva serie de procesadores RISC (Reduced Instruction System Code) para sus líneas de computadores personales y de oficina de alto rendimiento.

b. Memoria Principal: La memoria principal es el lugar en el cual el computador almacena la información mientras esta es procesada; se le conoce como memoria RAM (Memoria de Acceso Aleatorio - Random Access Memory). Se mide en bytes de información, donde un byte es la agrupación de 8 bits (unidad mínima de información on-off), es de carácter transitorio ya que es necesario tener el computador encendido para mantener la alimentación de dicha memoria y sostener la información que allí se almacena.

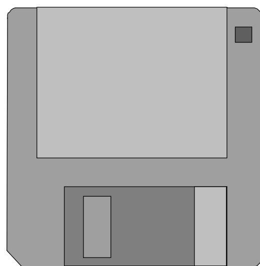


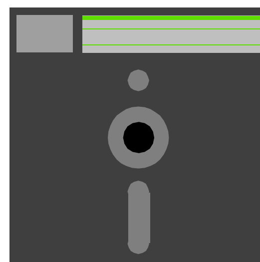
2.3.4. Dispositivos de Almacenamiento

Son los encargados de guardar la información (datos y programas) que se calcula y procesa en el computador. Debido al carácter volátil de la memoria RAM, y sus restricciones en cuanto a capacidad, es importante tener otros mecanismos de almacenamiento que permitan guardar una mayor cantidad de información y que sea independiente de la alimentación eléctrica.

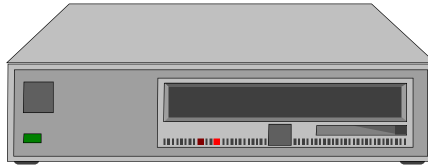
Los dispositivos de almacenamiento se clasifican según el medio que se utilice para guardar la información:

a. Discos Flexibles: conocidos como disquetes se caracterizan por ser removibles y almacenar volúmenes de información que varían entre 360 KBytes hasta 2.8 MBytes; son el medio más utilizado para este fin y permiten entre otras ventajas llevar la información de un lugar a otro y servir como medio de respaldo (backup) de otros medios de almacenamiento. La medida más popular es 3.5 pulgadas, aunque anteriormente se utilizaban 5.125 y 8 pulgadas con diferentes capacidades.

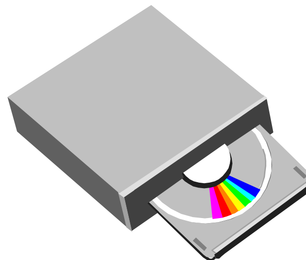




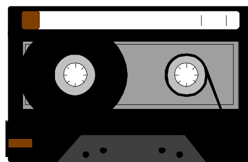
b. Discos Duros: son por excelencia el medio físico mas utilizado para almacenar información, se caracterizan por su velocidad y confiabilidad, permitiendo el almacenamiento de grandes volúmenes de información que van desde unos pocos MBytes hasta varios GBytes, dependiendo de la arquitectura y el modelo.



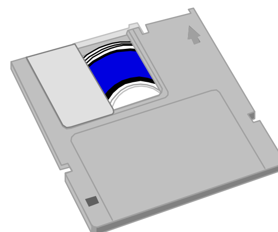
c. Discos Compactos: son discos de almacenamiento que solo permiten grabar la información una vez y luego se convierten en elementos de consulta. Son ideales para la instalación de nuevos programas cuyos volúmenes sean de tamaño considerable. (sistemas operacionales, programas extensos, bibliotecas de datos), su capacidad varía desde 600 Mbytes en adelante.



d. Cintas: son medios de carácter secuencial que permiten almacenar grandes volúmenes de información a manera de respaldo para discos duros u otros medios de almacenamiento. Se caracterizan por ser de baja velocidad pero con un costo muy bajo por unidad almacenada.



e. Discos Ópticos: son medios de caracter óptico que mediante el uso de lectores de láser permiten almacenar desde 680 Mbytes de información.



2.4. Software

El software como se mencionó en el glosario se refiere a los programas que funcionan en el computador, los cuales dependiendo de su función se dividen en:

2.4.1. Software del sistema

El software del sistema comúnmente conocido como sistema operacional es la base para el funcionamiento de todo computador permitiendo la interacción entre los diferentes recursos electromecánicos del hardware, sirviendo como interfaz entre el usuario y el computador. Adicionalmente, es el mecanismo de comunicación entre el software de aplicación y los dispositivos físicos de la máquina.

Los principales sistemas operativos son: UNIX, DOS, Windows (3.1, 95, 98, NT), OS/2, CPM, VMS, MAC/OS, OS/400, LINUX, entre otros. Este tema se analizará con mayor detalle en el capítulo III – Manejo de Computadores -

2.4.2. Software de aplicación

Son todos aquellos programas que pueden ser operados en el computador, independientemente del tipo, función o características que posean. Normalmente se clasifican en:

- Aplicaciones estándar: procesadores de palabra, hojas electrónicas, bases de datos, graficadores.
- Herramientas de trabajo: compiladores, interpretadores, utilidades operativas.
- Programas a la medida: contabilidades, nóminas, controles de producción, inventarios.
- Programas para el entretenimiento: juegos, simuladores.
- Páginas WEB: permiten la difusión y captura de información en las redes.

3. MANEJO DE COMPUTADORES

El manejo de los computadores se restringe básicamente a las tareas que se puedan desempeñar a nivel de sus dispositivos de procesamiento, entrada y salida, siendo el sistema operacional el encargado de controlar dichas acciones.

El sistema operacional está formado por una serie de programas que permiten el funcionamiento del computador mediante la interacción de las diferentes partes que lo conforman (hardware, software y firmware), proporcionando adicionalmente un mecanismo de interfaz entre el computador y el usuario conocido como interpretador de comandos, elemento mediante el cual se le imparten las órdenes o instrucciones al computador para que realice las tareas que el usuario requiere.

Las principales características que aporta un interpretador de comandos al usuario son entre otras:

- visualizar el contenido de las unidades de almacenamiento
- copiar información entre unidades
- borrar información de una unidad
- ejecutar programas o grupos de instrucciones
- visualizar el contenido de un archivo o conjunto de información
- preparar las unidades de almacenamiento para su utilización
- determinar el estado de la máquina y cada uno de sus componentes
- servir como medio para configurar sus componentes

Los sistemas operativos mas comúnmente utilizados y difundidos a lo largo de la historia de los micro computadores son:

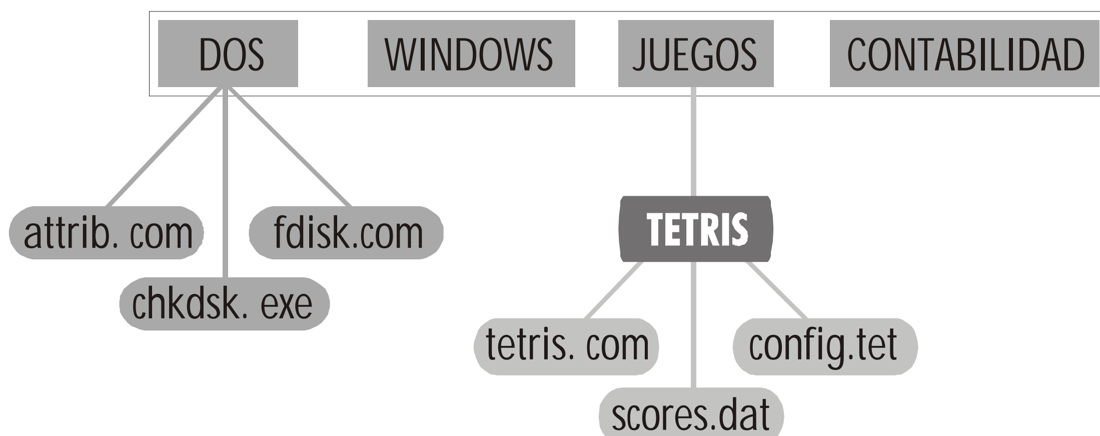
- DOS
- WINDOWS 3.1 / 3.11 (Grupos de Trabajo)
- WINDOWS 95 / 98 / 2000

3.1. DOS

Sigla cuyo significado es disco del sistema operacional (Disk Operating System); es el sistema operacional monousuario y monotarea más utilizado en el mundo, el cual proviene como una evolución del sistema CP/M (programas de control para microprocesadores). Se caracteriza por servir a computadores cuyo procesador sea de la familia Intel 80x86 y Pentium.

El DOS es un sistema que permite almacenar la información de manera jerárquica utilizando el concepto de directorios, los cuales agrupan la información que allí se almacena de acuerdo con las necesidades del usuario. Por esta razón la estructura del sistema de archivos se asemeja a la de un árbol invertido compuesto de archivos y directorios, donde un directorio está formado de archivos y si es necesario de otros directorios, generando una verdadera jerarquía para el almacenamiento de la información.

Esta jerarquía se visualiza gráficamente de la siguiente forma:



Las unidades de almacenamiento externas son representadas por letras (A, B, C, etc.), de tal manera que faciliten el uso de dichas unidades mediante los distintos comandos diseñados para tal fin.

El uso del sistema DOS basa su funcionamiento en el procesador de comandos (programa COMMAND.COM), que se encarga de realizar la interacción entre el usuario y el computador mediante un *prompt* o elemento indicador del estado del procesador de comandos, el cual indica la unidad de disco actual y el directorio en que se encuentra.

Un ejemplo de la forma del *prompt* es la siguiente:

C:\WINDOWS\SYSTEM>_

Indicando que la unidad actual es el disco C: y el directorio en el cual está posicionado es \WINDOWS\SYSTEM; por lo tanto todo comando que se especifique se relacionará directamente con dicho directorio de la unidad C.

El sistema operacional DOS posee dos tipos de comandos básicos, los internos y los externos; los internos son aquellos de uso mas frecuente y que permanecen continuamente en la memoria (residentes incluidos en el interpretador de comandos: COMMAND.COM), mientras que los externos son programas almacenados en los discos del computador (normalmente bajo el directorio C: \DOS) cuyo uso no es tan frecuente.

A nivel del DOS las principales instrucciones o comandos son:

COMANDO	**DESCRIPCIÓN**	**TIPO**
ATTRIB	modifica los atributos de un archivo	Externo
CD	cambia a un determinado directorio de archivos	Interno
CHKDSK	verifica el estado de un disco	Externo
CLS	borra la pantalla	Interno
COPY	copia un archivo o conjunto de archivos de un lugar a otro	Interno
DEFRAG	defragmenta el espacio en un disco	Externo
DEL	borra el contenido de un archivo	Interno
DELTREE	borra recursivamente un directorio y todo lo almacenado en él	Externo
DIR	muestra el contenido de una unidad lógica del computador	Interno
DISKCOPY	duplica el contenido de un disquete en otro	Externo
EDIT	utiliza el editor de archivos del sistema	Externo
FORMAT	prepara un disco nuevo para almacenar información	Externo
LABEL	asigna el nombre o etiqueta a un disco	Externo
MD	crea un directorio de archivos	Interno
MEM	presenta el estado de la memoria	Externo
MOVE	mueve un conjunto de archivos de un directorio a otro	Externo
PRINT	imprime un archivo	Externo
RD	remueve o elimina un directorio de archivos	Interno
REN	renombra un archivo	Interno
SYS	transfiere los archivos del sistema a un disco	Externo
TREE	muestra los directorios de un disco en forma de árbol	Externo
TYPE	muestra el contenido de un archivo	Interno
UNDELETE	recupera archivos previamente borrados	Externo
VOL	muestra el nombre de un disco	Interno
XCOPY	copia archivos por bloques	Externo

La estructura típica de archivos de un computador bajo el sistema DOS presenta la siguiente configuración mostrada mediante el comando DIR:

```
El volumen de la unidad C es EDUARDO
 El número de serie del volumen es 259E-3EE3
 Directorio de C:\

DOS           <DIR>          08/11/98   11:22
COMMAND  COM          56.539 30/09/93    6:20
WINA20   386           9.349 10/03/93    6:00
GMOUSE        <DIR>          31/12/98   10:53
FLTSIM5       <DIR>          02/01/99   11:25
JUEGOS        <DIR>          08/11/98   12:07
BORLANDC      <DIR>          05/12/98   20:20
BC       BAT              26 06/12/98    8:03
WINDOWS       <DIR>          30/12/98    8:10
CONFIG   SYS             388 02/01/99   11:25
AUTOEXEC BAT             179 02/01/99   11:25
TRIDENT  HGI  <DIR>          30/12/98    8:34
TRIDENT  UTL  <DIR>          30/12/98    8:34
XING          <DIR>          30/12/98    8:51
MSOFFICE      <DIR>          30/12/98   19:35
       15 archivo(s)           66.481 bytes
                         402.243.584 bytes libres
```

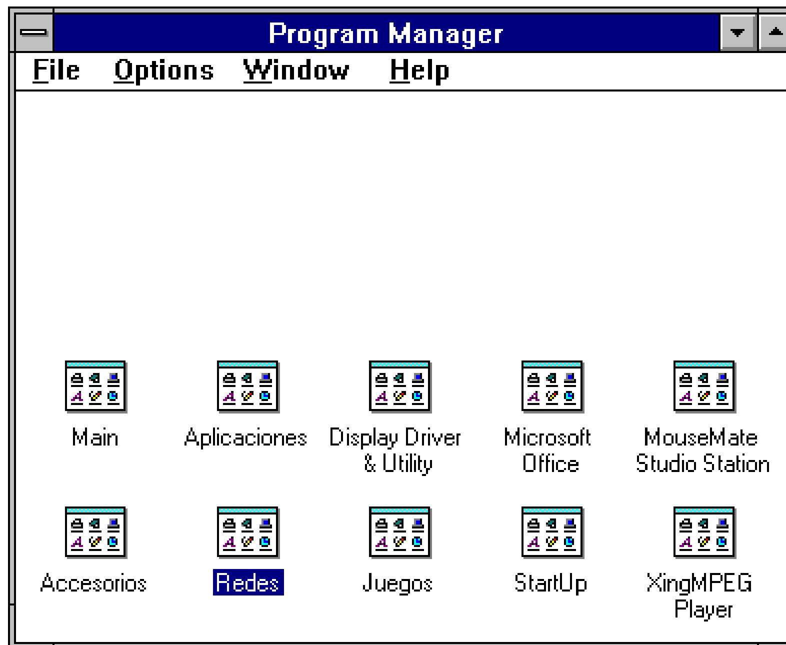
En la distribución de archivos se puede destacar: nombre del archivo o directorio (máximo 8 caracteres), extensión (máximo 3 caracteres), tamaño medido en bytes (los directorios no tienen tamaño y se reconocen por el componente <DIR>), fecha y hora de creación o última modificación. Al final se presenta un sumario o resumen en el cual se especifica la cantidad de archivos listados, el espacio ocupado por los mismos y el espacio disponible en dicha unidad.

3.2. Windows 3.1 / Windows 3.11 (para grupos de trabajo)

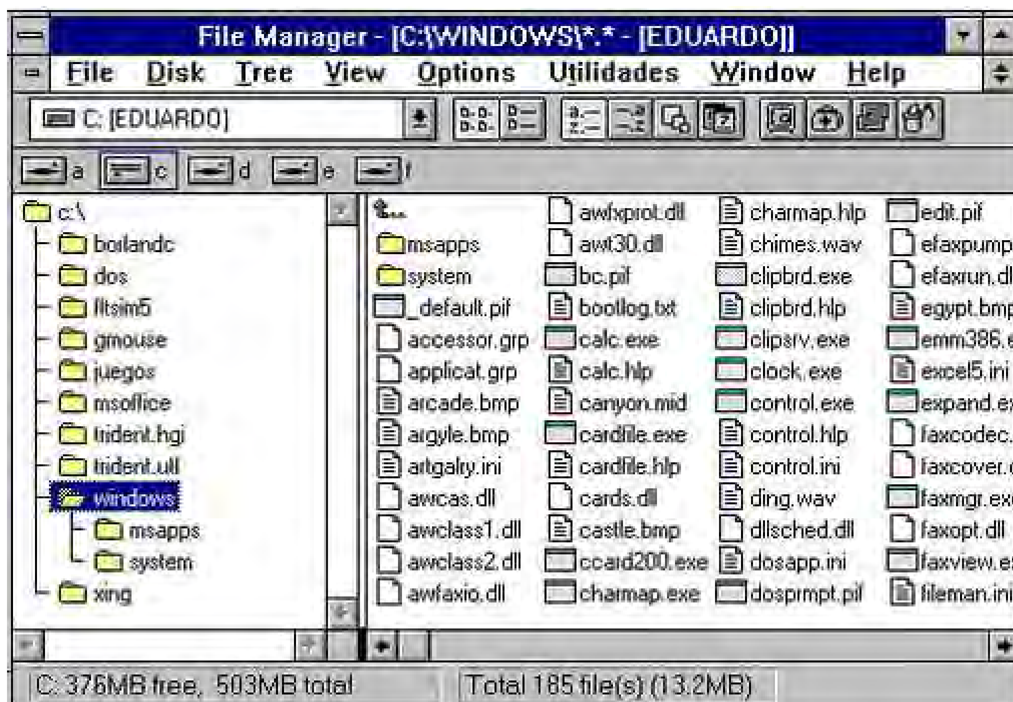
Surgió como un programa para manejar de una forma gráfica el DOS utilizando el mecanismo de ventanas y menús de tipo colgante, que puede ser manejados interactuando con el Mouse o Ratón para facilitar el trabajo; no es un sistema operativo, ya que éste se apoya en el DOS para la realización de todas las labores que desempeña, siendo necesario que éste se encuentre presente para su ejecución.

Los conceptos de organización de archivos se mantienen, pero se adiciona el concepto de grupos de programas; los cuales permiten almacenar conjuntos de programas según su funcionalidad.

La configuración básica de la pantalla en Windows 3.11 es la siguiente:



En las versiones 3.0, 3.1 y para grupos existe un icono denominado Manejador de Archivos (File Manager) el cual se encuentra situado en la carpeta Principal (Main), encargado de realizar la mayor parte de funciones que proveía el sistema operacional DOS, el cual tiene la siguiente apariencia:



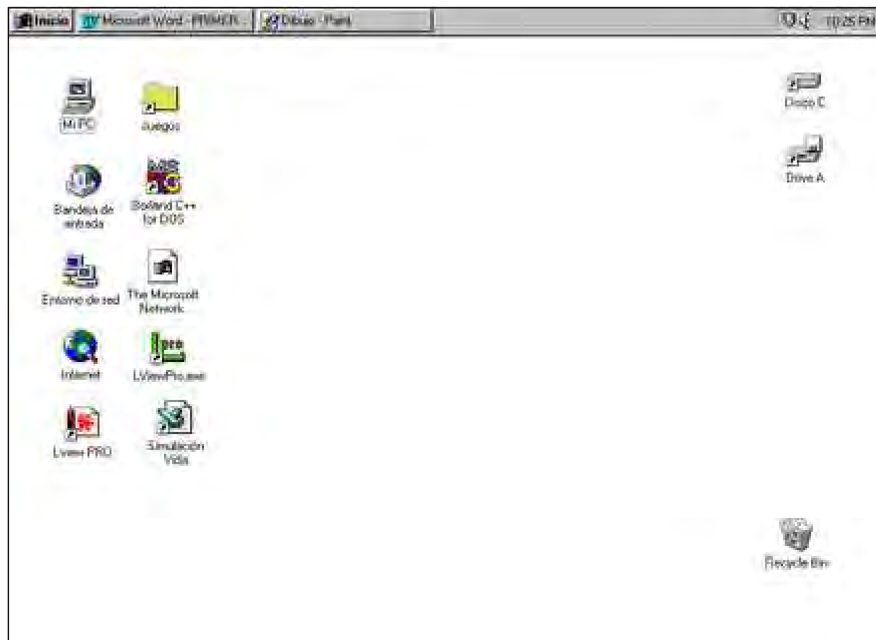
La versión 3.11 conocida como Windows para Grupos de Trabajo es la actualización de la 3.1, permite el manejo y acceso a redes, dando la posibilidad de interconectar computadores, mediante el uso de tarjetas de red y protocolos Ethernet; y permite compartir recursos tales como: unidades de disco (discos duros, disquetes, CD-ROM, DVD) e impresoras.

3.3. Windows 95 / 98

Surgió como una evolución del Windows 3.11 y brinda la posibilidad de manejar de una forma mas natural los diferentes mecanismos de ventanas y menús de tipo colgante, llegando a convertirse en un verdadero sistema operacional.

La clave de su manejo se basa en la utilización de ventanas gráficas y el uso del Mouse como dispositivo accionador de los diferentes comandos. La representación de los archivos y carpetas de archivos se realiza mediante iconos que se asocian a cada uno de ellos para facilitar su manejo.

La configuración típica de la pantalla o escritorio en Windows 95 / 98 es la siguiente:



Se destaca la barra de inicio, los diferentes iconos para ejecutar los programas y la basura o dispositivo para el borrado de archivos.

En esta versión existe un programa denominado Explorador, que de forma similar al Manejador de Archivos pero bastante mas elaborado permite un manejo completamente gráfico, sencillo y práctico del sistema operacional. Estas operaciones incluyen: copia y formato de discos, renombramiento, borrado y copia de archivos, visualización de directorios, ejecución de programas, etc.

La presentación del Explorador de Windows es la siguiente:

3.4. UNIX

Sistema operacional multiusuario desarrollado en 1968 por Ken Thompson en los laboratorios Bell, el cual permite el trabajo de varios usuarios sobre el mismo computador haciendo uso de terminales. Por estar escrito en lenguaje C, posee una gran portabilidad que le permite funcionar en computadores que van desde un PC hasta poderosos super computadores.

El intérprete de comandos de UNIX (denominado shell) se presenta normalmente con el caracter $ y admite entre otros los siguientes comandos:

COMANDO	**DESCRIPCIÓN**
cat	Muestra el contenido de un archivo.
cc	Compilador del lenguaje C.
cd	Cambia a un directorio de archivos.
cp	Copia un archivo o conjunto de archivos de un lugar a otro.
gcc	Compilador del lenguaje C++ (GNU)
rm	Borra el contenido de un archivo.
ls	Muestra el contenido de una unidad lógica del computador.
md	Crea un directorio de archivos.
mv	Renombra un archivo.
passwd	Permite cambiar la clave de acceso
ps	Muestra el estado de los procesos.
pwd	Directorio de trabajo del usuario.
rd	Remueve un directorio de archivos.
vi	Editor de archivos del sistema.
who	Muestra la lista de usuarios actuales.

Existen muchas versiones de UNIX, desarrolladas por otras empresas entre las cuales vale la pena destacar:

XENIX	versión reducida desarrollada por SCO
AIX	versión desarrollada por la IBM
SOLARIX	versión desarrollada por SUN Microsystem
HP-UX	versión de Hewlett Packard
IRIX	versión de Silicon Graphics
ULTRIX	versión propia de la DEC
LINUX	versión académica de libre uso
UNOS	versión propia de la Charles River Data System

3.5. Otros sistemas operacionales

Existen otros sistemas operacionales de bastante cobertura en el medio, tales como el MAC/OS y el OS/2, los cuales se caracterizan por ser sistemas gráficos multitarea desarrollados por Apple y la alianza Microsoft e IBM respectivamente. Su diseño se centró en el manejo de los ambientes gráficos y tienen excelentes características que facilitan su conexión a cualquier tipo de red.

Otro sistema de bastante uso en la actualidad, es el OS/400, el cual tiene como función operar los computadores IBM de la serie AS/400 (Aplication System 400), el cual basa su funcionamiento en una serie de menús orientados según las necesidades de cada uno de los usuarios.

SEGUNDA PARTE

CONCEPTOS BÁSICOS DE PROGRAMACIÓN

4. SOLUCIÓN DE PROBLEMAS

El principal objetivo de los computadores es servir como herramienta para que los usuarios puedan resolver sus problemas de una manera ágil, eficiente y segura; los computadores no realizan tareas milagrosas ni especiales, sus principales características son la velocidad de proceso y la capacidad de almacenamiento, donde toda tarea o programa que éste realiza puede ser ejecutada manualmente por un ser humano aunque esta le tarde muchos días o tal vez años.

Los computadores resuelven los problemas mediante conjuntos de instrucciones explícitas y no ambiguas expresadas bajo un lenguaje de programación denominados programas. Para escribir un programa que resuelva un problema o ejecute una determinada tarea, es necesario conocer muy bien el esquema de la solución, a lo cual se llega después de realizar el debido análisis y aplicar estrategias de solución que permitan encontrar elementos que resuelvan el problema planteado.

La solución a un determinado problema es un proceso creativo en el cual la mecanización y sistematización pueden no ser útiles ya que no existen métodos universales para la resolución de problemas; estrategias diferentes funcionan para personas y problemas diferentes.

4.1. Estrategias para alcanzar la solución de un problema

Existen muchos métodos o estrategias para llegar a al solución de un problema, como también existen varias soluciones para el mismo; siendo necesario conocer diferentes estrategias para buscar dichas soluciones, ya que cada una se acomoda mas a determinados problemas sin tener una regla general para su aplicación.

Antes de proceder a la solución de cualquier problema, independientemente de la estrategia que se utilice, es importante entender completamente el problema y hacer un pequeño análisis de lo que se quiere resolver y de la información que se tiene, con el fin de solucionar lo que realmente se necesita. Muchas veces se resuelven los problemas que no se requieren o se hacen operaciones innecesarias, únicamente debido a la falta de entendimiento del mismo.

Ej. : Una persona que viaja hacia Manizales se encuentra con un amigo, el cual tenia siete amigas; cada amiga tenía tres sacos; en cada saco había dos gatas y cada gata tenía cuatro gaticos. ¿Cuántos iban para Manizales?

Normalmente, cuando no se lee detalladamente el problema, se plantea la siguiente solución:

En cada saco hay dos gatas y cada una con cuatro gaticos para un total de diez (ocho gaticos + dos gatas); cada amiga tenía tres sacos, por lo tanto llevaba treinta gatos. Entre las siete amigas llevaban doscientos diez gatos, mas las siete amigas, mas el amigo, mas el viajero, generan en total:

210 + 7 + 1 + 1 = 219

Por lo tanto viajaban doscientos diecinueve, entre personas y gatos.

¿Dónde se especificó en el enunciado del problema que el amigo, sus amigas y los gatos viajaban para Manizales?. Únicamente se mencionó que se los encontró, por lo tanto la respuesta correcta es uno, el viajero.

Este tipo de planteamientos es muy común, por lo cual es importante analizar y entender completamente el problema que se desea solucionar y determinar la información relevante para dicha solución.

Entre las estrategias más comunes vale la pena estudiar y tener en cuenta: Particularización y Generalización, Dividir y Conquistar, Ensayo y Error.

4.2. Particularización y generalización

La idea central de esta estrategia es tratar de encontrar la solución al problema planteado, buscando elementos comunes y/o característicos en la solución de ejemplos específicos, para posteriormente dar una solución generalizada que abarque todos los casos posibles.

Para llegar a la solución se plantean los siguientes pasos:

• **Reflexionar:** La solución de un problema solo es posible una vez que se ha entendido lo que debe solucionarse, buscando encontrar que es lo que hay que hacer y no cómo se debe hacer. Es importante la formulación de preguntas tales como: ¿Qué información hay?, ¿Qué quiero saber?, ¿Qué puedo utilizar?

• **Particularizar:** Una buena manera de comenzar a buscar la solución es escogiendo ejemplos específicos y tratar de resolverlos determinando el mecanismo de solución de ese problema en particular, planteándolo en términos o pasos generales.

• **Generalizar:** Luego de la aproximación inicial al problema se busca una solución general agrupando bajo un mismo esquema todas las soluciones particulares.

• **Verificar:** Luego de obtener una solución general, se realizan pruebas con diferentes casos o ejemplos comprobando que efectivamente sea una solución satisfactoria al problema propuesto, de lo contrario se debe reiniciar el proceso con base en aquellos casos para los cuales no funcionó la solución planteada.

• **Reutilizar:** Cuando se tiene un determinado grado de experiencia, se puede buscar similitudes entre el problema planteado y otros problemas resueltos previamente, de tal manera que con pequeñas modificaciones se pueda adaptar a las necesidades del nuevo problema, encontrando así una solución satisfactoria.

Ej. : En cierto almacén por cada venta ofrecen un descuento del 20%, pero al mismo tiempo se debe pagar un impuesto del 15%. ¿Qué es mejor, Calcular primero el impuesto o el descuento?

Se procede a particularizar, haciendo una prueba determinada. Suponiendo ventas por $1.000 pesos:

a) Se calcula primero el descuento y luego el impuesto

valor venta		1000
valor descuento	1000 * 20% =	- 200
subtotal		800
valor impuesto	800 * 15% =	+ 120
total		920

b) Se calcula primero el impuesto y luego el descuento

valor venta		1000
valor impuesto	1000 * 15% =	+ 150
subtotal		1150
valor descuento	1150 * 20% =	- 230
total		920

c) Se deduce para cada caso lo siguiente:

valor a pagar = valor venta * 0.8 * 1.15

valor a pagar = valor venta * 1.15 * 0.8

Teniendo en cuenta que la multiplicación es conmutativa, se generaliza que para cualquier venta con diferentes factores de descuento e impuesto, sin importar el orden que éstos se calculen, el resultado será el mismo.

Ej. : Los números capicúas son aquellos que se leen igual de izquierda a derecha que de derecha a izquierda, tales como 12321, 1221, 3467643, 343. Determinar si es cierto que todos los números capicúas de cuatro cifras son divisibles exactamente por 11.

se hacen algunos ensayos en forma aleatoria:

1221 / 11 = 111
3003 / 11 = 273
2112 / 11 = 192
5885 / 11 = 535

aparentemente la afirmación es correcta, pero se debe buscar una forma sistemática para asegurar que todos los capicúas de cuatro cifras cumplen con dicha regla.

¿Cuál es el capicúa de cuatro cifras más pequeño?
1001 / 11 = 91

¿Cuál es el siguiente?
1111 / 11 = 101

¿Cuál sigue?
1221 / 11 = 111

¿Cuál sigue?
1331 / 11 = 121

Parece que funciona, pero ¿porque?

dados los primeros capicúas se tiene la siguiente relación basada en la diferencia entre uno y otro:

1001	1111	1221	1331	1441	1551	1661

110	110	110	110	110	110

Cómo se aprecia hay una diferencia de 110 entre un capicúa y el siguiente, pero, ¿se cumple para todos?

¿Si se toma el 1991 que pasa con el siguiente?

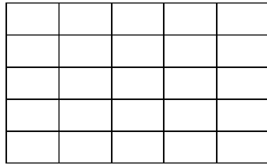
1991	2002	2112	2222	2332	2442	2552	2662	2772	2882	2992	3003

11	110	110	110	110	110	110	110	110	110	11

Se puede apreciar que la diferencia entre dos capicúas cuando se cambia la cifra de los miles es de 11, por lo tanto todo número capicúa de cuatro cifras se consigue sumando múltiplos de 11 o 110 al primer capicúa de cuatro cifras que es divisible por 11; y como 11 y 110 son divisibles por 11, entonces todos los capicúas de cuatro cifras son divisibles por 11.

Ej. : ¿Cuántos cuadros de cualquier tamaño se pueden encontrar en un tablero de N x N?

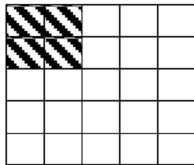
Se particulariza con un tablero de un tamaño pequeño como 5, entonces:

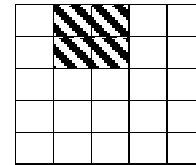


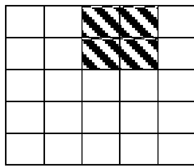
Se debe determinar cuantos cuadrados hay de 1x1, 2x2, 3x3, 4x4, 5x5 y calcular la suma de cada uno de ellos.

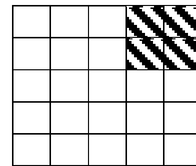
De lado 1 hay: 5*5 = 25 cuadrados de lado 1

De lado 2 hay:

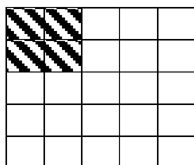


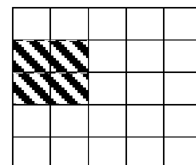


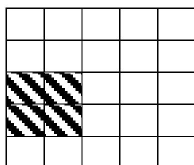


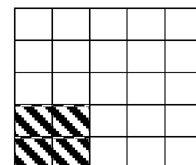


Como se puede ver se pueden disponer 4 cubos de 4x4 por cada fila del tablero de 5x5; por lo tanto en las columnas pasa algo similar así:



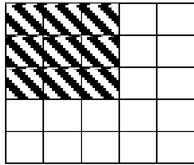


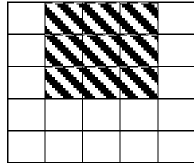


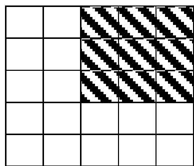


Entonces en un tablero de 5x5 se pueden encontrar 4 * 4 = 16 cuadrados de lado 2.

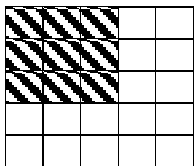
Cuadrados de lado 3:

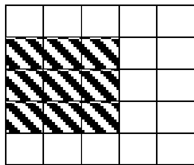


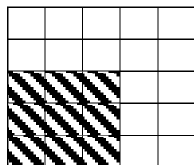




Como se puede apreciar, por cada fila se localizan 3 cuadrados de lado 3 en un tablero de 5 x 5; análogamente para las columnas así:

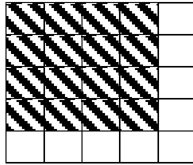


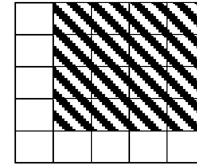




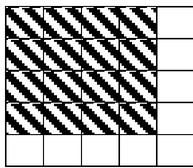
Entonces en un tablero de 5x5 se pueden encontrar 3 * 3 = 9 cuadrados de lado 3.

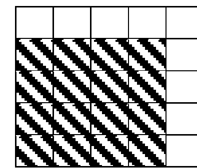
Cuadrados de lado 4:





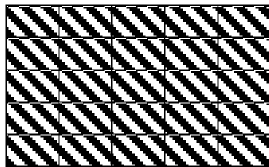
Como se puede apreciar, por cada fila se localizan 2 cuadrados de lado 4 en un tablero de 5 x 5; análogamente para las columnas así:





Entonces en un tablero de 5x5 se pueden encontrar 2 * 2 = 4 cuadrados de lado 4.

Cuadrados de lado 5:



En un tablero de 5x5 se puede encontrar 1 * 1 = 1 cuadrado de lado 5.

En resumen se tiene el siguiente cuadro:

Lado	1x1	2x2	3x3	4x4	5x5
Cantidad	25	16	9	4	1

Dando como suma = 25 + 16 + 9 + 4 + 1 = 55

Generalizando se tiene:

Lado 1 = 25 que es igual a 5*5

Lado 2 = 16 que es igual a 4*4
Lado 3 = 9 que es igual a 3*3
Lado 4 = 4 que es igual a 2*2
Lado 5 = 1 que es igual a 1*1

Buscando la forma general teniendo en cuenta que es un tablero de 5 x 5, en términos de la longitud del lado se puede deducir:

Lado k = (5 – k + 1) * (5 – k + 1)

Donde el resultado es:

$$\sum_{k=1}^{5}(5-k+1)^2$$

Generalizando para un tablero de N x N, la cantidad de cuadrados de cualquier lado es:

$$\sum_{k=1}^{N}(N-k+1)^2 = \sum_{k=1}^{N} k^2$$

Ej.: Para enumerar las páginas de un libro se han empleado 39.994 dígitos o cifras. ¿Cuántas páginas tiene el libro?

Se particulariza para cada uno de los subproblemas:

para las primeras 9 páginas se utiliza un dígito por cada una.
9 x 1 = 9

para las siguientes 90 páginas (de la 10 a la 99) se utilizaron dos dígitos por cada una.
90 x 2 = 180

para las siguientes 900 páginas (de la 100 a la 999) se utilizaron tres dígitos por cada una.
900 x 3 = 2700

de esta manera se pude establecer una regla general de tal manera que se cumpla:

Páginas	1-9	10-99	100-999	1.000-9.999	10.000-99.999
Valor	9 x 1	90 x 2	900 x 3	9000 x 4	90.000 x 5
Fórmula	$9 \times 10^0 \times 1$	$9 \times 10^1 \times 2$	$9 \times 10^2 \times 3$	$9 \times 10^3 \times 4$	$9 \times 10^4 \times 5$
Cifras	9	180	2.700	36.000	450.000
Acumulado	9	189	2.889	38.889	488.889

La fórmula general de cantidad de cifras es: $9 * 10^{(N-1)} * N$

Como se aprecia en el acumulado, el número de páginas del libro está entre 10.000 y 999.999, por lo tanto es necesario determinar cuantas cifras faltan por ser utilizadas luego de las primeras 9.999 páginas y calcular el remanente.

para las primeras 9.999 páginas se han utilizado 38.889 cifras, por lo tanto faltan por utilizar:

total cifras:		39.994
utilizadas:	-	38.889
remanente:		1.105

Teniendo en cuenta la fórmula encontrada, es necesario determinar si con 450.000 cifras se numeran 90.000 páginas de 5 dígitos cada una, con 1.105 cifras, ¿cuántas páginas se pueden numerar?

por regla de tres simple:

cifras	450.000	1.105
páginas	90.000	X

$$X = \frac{90.000 * 1.105}{450.000} = 221 \text{ páginas}$$

o simplemente se dividen las 1.105 cifras restantes para enumerar páginas de cinco dígitos entre cinco, obteniendo el valor de 221 páginas.

Entonces, el resultado es: 9.999 páginas enumeradas con 38.889 cifras sumado a 221 páginas enumeradas con 1.105 cifras para un total de 10.220 páginas.

Ej.: Según dice la historia, el juego del ajedrez fue inventado en la India, y cierto rey quiso recompensar a su inventor, el cual le hizo la siguiente propuesta: *"dame un grano de trigo por la primer casilla del tablero, dos por la segunda, cuatro por la tercera, ocho por la cuarta y así sucesivamente para cada una de las 64 casillas del tablero"*. ¿Cuántos granos de trigo conforman la recompensa?

particularizando la solución del problema se tiene:

para la primer casilla, un grano de trigo, para la segunda dos, para la tercera cuatro, para la quinta ocho, organizados de la siguiente manera:

Casilla	1	2	3	4	5	6	7
Fórmula	2^0	2^1	2^2	2^3	2^4	2^5	2^6
Granos	1	2	4	8	16	32	64

Generalizando para la casilla N se tiene: 2^{N-1}

de esta manera la cantidad total de granos de trigo que conforman la recompensa se obtiene mediante la siguiente expresión:

$$\sum_{n=1}^{64} 2^{n-1}$$

lo que produce como resultado la suma de dieciocho trillones, cuatrocientos cuarenta y seis mil setecientos cuarenta y cuatro billones, setenta y tres mil setecientos nueve millones, quinientos cincuenta y un mil seiscientos quince.

Ej. : Dada una tira de papel de una longitud determinada; si se dobla en la mitad se obtienen dos partes; si se dobla nuevamente a la mitad se obtienen cuatro partes, si se repite la acción 3, 4, 5 o más veces, ¿cuántas partes se consiguen? es necesario generalizar para un valor N cualquiera de dobleces.

0 dobleces, 1 parte; 1 doblez, 2 partes; 2 dobleces, 4 partes

Dobleces	0	1	2	3	4	5	6
Fórmula	2^0	2^1	2^2	2^3	2^4	2^5	2^6
Partes	1	2	4	8	16	32	64

generalizando para un doblez N cualquiera se obtiene la siguiente fórmula:

El doblez N genera 2^N partes.

Ej. : Se le propone a una persona que piense y anote un número cualquiera de tres cifras que no termine en cero y las cifras no sean todas iguales, y le pide que ponga debajo las tres cifras en orden contrario. Luego debe restar el número menor del mayor, y la diferencia obtenida (en tres cifras) se suma con dicha diferencia escrita en orden inverso. Sin mas preguntas, ¿Cuál es el número?

particularizando con el número 467, se realizan las siguientes operaciones:
el inverso es 764, por lo tanto:
764 – 467 = 297
297 + 792 = 1089

particularizando con otro número 321, se realizan las siguientes operaciones:
el inverso es 123, por lo tanto:
321 – 123 = 198
198 + 891 = 1089

ahora con el número 544, se realizan las siguientes operaciones:
el inverso es 445, por lo tanto:

544 – 445 = 099
099 + 990 = 1089

parece que funciona con todos, ahora se busca la regla general:

si un número tiene tres cifras a, b y c.

El número es: $100a + 10b + 1c$
El número con las cifras invertidas es: $+ 100c + 10b + 1a$
La diferencia entre los dos es: $99a - 99c$

Aplicando algunas transformaciones matemáticas buscando expresar el número resultante en términos de centenas, decenas y unidades: $99a - 99c$

$$= 99(a - c)$$
$$= 100(a - c) - (a - c)$$
$$= 100(a - c) - 100 + 100 - 10 + 10 - a + c$$
$$= 100(a - c - 1) + 90 + (10 - a + c)$$
$$= 100(a - c - 1) + 10(9) + 1(10 - a + c)$$

el número invertido es: $100(10 - a + c) + 10(9) + 1(a - c - 1)$

sumando las dos expresiones:

$$\begin{array}{rl} & 100(a - c - 1) + 10(9) + 1(10 - a + c) \\ + & 100(10 - a + c) + 10(9) + 1(a - c - 1) \\ \hline & 1089 \end{array}$$

Se obtiene entonces como resultado que cualquier número de tres cifras en la cual la última sea diferente de cero, se invierta y reste el mayor del menor, y luego el resultado se sume con su inverso, siempre dará como resultado: 1089.

4.3. Dividir y conquistar

El esquema de solución de esta estrategia se basa en descomponer un problema en dos o más subproblemas de complejidad menor al problema inicial, y así sucesivamente descomponer cada subproblema hasta llegar a problemas de solución simple, trivial o previamente conocida. Una vez encontradas las soluciones a todos los subproblemas en los cuales se descompuso el problema inicial, se procede a integrar cada una de las soluciones parciales de tal manera que permitan llegar a la solución total del problema.

Cuando un problema se divide en dos o más subproblemas, no implica que cada uno de ellos se deba resolver bajo el mismo método, siendo posible aplicar otro esquema de solución de tal forma que solucione el problema.

Ej. : Encuentre una solución para calcular las posibles combinaciones de los números o elementos de un conjunto. Conocido como el problema del combinatorio.

La primer descomposición es la de aplicar la solución matemática generada mediante la fórmula:

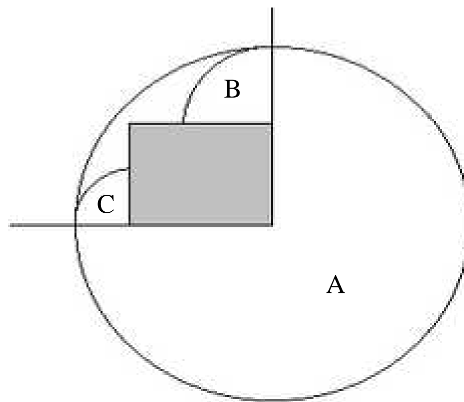
$$\binom{N}{K} = \frac{N!}{(N-K)! * K!}$$

Se procede a solucionar cada una de las operaciones allí planteadas de forma independiente y después se unen las soluciones encontradas, así:

Se debe calcular el factorial de los valores N, (N - K) y K respectivamente. Una vez obtenidos dichos resultados, se procede a dividir el resultado de N! por la multiplicación de los factoriales de (N-K) y de K, obteniendo la solución deseada. Si no se conoce la solución de un factorial, éste se calcula mediante la multiplicatoria de todos los números enteros que hay desde uno hasta el número al cual se le desea encontrar el factorial.

Ej. : Una cabra está atada con una cuerda de 6 mt. de longitud a la esquina externa de una casa que mide 4 mt. de largo por 5 mt. de ancho, la cual se encuentra totalmente rodeada por un campo de hierba; ¿En que área puede pastar la cabra?

se analiza el problema desde el punto de vista gráfico, de tal manera que se logre su comprensión y entendimiento:



la solución del problema se puede descomponer o dividir en tres subproblemas de complejidad menor, tales como calcular el área de las secciones A, B y C; una vez calculadas dichas áreas se procede a integrar las tres soluciones en una sola dando como resultado la solución del problema total.

solución problema = solución A + solución B + solución C

donde las soluciones de los problemas A, B y C se reducen a encontrar las áreas de cada una de las partes que le corresponde, así:

el área de A es: 3 / 4 del área del círculo de radio 6 (longitud de la cuerda)

el área de B es: 1 / 4 del área del círculo de radio 2 (6 del largo de la cuerda menos 4 del largo de la casa)

el área de C es 1 / 4 del área del círculo de radio 1 (6 del largo de la cuerda menos 5 del ancho de la casa)

teniendo en cuenta que el área de un círculo se calcula mediante la fórmula:

área = π * radio ^{2}

entonces:

el área de A = $3 / 4 * \pi * 6^2 = 84.82$ mt.^{2}
el área de B = $1 / 4 * \pi * 2^2 = 3.14$ mt.^{2}
el área de C = $1 / 4 * \pi * 1^2 = 0.78$ mt.^{2}

donde la respuesta total es la suma de las tres respuestas parciales así:

área total = 84.82 + 3.14 + 0.78 = 88.74 mt.^{2}

Ej. : ¿Cómo se pueden verter 6 litros de un determinado líquido que fluye de una llave sobre una tina con capacidad de 100 litros, si se tienen otras dos tinas de 4 y 9 litros respectivamente?

teniendo en cuenta el siguiente estado inicial:

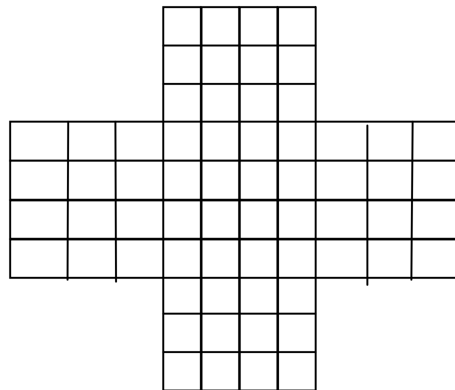
(0,9) (0,4) (0,100)

cuyo significado es la capacidad utilizada de cada una de las tinas(capacidad utilizada, capacidad máxima), las cuales se encuentran inicialmente vacías. (0 litros cada una), se realizan los siguientes pasos:

a)	lleno la tina de 9 desde el grifo	(9,9)	(0,4)	(0,100)
b)	lleno la tina de 4 desde la tina de 9	(0,9)	(4,4)	(5,100)
d)	boto el contenido de la tina de 4	(0,9)	(0,4)	(5,100)
e)	lleno la tina de 9 desde el grifo	(9,9)	(0,4)	(5,100)
f)	lleno la tina de 4 desde la tina de 9	(5,9)	(4,4)	(5,100)
g)	boto el contenido de la tina de 4	(5,9)	(0,4)	(5,100)
h)	lleno la tina de 4 desde la tina de 9	(1,9)	(4,4)	(5,100)
i)	paso el contenido de la tina de 9 a la de 100	(0,9)	(4,4)	(6,100)

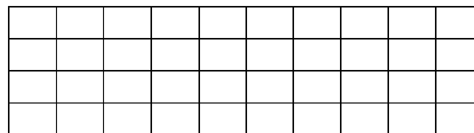
de esta forma se logra verter 6 litros en una tina, utilizando únicamente dos tinas adicionales con capacidades de 4 y 9 litros respectivamente.

Ej.: Cuantos cuadrados de diferente lado se encuentran en la siguiente figura:

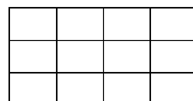


El problema se divide en tres subproblemas de menor orden, tales como:

a) Calcular el número de cuadros en un rectángulo de 10 x 4



b) Calcular el número de cuadros en un rectángulo de 4 x 3



c) Calcular el número de cuadros que forman parte de la intersección de los rectángulos de 10 x 4, y 4 x 3.

Una vez calculadas las soluciones a), b) y c), se procede a determinar la solución total, la cual está dada por la siguiente fórmula:

solución total = solución de a) + dos veces la solución de b) + dos veces la solución de c).

Estas soluciones se plantearon previamente en particulaP' rizar y generalizar, dando como resultados lo siguiente:

Solución a):

Dimensión	1 x 1	2 x 2	3 x 3	4 x 4
Cantidad	40	27	16	7

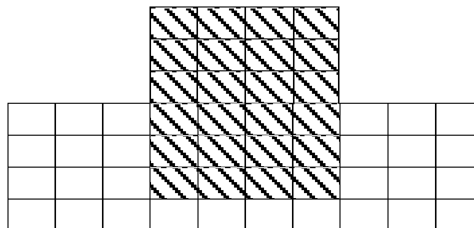
Total cuadros en parte a) es: 40 + 27 + 16 + 7 = 90

Solución b):

Dimensión	1 x 1	2 x 2	3 x 3
Cantidad	12	6	2

Total cuadros en parte b) es: 12 + 6 + 2 = 20

Solución c):



Se debe determinar cuantos cuadros de lado 2, 3 y 4 se encuentran en la intersección de las zonas de los subproblemas a) y b) (parte en cada una de las soluciones)

Dimensión	2 x 2	3 x 3	4 x 4
Cantidad	3	4	3

Total de cuadros en la parte c) es: 3 + 4 + 3 = 10

entonces, la respuesta total del problema es la integración de las respuestas generando la siguiente solución:

solución de a) = 90
solución de b) = 20
solución de c) = 10

La respuesta es: 90 + 2 (20) + 2 (10) = 150 cuadrados.

Ej. : Calcular la suma de los valores almacenados en la siguiente figura:

1	2	3	4	5
2	2	3	4	5
3	3	3	4	5
4	4	4	4	5
5	5	5	5	5

Se procede a dividir el problema en cuantos tipos de valores hay:

Valor	1	2	3	4	5
Cantidad	1	3	5	7	9

Se multiplican y suman las respuestas, de tal manera que la solución es:

= (1 * 1) + (2 * 3) + (3 * 5) + (4 * 7) + (5 * 9)
= 1 + 6 + 15 + 28 + 45
= 95

4.4. Ensayo y error

Cuando no es posible aplicar una técnica sistemática para solucionar algunos problemas, existe una forma de intentar resolverlos mediante el ensayo y el error; esta técnica se basa en aprovechar las diferentes heurísticas, información del problema, soluciones similares, los hechos, y la lógica con el fin de "lanzar" una posible solución y tratar de acomodarla al problema planteado.

Ej.: Dado el siguiente cuadro conformado por nueve casillas en filas y columnas de 3 x 3, acomodar los dígitos de 1 al 9 sin repetir, de tal manera que la suma de cada fila y columna de cómo resultado 15.

Como se puede apreciar no es posible aplicar algún tipo de metodología como particularizar y generalizar o dividir y conquistar, siendo necesario hacer uso de la lógica y algunos trucos matemáticos de tal manera que se pueda plantear alguna solución o ensayando y descartando valores en las diferentes casillas hasta lograr el objetivo propuesto:

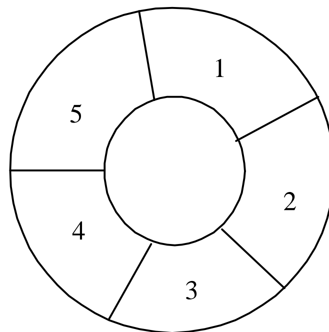
			15
8	3	4	**15**
1	5	9	**15**
6	7	2	**15**
15	**15**	**15**	**15**

Ej. : Cinco señoras comen en una mesa redonda, bajo las siguientes reglas:

a) La Sra. García está sentada entre la Sra. López y la Sra. Martínez.
b) Elena está sentada entre catalina y la Sra. Pérez.
c) La Sra. López esta sentada entre Elena y Alicia.
d) Catalina y Doris son hermanas.
e) Isabel está sentada con la Sra. Gómez a su izquierda y la Sra. Martínez a su derecha.

Determine el nombre y apellido de cada señora, así como su ubicación en la mesa.

aprovechando la información suministrada se comienzan a verificar las diferentes reglas, tratando de encontrar heurísticas que determinen hechos reales con los cuales se pueda iniciar.



aplicando la regla e) que permite asegurar la información se obtiene

PUESTO	1	2	3	4	5
NOMBRE		Isabel			
APELLIDO	Gómez		Martínez		

la regla a) permite ubicar a la Sra. García en el puesto 4 dadas las condiciones anteriores:

PUESTO	1	2	3	4	5
NOMBRE		Isabel			
APELLIDO	Gómez		Martínez	García	López

la regla b) tiene dos posibilidades, ya que el apellido de Isabel es de Pérez (el único que queda), y las posibilidades de ubicar a Elena son en el puesto 1 o el puesto 3; se debe ensayar uno como el puesto 1, quedando así:

PUESTO	1	2	3	4	5
NOMBRE	Elena	Isabel			Catalina
APELLIDO	Gómez	Pérez	Martínez	García	López

aplicando la regla c) se obtiene:

PUESTO	1	2	3	4	5
NOMBRE	Elena	Isabel		Alicia	Catalina
APELLIDO	Gómez	Pérez	Martínez	García	López

y aplicando la regla resultante d), completa el cuadro así:

PUESTO	1	2	3	4	5
NOMBRE	Elena	Isabel	Doris	Alicia	Catalina
APELLIDO	Gómez	Pérez	Martínez	García	López

Ej.: Dadas cinco casas, cada una de color diferente habitadas por personas de diferente nacionalidad, bebida, marca de cigarrillo y víctima, determinar los datos de cada uno basados en las siguientes reglas:

a) El inglés vive en la casa roja
b) El español mató a Rogelio
c) En la casa verde se toma café
d) El ucraniano toma té
e) La casa verde está inmediatamente a la derecha de la casa blanca
f) El que fuma winston mató a Cenelia

g) En la casa amarilla se fuma pielroja
h) En la casa del medio se toma leche
i) El noruego vive en la primera casa de la izquierda
j) El hombre que fuma royal vive en la casa contigua a la del hombre que mató a Foción
k) Pielroja se fuma en la casa contigua a donde se mató a Hortensia
l) El que fuma lucky toma jugo de naranja
m) El japonés fuma president
n) El noruego vive en la casa contigua a la casa azul
o) Otra víctima es Zenón
p) La otra bebida es agua

Mediante la estrategia de ensayo y error, y aplicando la información suministrada, se procede a llenar el respectivo cuadro logrando el siguiente resultado:

COLOR	Amarilla	Azul	Roja	Blanca	Verde
NACIÓN	Noruego	Ucraniano	Inglés	Español	Japonés
BEBIDA	Agua	Té	Leche	Jugo Naranja	Café
CIGARRILLO	Pielroja	Royal	Winston	Lucky	President
VICTIMA	Foción	Hortensia	Cenelia	Rogelio	Zenon

Ej. : Expresar el número 1000 utilizando ocho cifras iguales, además de las cifras se permite también utilizar los signos de las operaciones matemáticas.

888 + 88 + 8 + 8 + 8 = 1000

Ej. : Utilizando cuatro cuatros y operaciones matemáticas entre ellos, exprese los números del cero al diez.

el cero se logra: 44 – 44

el uno: 44 / 44

el dos: (4 / 4) + (4 / 4)

el tres: (4 + 4 + 4) / 4

el cuatro: 4 + ((4 – 4) / 4)

el cinco: ((4 * 4) + 4) / 4

el seis: ((4 + 4) / 4) + 4

el siete: (44 / 4) - 4

el ocho: 4 + 4 + 4 – 4

el nueve: 4 + 4 + (4 / 4)

el diez: (44 – 4) / 4

4.5. Ejercicios propuestos

1) Dada la siguiente operación e información, encontrar el valor de cada una de las letras involucradas, teniendo en cuenta que se conoce el valor de la letra D, el cual es 5.

```
     DONALD
+    GERALD
  -----------
     ROBERT
```

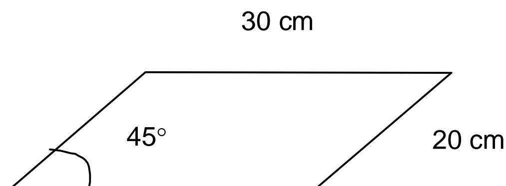
2) Dados los tres valores que representan las longitudes de los lados de un triángulo, determinar el tipo de triángulo.

3) Calcule la suma de los primeros N números naturales.

4) ¿Cuántos ratones se comen tres gatos en tres minutos, si un gato y medio en minuto y medio se come ratón y medio?

5) Una persona debe transportar en un bote de una orilla a otra de un río, un tigre, una cabra y un bulto de repollos; teniendo en cuenta que el tigre come cabras y las cabras comen repollos (si se dejan solos), y que en el bote solo puede llevar una cosa a la vez (tigre, cabra o repollos), ¿Cómo hace el hombre para transportar y mantener sus bienes?

6) Calcular el área de un trapecio cuyo lado mayor es de 30 cm, el lado menor es de 20 cm y el ángulo entre el lado mayor y el lado menor es de 45 grados.



7) Dos obreros, uno viejo y otro joven, viven en la misma casa y trabajan en el mismo sitio. El joven va desde la casa al trabajo en 20 minutos; el viejo, en 30 minutos. ¿En cuántos minutos alcanzará el joven al viejo, si éste sale de casa 5 minutos antes que el joven?

8) Entre dos ciudades A y B hay 5 caminos diferentes, entre B y C hay 4 caminos diferentes. ¿Cuántos caminos diferentes hay entre A y C?

9) ¿Cuantos metros cúbicos de agua hay almacenados en un estanque rectangular, cuyas medidas son 2 metros de largo, 3 metros de ancho y 2.5 metros de profundidad?

10) A las ocho de la mañana llegó a la ciudad de Manizales que tiene una población de 450.000 habitantes un ciudadano con cierta noticia importante; en la siguiente hora este se la comunica a cinco

personas, a la hora siguiente esas cinco personas se la comunican a otras cinco personas cada una. ¿En cuantas horas toda la población conoce la noticia?

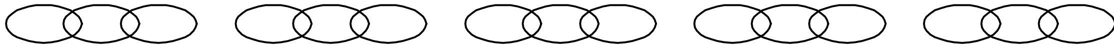
11) Dados seis periodistas es necesario determinar: el apellido, el diario en que trabajan, el canal de televisión y el día en que se emiten sus programas, y el libro que escribieron; dadas las siguientes reglas:

a) El periodista de apellido Castro escribe en el diario El Nacional y su programa de televisión sale al aire los días sábado.
b) El periodista de apellido Hoyos, colabora con el periódico El Diario y es autor del libro titulado El Precio.
c) Bernardo conduce el programa periodístico que se emite los martes por el canal 11.
d) El programa del canal 4 es conducido por el autor del libro Palabra Fácil, que además escribe en el periódico El Clarinete.
e) Mariano conduce el programa del canal 13, es autor del libro La Corrupción y no se apellida Castro.
f) Sánchez es el autor del libro Línea Recta y escribe en el diario Página 15.
g) El periodista que colabora con el diario La Prensa, conduce el programa de los jueves, pero su libro no es Política Clara.
h) Samuel se apellida Gómez y es autor del libro Cómo Llegar.
i) Nelson no es el autor del libro El Precio y su programa no se emite por el canal 2.
j) El programa de televisión de los miércoles se transmite por el canal 9, su conductor no es Marcelo, pero sí el que escribe en el diario El Popular.
k) Cortés es el apellido de quien conduce el programa de los viernes y no se llama Daniel.
l) Marcelo no conduce los programas de los canales 2 ni 7.
m) Sánchez no es el apellido de Daniel ni de Nelson.
n) Nelson no se apellida González, y ni él, ni González son los conductores del programa que se emite los lunes.

12) Cinco parejas deciden ir a un restaurante para celebrar cierto acontecimiento; una vez reunidos, se entabló entre ellos una discusión sobre el orden en que habían de sentarse a la mesa. Unos propusieron por orden alfabético, otros que por edad; otros por estatura, etc. Al cabo de cierto tiempo el mesero hizo la siguiente propuesta: siéntense en cualquier orden y anoten como fue, mañana se sientan en un orden diferente y así sucesivamente hasta que prueben todas las formas de sentarse. Una vez probadas todas las formas de sentarse, la comida será gratis en adelante. ¿Cuántos días tendrán que esperar para comer gratis en dicho restaurante?

13) Exprese el número cien de cuatro modos diferentes, empleando cinco cifras iguales, haciendo uso de los diferentes operadores matemáticos.

14) A un herrero le trajeron cinco trozos de cadena, de tres eslabones cada uno, y le encargaron que los uniera formando una cadena continua.



¿Cuántos anillos deberán ser cortados y soldados para realizar dicha labor con el menor trabajo?

15) En cierta reserva forestal existen arboles que son visitados por las aves cada año luego de la migración. El guardabosque quien vive cerca de los árboles, cuenta que pareciera que las aves hubieran enumerado los árboles.

Para ellas el árbol más frondoso es el primero; todas las tardes llegan a él en una proporción 3d (d representa el día).

Al árbol que para las aves es el segundo, arriban cada dos días en una proporción de 3d (d representa el día); según el guardabosque el primer día llegaron tres pájaros, el tercer día llegaron nueve, el quinto quince y así para cada día.

Al tercer árbol las aves llegan en una proporción 3d (d representa el día) cada tres días.

Ante una posible devastación del bosque, el ministerio del Medio Ambiente decide realizar un estudio para el cual necesita responder a las siguientes preguntas:

a) ¿Cuantas aves han arribado al árbol más frondoso hasta el día m-ésimo?
b) Considerando que el arribo de las aves a los demás arboles cumple con la proporción enunciada antes. Si existen L árboles. ¿Cuantas aves se reunirán en el bosque al m-esimo día?
c) Si existen L árboles, ¿Cuantas aves habrán visitado el bosque hasta el día m-ésimo?

5. ALGORITMOS

Se define un algoritmo como el conjunto finito y ordenado de pasos, reglas o procedimientos lógicos que se deben realizar en un orden determinado para llegar a la solución de un problema [ALB96][TAM96]. El origen de la palabra algoritmo proviene del nombre de un matemático árabe del siglo IX llamado Mohammed al-Khowârizmî quien enunció un conjunto de reglas básicas para la realización de cálculos con números y ecuaciones ya sea de forma manual o automática[BRA97].

Un algoritmo debe cumplir con las siguientes características:

- No debe tener ambigüedades o problemas de interpretación
- El número de pasos debe ser limitado o finito (no implica que no pueda ser muy grande)
- Debe alcanzar el fin propuesto (cumplir con el objetivo)
- Si se sigue dos o más veces el resultado debe ser siempre el mismo

5.1. Elementos de un algoritmo

Un algoritmo está compuesto de un determinado conjunto de elementos que permiten expresar mediante pasos o instrucciones la solución de un problema; entre estos elementos se tienen: variables, constantes, operaciones, expresiones e instrucciones.

El esquema general de un algoritmo es el siguiente:

```
Algoritmo:  nombre
Inicio
            Instrucciones
Fin
```

5.1.1. Variables y Constantes

Para llegar a la solución de un problema es necesario representar los números, letras y valores mediante objetos que faciliten su manipulación en el desarrollo del algoritmo.

Las constantes son objetos que siempre presentan el mismo valor a lo largo de proceso, mientras que las variables son objetos que pueden tomar diferentes valores durante el proceso.

A los objetos (variables y/o constantes) se les asigna un nombre o identificador único compuesto de varias letras o caracteres alfanuméricos que deben empezar por una letra, no pueden tener espacios ni signos de puntuación y preferiblemente que indiquen la tarea o función que desempeña dicho objeto en el contexto del problema.

Ej.:

- un objeto que cuente un determinado evento se le puede llamar contador
- un objeto que acumule resultados de un proceso se le puede llamar acumulador
- un objeto constante que representa el número 3,1415926 se puede llamar PI o número_pi

5.1.2. Operaciones y Expresiones

La asignación es la operación básica de todo algoritmo y tiene como fin llevar el valor de una determinada expresión o de una constante a una variable; la estructura básica de la asignación es:

variable ← expresión

donde variable es el nombre de un determinado objeto, y expresión es un conjunto de operaciones aritméticas entre los objetos (variables y/o constantes).

En las expresiones matemáticas se utilizan símbolos para representar cada una de las operaciones así:

OPERACIÓN	SÍMBOLO
Suma	+
Resta	-
Multiplicación	*
División	/
Potenciación	^
División entera	div
Residuo (módulo) de la división	mod

Se debe tener en cuenta al evaluar expresiones, que no todos los operadores tienen el mismo nivel o jerarquía para realizar su evaluación; teniendo en cuenta el siguiente esquema:

NIVEL	SÍMBOLO	EXPLICACIÓN
1	( )	Las expresiones encerradas entre paréntesis tienen prelación (se realizan primero)
2	**^**	La potenciación tiene la mayor prioridad entre los operadores.
3	***, /**	La multiplicación y la división tienen la misma prioridad; si hay dos o más operadores de este tipo, se evalúan de izquierda a derecha.
4	**div, mod**	La división entera y el residuo tienen la misma prioridad, si hay dos o más operadores de este tipo, se evalúan de izquierda a derecha.
5	**+ , -**	La suma y resta tienen la misma prioridad; si hay dos o más operadores de este tipo, se evalúan de izquierda a derecha.

Ej. : Calcular el área de un triángulo cuya base es 10 y su altura es 30

base ← 10	Guarde en la variable base el valor constante 10
altura ← 30	Guarde en la variable altura el valor constante 30
área ← (base * altura) / 2	Guarde en la variable área el resultado de (base * altura) / 2

Ej. : Calcular el promedio de cinco números determinados

promedio ← (3 + 4 + 12 + 8 + 9) / 5

Se debe tener en cuenta la diferencia que hay entre:

promedio ← (3 + 4 + 12 + 8 + 9) / 5

y

promedio ← 3 + 4 + 12 + 8 + 9 / 5

En la primera el resultado se logra sumando los cinco números (3, 4, 12, 8, 9) y dividiendo el resultado por 5; mientras que en la segunda el resultado se logra sumando 3 + 4 + 12 + 8 + el resultado de dividir 9 entre 5. En el buen uso de los paréntesis y los operadores aritméticos puede estar la solución de un problema. Esto se debe a la mayor prioridad que tiene la división sobre la suma, haciendo que esta se realice primero,

5.1.3. Instrucciones de Entrada y Salida

Todo algoritmo requiere de unos ciertos valores de entrada provenientes del medio externo, los cuales deben ser ingresados por parte del usuario; a esta operación se le denomina lectura, la cual tiene como fin leer un dato del medio externo y asignarlo a una variable para su posterior procesamiento.

La estructura básica de la instrucción de entrada es:

Leer (var1, var2, ..., varn)

Donde *var1, var2, ... varn*, son los nombres de cada una de las variables separadas por comas (,) en las cuales se desean almacenar los diferentes datos obtenidos de quien utiliza el algoritmo.

Computacionalmente es equivalente una instrucción de lectura con n variables, a n instrucciones de lectura con una variable cada una.

Ej.: Escribir el algoritmo para calcular el área de cualquier triángulo dadas su base y altura

```
Algoritmo:  Area
Inicio
            Leer (base)
            Leer (altura)
            área ← base * altura / 2
Fin
```

o su equivalente:

```
Algoritmo:  Area
Inicio
            Leer (base, altura)
            área ← base * altura / 2
Fin
```

Análogamente a las instrucciones de entrada, existen instrucciones de salida que permiten dar a conocer los valores almacenados en las variables, siendo este el mecanismo para presentar las soluciones obtenidas luego de la realización de cada uno de los pasos del algoritmo.

La estructura básica de la instrucción de salida es:

Escribir (var1, var2, ..., varn)

donde *var1, var2, ... varn*, son los nombres de cada una de las variables separadas por comas, de las cuales se desea presentar su valor o contenido. Adicionalmente es posible utilizar la instrucción *Escribir* con el fin de presentar mensajes textuales o avisos, los cuales deben estar enmarcados entre comillas dobles (").

Como en el proceso de lectura, para escribir es equivalente una instrucción de escritura con n variables, a n instrucciones de escritura con una variable cada una.

Ej. : Escribir el algoritmo que permita calcular el área de cualquier triángulo, dadas su base y altura, presentando el resultado

```
Algoritmo:  Area
Inicio
            Leer (base)
            Leer (altura)
            área ←  base * altura / 2
            Escribir (área)
Fin
```

Ej. : Calcular la nota definitiva para un estudiante que realiza tres exámenes; donde el primero y el segundo valen 35% c/u, mientras que el tercero vale el 30%.

```
Algoritmo:  Nota
Inicio
            Leer (nota1, nota2, nota3)
            notadef ← nota1 * 0.35 + nota2 * 0.35 + nota3 * 0.30
            Escribir (notadef)
Fin
```

Ej. Escribir el mensaje "Primer algoritmo"

```
Algoritmo:  Mensaje
Inicio
            Escribir ("Primer algoritmo")
Fin
```

Ej. : Calcular el área de un círculo dado su radio.

```
Algoritmo:  Area_Radio
Inicio
            Leer (radio)
            area ← 3.1416 * radio ^ 2
            Escribir ("El area del círculo es : ", area)
Fin
```

Ej. : Escribir un algoritmo que convierta una medida de cm a pulgadas.

```
Algoritmo:  Cm_a_Pulgadas
Inicio
            Leer (centímetros)
            pulgadas ← centímetros / 2.54
            Escribir (pulgadas)
Fin
```

Ej. : Calcular el cubo de un número.

```
Algoritmo:  Cubo
Inicio
            Leer (numero)
            cubo ← numero * numero * numero
            Escribir (cubo)
Fin
```

5.1.4. Instrucciones Condicionales

Son aquellas instrucciones que permiten condicionar la secuencia de pasos del algoritmo dado un determinado criterio de comparación.

a) Instrucción Condicional Simple: Es aquella que dependiendo de una condición que puede ser evaluada como verdadera o falsa permite la realización de uno de los dos posibles conjuntos de instrucciones.

La estructura básica de la instrucción condicional es:

```
Si (condición) entonces
      instrucciones1
Sino
      instrucciones2
Fin_Si
instrucciones3
```

donde *condición* es la consulta o elemento condicional por el cual se pregunta, y en caso de ser evaluado como verdadero o cierto, se realizan los pasos del algoritmo agrupados bajo *instrucciones1*, de lo contrario (no se cumple), se realizan los pasos agrupados bajo *instrucciones2*. Una vez realizados los pasos agrupados en *instrucciones1* o en *instrucciones2*, el algoritmo continúa con los pasos agrupados en *instrucciones3*.

Algunas veces no es necesario realizar algún tipo de acción cuando la *condición* no se cumple, presentándose una instrucción condicional más sencilla con la siguiente estructura básica:

```
Si (condición) entonces
        instrucciones1
Fin_Si
instrucciones2
```

de cumplirse la condición se efectúan los pasos agrupados bajo *instrucciones1* y luego continúa con *instrucciones2*; de no cumplirse la condición, se ejecutan directamente los pasos agrupados en *instrucciones2*.

Teniendo en cuenta que las condiciones son expresiones que al evaluarse pueden generar un valor verdadero o falso, según las condiciones, es necesario definir una serie de operadores relacionales que permitan realizar las diferentes comparaciones, así:

OPERADOR RELACIONAL	SIGNIFICADO
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
=	determina si dos valores numéricos o dos caracteres son iguales
<>	determina si dos valores numéricos o dos caracteres son diferentes

El formato general para las condiciones es:

```
expresión1   operador_relacional   expresión2
```

donde son ciertas las siguientes condiciones:

```
4 < 10
1234 <> 123
12 + 2 = 14
234 <= 234
9 > 1
123 >=28
```

Ej. : Determinar y escribir el mayor entre dos números leídos previamente

```
Algoritmo:  Mayor
Inicio
            Leer (número1)
            Leer (número2)
            Si (número1 > número2) entonces
                    Escribir (número1)
            Sino
```

```
                    Escribir (número2)
          Fin_Si
Fin
```

Ej. : Calcular la nota definitiva para un estudiante que realiza tres exámenes; el primero y segundo valen 35% c/u, mientras que el tercero vale el 30%. Determinar si el estudiante aprobó la materia, teniendo en cuenta que las notas mayores o iguales a tres son aprobatorias.

```
Algoritmo:  Nota_Definitiva
Inicio
          Escribir ("Teclee las notas : ")
          Leer (nota1, nota2, nota3)
          notadef ← nota1 * 0.35 + nota2 * 0.35 + nota3 * 0.30
          Escribir ("La nota definitiva es : ", notadef)
          Si (notadef  >= 3) entonces
                    Escribir ("Aprobó la Materia")
          Sino
                    Escribir ("No Aprobó la Materia")
          Fin_Si
Fin
```

Ej. : Calcular el precio final de venta de un artículo, teniendo en cuenta que para artículos con valor superior a $ 20.000, se realiza un descuento del 10%.

```
Algoritmo:  Precio
Inicio
          Escribir ("Teclee el precio base : ")
          Leer (precio)
          Si (precio  >= 20000) entonces
                    precio ← precio - (precio * 0.10)
          Fin_Si
          Escribir (precio)
Fin
```

Es posible encontrar instrucciones condicionales anidadas, en las cuales una vez cumplida una condición es necesario verificar otras diferentes (una o más) con el fin de obtener una determinada respuesta.

Ej. : Encontrar y escribir el mayor entre 3 números.

```
Algoritmo:  Mayor
Inicio
          Leer (número1, número2, número3)
          Si (número1 > número2) entonces
                    Si (número1 > número 3) entonces
```

```
                Escribir ("El mayor es ", número1)
            Sino
                Escribir ("El mayor es", número3)
            Fin_Si
        Sino
            Si (número2 > número3) entonces
                Escribir (número2)
            Sino
                Escribir (número3)
            Fin_Si
        Fin_Si
Fin
```

Adicionalmente las condiciones pueden ser compuestas, dando la posibilidad de evaluar varias condiciones simultáneamente unidas mediante operadores lógicos. Los operadores lógicos básicos son: el operador Y, que sólo es verdadero cuando las dos condiciones que une son verdaderas, de lo contrario su valor es falso; el operador O, que solo es falso cuando las dos condiciones que une son falsas, de lo contrario es verdadero; el operador unitario NO, que es falso cuando su condición es verdadera y verdadero cuando su condición es falsa.

OPERADOR LÓGICO	**SIGNIFICADO**
Y (AND)	Se deben cumplir las dos condiciones
O (OR)	Se debe cumplir una de las condiciones
NO (NOT)	Se niega el valor de la condición

El formato general para las condiciones compuestas es:

```
condición1   operador_lógico   condición2     (para los operadores Y, O)
operador_lógico  condición                     (para el operador NO)
```

donde se evalúan como verdaderas las siguientes operaciones:

```
( 20 > 12) Y (10 <= 7)
(10 < 5) O (4 <> 6)
NO (30 < 24)
NO ((4 + 3 – 2 ) > (12 – 2))
(12 + 4 <= 123) Y ( 23 > 4)
(12 < 2) O (5 > 3)
```

Ej. : Determinar si un valor previamente leído está entre 10 y 20.

```
Algoritmo:  Determinar_Rango
Inicio
        Escribir ("Teclee el numero : ")
```

```
        Leer (numero)
        Si ((numero >=10) Y (numero <= 20)) entonces
            Escribir ("Esta en el rango")
        Sino
            Escribir ("Esta fuera de rango")
        Fin_Si
Fin
```

Ej. : Ordenar y escribir tres números descendentemente.

```
Algoritmo:  Ordenar
Inicio
        Escribir ("Teclee los números : ")
        Leer (a, b, c)
        Si (a > b Y b > c) entonces
            Escribir (a, b, c)
        Fin_Si
        Si (a > b Y c > b Y  a > c) entonces
            Escribir (a, c, b)
        Fin_Si

        Si (b > a Y a > c) entonces
            Escribir (b, a, c)
        Fin_Si
        Si (b > a Y c > a Y b > c) entonces
            Escribir (b, c, a)
        Fin_Si
        Si (c > a Y a > b) entonces
            Escribir (c, a, b)
        Fin_Si
        Si (c > a Y b > a Y  c > b) entonces
            Escribir (c, b, a)
        Fin_Si
Fin
```

b) Instrucción Condicional Múltiple: Es aquella que dependiendo de una determinada expresión que se evalúa como un valor numérico, permite escoger un conjunto de instrucciones entre las diferentes posibilidades que se plantean.

La estructura básica de la instrucción condicional múltiple es:

```
En_caso_de ( expresión )
    valor1: instrucciones1
    valor2: instrucciones2
    valor3: instrucciones3
    .....
```

```
        valorN: instruccionesN
        Sino   instrucciones
    Fin_En_caso_de
```

donde *expresión* se evalúa para obtener un determinado valor numérico, o de tipo carácter, el cual será comparado con cada una de las diferentes opciones que se presentan *(valor1, valor2, ..., valorN)* y en caso de ser igual a una de ellas, se realizan las instrucciones asociadas a cada una. De no concordar con ninguna de las opciones, se tiene la posibilidad de un Sino, el cual realizaría las instrucciones que a éste se le asocien (es opcional).

Ej. : Dado el número que identifica un mes, escribir el nombre del mes correspondiente.

```
Algoritmo: Meses
Inicio
    Escribir ("Teclee el número que identifica al mes: ")
    Leer (numero)
    En_caso_de (numero)
        1: Escribir ("enero")
        2: Escribir ("febrero")
        3: Escribir ("marzo")
        4: Escribir ("abril")
        5: Escribir ("mayo")
        6: Escribir ("junio")
        7: Escribir ("julio")
        8: Escribir ("agosto")
        9: Escribir ("septiembre")
        10: Escribir ("octubre")
        11: Escribir (noviembre")
        12: Escribir ("diciembre")
        Sino Escribir ("Ese numero no corresponde a un mes")
    Fin_En_caso_de
Fin
```

Ej. : Dadas las notas de un colegio, las cuales se representan mediante números enteros entre 1 y 5, se debe especificar una nota cualificativa según la siguiente regla:

1 es pésima
2 es mala
3 es regular
4 es buena
5 es excelente

```
Algoritmo Notas_Cualificativas
 Inicio
        Escribir ("Teclee la nota numérica : ")
        Leer (nota)
        En_caso_de (nota)
```

```
                    1: Escribir ("pésima")
                    2: Escribir ("mala")
                    3: Escribir ("regular")
                    4: Escribir ("buena")
                    5: Escribir ("excelente")
          Fin_En_caso_de
Fin
```

5.1.5. Instrucciones Repetitivas

Son aquellas instrucciones que permiten realizar una misma secuencia de pasos del algoritmo varias veces, mientras que se cumpla un determinado criterio de comparación.

a) Instrucción repetitiva *Mientras_que .. haga:* Es aquella que permite realizar un conjunto de instrucciones siempre y cuando la condición asociada se evalúe como cierta, de lo contrario continua con el siguiente grupo de instrucciones.

La estructura básica de la instrucción repetitiva *Mientras_que .. haga es:*

```
Mientras_que (condición) haga
     instrucciones1
Fin_Mientras_que
instrucciones2
```

donde *condición* es la expresión condicional que debe cumplirse para que se realicen los pasos del algoritmo agrupados bajo *instrucciones1*, de lo contrario (cuando no se cumpla la *condición*), se realizan los pasos descritos en *instrucciones2* (es opcional y puede no ser necesario

Ej. : Escribir los números enteros existentes entre el 1 y el 10.

```
Algoritmo:  Enteros
Inicio
            numero ← 1
            Mientras_que (número <= 10) haga
                 Escribir (número)
                 número ← número + 1
            Fin_Mientras_que
Fin
```

Ej. : Determinar entre 20 números leídos, cuantos son pares.

```
Algoritmo:  Pares
Inicio
            contador ← 0
```

```
        pares ← 0
        Mientras_que(contador < 20) haga
                Escribir ("Teclee un número : ")
                Leer(numero)
                Si (numero MOD 2 = 0) entonces
                        pares ← pares + 1
                Fin_Si
                contador ← contador + 1
        Fin_Mientras_que
        Escribir ("Se digitaron : ", pares, " números pares")
Fin
```

Ej. : Escribir la suma de los números enteros existentes entre el 1 y el 100.

```
Algoritmo: Suma_Enteros
Inicio
        suma ← 0
        numero ← 1
        Mientras_que (número <= 100) haga
                suma ← suma + número
                número ← número + 1
        Fin_Mientras_que
        Escribir ("La suma de los 100 primeros números es : ", suma)
Fin
```

Ej. : Escribir los números enteros pares existentes entre el 1 y el 200.

```
Algoritmo: Pares
Inicio
        numero ← 2
        Mientras_que (número <= 200) haga
                Escribir(número)
                número ← número + 2
        Fin_Mientras_que
Fin
```

Ej. : Escribir el resultado de multiplicar los 20 primeros números pares, menos la suma de los 10 primeros números impares

```
Algoritmo:  Multiplicación_Pares
Inicio
        mult_pares ← 1
        contador ← 1
        Mientras_que (contador <= 20) haga
```

```
                par ← contador * 2
                mult_pares ← mult_pares * par
                contador ← contador + 1
        Fin_Mientras_que

        suma_impares ← 1
        contador ← 1
        Mientras_que (contador <= 10) haga
                impar ←   (contador * 2) - 1
                suma_impares ← suma_impares + ((2 * contador) - 1)
                contador ← contador + 1
        Fin_Mientras_que
        resultado ←  mult_pares - suma_impares
        Escribir(resultado)
Fin
```

Ej. : Escriba un algoritmo que dado un número entero positivo, determine la cantidad de cifras que lo componen (para cualquier cantidad de cifras).

```
Algoritmo: Cifras
Inicio
        Escribir ("Teclee el número : ")
        Leer (N)
        contador ← 0
        Mientras_que ( N >= 1) haga
                contador ← contador + 1
                N ←  N / 10
        Fin_Mientras_que
        Escribir ("El número de cifras es : ", contador)
Fin
```

Ej. : Escriba un algoritmo que calcule la multiplicatoria de los primeros N términos de la siguiente serie : -1, 4, -9, 16, -25, 36, -49, 64, ...

```
Algoritmo: Serie_Cuadrados
Inicio
    Leer (N)
    producto ← 1
    contador ← 1
    signo ← -1
    Mientras_que ( contador <= N ) haga
        termino ← signo * (contador ^ 2)
        producto ← producto * termino
        contador ← contador + 1
    Fin_Mientras_que
```

```
        Escribir ("El valor de la multiplicatoria es : ", producto)
Fin
```

b) Instrucción repetitiva *Repita .. Hasta_que:* Es aquella que de forma similar a la instrucción *Mientrasque*, permite realizar un conjunto de instrucciones varias veces, siempre y cuando la condición de terminación o salida se evalúe como falsa.

La estructura básica de las instrucción repetitiva *Repita .. Hasta_que* es la siguiente:

```
Repita
        instrucciones1
Hasta_que (condición)
instrucciones2
```

donde *condición* es la expresión condicional que no debe cumplirse para que se realicen los pasos del algoritmo agrupados bajo *instrucciones1*, de lo contrario (cuando se cumpla la *condición*), se realizan los pasos agrupados bajo *instrucciones2* (es opcional y puede no ser necesario). La diferencia básica entre la instrucción *Mientras_que y Repita .. Hasta_que,* radica en que la verificación de la *condición* se hace al final del ciclo, lo que implica que el *Repita .. Hasta_*que siempre se realice por lo menos una vez y adicionalmente las condiciones de terminación son contrarias, ya que el *Mientras_que* termina cuando la *condición* es falsa, mientras que el *Repita*, termina cuando la *condición* es verdadera.

Ej. : Escribir un algoritmo para leer y sumar 10 números, calculando el respectivo promedio

```
Algoritmo:  Leer_Sumar_Numeros
Inicio
            contador ← 0
            suma ← 0
            Repita
                    Escribir ("Teclee un número : ")
                    Leer (numero)
                    suma ← suma + numero
                    contador ← contador + 1
            Hasta_que (contador = 10)
            promedio ← suma / 10
            Escribir ("La suma es ", suma)
            Escribir ("El promedio es : ", promedio)
Fin
```

Ej. : Calcular el factorial de un número previamente leído.

```
Algoritmo: Factorial
Inicio
            Leer (numero)
```

```
                fac ← 1
                Repita
                        fac ← fac * numero
                        numero ← numero – 1
                Hasta_que (numero <= 0)
                Escribir (fac)
Fin
```

Ej. : Escriba un algoritmo que determine si N valores leídos son positivos, negativos o cero.

```
Algoritmo: Clasificar
Inicio
        Leer (N)
        contador ← 0
        Repita
                Leer (numero)
                Si (numero > 0) entonces
                        Escribir("positivo")
                Sino
                        Si (numero < 0) entonces
                                Escribir ("negativo")
                        Sino
                                Escribir ("cero")
                        Fin_Si
                Fin_Si
                contador ← contador + 1
        Hasta_que (contador = N)
Fin
```

Ej. : Calcular los primeros N elementos de la siguiente serie : 1, -3, 5, -7, 9, -11, 13 ...

```
Algoritmo: Serie_Impares_Negativos
Inicio
        Escribir ("Cantidad de elementos de la serie : ")
        Leer (N)
        contador ← 0
        numero ← 1
        Repita
                Escribir (numero)
                numero ← (numero + 2) * (-1)
                contador ← contador + 1
        Hasta_que (contador = N)
Fin
```

Ej. : Calcular los N primeros términos de la serie de Fibonacci, la cual se presenta de la siguiente manera:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
Algoritmo: Fibonacci
Inicio
        Escribir ("Número de términos a calcular : ")
        Leer (N)
        anterior ← 0
        actual ← 1
        Si (N = 1) entonces
                Escribir (actual)
        Sino
                Escribir (actual)
                contador ← 1
                Repita
                        siguiente ← anterior + actual
                        Escribir (siguiente)
                        anterior ← actual
                        actual ← siguiente
                        contador ← contador + 1
                Hasta_que (contador = N)
        Fin_Si
Fin
```

c) Instrucción repetitiva *Para:* Permite la realización de un ciclo con conjunto de instrucciones, teniendo como referencia un contador para determinar el número de veces que se realizará dicho ciclo.

La estructura básica de la instrucción repetitiva *Para* es la siguiente:

```
Para variable desde valor_inicial Hasta valor_final
        instrucciones1
Fin_Para
instrucciones2
```

donde *variable* toma valores numéricos que van desde *valor_inicial* hasta *valor_final* con incrementos progresivos de uno (1); cada vez que se realiza un ciclo, se ejecuta el conjunto *instrucciones1*, se incrementa el valor de la variable y se verifica si dicho valor no sobrepasa el *valor_final.* Una vez que el valor de la *variable* sobrepasa el límite del *valor final*, se realizan las instrucciones agrupadas en *instrucciones2*. Los valores representados por *valor_inicial* y *valor_final* pueden ser variables, constantes numéricas o expresiones que regresen un valor numérico.

Ej. : Escribir los números enteros existentes entre el 1 y el 10.

```
Algoritmo: Numeros_Enteros
Inicio
```

```
        Para numero desde 1 Hasta 10
                Escribir (número)
        Fin_Para
Fin
```

Ej. : Escribir los números enteros que se encuentran entre dos cantidades A y B, de tal forma que se presenten en forma descendente

```
Algoritmo: Rango
Inicio
        Escribir ("Teclee el rango : ")
        Leer (A, B)
        Si (A < B) entonces
                Para contador desde A hasta B
                        Escribir (B – contador + A)
                Fin_Para
        Sino
                Para contador desde B hasta A
                        Escribir (A – contador + B)
                Fin_Para
        Fin_Si
Fin
```

Ej. : Escribir la suma de los números enteros existentes entre el 1 y el 100.

```
Algoritmo:  Suma_Enteros
Inicio
        suma ← 0
        Para numero desde 1 Hasta 100
                suma ← suma + número
        Fin_Para
        Escribir (suma)
Fin
```

Ej. : Escribir los números enteros pares existentes entre el 1 y el 200.

```
Algoritmo: Numeros_Pares
Inicio
        Para numero desde 1 Hasta 100
                Escribir (2*número)
        Fin_Para
Fin
```

Ej. : Calcular el factorial de un número determinado

```
Algoritmo: Factorial
Inicio
        Escribir ("Teclee el número : ")
        Leer (N)
        factor ← 1
        Para numero desde 1 Hasta N
                factor ← factor * numero
        Fin_Para
        Escribir ("El factorial es: ", factor)
Fin
```

Ej. : Hallar la suma de los primeros N términos de la siguiente serie: 1/1!, 2/3!, 3/5!, 4/7!, 5/9!, ... (recuerde que el símbolo !, representa el factorial)

```
Algoritmo: Serie
Inicio
        Escribir ("Teclee el número de elementos de la serie: ")
        Leer (N)
        suma ← 0
        valor ← 1
        Para contador desde 1 Hasta N
                numerador ← contador
                factor ← 1
                Para numero desde 1 Hasta valor
                        factor ← factor * numero
                Fin_Para
                denominador ← factor
                suma ← suma + numerador / denominador
                valor ← valor + 2
        Fin_Para
        Escribir ("La suma es : ", suma)
Fin
```

5.2. Verificación de algoritmos

La experiencia derivada al trabajar en el desarrollo de algoritmos y programas de computador ha demostrado que la mayor parte de los esfuerzos y recursos se gastan en la corrección de errores y modificación de programas. Estos problemas aumentan cuanto más grandes y complejos son los programas.

5.2.1. Pruebas de Escritorio

Normalmente se verifica el funcionamiento de un programa (ó algoritmo), mediante pruebas con innumerables datos de entrada y su correspondiente salida, demostrando que funciona para muchos casos sin garantizar que funcione para otros. Esto se logra disponiendo de una lista con todas las variables involucradas en el algoritmo y realizando una a una de las instrucciones que lo conforman, modificando el valor de las variables cada que el algoritmo realiza una asignación u operación sobre ellas y presentando los resultados que éste produce.

Ej. : Intercambiar el valor de dos variables previamente leídas utilizando un temporal y escribir el resultado que generan:

```
Algoritmo: Intercambio
Inicio
        Leer (A, B)
        Temporal ← A
        A ← B
        B ← Temporal
        Escribir (A, B)
Fin
```

Valores iniciales de las variables

Variables	**Valor**
A	Indeterminado
B	Indeterminado
Temporal	Indeterminado
Entrada	**10 20**
Salida:	

Paso 1: Para una entrada de 10 en la variable A y 20 para la variable C

```
          Algoritmo: Intercambio
          Inicio
=>                Leer (A, B)
                  Temporal ← A
                  A ← B
                  B ← Temporal
                  Escribir (A, B)
          Fin
```

Valores de las variables

VARIABLES	VALOR
A	10
B	20
Temporal	Indeterminado
Entrada:	
Salida:	

Paso 2: Asignando a la variable Temporal el valor de la variable A

```
        Algoritmo: Intercambio
        Inicio
                Leer (A, B)
=>              Temporal ← A
                A ← B
                B ← Temporal
                Escribir (A, B)
        Fin
```

Valores de las variables

VARIABLES	VALOR
A	10
B	20
Temporal	10
Entrada:	
Salida:	

Paso 3: Asignando a la variable A, el valor de la variable B

```
        Algoritmo: Intercambio
        Inicio
                Leer (A, B)
                Temporal ← A
=>              A ← B
                B ← Temporal
                Escribir (A, B)
        Fin
```

Valores de las variables

VARIABLES	VALOR
A	20
B	20
Temporal	10
Entrada:	
Salida:	

Paso 4: Asignando a B el valor de Temporal:

```
        Algoritmo: Intercambio
        Inicio
               Leer (A, B)
               Temporal ← A
               A ← B
=>             B ← Temporal
               Escribir (A, B)
        Fin
```

Valores de las variables

VARIABLES	VALOR
A	20
B	10
Temporal	10
Entrada:	
Salida:	

Paso 5: Escribiendo el valor de A y B :

```
        Algoritmo: Intercambio
        Inicio
               Leer (A, B)
               Temporal ← A
               A ← B
               B ← Temporal
=>             Escribir (A, B)
        Fin
```

Valores de las variables

VARIABLES	**VALOR**
A	20
B	10
Temporal	10
Entrada:	
Salida:	**20 10**

5.2.2. Verificación formal

Existe un método de verificación de algoritmos que basa su funcionamiento en principios matemáticos que permite determinar si los resultados obtenidos por la ejecución de un programa independientemente de las entradas cumple con las especificaciones previamente definidas.

El primer paso para verificar formalmente el funcionamiento de un programa es el de especificar las condiciones de entrada y salida, de manera que describan el estado de ejecución de cada una de las variables del programa o algoritmo según sea el caso. Estas condiciones se denominan: precondición y postcondición:

a) La precondición o condición inicial debe especificar cualquier tipo de restricción que deben cumplir las variables de entrada al programa, subprograma, módulo e inclusive para cada conjunto de instrucciones.

b) La postcondición o condición final debe especificar los resultados que se espera que genere el programa, subprograma, módulo e inclusive conjuntos de instrucciones, en términos de sus variables.

El método basa su funcionamiento en que la postcondición debe ser inferida o deducida a partir de la precondición y la ejecución del conjunto de instrucciones al cual se refieren dichas condiciones, de tal manera que se pueda asegurar que si la precondición es cierta antes de la realización de las instrucciones, en la medida que dichas instrucciones se ejecuten independientemente de los datos, y formalmente se llegue a la postcondición, se asegura que dicho proceso está correcto.

El principal inconveniente del método radica en que se debe realizar una verificación formal de cada una de las instrucciones del algoritmo, lo cual se vuelve impracticable debido a la cantidad de líneas de código que conforman un algoritmo o programa.

Debido a la complejidad de este método, únicamente se mencionan sus principales características ya que no es el objetivo de este libro; pero para mayor información se sugiere la siguiente bibliografía:

⇒ AHO, A., HOPCROFT, J., ULLMAN, J., "The Design and Analysis of Computer Algorithms", Addison-wesley, 1974.
⇒ CARDOSO, R., "Verificación y Desarrollo de Programas", Ediciones Uniandes, 1991.
⇒ DIJKSTRA, E. W., "A Discipline of Programming", Prentice-Hall, 1976.
⇒ GRIES, D., "The Science of Programming", Springer-Verlag, 1981

5.3. Ejercicios propuestos

1) Calcule el volumen de una caja dadas las dimensiones que la conforman, escribiendo los datos de entrada y la respuesta.

2) Convierta el valor de una temperatura medida en grados Fahrenheit a su correspondiente en grados centígrados.

3) Intercambie el contenido de dos variables numéricas sin utilizar variables auxiliares.

4) Calcule el valor máximo entre N valores leídos.

5) Calcule la suma, resta, multiplicación y división entre dos valores previamente leídos, teniendo en cuenta que la división por cero no es válida, escribiendo los resultados obtenidos.

6) Determine el nivel de la nota definitiva de un estudiante, cuya definitiva se calcula teniendo en cuenta que la primer nota vale el 35%, la segunda el 35% y la tercera el 30% y los niveles son: de 0 a 2 mala, de 2 a 3 deficiente, de 3 a 4 buena, y entre 4 y 5 excelente.

7) Calcule los números factoriales entre uno y un valor previamente leído, escribiendo el número y su correspondiente valor factorial, y al final, la suma de los números y la suma de sus factoriales.

8) Determine las estadísticas para N estudiantes de un colegio, según los siguientes requerimientos:

- estudiantes menores de 5 años
- estudiantes entre 5 y 7 años
- estudiantes entre 7 y 10 años
- estudiantes entre 10 y 12 años
- estudiantes mayores de 12 años

9) Calcule el número del medio entre tres números previamente conocidos.

10) Calcule el valor de una venta compuesta por N artículos y sus valores correspondientes, teniendo en cuenta que se cobra un impuesto adicional de 16% y que si el valor de las ventas antes del impuesto está entre $20.000 y $50.000 se le hace un descuento del 5%, y para ventas superiores de $50.000 el descuento es del 8%.

11) Escriba los primeros N términos de la siguiente serie: 1, 2, -3, -4, 5, 6, -7, -8, 9, 10, -11, -12, 13, ...

12) Haciendo uso de cada una de las instrucciones repetitivas (*Mientrasque, Repita, Para*), calcule los primeros N términos de la siguiente serie: $1^2 / 2$, $- 3^2 / 4$, $5^2 / 6$, $- 7^2 / 8$, $9^2 / 10$, $- 11^2 / 12$, ...

13) Calcule el N-esimo término de la serie de Fibonacci.

14) Determine si un número tiene raíz cuadrada exacta.

15) Determine si un número tiene raíz cúbica exacta; extiéndalo para cualquier raíz.

16) Calcule la raíz cuadrada de un número hasta N cifras decimales.

17) Determine si un número es primo.

18) Calcule los factores primos de un número entero positivo.

19) Determine si un número cumple con las características de capicúa (se lee igual de izquierda a derecha, que de derecha a izquierda, Ej. 121, 23432, 44, 49233294)

20) Dada una especificación de tiempo en: horas, minutos y segundos, calcule dicha especificación un segundo después.

21) Utilizando el ejercicio anterior, desarrolle el simulador de un reloj, de tal manera que presente las diferentes variaciones del tiempo.

22) Calcule la diferencia entre dos períodos de tiempo dados en: horas, minutos y segundos.

23) Calcule la cantidad de monedas de cada denominación: 100, 25, 10, 5 y 1 peso respectivamente, que deben ser entregadas para pagar una suma cualquiera de N pesos.

24) Una persona desea invertir su dinero en un banco, el cual le otorga un 2% de interés mensual. ¿Cuál será la cantidad al cabo de un año, sí la ganancia de cada mes sé reinvierte?

25) Una empresa de construcción ofrece casas bajo las siguientes condiciones: si los ingresos del comprador son menores o iguales a $120.000, la cuota inicial será del 15% del costo de la casa y el resto se distribuirá uniformemente en pagos mensuales a 10 años. Si los ingresos son mayores de $120.000, la cuota inicial será del 30% del costo de la casa y el resto se distribuirá en pagos mensuales a 7 años. Dado el valor de la casa y salario, ¿Cuánto debe pagar de cuota inicial y cuanto de cuota mensual?

TERCERA PARTE

EL LENGUAJE C

6. CONCEPTOS BÁSICOS DEL LENGUAJE

6.1. Historia

C es un lenguaje de propósito general originalmente diseñado por Dennis Ritchie de los laboratorios Bell e implementado en un computador DEC PDP-11 de la Digital Equipment Corporation en 1.972, con el fin de solucionar las deficiencias y limitaciones del anterior lenguaje B, el cual se basaba en el lenguaje BCPL desarrollados en 1.967 por Martin Richards.

Durante muchos años el estándar de C fue la versión proporcionada con el sistema operativo UNIX versión 5, pero pronto empezaron a surgir muchas implementaciones del lenguaje a raíz de la popularidad creciente de los microcomputadores. Por este motivo, fue necesario definir un C estándar que está representado hoy por el ANSI C.

Sus principales características son:

- Un reducido conjunto de expresiones, estructuras de control, estructuras de datos avanzadas y un completo grupo de operadores aritmético-lógicos.
- C no es un lenguaje de alto nivel y no se especializa en un área determinada, pero es el lenguaje de desarrollo estándar para la mayor parte del software de hoy. Muchos programas que funcionan en los sistemas operativos DOS, OS/2, Windows y UNIX, son escritos utilizando este lenguaje.
- El sistema operacional UNIX y todas sus variantes se han escrito en C, siendo éste su lenguaje nativo y el que más fácil se acomoda para el desarrollo de aplicaciones bajo este sistema operacional.
- Es un lenguaje de propósito general el cual se ha utilizado para el desarrollo de aplicaciones tan diversas como: hojas de electrónicas, gestores de bases de datos, compiladores, sistemas operativos, etc.
- Es un lenguaje de nivel medio, ya que permite programar a alto nivel (pensando en el ámbito lógico y no en la máquina física) y a bajo nivel (con lo que se puede obtener la máxima eficiencia y un control absoluto de cuanto sucede en el interior del computador).

▪ Es un lenguaje portable, ya que los programas escritos en C son fácilmente transportables a otros sistemas operacionales.

▪ Es un lenguaje potente y eficiente, ya que haciendo uso del C, un programador puede casi alcanzar la eficiencia del código ensamblador junto con la estructura de lenguajes de alto nivel como AlGOL y/o PASCAL; adicionalmente es la plataforma de los lenguajes C++ y JAVA.

Como desventajas habría que reseñar que es más complicado de aprender que otros lenguajes tradicionales como PASCAL o BASIC, además, permite un uso exhaustivo de los recursos del sistema operacional y del hardware, lo que fácilmente puede acarrear errores de funcionamiento sino se tiene la suficiente experiencia, ya que requiere de un buen conocimiento para poder aprovecharlo a fondo.

6.2. Estructura del lenguaje

Todo programa en C debe cumplir con la siguiente estructura (el orden no es determinante):

- declaraciones externas
- instrucciones para el preprocesador
- definición de constantes y macroinstrucciones
- definición de estructuras y tipos
- declaración de prototipos para las funciones
- declaración de variables globales
- definición de funciones

6.2.1. Declaraciones externas

Es la parte donde se incluyen los archivos encabezados de las diferentes librerías que se utilizan para el desarrollo del programa; estos archivos se caracterizan por tener un nombre cuya extensión es: .h (header o encabezado) y contienen los prototipos o encabezados de las funciones que indican el tipo o valor a retornar, así como el tipo y cantidad de cada uno de los parámetros o argumentos, también se encuentra la definición de estructuras, declaración de variables y constantes requeridas para el normal desempeño de las funciones allí referenciadas.

El prototipo de una función es el encabezado de la misma, incluyendo el valor de retorno, nombre de la función y el tipo de cada uno de los parámetros que la conforman, terminada con punto y coma (;).

Los archivos de encabezados más comunes son:

ARCHIVO	**DESCRIPCIÓN**
alloc.h	rutinas administradoras de la memoria dinámica
bios.h	rutinas para el control y manejo del BIOS (Basic Input Output System)
complex.h	funciones para el manejo de números complejos
conio.h	entradas y salidas por la consola utilizando rutinas del DOS
ctype.h	clasificación y conversión de tipos de datos
dir.h	estructuras y macros para trabajar con directorios, caminos y nombres
dos.h	funciones propias del sistema DOS bajo procesadores tipo 80x86, Pentium
errno.h	constantes y/o mnemónicos para el manejo de los códigos de error
float.h	parámetros para las rutinas de punto flotante
io.h	estructuras y declaraciones para rutinas de entrada y salida a bajo nivel
locale.h	información sobre el país y el lenguaje seleccionado en el sistema
math.h	cálculos matemáticos
mem.h	macros, funciones y constantes para el manejo de memoria
memory.h	manipulación de la memoria
process.h	estructuras y declaraciones para el manejo de procesos
search.h	funciones para búsqueda y ordenamiento
share.h	parámetros y estructuras de las funciones para compartir archivos
stdarg.h	manejo de funciones con número variable de argumentos
stdio.h	manejo estándar de las entradas y salidas
stdlib.h	manejo de rutinas estándar del sistema
string.h	manejo de cadenas de caracteres
time.h	cálculo y manejo de horas y fechas

La sintaxis para incluir estos archivos es la siguiente:

```
#include <nombre>
```

o también es válido indicar el nombre del archivo de encabezado entre comillas dobles como una cadena de caracteres con el fin de indicar el camino completo, así:

```
#include "nombre con camino"
```

Ej. : Si se requiere manejo de cadenas, entradas y salidas; el formato es el siguiente:

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

Ej. : Si se requiere incluir un encabezado para el manejo de la memoria que se encuentra en el directorio c:\borlandc\include; el formato es el siguiente:

```
#include "c:\\borlandc\\include\\memory.h"
```

6.2.2. Instrucciones para el preprocesador

Existe un conjunto de instrucciones que son utilizadas por el preprocesador del lenguaje (primera fase de la compilación) que permiten establecer características del sistema operacional, del programa, del microprocesador utilizado, determinar el tipo de computador, de tarjeta graficadora o establecer si un archivo de encabezados ya está definido, de tal manera que estas puedan ser tenidas en cuenta realizando las acciones pertinentes y permitan la normal compilación y posterior ejecución del programa.

Ej. : Determinar si la constante DOS ya está definida, y en caso de estarlo definir una constante VALOR con un contenido de uno (1), de lo contrario definir la constante con cero (0):

```
#ifdef    DOS
#define   VALOR   1
#elseif
#define   VALOR   0
#endif
```

6.2.3. Definición de constantes y macroinstrucciones

Es el lugar donde se deben definir y declarar las constantes, macroinstrucciones del programa (*#define*),

Ej. : Definir la constante PI:

```
#define        PI        3.141592654
```

La explicación del anterior ejemplo es: reemplace todas las ocurrencias de PI en el código del programa por el valor 3.141592654.

Ej. : Definir una constante denominada TAMAÑO con una valor de 100

```
#define        TAMAÑO   100
```

también se pueden definir macroinstrucciones que lleven parámetros, dando la posibilidad de reemplazar el valor de los parámetros en las expresiones como si fueran funciones

Ej. : Definir una macroinstrucción que permita elevar un valor cualquiera al cuadrado

```
#define    elevar(X)    ((X)*(X))
```

6.2.4. Definición de las estructuras o registros

y los nuevos tipos que se necesitan en el programa para la formación de estructuras de datos (*structs*) y la definición de nuevos tipos de datos mediante la instrucción *typedef*, siendo posible extender

el conjunto de tipos básicos que provee el lenguaje C, de tal manera que se definan otros de acuerdo con las necesidades o requerimientos de cada programa.

Ej.: Si se desea definir el tipo entero sin signo, para denominarlo NumEntero, se expresa de la siguiente manera:

```
typedef    unsigned  int    NumEntero;
```

6.2.5. Declaración de prototipos para las funciones

Con el fin de que el compilador pueda determinar si la estructura y la sintaxis en los llamados a las funciones definidas en cada programa es correcta, es necesario establecer la estructura de todas y cada una de las funciones, especificando el tipo de la función, nombre de la función, la cantidad y el tipo de cada uno de los parámetros.

sintaxis de un prototipo:

```
tipo_funciónnombre_función( lista de tipos de los parámetros );
```

Ej. : Dada la función que calcula y retorna el factorial para un número entero que recibe como entrada, el prototipo correspondiente es:

```
int    factorial(int);
```

6.2.6. Declaración de variables globales

Si el programa requiere de variables globales, estas se deben declarar, luego de la definición de las estructuras de datos y constantes y antes de las funciones que conforman el programa; de esta manera las variables tendrán un alcance sobre las funciones que a partir de allí se definan.

6.2.7. Definición de funciones

Se definen las diferentes funciones que conforman el programa, incluyendo siempre la función main que corresponde al programa principal.

Adicionalmente, C como los demás lenguajes modernos permite manejar comentarios o instrucciones no ejecutables, utilizadas para escribir explicaciones o clarificar algún detalle del código, lo cual se logra encerrando el texto explicativo o el conjunto de instrucciones que no se desean ejecutar entre las parejas de caracteres /* al inicio y */ al final, pudiendo abarcar parte de un renglón, un renglón completo o varios renglones.

Ej. : Si se desea adicionar un texto explicativo para una función, se procede de la siguiente manera:

```
/* El objetivo de este conjunto de instrucciones es el de
        establecer el número de  ......   */
```

En las versiones de programación por objetos (conocidas como C++) hay otro método para indicar comentarios, el cual permite su definición para aquellas líneas explicativas que ocupan un solo renglón o parte del mismo, esto se logra adicionando dos caracteres diagonal (//), haciendo que el compilador ignore todo lo que esté escrito a partir de estos caracteres y el final de la línea.

Ej. : Explicar la siguiente asignación

```
x = 10;   // valor inicial de la variable x según .....
```

6.3. Tipos de datos básicos

EL lenguaje C, como la mayor parte de lenguajes de programación permite representar la información en variables cuyo tipo depende de las características que el usuario necesite; los tipos más utilizados son:

int	almacena información numérica entera (sin decimales).
char	almacena todo tipo de caracteres definidos en el código ASCII.
float	almacena información numérica con decimales.
double	almacena información numérica de doble precisión.

El tipo cadena de caracteres es una estructura de datos compuesta, basada en el uso de vectores de caracteres que permite el almacenamiento de letras, números o símbolos para la manipulación y almacenamiento de textos o nombres. (ver capítulo VIII: ARREGLOS)

Existe otro tipo de datos especial denominado *void*, el cual representa variables o funciones sin tipo; es utilizado para representar objetos cuando se desconoce el tipo o clase de información que manejan o en funciones que no retornan valor alguno.

Uno de los usos del tipo *void* se puede observar al comparar estos dos programas:

```
#include <stdio.h>
main ( )
{
    printf (“Versión 1.”);
    getchar( );
}
```

```
#include <stdio.h>
void main (void)
{
    printf (“Versión 2.”);
    getchar( );
}
```

Al poner *void* entre los paréntesis de la definición de una función, se especifica que es una función que no tiene argumentos. No se debe confundir con la llamada a una función, en cuyo caso no se puede utilizar el tipo *void*.

Del mismo modo, al anteponer *void* al nombre de la función en la definición, se está declarando como una función que no devuelve ningún tipo de valor.

Adicionalmente con los tipos básicos, se definen los modificadores de tipo, los cuales se usan para alterar el significado del tipo básico para que se ajuste más preciso según las necesidades.

Modificadores de tipo:

Modificador	**Descripción**	**Tipos a los que aplica**
signed	con signo	int, char
unsigned	sin signo	int, char
long	largo	int, char, double
short	corto	int, char

El uso de *signed* con enteros es redundante aunque está permitido, ya que la declaración hace implícito que los enteros asumen un número con su respectivo signo.

El estándar ANSI (American National Standar Institute) elimina el *long float* por ser equivalente al *double*. Sin embargo, para escribir un *double* con la función *printf* es necesario utilizar el código de formato de *printf:* %lf; que significa: *long float*.

La longitud (y por tanto, también el rango) de los tipos depende del sistema operacional y la implementación del lenguaje que se utilice; no obstante, la siguiente tabla es válida para la mayoría de sistemas:

TIPO	LONGITUD- BITS	VALOR MÍNIMO	VALOR MÁXIMO
unsigned char	8	0	255
signed char	8	-128	127
char	8	-128	127
enum	16	-32,768	32,767
unsigned int	16	0	65,535
short int	16	-32,768	32,767
signed int	16	-32,768	32,767
unsigned short int	8	0	255
signed short int	8	-128	127
int	16	-32,768	32,767
unsigned long	32	0	4,294,967,295
long	32	-2,147,483,648	2,147,483,647
long int	32	-2,147,483,648	2,147,483,647
signed long int	32	-2,147,483,648	2,147,483,647
unsigned long int	32	0	4,294,967,295
float	32	3.4 * (10**-38)	3.4 * (10**+38)
double	64	1.7 * (10**-308)	1.7 * (10**+308)
long double	80	3.4 * (10**-4932)	1.1 * (10**+4932)

6.4.1. Las constantes

Son aquellos valores que se quieren definir cuyo valor no cambia durante la ejecución del programa, ya que corresponden a valores previamente establecidos y que por razones de claridad y facilidad se definen y se les asigna un identificador.

La sintaxis para la definición de una constante es:

```
#define    identificadorvalor
```

Cuyo significado es el de reemplazar en todo el programa el identificador dado, por el respectivo valor asignado.

Ej. : Si se quiere definir una constante con el número pi, se procede de la siguiente manera:

```
#define        PI    3.141592654
```

Una vez definida la constante, es posible utilizarla para la realización de todo tipo de operaciones en las cuales se requiera, referenciándola por su nombre.

Otra forma de definir constantes es utilizando el prefijo *const* asociado al tipo de valor que se quiere almacenar (utilizado principalmente en la programación por objetos).

Ej. : Definir la constante PI como un valor flotante

```
const float PI = 3.141592654;
```

Adicionalmente, existe la instrucción *enum* que permite la definición de un conjunto de constantes de tipo entero *(int)*, dando la posibilidad de generar secuencias, mediante la inicialización de uno de sus componentes.

La sintaxis para el uso del *enum* es la siguiente:

```
enum  nombre_conjunto { lista de constantes } lista de variables;
```

Ej. : Si se quiere definir constantes para los nombres de los meses comenzando en 1:

```
enum { ENERO=1, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO,
       SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE };
```

si no se especifica un valor inicial, el compilador asume cero (0) para el primer elemento, incrementando en uno para los siguientes.

los conjuntos enumerados permiten definir variables que cumplen con el tipo de valores especificados en el conjunto.

6.4.2. Las macros

Son instrucciones que permiten utilizar variables para realizar operaciones, efectuando los respectivos reemplazos en el momento de la compilación.

La sintaxis es similar a la de las constantes, haciendo uso de la instrucción *#define* así:

```
#define   identificador (lista de argumentos) expresión
```

Cuyo objetivo es el de reemplazar las ocurrencias del identificador con su lista de argumentos por la evaluación de la expresión asociada con los argumentos establecidos.

Ej. : Una macro para calcular el mayor entre dos números mediante la instrucción condicional:

```
#define MAYOR(X, Y)   (((X)<(Y)) ? (X) : (Y))
```

Ej. : Una macro para elevar un número al cuadrado:

```
#define ELEVAR2(X)    ((X) * (X))
```

6.4.3. Las variables

Son aquellas porciones de memoria destinadas para el almacenamiento de la información, a las cuales se les asigna un nombre o identificador para su referenciamiento y posterior utilización. Toda variable debe ser definida o asociada con un tipo que determine la clase de información que allí se almacena.

Los nombres que se asignan a las variables tienen una longitud máxima de 32 caracteres (dependiendo de la implementación del compilador), los cuales deben comenzar con una letra, no pueden tener espacios, signos de puntuación ni caracteres especiales tales como: (,), *, -, +, /, &, !, {, }, [,], =, <, >,?.

Es importante tener en cuenta que el lenguaje C, hace diferencia entre las letras mayúsculas y las minúsculas, razón por la cual las variables deben ser escritas y referenciadas siempre de la misma manera, ya que el cambio de una sola de sus letras de minúscula a mayúscula o viceversa, implican nombres o identificadores diferentes.

Ej. : la expresión:

```
int     edad;
```

significa que la variable cuyo nombre es edad, es de tipo entero *(int)* y que por lo tanto puede almacenar números sin decimales entre -32,768 y 32,767.

Ej. : la expresión:

```
float   salario, descuento, iva;
```

significa que las variables salario, descuento e iva, son de tipo flotante (float).

Ej. : la expresión:

```
char   nombre[30];
```

significa que la variable nombre es de tipo cadena de caracteres y permite almacenar una combinación hasta de 30 letras, números y/o símbolos (elementos del código ASCII).

Ej. : los siguientes nombres de variables no son válidos:

cuenta puntos	: las variables no pueden tener espacios.
2pi	: las variables no pueden comenzar con dígitos.
var*	: las variables no pueden tener operadores matemáticos.
for	: las variables no pueden ser palabras reservadas del lenguaje.
la+casa	: las variables no pueden tener operadores matemáticos.

6.5. Operaciones simples

6.5.1. Asignación

La operación más simple que se puede realizar es la asignación, la cual consiste en dar a una variable un determinado valor resultante ya sea de una constante, de evaluar una operación aritmética o de la respuesta que genera la evaluación de una función.

Las asignaciones requieren de una variable como elemento receptor, la cual debe ir en la parte izquierda de la operación, seguida del operador de igualdad (=) y luego el conjunto de primitivas que producen el valor que se asigna.

Ej. : la expresión:

```
x = 124;
```

tiene como significado que a la variable x se le asigne el valor de 124.

Ej. : la expresión:

```
costo = articulo * 1.14;
```

significa que a la variable costo se le asigne el contenido de la variable artículo multiplicado por el valor numérico 1.14

Ej. : la expresión:

angulo = sin(45 * PI / 180.0) ;

tiene como significado que a la variable ángulo se le asigne el resultado de evaluar la función *sin* (seno de un ángulo en radianes) cuyo argumento es la expresión : 45 multiplicado por la constante PI y dividido por 180.

El lenguaje C utiliza los siguientes operadores aritméticos para la realización de cálculos:

OPERADOR	SIMBOLO	SIGNIFICADO
Suma	+	Suma dos valores numéricos de cualquier tipo
Resta	-	Resta dos valores numéricos de cualquier tipo
Multiplicación	*	Multiplica dos valores numéricos de cualquier tipo
División	/	Divide dos valores numéricos de cualquier tipo
Residuo	%	Calcula el residuo de la división de dos números enteros

El lenguaje C realiza una evaluación de las expresiones de izquierda a derecha, pero los operadores tienen diferente precedencia y orden de evaluación, regidos por la siguiente clasificación

<table>
<tr><th>NIVEL</th><th>OPERADOR</th></tr>
<tr><td>1</td><td>( ), [ ], -></td></tr>
<tr><td>2</td><td>!, ~, ++, --</td></tr>
<tr><td>3</td><td>*, /, %</td></tr>
<tr><td>4</td><td>+, -</td></tr>
<tr><td>5</td><td><<, >></td></tr>
<tr><td>6</td><td><, <=, >, >=</td></tr>
<tr><td>7</td><td>==, !=</td></tr>
<tr><td>8</td><td>&</td></tr>
<tr><td>9</td><td>|</td></tr>
<tr><td>10</td><td>&&</td></tr>
<tr><td>11</td><td>||</td></tr>
<tr><td>12</td><td>=, +=, -=, *=, /=, %=</td></tr>
</table>

cuando dos operadores del mismo nivel se encuentran continuos en una expresión, estos son evaluados de izquierda a derecha a medida que se encuentren.

Ej. : La expresión:

```
x = a / b + c * d % e++;
```

es equivalente a:

```
x = (a / b) + ((c * d) % e++);
```

evalúa primero la división (a / b), luego la multiplicación (c * d), luego el residuo del resultado de (c*d) % e++, y por último la suma total.

Ej. : La expresión:

```
x = a+b+c-d+e-f+g
```

es equivalente a:

```
x = (((((a + b) + c) - d) + e) - f) + g
```

Existen otras formas de asignación que permiten incrementar el valor de las variables según las necesidades: incremento ó decremento unitario y operaciones en términos de la misma variable.

6.5.2. Incremento ó decremento unitario

Permite incrementar o decrementar en uno el valor de una variable; se realiza en dos modalidades: prefija o postfija. La primera o prefija incrementa o decrementa en uno el valor de la variable antes de su evaluación, mientras que la segunda, evalúa la expresión y luego aumenta o decrementa el valor de la variable.

La forma general de la modalidad prefija es:

```
++ variable
– – variable
```

Ej. : Si se tienen las siguientes instrucciones:

```
x = 10;
printf("\n Valor de x antes : %d", ++x);
printf("\n Valor de x después : %d", x);
```

El resultado obtenido sería:

```
valor de x antes : 11          /* Se incrementa el valor antes de escribirlo */
valor de x después : 11        /* El valor ya está incrementado */
```

La forma general de la modalidad postfija es:

```
variable++
variable- -
```

Ej. : Si se tiene las siguientes instrucciones:

```
x = 10;
printf("\n Valor de x antes : %d", x++);
printf("\n Valor de x después : %d", x);
```

El resultado obtenido sería:

```
valor de x antes : 10       /* Se escribe antes de incrementar */
valor de x después : 11    /* El valor se incrementa luego de escribirlo */
```

6.5.3. Operaciones en términos de la misma variable

Cuando se requieren operaciones sucesivas sobre una misma variable tales como sumatorias o multiplicatorias, es posible utilizar otro tipo de asignación cuya sintaxis es:

```
variable  operador= expresion;
```

donde operador es uno de los siguientes símbolos u operadores aritméticos : (+, -, *, /, %).

Ej. : Si se desea incrementar en 10 la variable x, se escribe de la siguiente manera:

```
x += 10;
```

Ej. : Si se desea multiplicar por -1 el valor de la variable x:

```
x *= (-1);
```

Ej. : Si se desea restar a una variable x el valor de la expresión z+k-r, se escribe de la siguiente forma:

```
x -= (z+k-r);
```

6.6. Operaciones de salida

Las operaciones de salida permiten desplegar información a través de los dispositivos destinados para tal fin, siendo el mas importante la pantalla o monitor; estas operaciones se realizan mediante las funciones *printf, puts* y *putchar* principalmente.

6.6.1. Función printf

Es el principal medio de despliegue de información siendo necesario especificar el tipo de información que se quiere presentar y las variables o valores asociadas.

El formato general de la operación de salida printf es:

printf(cadena de control, lista de argumentos);

donde la cadena de control es una serie de caracteres encerrados entre comillas dobles, en la cual se especifican uno a uno los formatos de las diferentes variables o expresiones que se desean escribir, así como los diferentes caracteres para acciones especiales. Las variables o expresiones que se desean desplegar se designan en la lista de argumentos, donde es importante tener en cuenta que para cada variable que se involucra en la lista, se debe especificar su respectivo caracter de control, con el fin de determinar el tipo o formato de la variable a escribir, de tal manera que el primer caracter de control se asocia con la primer variable, el segundo caracter con la segunda variable y así sucesivamente.

En las cadenas de control se permite combinar el símbolo de porcentaje % con diferentes caracteres para especificar el tipo de datos que se quiere escribir, de la siguiente manera:

CARACTER	EXPLICACIÓN / FUNCIÓN
%c	como un caracter
%d, %i	como un entero decimal
%u	como un entero decimal sin signo
%o	como un entero octal
%x	como un entero hexadecimal letras minúsculas
%X	como un entero hexadecimal letras mayúsculas
%e	como un número de punto flotante (e)
%E	como un número de punto flotante (E)
%f	como un número de punto flotante sin exponente
%g	modificador corto del formato e
%G	modificador corto del formato E
%s	como una cadena de caracteres
%p	como un apuntador en hexadecimal

Existen otros caracteres de control que tienen como finalidad realizar acciones especiales:

CÓDIGO	SIGNIFICADO O ACCIÓN
\n	cambia a la línea siguiente
\\	escribe el carácter (\)
\"	escribe el carácter doble comilla (")
\’	escribe el carácter comilla simple o apóstrofo
\t	realiza un tabulador horizontal
\a	señal acústica que emite una alerta o campana
\b	realiza un backspace o retroceso
\r	cambia de línea o realiza un retorno de carro
\f	efectúa un cambio de página en la impresora (FORM FEED)
\0	especifica el caracter fin de cadena o caracter nulo
\ddd	constante octal (ddd son tres dígitos como máximo)
\xddd	constante hexadecimal (ddd son tres dígitos como máximo)

Hay otros caracteres no imprimibles que no tienen correspondencia en la tabla anterior. Estos caracteres se pueden utilizar mediante los códigos \ddd, \xddd o simplemente usando el número del código ASCII (Ver Anexo 1).

Ej. : la expresión:

```
printf("%s", "Buenos días");
o
printf("Buenos días");
```

tiene como significado que muestre en la pantalla la cadena de caracteres : Buenos días.

Ej. : las expresiones:

```
int     edad;

edad = 21;
printf("%d", edad);
```

significan que se quiere mostrar en la pantalla la variable entera edad, cuyo valor es el número 21.

Ej. : las expresiones:

```
int     x;
float   y;

x = 122;
y = x / 2;
printf("El valor de X es : %d y el de Y es : %f", x, y);
```

tienen como significado que muestren en la pantalla el valor de las variables x, y que son de tipo *int* y *float* respectivamente, donde %d se asocia con la variable x, %f se asocia con la variable y, dando como resultado:

El valor de X es : 122 y el de Y es : 61

6.6.2. Función puts

Presenta el contenido de una cadena de caracteres en la pantalla, donde la cadena de caracteres debe incluir el centinela de fin de cadena o caracter \0. (ver 8.2 CADENAS DE CARACTERES). La sintaxis de la función puts es la siguiente:

```
puts(cadena);
```

Ej. : Escribir una cadena constante de caracteres.

```
puts("Prueba de las instrucciones de salida con la función puts");
```

Ej. : Declarar una cadena de caracteres, asignarle una expresión en el momento de la declaración y escribir el contenido de la variable.

```
char   nombre[ ] =  "Prueba de la función puts";
puts(nombre);
```

la variable nombre se declara como un vector de 26 caracteres. (25 del texto y uno del fin de cadena).

6.6.3. Macroinstrucción putchar

Es una instrucción definida como una macro sobre la función *putc*, la cual tiene como objetivo presentar en la pantalla el caracter especificado. La sintaxis de la instrucción *putchar* es la siguiente:

```
putchar(caracter)
```

Ej. : Escribir un caracter de cambio de línea.

```
putchar('\n');
```

Ej. : Desplegar la letra A.

```
putchar('A');
```

Ej. : Presentar en la pantalla el caracter cuyo código ASCII es 165.

```
putchar(165);
```

Ej. : Presentar el contenido de la variable ch de tipo char.

```
char   ch;
ch='$';
putchar(ch);
```

6.7. Operaciones de entrada

Las operaciones de entrada permiten básicamente la lectura de datos provenientes del teclado, puertos, archivos y demás dispositivos destinados con este fin. Para la lectura de datos por el teclado se utilizan básicamente las instrucciones *scanf, gets, getchar, getch, getche.*

6.7.1. Función scanf

La operación de entrada básica del lenguaje C, se realiza mediante la función scanf en el cual es necesario especificar para cada parámetro el tipo con que fue definido y/o como se quiere presentar. El formato general es el siguiente:

```
scanf(cadena de control, lista de argumentos);
```

donde la cadena de control es una serie de caracteres encerrados entre comillas dobles, en la cual se especifican uno a uno los formatos de las diferentes variables o expresiones que se desean leer y la lista de argumentos permite especificar las direcciones de las variables que se desean leer o el lugar donde se quiere almacenar la información. Adicionalmente, la función scanf utiliza como delimitador o separador para la entrada de cada una de las variables los caracteres espacio, tabulador o cambio de línea.

La siguiente tabla especifica los caracteres de control que se asocian al caracter porcentaje %:

CARACTER	EXPLICACIÓN / FUNCIÓN
%c	como un caracter
%d, %i	como un entero decimal
%u	como un entero decimal sin signo
%o	como un entero octal
%x, %X	como un entero hexadecimal sin signo
%e	como un número de punto flotante (e)
%E	como un número de punto flotante (E)
%f	como un número de punto flotante sin exponente
%g	modificador corto del formato e
%G	modificador corto del formato E
%s	como una cadena de caracteres
%p	como un apuntador en hexadecimal

Las direcciones de las variables para todos los casos menos las cadenas de caracteres se obtienen anteponiendo el símbolo & a la variable.

Ej. : las expresiones:

```
int     x;
scanf("%d", &x);
```

tienen como significado que se requiere leer un valor de tipo entero, el cual se almacena en la variable x. El símbolo & que se antepone a la variable x tiene como objetivo extraer la dirección en la cual se debe almacenar el valor a leer (se explica con mas detalle en el capítulo XII)

Ej. : las expresiones:

```
float   valor;
int     cantidad;

scanf("%d %f", &cantidad, &valor);
```

permiten leer por intermedio del teclado los valores requeridos para las variables cantidad y valor definidas de tipo *int* y *float* respectivamente.

Ej. : las expresiones:

```
char   nombre[30];

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

permite leer por intermedio del teclado el valores requerido para la variable nombre definida de tipo cadena de caracteres (por ser cadena de caracteres no requiere el símbolo & para extraer la dirección de la variable)

6.7.2. Función gets

Lee una cadena de caracteres proveniente de la consola o teclado y la almacena en la variable especificada como argumento, adicionando automáticamente el centinela de fin de cadena al final de la misma. La sintaxis de la función *gets* es la siguiente:

```
gets(cadena);
```

Ej. : Leer una cadena de caracteres y almacenarla en una variable.

```
char   nombre[30];

       printf("Teclee un nombre : ");
       gets(nombre);
```

6.7.3. Función getch

Retorna un caracter proveniente de la consola o teclado sin mostrarlo en la pantalla y sin la necesidad de presionar la tecla <ENTER> para su realización. Este tipo de instrucción se utiliza principalmente para la lectura de claves de acceso, en los cuales no se quiere visualizar en la pantalla los caracteres digitados.

Ej. : Leer un caracter del teclado y almacenarlo en la variable ch:

```
ch = getch( );
```

6.7.4. Función getche

Retorna un caracter proveniente del teclado mostrándolo en la pantalla sin la necesidad de presionar la tecla <ENTER> para su realización.

Ej. : Leer un caracter del teclado, mostrándolo en la pantalla y almacenarlo en la variable ch:

```
ch = getche( );
```

NOTA: Es importante tener en cuenta que la función scanf lee valores hasta que el usuario presione la tecla <ENTER>, lo cual genera un pequeño problema ya que éste (la tecla <ENTER>) también genera un carácter (‘\n’) el cual se queda a la espera en un lugar que se denomina el buffer del teclado, siendo necesaria su evacuación.

Ej. : Si se necesitara leer un número entero y dos cadenas de caracteres se escribirían las siguientes instrucciones:

```
printf("\n Teclee el número : ");
scanf("%d", numero);
printf("\n Teclee la primer cadena : ");
gets(cadena1);
printf("\n Teclee la segunda cadena : ");
gets(cadena2);
```

al ejecutar las anteriores líneas de programa se puede apreciar que el computador pedirá el número, pero una vez que se ingresa aparecerá el mensaje de la primer cadena e inmediatamente el mensaje de la segunda cadena, sin dar posibilidad de ingresar información para la primer cadena. ¿ Qué pasó ?. Cuando se teclea el número, este se ingresa mediante la tecla <ENTER>, pero la función *scanf* no toma dicho carácter (‘\n’), el cual queda pendiente para una próxima instrucción de entrada, pero cuando se va a ejecutar la lectura de la primer cadena, esta se realiza mediante la instrucción *gets*, la cual lee caracteres hasta que encuentre la tecla <ENTER>, pero como en el buffer del teclado (o la lista de espera) hay un <ENTER>, la función lo toma e inmediatamente continúa con la siguiente instrucción, hasta ejecutar la instrucción *gets* de la segunda cadena.

La solución a este problema se logra eliminando el caracter '\n' que queda pendiente en el teclado, luego de cada lectura con la función *scanf*, lo cual se logra de dos formas diferentes: adicionando una instrucción *getchar( )* para que tome dicho carácter del buffer del teclado o mediante la instrucción *fflush(stdin)*, la cual desocupa el buffer del teclado (ver 11.6.4. Grabación de los Buffers)

6.8. Tipos de operadores

El lenguaje C provee diferentes tipos de operadores agrupados según su funcionalidad y utilizables dependiendo de los requerimientos de programación.

Estos operadores de clasifican en: relacionales, de igualdad, lógicos y de bits.

6.8.1. Operadores Relacionales

Se utilizan para establecer relaciones de orden entre valores numéricos o caracteres. (no funcionan para cadenas de caracteres)

<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

6.8.2. Operadores de igualdad

Se utilizan para determinar la igualdad entre valores numéricos o caracteres. (no funcionan para cadenas de caracteres)

= =	determina si dos valores numéricos o dos caracteres son iguales
!=	determina si dos valores numéricos o dos caracteres son diferentes

6.8.3. Operadores lógicos

Se utilizan como conectores lógicos en expresiones compuestas.

!	Operador negación NOT
&&	Operador AND
||	Operador OR
^	Operador XOR

Las tablas de verdad para cada uno de los operadores lógicos son:

• Operador NOT (!)

OPERADOR	OPERANDO	RESULTADO
NOT	VERDADERO	FALSO
NOT	FALSO	VERDADERO

• Operador AND (&&), (&)

OPERANDO 1	OPERADOR	OPERANDO 2	RESULTADO
VERDADERO	AND	VERDADERO	VERDADERO
VERDADERO	AND	FALSO	FALSO
FALSO	AND	VERDADERO	FALSO
FALSO	AND	FALSO	FALSO

• Operador OR (||), (|)

OPERANDO 1	OPERADOR	OPERANDO 2	RESULTADO
VERDADERO	OR	VERDADERO	VERDADERO
VERDADERO	OR	FALSO	VERDADERO
FALSO	OR	VERDADERO	VERDADERO
FALSO	OR	FALSO	FALSO

• Operador XOR (^)

OPERANDO 1	OPERADOR	OPERANDO 2	RESULTADO
VERDADERO	XOR	VERDADERO	FALSO
VERDADERO	XOR	FALSO	VERDADERO
FALSO	XOR	VERDADERO	VERDADERO
FALSO	XOR	FALSO	FALSO

6.8.4. Operadores de bits

Se utilizan para la manipulación de información a nivel de bits, basados en su representación como números binarios.

>>	Desplazamiento a la derecha (shift-r)
<<	Desplazamiento a la izquierda (shift-l)
!	Operador NOT (negación)
|	Operador OR (disyunción)
&	Operador AND (conjunción)
^	Operador XOR (o exclusivo)

6.9. Bloques de instrucciones

El lenguaje C maneja el concepto de bloque, que permite agrupar un conjunto de instrucciones dando la posibilidad de definir variables con alcance exclusivo para el bloque. Un bloque es todo conjunto de una o más instrucciones encerradas entre llaves ({ , }).

Ej. :

```
< instrucciones >
{
        int x;

        x = (a + b + c) / 2;
        printf("\n %d", x);
 }
< instrucciones >
```

la variable X definida en este bloque solo tiene alcance y validez mientras se ejecuta el correspondiente bloque; una vez terminadas las instrucciones que lo conforman, la variable X y sus valores desaparecen, no siendo posible su acceso.

6.10. Ejercicios propuestos

1) Determine ¿Cuales de los siguientes nombres de variables no son válidos y porqué?

m1a2p3a	1piso
CASA1	%porcentaje
dinero$	else
la-suma	el contador
las_sumas	supervariables

2) Calcule y almacene el promedio de cinco variables.

3) Especifique los diferentes métodos para incrementar en un valor de 10 una variable.

4) Presente los diferentes métodos para leer una variable de tipo carácter.

5) Como se puede escribir mediante la función *printf* una variable de tipo *float* de tal manera que solo presente dos decimales.

6) Dadas las longitudes de los tres lados de un triángulo, determine el tipo que es : equilátero, isósceles o escaleno.

7) Dadas las longitudes de los dos catetos de un triángulo rectángulo, determinar la hipotenusa.

8) Dados dos puntos del plano cartesiano con sus correspondientes componentes X,Y ; determine la distancia entre dichos puntos.

9) Escriba una variable de tipo entero de tal manera que se visualice en octal y en hexadecimal utilizando letras mayúsculas.

10) Establezca una condición basada en operadores relacionales que defina un filtro para una variable, cuando los valores solo sean mayores que cinco y menores que diez, o menores que cero, o diferentes de cincuenta.

7. ESTRUCTURAS DE CONTROL

Con el fin de permitir la implementación de todo tipo de algoritmos y desarrollo de soluciones, el lenguaje C provee instrucciones y estructuras que facilitan el control de la ejecución de los programas, brindando herramientas para la comparación de valores y realización de ciclos repetitivos, agrupados de la siguiente manera:

7.1. Estructuras condicionales

Son aquellas que permiten modificar la secuencia de instrucciones que realiza el computador con el fin de cumplir con un determinado requerimiento o condición.

7.1.1. Condicional Simple

El *if* o estructura condicional simple conocida algorítmicamente como: "si ... entonces...sino...", es aquella estructura condicional que luego de evaluar una expresión y dependiendo de su resultado (VERDADERO o FALSO), permite escoger entre dos posibles alternativas.

La forma general de la expresión condicional es la siguiente:

```
if (expresión)
        sentencias1
else
        sentencias2
```

donde al evaluar expresión, esta debe regresar un valor verdadero o diferente de cero para ejecutar el conjunto de instrucciones albergadas en sentencias1, de lo contrario, se realizan las instrucciones en sentencias2, se debe tener en cuenta que una expresión condicional puede no requerir de la expresión else (de lo contrario), en los casos que ésta no lo amerite.

Si el grupo de instrucciones contenidos en sentencias1 o sentencias2 e mayor que una, es necesario utilizar el concepto de bloque, por lo tanto requiere que dichas instrucciones estén encerradas entre llaves ({ }).

Las condiciones pueden tener uno o varios operadores relacionales; para utilizar 2 o mas operadores relacionales se deben enlazar mediante los operadores lógicos definidos como (&&) para la Y, (||) para la O y (!) para la negación.

Se debe tener en cuenta que para la evaluación de expresiones compuestas cuyo operador lógico principal es la O (||), cuya tabla de verdad indica que la condicional es falsa únicamente cuando las dos expresiones son falsas, cuando la primer expresión resulta verdadera, independientemente del resultado de la segunda, la condicional resulta verdadera, razón por la cual la segunda expresión no es evaluada. Análogamente, en las expresiones compuestas cuyo operador principal es la Y (&&), de cuya tabla de verdad se deduce que la condicional es verdadera únicamente cuando las dos expresiones son verdaderas, si al evaluar la primera expresión su resultado es falso, la segunda no será evaluada y retornará un valor falso para la expresión.

Ej. : Determinar el mayor entre dos números a y b.

```
if (a>b)
        mayor = a;
else
        mayor = b;
```

Ej. : Si un número a es mayor que otro b, intercambiar sus valores

```
if (a>b)
{
        temp = a;
        a = b;
        b = temp;
}
```

Ej. : Si un caracter es una letra minúscula, pasarla a su correspondiente mayúscula

```
if (ch>='a' && ch <='z')
        ch = (char) ((int) ch - 32);

/* 32 es la diferencia que hay entre una letra minúscula y su correspondiente mayúscula
   según la tabla de caracteres ASCII (ver anexo # 1) */
```

Ej. : En las siguientes expresiones, la segunda parte de la condición no se evalúa:

```
h=10;
k=0;
if (h>5 || ++k > 0)
        w=10;
printf("\n El valor de k es : %d", k);
```

El resultado generado es: El valor de k es 0, apreciándose que la instrucción ++k no se realiza, ya que la primera parte de la expresión es verdadera, no siendo necesaria su evaluación.

```
h=10;
k=0;
if (h>25 && ++k > 0)
        w=10;
printf("\n El valor de k es : %d", k);
```

El resultado generado es: El valor de k es 0, apreciándose que como en el ejemplo anterior la instrucción ++k no se realiza, ya que la primera parte de la expresión es falsa, no siendo necesaria su evaluación.

7.1.2. Condicional Múltiple

El *switch* es una estructura condicional de múltiple selección conocida algorítmicamente como: "en caso de". Se utiliza cuando existen mas de dos posibles valores (verdadero o falso) como resultado de la evaluación de una condición.

La forma general del condicional múltiple es:

```
switch (expresión)
{
        case valor1: sentencia-1
                     break;
        case valor2: sentencia-2
                     break;
        case valor3: sentencia-3
                     break;
                     ..
                     ..
        default:     sentencia-n
                     break;
}
```

La cual se interpreta que dependiendo del valor resultante de evaluar la expresión enmarcada en el parámetro del *switch*, la cual debe arrojar un valor numérico entero y si el resultado es valor1, debe realizar las instrucciones agrupadas en sentencia-1 hasta encontrar la instrucción *break*; si el resultado

es valor2, debe realizar las instrucciones en sentencia-2 y así sucesivamente hasta encontrar un valor denominado *default*, el cual se realiza cuando ninguno de los valores cumple con el resultado de la expresión. (el *default* es opcional)

Ej. : Dado un valor n que contiene un número de 1 a 12 que representa los meses del año, escribir el nombre del mes correspondiente.

Utilizando el condicional simple:

```
if (n==1)
        printf("\n Enero");
else
        if (n==2)
                printf("\n Febrero");
        else
                if (n==3)
                        printf("\n Marzo");
                        else
                                ..
                                ..
                                ..
                                        else
                                                printf("\n Diciembre");
```

Utilizando el condicional múltiple:

```
switch (n)
{
        case 1 :        printf("\n Enero");
                        break;
        case 2:         printf("\n Febrero");
                        break;
        case 3:         printf("\n Marzo");
                        break;
        ..
        ..
        default :       printf("\n Diciembre");
                        break;
}
```

NOTA : Los puntos suspensivos no forman parte del lenguaje C, se especifican como continuación de cada una de las instrucciones para cada uno de los meses del año.

El *break* es una instrucción que causa la salida o terminación inmediata de un bloque de instrucciones, continuando con la primer instrucción luego de terminado dicho bloque. También se aplica para la terminación de ciclos tales como: *do .. while, while y for*.

El condicional, ya sea simple o múltiple únicamente es válido para evaluar expresiones cuyo resultado sea numérico (caracteres independientes, son tratados como números mediante su código ASCII), por lo tanto no aplican sobre cadenas de caracteres.

7.1.3. El operador condicional

Algunas veces es necesario evaluar una determinada condición y dependiendo de su resultado se requiere algún tipo de acción específica (asignación, impresión). Para estos casos se tiene la opción de utilizar el operador condicional, cuyo fin es agrupar las instrucciones de comparación y evaluación para el caso verdadero y para el caso falso bajo un solo comando o instrucción. La estructura general del operador condicional es la siguiente:

condición ? expresión1 : expresión2

su significado es:

Si se cumple la condición se ejecutará la expresión1, de lo contrario se realizará la expresión2 y como efecto de borde regresa el resultado de la expresión ejecutada (expresión1 o expresión2 según sea el caso).

Ej. : Determinar si un caracter es un dígito

```
(ch>='0' && ch<='9') ? printf("\n DIGITO"); : printf("\n NO DIGITO");
```

Ej. : Calcular el valor absoluto de un número y asignarlo a otra variable

```
absoluto = (numero < 0) ? -numero : numero;
```

7.2. Estructuras repetitivas

Son aquellas estructuras condicionales utilizadas para realizar un conjunto de instrucciones una o mas veces dependiendo de una determinada condición denominada condición de salida.

7.2.1. La estructura while

Conocida algorítmicamente como el "mientras que" es la instrucción repetitiva mas general del lenguaje C y se utiliza para realizar un conjunto de instrucciones cero o más veces hasta que su expresión condicional o de salida retorne un valor falso.

La forma general es:

```
while (expresión)
      sentencias
```

y se interpreta como la realización de las instrucciones agrupadas bajo sentencias mientras que la evaluación de la expresión sea verdadera.

Ej. : Escribir los números del 1 al 10

```
x = 1;
while ( x <= 10 )
{
        printf("\n%d", x);
        x++;
}
```

Ej. : Calcular el factorial de un número

```
printf("\n Teclee el valor a calcular el factorial: ");
scanf("%d", &n);
fac = 1;
contador = 1;
while (contador <= n)
{
        fac = fac * contador;
        contador = contador + 1;
}
printf("\n El factorial es : %d", fac);
```

7.2.2. La estructura repetitiva for

Conocida algorítmicamente como "para", la cual se realiza cero o mas veces, permitiendo agrupar bajo una sola instrucción: las condiciones iniciales, la condición de salida y las condiciones de avance para cada ciclo.

Su forma general es:

```
 for (expresión-1; expresión-2; expresión-3)
      sentencias
```

La cual se puede interpretar de la siguiente manera: realice como condiciones iniciales el conjunto de instrucciones agrupadas bajo expresión-1 (si es mas de una deben ir separadas por comas), mientras que la expresión-2 arroje un resultado verdadero realice el bloque de sentencias del ciclo, y por cada iteración ejecute las instrucciones de expresión-3 (si es mas de una deben ir separadas por comas).

Las expresiones 1,2, y 3 se pueden omitir parcial o totalmente dependiendo de las necesidades del usuario.

Ej. : Escribir los primeros 20 números impares (2 formas diferentes)

```
for (contador = 1; contador <= 20; contador ++)
        printf("\n%d", 2 * contador - 1);

for (contador=1, impar = 1; contador <= 20; contador ++)
{
        printf("\t %d", impar);
        impar += 2;
}
```

Ej. : Escribir los caracteres de la tabla de códigos ASCII

```
for (contador=1; contador <= 255; contador++)
        printf("\n Caracter # %d = %c", contador, contador);
```

Ej. : Escribir los datos desde el valor inicial de resultado hasta cero, retomando el valor previamente calculado

```
resultado = x + y - z;
resultado += valor;
for ( ; resultado > 0; resultado — — )
        printf("\n El resultado es : %d", resultado);
```

Ej. : Utilizar múltiples argumentos; calcular el factorial de un número n

```
for (factorial=1, contador=1; contador <= n; factorial *= contador, contador ++)
;  // Instrucción nula
printf("\n El factorial es %d", factorial);
```

El punto y coma (;) luego del *for* se asume como la instrucción nula, ya que el ciclo *for* debe realizar al menos una instrucción y como en este caso no hay instrucciones, se debe adicionar la instrucción nula.

Ej. : Realizar un ciclo infinito

```
for ( ; ; )
{
        printf("\n Estoy en un ciclo infinito - ");
}
```

Ej. : Realizar un ciclo hasta que se presione alguna tecla

```
for ( ; ; )
{
```

```
        printf("\n Estoy en un ciclo (presione una tecla para terminar) ");
        if (kbhit( ))
            break;
    }
```

Es posible hacer una analogía directa entre la estructura *for* y la estructura *while*, si se reescribe la forma general del *while* de la siguiente manera:

expresión-1
while (expresión-2)
{
    sentencias
    expresión-3;
}
	for (expresión-1; expresión-2; expresión-3)
	    sentencias

se consigue un comportamiento exactamente igual bajo las dos estructuras.

Se debe tener en cuenta que las condiciones de inicio, avance y terminación en un ciclo *for* pueden ser parcial o totalmente independientes entre sí.

Ej. :

```
    for (cantidad=0, velocidad=1; contador < 10; veces += 2, cantidad — —)
    {
        /*  instrucciones del ciclo */
    }
```

7.2.3. La estructura repetitiva do ... while

Conocida algorítmicamente pomo "haga .. mientras que" o "haga ... hasta", que se realiza una o mas veces debido a que la posición de la condición de salida se encuentra justo al final del ciclo, razón por la cual siempre se ejecutarán la totalidad de las instrucciones del ciclo por lo menos una vez.

Su forma general es:

```
    do
        sentencias
    while  (expresión);
```

Uno de los principales usos es en la presentación de menús, con su respectivo criterio de selección:

Ej. :

```
    do
    {
```

```
            clrscr();
            printf("\n Entrar datos ...................................     1");
            printf("\n Modificar datos ..............................      2");
            printf("\n Mostrar datos .................................     3");
            printf("\n Terminar .......................................    4");
            printf("\n\n");
            printf("\n Teclee su opción : ");
            opcion = getchar( );

        switch (opcion ) {
            case '1' :     entrar( );
                           break;
            case '2' :     modificar( );
                           break;
            case '3' :     mostrar( );
                           break;
        }
    } while (opcion != '4');
```

NOTA : El uso de llaves ({,}) es necesario en todas las estructuras repetitivas cuando el número de instrucciones cobijadas por el ciclo sea mayor que uno, de lo contrario éstas se pueden omitir. El concepto de bloque también se aplica a este tipo de estructuras.

La instrucción *continue* asociada comúnmente a las estructuras repetitivas, provoca un salto incondicional entre un ciclo repetitivo, ejecutando la siguiente iteración del ciclo mas interno.

Ej. : Escribir un ciclo que permita leer 10 parejas de números y calcule el residuo de su división siempre y cuando este se pueda calcular

```
        for (x=1; x <= 10; x++)
        {
            printf("\n Teclee dividendo y divisor : ");
            scanf("%d%d", &dividendo, &divisor);
            if (divisor == 0)
                continue;
            residuo = dividendo % divisor;
            printf("\n El resultado es : %d", residuo);
        }
```

se debe notar en el ejemplo que la instrucción condicional *if* no requiere de su correspondiente *else*, ya que la instrucción *continue* rompe la secuencia de ejecución, pasando a la siguiente iteración del ciclo.

7.3. Ejercicios propuestos

1) Calcular el valor absoluto de un número determinado.

2) Calcular los dos valores que cumplen con la solución de la ecuación cuadrática:
$AX^2 + BX + C = 0$

3) Determinar si un caracter es una letra, un dígito o un símbolo cualquiera.

4) Determinar si un número es o no primo, donde un número primo es aquel que solo es divisibles por si mismo y por la unidad.

5) Determinar si un número tiene raíz cuadrada exacta.

6) Calcular los factores primos que descomponen un determinado número.

7) Determinar si un número entero positivo es par o impar (sin utilizar los operadores % o /)

8) Generar y escribir los primeros n términos de la siguiente serie:
1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - ....

9) Leer las notas definitivas para n estudiantes de una universidad y calcular la siguiente información:

- promedio
- mayor nota
- menor nota
- número de notas aprobadas (mayor o igual a 3.0)

10) Dada una pareja de números calcular el máximo común divisor (MCD) y el mínimo común múltiplo (MCM) entre ellos.

8. ARREGLOS: VECTORES Y MATRICES

Los arreglos son un tipo de datos que utiliza variables indexadas haciendo posible la representación y manipulación de gran cantidad de datos homogéneos bajo un mismo nombre o identificador de variable, haciendo uso de subíndices con el fin de alcanzar las diferentes posiciones de memoria reservadas para cada uno de los datos.

En el lenguaje C los arreglos y los apuntadores se encuentran muy ligados, de tal manera que una variable de tipo arreglo tiene el mismo manejo de un apuntador. (ver capítulo XII, APUNTADORES)

8.1. Arreglos unidimensionales

Los programas normalmente requieren la manipulación de grandes volúmenes de información homogénea (del mismo tipo o clase), siendo necesaria la definición de numerosas variables de igual tipo, lo cual es posible pero dificulta su manejo debido a la personalización de cada dato mediante el nombre de una variable diferente, no siendo posible la relación entre los nombres de las variables y los valores de la información allí almacenada.

Si se desean almacenar tres datos numéricos, sería necesario definir tres variables diferentes:

```
int     valorA, valorB, valorC;
```

implicando un manejo personalizado para cada una de ellas. Si se quisiera leer valores y almacenarlos en dichas variables se debería proceder de la siguiente manera:

```
scanf("%d%d%d", &valorA, &valorB, &valorC);
```

Si el número de datos a almacenar no fuera 3 sino 500, Cómo se debería realizar la definición y lectura de cada una de estas variables ?

Con el esquema anterior la solución sería:

```
int     valor1, valor2, valor3, valor4, valor5, ........, valor 500;
```

NOTA : los puntos suspensivos no forman parte del lenguaje C, siendo utilizados para mostrar la inutilidad del método para realizar este tipo de operación.

Los arreglos surgen como elemento integrador de información, dando la posibilidad de agilizar su definición y manipulación.

El formato general o sintaxis para la definición de las variables tipo arreglo es la siguiente:

```
tipo_datos    nombre_variable[ número_valores ];
```

La interpretación de dicho formato es: defina un conjunto de variables representados por un identificador cuyo nombre es nombre_variable y puede albergar hasta número_valores de variables homogéneas cuyo tipo es tipo_datos. Se debe tener en cuenta que en C los subíndices de los arreglos de las variables se definen desde cero (0) hasta uno menos que el tamaño del arreglo.

Es importante tener claro que debido al caracter enumeratorio de los arreglos en C, comenzando desde la posición o elemento cero [0], es diferente referirse al n-ésimo elemento de un arreglo que al elemento n del mismo. Por ejemplo: el cuarto elemento de un vector se encuentra en la posición cuyo subíndice es el tercero, mientras que el elemento de subíndice cuatro se encuentra en la posición quinta del mismo.

Ej. : Definir un arreglo con nombre vector para almacenar hasta 5 números flotantes.

```
float   vector [5];
```

donde el primer elemento es vector[0] y el último es vector[4].

vector

valor 1	valor 2	valor 3	valor 4	valor 5
vector[0]	vector[1]	vector[2]	vector[3]	vector[4]

Ej. : Una solución basada en vectores para el problema de representar y leer 500 números enteros es :

```
int     valor[500];

for (x=0; x<500; x++)
        scanf("%d", &valor[x]);
```

Existen diferentes formas para inicializar las variables tipo arreglo:

a) Inicialización por asignación:

```
float   vector[5];

vector[0] = 8.0;
vector[1] = 7.0;
vector[2] = 6.0;
vector[3] = 5.0;
vector[4] = 4.0;
```

vector

8.0	7.0	6.0	5.0	4.0
vector[0]	vector[1]	vector[2]	vector[3]	vector[4]

b) Inicialización por definición con tamaño especificado:

```
float   vector[10] = { 8.0, 7.0, 6.0, 5.0, 4.0 };
```

vector

8.0	7.0	6.0	5.0	4.0	0.0	0.0	0.0	0.0	0.0
vector[0]	vector[1]	vector[2]	vector[3]	vector[4]	vector[5]	vector[6]	vector[7]	vector[8]	vector[9]

c) Inicialización por definición sin tamaño especificado:

```
float   vector[ ] = {  8.0, 7.0, 6.0, 5.0, 4.0 };
```

vector

8.0	7.0	6.0	5.0	4.0
vector[0]	vector[1]	vector[2]	vector[3]	vector[4]

si la lista de inicializadores es mas corta que el dimensionamiento del arreglo (caso b.), los elementos restantes se inicializan con valor de cero (0); también se debe notar que en una inicialización sin la definición del tamaño del arreglo (caso c.), el tamaño se ajusta al número de elementos inicializadores.

Para inicializar un arreglo con valores de cero para todas sus componentes es suficiente con:

```
#define          MAXIMO      20

int              arreglo[ MAXIMO ] = { 0 };
```

Es una buena práctica de programación el no especificar valores numéricos constantes al definir el tamaño de un arreglo, siendo aconsejable el uso de constantes definidas mediante la instrucción *#define*.

Ej. : Escribir el programa completo que permita leer n números enteros y mostrarlos en sentido contrario a la lectura.

```
#include <stdio.h>

#define MAXIMO 20      /* El vector puede tener hasta 20 valores */

int     vector[MAXIMO], n, x;

void main ( void )
{
        printf("\n Teclee la cantidad de datos : ");
        scanf("%d", &n);
        for (x=0; x<n; x++)
        {
                printf("\n Teclee dato : ");
                scanf("%d", &vector[x]);
        }
        for (x=n-1; x>=0; x—)
                printf("\n %d ", vector[x]);
}
```

Ej. : Escribir el programa completo para leer y ordenar ascendentemente n números.

```
#include <stdio.h>

#define MAXIMO 20

int     vector[MAXIMO], n, x, y;

void main ( void )
{
        /*  Lectura de datos */
        printf("\n Teclee la cantidad de datos : ");
        scanf("%d", &n);
        for (x=0; x<n; x++)
        {
                printf("\n Teclee dato [%d] : ", x);
                scanf("%d", &vector[x]);
        }

        /* Ordenamiento de los datos mediante el método de la burbuja */
        for (x=0; x<n-1; x++)
                for (y=x+1; y<n; y++)
                        if (vector[x] > vector[y])
                        {
                                temp = vector[x];
                                vector[x] = vector[y];
                                vector[y] = temp;
                        }

        /* Presentación de los resultados */
        printf("\n Los valores ordenados quedan de la siguiente forma : \n");
        for (x=0; x<n; x++)
                printf("\n%d", vector[x]);
}
```

Ej. : Dado un número binario de N bits, encontrar su correspondiente decimal

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

#define MAXIMO 20

int     binario[MAXIMO], decimal, n, x, y;
```

```
void main ( void )

{
                /*  Lectura de datos */

                printf("\n Teclee el número de cifras del binario : ");
                scanf("%d", &n);
                for (x=0; x<n; x++)
                {
                        printf("\n Teclee la cifra [%d] : ", x);
                        scanf("%d", &binario[x]);
                }

                /* Cálculo del número decimal */

                decimal = 0;
                for (x=n-1, y=0  ; x>=0; x– –, y++)
                {
                        decimal += binario[x] * pow(2, y);
                }
                /* Presentación de los resultados */
                printf("\n El correspondiente número decimal es : %d ", decimal);
 }
```

Ej. : Dado un número decimal, encontrar su correspondiente binario

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

        #define MAXIMO 20

        int     binario[MAXIMO], decimal, n, x;

        void main ( void )
        {
                /*  Lectura de datos */
                printf("\n Teclee el número decimal : ");
                scanf("%d", &decimal);
                x=0;
                while (decimal > 0)
                {
                        binario[x] = decimal % 2;
                        decimal = (int) (decimal / 2);
                        x + = 1;
                }
```

```
/* Escritura del número binario, teniendo en cuenta que al generarlo queda en
orden inverso */

            printf("\n El correspondiente número binario es : \n");
            for (x=n-1; x>=0; x — — )
            {
                  printf("%d ",binario[x]);
            }
      }
```

8.2. Cadenas de caracteres

Las cadenas de caracteres son arreglos unidimensionales con elementos tipo caracter. Por convención el lenguaje C utiliza un centinela para la terminación de las cadenas de caracteres denominado caracter de fin de cadena '\0' o caracter nulo, evitando de esta manera el manejo de una variable adicional para determinar la cantidad real de caracteres utilizados por la cadena; dicho caracter ocupa un espacio entre el vector de caracteres siendo necesario tenerlo en cuenta en el momento de especificar su tamaño. Las cadenas constantes se escriben entre comillas dobles.

El formato general de las variables tipo cadena o arreglo de caracteres es la siguiente:

```
char   nombre_variable[ número_elementos ];
```

el cual es equivalente a un vector de caracteres, diferenciándose únicamente en su manejo y es importante aumentar en uno el máximo tamaño del vector para albergar el caracter de fin de cadena.

Existen diferentes formas para la inicialización de una cadena de caracteres:

Ej. : Inicializar la cadena nombre con el texto "prueba"

```
char nombre[ ] = "prueba";
```

es equivalente a:

```
char nombre[ ] = { 'p', 'r', 'u', 'e', 'b', 'a', '\0'} ;
```

nombre

p	r	u	e	b	a	\0
vector[0]	vector[1]	vector[2]	vector[3]	vector[4]	vector[5]	vector[6]

Las cadenas siendo variables tipo arreglo de caracteres requieren de funciones especiales para su manejo ya sea en lectura, escritura, asignación y comparación principalmente.

Las cadenas se leen mediante las funciones *scanf* y *gets*; el scanf requiere del modificador %s y solo permite la lectura de cadenas de caracteres que no contengan el caracter espacio (ASCII 32) debido a que dicha función asume el caracter espacio como elemento separador para los argumentos de lectura. Se debe tener en cuenta que al utilizar *scanf* no es necesario anteponer el símbolo (&) dirección, ya que el nombre de la variable en todo vector es su dirección.

La instrucción *gets* permite la lectura de todo tipo de cadenas hasta encontrar el caracter de fin de línea '\n'. (ASCII 13 ó 10 dependiendo del compilador)

Ej. : Leer un nombre con su apellido y almacenarlo en una sola variable.

a) Utilizando scanf:

```
#include <stdio.h>

#define MAX_NOMBRE        30

void main ( void )
{
        char   nombre[MAX_NOMBRE];
        printf("\n Teclee el nombre y el apellido : ");
        scanf("%s", nombre);
        printf("\n El nombre leído es : %s", nombre);
}
```

En el anterior programa para una entrada de "Carlos Pérez" obtendrá una salida de "Carlos". Para solucionar este problema se deben leer el nombre y el apellido en variables diferentes y luego se procede a unirlas en una sola. (proceso de concatenación)

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

#define MAX_NOMBRE        30

void main ( void )

{
        char   nombre[MAX_NOMBRE], apellido[MAX_NOMBRE];
        printf("\n Teclee el nombre y el apellido : ");
        scanf("%s%s", nombre, apellido); /* Se leen cada uno en una variable diferente */
        strcat(nombre," ");              /* Espacio que separa nombre y apellido */
        strcat(nombre, apellido);        /* Se unen o concatenan el nombre y el apellido */
        printf("\n El nombre leído es : %s", nombre);
}
```

Obteniendo así una salida de "Carlos Pérez".

Hay otra solución para la lectura de cadenas que contengan espacios y es utilizando modificadores de formato, los cuales permiten generar nuevos formatos personalizados para la lectura. De esta manera se puede leer una cadena de caracteres hasta que encuentre el caracter de fin de línea mediante el siguiente llamado a la función *scanf*:

```
scanf("[^\n]", nombre);
```

el cual se interpreta como: lea caracteres hasta que encuentre el caracter de fin de línea o '\n'.

b) Utilizando *gets*:

```
#include <stdio.h>
#define MAX_NOMBRE        30
void main ( void )
{
        char   nombre[MAX_NOMBRE];

        printf("\n Teclee el nombre y el apellido : ");
        gets( nombre);
        printf("\n El nombre leído es : %s", nombre);
}
```

Para la misma entrada del ejemplo anterior se obtendrá una salida de "Carlos Pérez"

La escritura de las cadenas se puede hacer de dos formas diferentes comportándose igual para cualquier caso; mediante la función *printf* con modificador %s o con la instrucción *puts*.

Ej. : Escribir la cadena de caracteres definida en la variable nombre.

```
puts(nombre);
ó
printf("%s", nombre);
```

La comparación de cadenas debido a su manejo como apuntadores no se puede realizar mediante el operador de igualdad (==) ya que la operación tendría como significado verificar si las dos cadenas apuntan a la misma región o posición de memoria, en lugar de determinar si el contenido de cada una de sus posiciones es el mismo. Para realizar dicha comparación es necesario utilizar la función strcmp(cadena1, cadena2), la cual compara el contenido de cada una de las posiciones de dos cadenas de caracteres y regresa valores enteros dependiendo del contenido de las mismas así:

cero : si las dos cadenas son iguales
menor que cero : si cadena1 es menor lexicográficamente que cadena2
mayor que cero : si cadena1 es mayor lexicográficamente que cadena2

Ej. : Leer dos cadenas y comparar su contenido.

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

#define  MAX_NOMBRE        30

void main ( void )
{
        char   nombre1[MAX_NOMBRE], nombre2[MAX_NOMBRE];
        int     valor;

        printf("\n Teclee la cadena 1 : ");
        gets( nombre1);
        printf("\n Teclee la cadena 2 : ");
        gets( nombre2);
        valor = strcmp(nombre1, nombre2);
        if (valor == 0)
                printf("\n Las dos cadenas son iguales");
        else
                if (valor < 0)
                        printf("\n La cadena 1 es menor que la segunda");
                else
                        printf("\n La cadena 1 es mayor que la segunda");
}
```

Análogamente, para copiar el contenido de una cadena en otra no es posible utilizar el operador asignación (=), ya que modificaría el apuntador de la cadena destino, implicando que las dos cadenas se conviertan en una sola, con dos nombres diferentes y en caso de modificar una de ellas la otra se modifica automáticamente (físicamente son la misma). Para solucionar este inconveniente se utiliza la función *strcpy* con parámetros destino y origen, que duplica el contenido de la cadena origen en la cadena destino copiando cada caracter hasta encontrar el centinela de fin de cadena. (también lo copia)

Ej. : leer una cadena y duplicarla.

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

#define MAX_NOMBRE 30

void main ( void )
{
        char   nombre1[MAX_NOMBRE], nombre2[MAX_NOMBRE];

        printf("\n Teclee la cadena  : ");
        gets( nombre1);
```

```
strcpy(nombre2, nombre1);
		printf("\n La cadena original es %s", nombre1);
		printf("\n La cadena duplicada es %s", nombre2);
	}
```

Adicionalmente, el lenguaje C provee un completo conjunto de funciones para la manipulación de cadenas de caracteres tales como:

strncpy(cadena1, cadena2, n)	copia los primeros n caracteres de la cadena2 a la cadena1, dejando el resultado en cadena1.
strcmpi(cadena1, cadena2)	compara las cadenas: cadena1 y cadena2 sin diferenciar mayúsculas o minúsculas, regresando: un valor < 0, si cadena1 < cadena 2 un valor > 0, si cadena1 > cadena2 un valor = 0, si cadena1 == cadena2
strcat (cadena1, cadena2)	concatena el contenido de cadena2 al final de cadena1, dejando el resultado en cadena1.
strncat (cadena1, cadena2, n)	concatena los primeros n caracteres de cadena2 al final de cadena1.
strchr (cadena, caracter)	busca en la cadena la primer ocurrencia de un carácter, retornando un apuntador a dicho carácter en la cadena.
strdup (cadena)	duplica la cadena, regresando un apuntador al duplicado (separa la memoria con malloc)
strtok (cadena, separadores)	busca palabras en la cadena dado el conjunto de separadores (es una cadena)
strstr (cadena1, cadena2)	busca la primer ocurrencia de cadena2 entre la cadena1, regresando un apuntador a la subcadena en la cadena1.
strlwr (cadena)	convierte una cadena a su correspondiente en minúsculas, regresando adicionalmente un apuntador a la cadena.
strupr (cadena)	convierte una cadena a su correspondiente en mayúsculas, regresando adicionalmente un apuntador a la cadena.
strrev (cadena)	Invierte el contenido de la cadena, regresando un apuntador a la cadena.
strlen (cadena)	calcula y retorna la longitud o cantidad de caracteres almacenados en una cadena.

Ej. : Escribir una cadena en orden inverso sin modificarla

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

void main(void)
{
        int cantidad;
        char cadena[10];

        printf("\nDigite la cadena a escribir en orden inverso: ");
        gets(cadena);
        cantidad = strlen(cadena) - 1;  // Se resta uno para ubicarse en el último de la cadena
        while (cantidad >=0)
        {
                printf("%c", cadena[cantidad]);
                cantidad – –;
        }
}
```

Otras funciones importantes en el manejo de cadenas de caracteres, tienen que ver con la conversión de tipos, dando la posibilidad de pasar una cadena compuesta de dígitos (0-9) a un valor numérico y viceversa, mediante las siguientes funciones:

valor=atoi(cadena)	convierte la cadena de caracteres a la variable tipo *int* valor.
valor=atof(cadena)	convierte la cadena de caracteres a la variable tipo *float* valor.
valor=atof(cadena)	convierte la cadena de caracteres a la variable tipo *long* valor.
itoa(valor, cadena, base)	convierte el contenido de la variable valor de tipo *int* a la cadena según la base especificada.
ftoa(valor, cadena, base)	convierte el contenido de la variable valor de tipo *float* a la cadena según la base especificada.
ltoa(valor, cadena, base)	convierte el contenido de la variable valor de tipo *long* a la cadena según la base especificada.

Ej. : Leer una serie de dígitos y almacenarlos en una cadena de caracteres, escribir la cantidad leída en formato numérico, sumarle 35 a dicha cantidad y presentarlo como cadena de caracteres.

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

```
void main(void)
	{
		int numero;
		char cadena[8];

		printf("\nDigite el valor a almacenar : ");
		gets(cadena);
		numero=atoi(cadena);
		printf("\nEl valor leido en numeros es : %d", numero);
		numero += 35;
		itoa(numero, cadena, 10);
		printf("\nEl nuevo valor luego de sumarle 35 es: %s", cadena);
	}
```

8.3. Arreglos bidimensionales

Con el fin de proporcionar estructuras de datos más ágiles, el lenguaje C tiene la posibilidad de definir arreglos de mas de una dimensión, siendo los mas utilizados los de dos dimensiones comúnmente denominados matrices. Las matrices son arreglos que requieren de dos subíndices para el referenciamiento de las diferentes posiciones de memoria allí definidas. Una de las dimensiones especifica el número de filas y la otra el número de columnas.

La definición de un arreglo bidimensional es similar a la de los vectores, siendo necesario añadir el tamaño de la segunda dimensión.

El formato general de las variables tipo arreglo bidimensional o matriz es la siguiente:

```
tipo_datos	nombre_variable[numero_filas][numero_columnas];
```

Ej. : Definir una matriz de 4 filas y 6 columnas para almacenar valores enteros.

```
#define MAXFILAS		4
#define MAXCOLUMNAS	5

int	valores[MAXFILAS][MAXCOLUMNAS];
```

Representación gráfica:

valores

	columna 0	**columna 1**	**columna 2**	**Columna 3**	**columna 4**
fila 0	valores[0][0]	valores[0][1]	valores[0][2]	Valores[0][3]	valores[0][4]
fila 1	valores[1][0]	valores[1][1]	valores[1][2]	Valores[1][3]	valores[1][4]
fila 2	valores[2][0]	valores[2][1]	valores[2][2]	Valores[2][3]	valores[2][4]
fila 3	valores[3][0]	valores[3][1]	valores[3][2]	Valores[3][3]	valores[3][4]

De forma similar a como sucede con los vectores, en las matrices las filas se recorren desde cero hasta el número de filas - 1 y las columnas desde cero hasta número de columnas -1.

El proceso para recorrer los elementos de una matriz involucra la realización de dos ciclos anidados, el primero para moverse sobre las filas y el segundo sobre las columnas. Este proceso es útil para la lectura, el procesamiento y la escritura de cada uno de los elementos que conforman la matriz.

El recorrido por filas con respecto a la matriz representada gráficamente, presenta la siguiente variación en los subíndices:

valores[0][0], valores[0][1], valores[0][2], valores[0][3], valores[0][4], valores[1][0], valores[1][1], valores[1][2], valores[1][3], valores[1][4], valores[2][0], valores[2][1], valores[2][2], valores[2][3], valores[2][4], valores[3][0], valores[3][1], valores[3][2], valores[3][3], valores[3][4].

Lo cual se logra mediante dos ciclos así (su escritura):

```
for (x=0; x<4; x++)
        for (y=0; y<5; y++)
                printf("%d", valores[x][y]);
```

Las matrices como los vectores se pueden inicializar en el momento de la definición, especificando cada uno de sus elementos fila por fila encerrando cada fila entre llaves { }, y separados por coma (,).

Ej. : Definir e inicializar una matriz con el nombre datos de dos filas y tres columnas con los valores de: 2, 4 y 6 para la primer fila y 1, 3 y 5 para la segunda.

```
int  datos[2][3] = {{ 2, 4, 6 }, {1, 3, 5 }};
```

También se pueden inicializar la totalidad de elementos de una matriz con el mismo valor, asignando el único valor entre doble juego de llaves {{}}, de la siguiente manera:

Ej. : Definir e inicializar una matriz con el nombre tablero de ocho filas y ocho columnas de números flotantes con un valor de cero (0), en todas sus posiciones:

```
float  tablero[8][8] = {{ 0.0 }};
```

Adicionalmente, el lenguaje C permite el manejo de arreglos de mas de dos dimensiones, dependiendo del compilador y la versión, facilitando la creación de estructuras de datos mas complejas que permitan modelar y solucionar un conjunto mas grande de problemas.

Ej. : Definir un arreglo tridimensional de cuatro números enteros de largo, cinco de ancho y seis de profundidad.

```
Int     cubo[4][5][6];
```

En la anterior estructura se pueden almacenar valores (obtenidos del teclado) haciendo uso de tres ciclos anidados así:

```
for (x=0; x<4; x++)
  for (y=0; y<5; y++)
    for (z=0; z<6; z++)
      scanf("%d", &cubo[x][y][z]);
```

8.4. Ejercicios resueltos

Ej. : Definir y leer una matriz con M filas y N columnas (M y N menores que 10) para el almacenamiento de números flotantes, y una vez leídos efectuar la suma de todos los elementos, escribir los elementos de la diagonal principal y encontrar el mayor número almacenado en la matriz.

```
#include <stdio.h>

#define MAXIMO  10

void main ( void )
{
        float   matriz[MAXIMO][MAXIMO], suma, mayor;
        int     m, n, x, y, t;

        /* Lectura de las dimensiones a utilizar de la matriz */
        printf("\n Teclee el  número de filas y el número de columnas : ");
        scanf("%d%d", &m, &n);

        /* Lectura de los elementos de la matriz */
        for (x=0; x<m; x++)
                for (y=0; y<n; y++)
                {
                        printf("\n Teclee el elemento [%d][%d] : ", x , y);
                        scanf("%f", &matriz[x][y]);
                }

        /* Cálculo de la suma de cada uno de los elementos */
        suma = 0;
        for (x=0; x<m; x++)
                for (y=0; y<n; y++)
                        suma += matriz[x][y];

        /* Presentación del resultado de la suma */
        printf("\n La suma de las componentes de la matriz es : %f", suma);

        /* Elementos que conforman la diagonal */
```

```
        printf("\n Elementos de la diagonal : ");
        if (m>n)
                t = n;
        else
                t = m;
        for (x=0; x<t; x++)
                printf("\n%f", matriz[x][x]);

        /* Encuentra y escribe el mayor valor de la matriz */
        mayor = matriz[0][0];
        for (x=0; x<m; x++)
                for (y=0; y<n; y++)
                        if (mayor <  matriz[x][y])
                                mayor = matriz[x][y];
        printf("\n El mayor número de la matriz es : %f", mayor);
}
```

Ej. : Definir y leer una matriz con M filas y N columnas (M y N menores que 10) para el almacenamiento de números enteros, y una vez leídos escribir los elementos de la periferia (frontera) en sentido de las manecillas del reloj, comenzando en el elemento [0][0], escribiendo la primera fila de izquierda a derecha, luego la columna de la derecha de arriba hacia abajo, luego la fila de debajo de derecha a izquierda y por último la primer columna de abajo hacia arriba.

```
#include <stdio.h>

#define MAXIMO  10

void main ( void )
{
        int     matriz[MAXIMO][MAXIMO];
        int     m, n, x, y;

        /* Lectura de las dimensiones a utilizar de la matriz */
        printf("\n Teclee el  número de filas y el número de columnas: ");
        scanf("%d%d", &m, &n);

        /* Lectura de los elementos de la matriz */
        for (x=0; x<m; x++)
                for (y=0; y<n; y++)
                {
                        printf("\n Teclee el elemento [%d][%d] : ", x , y);
                        scanf("%d", &matriz[x][y]);
                }

        /* Cálculo de la periferia de la matriz */
```

```
        /* Primero se escribe la fila de arriba (fila 0) */
        for (x=0; x<n; x++)
                printf("%d", matriz[0][x]);

        /* Segundo se escribe la columna de la derecha (columna m-1) de arriba hacia abajo,
        quitándole el primer elemento que ya fue escrito */
        for (x=1; x<m; x++)
                printf("%d", matriz[x][n-1]);

        /* Tercero se escribe la fila de abajo (fila m-1), de derecha a izquierda, quitándole el
        último elemento de la fila, el cual se escribió en el segundo paso (como la última fila es n-1, la
        penúltima es n-2) */
        for (x=n-2; x>=0; x - -)
                printf("%d", matriz[m-1][x]);

        /* Cuarto se escribe la columna de la izquierda (columna 0) de abajo hacia arriba,
        comenzando en la  penúltima fila (como la última es m-1, la penúltima es m-2) */
        for (x=m-2; x>=1; x - -)
                printf("%d", matriz[x][0]);
    }
```

El anterior programa para una entrada de 3x3 al número de filas y columnas y valores de 1,2,3,4,5,6,7,8,9, genera como resultado: 1,2,3,6,9,8,7,4.

8.5. Ejercicios propuestos

1) Leer una serie de elementos, almacenarlos en un vector de números enteros e invertirlo.

2) Calcular la multiplicatoria de todos los elementos que conforman un vector.

3) Hallar el producto punto de dos vectores.

4) Hallar la longitud de una cadena de caracteres sin utilizar la función strlen.

5) Pasar un número de decimal a hexadecimal.

6) Pasar un número de hexadecimal a decimal.

7) Sumar dos matrices de tamaño M x N.

8) Hallar el determinante de una matriz de N x N.

9) Hallar la inversa de una matriz de N x N.

10) Buscar el elemento que más se repite dentro de una matriz de N x N

9. MANEJO DE REGISTROS

Se conoce como registro o estructura al conjunto de variables relacionadas de diferentes tipos agrupadas bajo un único nombre o identificador, donde cada uno de los elementos o variables allí almacenadas se denomina campo. Su principal utilidad radica en la posibilidad de construir estructuras de datos de mayor complejidad, tales como: listas, colas, pilas, árboles, grafos, etc. permitiendo almacenar información de una forma mas clara y ordenada, facilitando su manipulación mediante arreglos, listas y posteriormente archivos.

9.1. Declaración

Los registros se definen mediante la palabra reservada struct con la siguiente disposición o formato:

```
struct nombre_estructura {
        definición de los miembros, campos o variables de la estructura
} lista de variables;
```

cuya interpretación es : defina una estructura cuyo nombre es nombre_estructura, conformada por la lista de variables (campos) allí referenciadas; declarando una lista de variables (opcional) que tienen como tipo la estructura previamente definida.

También se puede definir como un nuevo tipo haciendo uso de la instrucción typedef, con el siguiente formato:

```
typedef struct {
        definición de los miembros, campos o variables de la estructura
} nuevo_tipo;
```

donde nuevo_tipo pasa a ser un nuevo tipo de datos del lenguaje, permitiendo la declaración de variables, así:

```
nuevo_tipo  lista de variables;
```

Ej. : Definir una estructura llamada estudiante que permita almacenar las variables nombre de 30 caracteres, código de 6 caracteres, edad como un entero y promedio como un flotante; adicionalmente declare dos variables denominadas prueba1 y prueba2 que sean de la estructura definida.

```
struct estudiante {
        char   nombre[31];
        char   codigo[7];
        int    edad;
        float  promedio;
} prueba1, prueba2;
```

Cuya representación gráfica se asemeja a:

Si posteriormente se desean declarar variables de una estructura previamente definida, la sintaxis es la siguiente:

```
struct nombre_estructura  lista de variables;
```

Ej. : Declarar las variables temporal1 y temporal2 cuyo tipo es la estructura estudiante definida en el ejemplo anterior:

```
struct estudiante temporal1, temporal2;
```

9.2. Inicialización de estructuras

Las estructuras pueden inicializarse en el momento de la declaración de las variables, asignando listas de inicialización, mediante la siguiente asignación:

```
struct nombre_estructura  nombre_variable = { lista de valores } ;
```

Ej. : Inicializar una variable temporal tipo estudiante con los valores: "Marcela Chavez" para el nombre, "495012" para el código, 19 para la edad y 4.3 para el promedio de notas:

```
struct  estudiante  temporal = {  "Marcela Chavez", "495012", 19, 4.3 };
```

9.3. Acceso a los campos de las estructuras

Teniendo como base la definición de una estructura en la cual se especifica que mediante un identificador o variable se tiene acceso a los diferentes componentes o campos de una estructura, se requiere del nombre de la variable seguida del caracter punto (.) y a continuación el nombre del campo sobre el cual se quiere operar:

```
nombre_variable.nombre_del_campo
```

Ej. : Para leer cada uno de los campos de un registro denominado temporal, cuyo tipo es la estructura estudiante previamente definida, se tiene el siguiente programa:

```
#include <stdio.h>

struct estudiante {
        char   nombre[30];
        char   codigo[7];
        int    edad;
        float  promedio;
} temporal;

void main (void)
{
        printf("\n Teclee el nombre : ");
        gets(temporal.nombre);
        printf("\n Teclee el código : ");
        gets(temporal.codigo);
        printf("\n Teclee la edad : ");
        scanf("%d", &temporal.edad);
        printf("\n Teclee el promedio : ");
        scanf("%f", &temporal.promedio);
}
```

Si se le asigna información al ejemplo anterior tal como: Marcela para el nombre, 495012 para el código, 19 para la edad y 4.3 para el promedio, su representación sería la siguiente:

temporal

```
nombre : Marcela
codigo : 495012
edad :    19
promedio : 4.3
```

los campos o valores allí almacenados se pueden alcanzar mediante las siguientes expresiones:

```
temporal.nombre
temporal.codigo
temporal.edad
temporal.promedio
```

El contenido de una estructura se puede copiar de una variable a otra, mediante la operación de asignación, siempre y cuando las dos variables sean del mismo tipo, sin necesidad de copiar campo por campo.

Ej. : dada la estructura estudiante del ejemplo anterior, si la variable temporal se desea copiar a otra variable del mismo tipo, se procede de la siguiente manera:

```
struct estudiante   copia, temporal;

        strcpy(temporal.nombre,  "Marcela");
        strcpy(temporal.codigo,  "495012");
        temporal.edad = 19;
        temporal.promedio = 4.3;

        copia = temporal;
```

9.4. Ejercicios resueltos

1) Escriba un programa que lea y almacene los datos para n estudiantes según la estructura de estudiante previamente definida, los ordene ascendentemente según la nota y escriba aquellos con edad mayor o igual a 18 y promedio inferior a 3.5

```
/* ——————————————————————————————————————————————————————————————————————
Programa para leer y almacenar datos para N estudiantes, dando la posibilidad de
ordenarlos ascendentemente según la nota y escribir aquellos con edad mayor
o igual a 18 y con promedio inferior a 3,5.
—————————————————————————————————————————————————————————————————————— */
#include <stdio.h>

#define MAXIMO  30

/*——————————————————————————————————————————————————————————————————————*/

struct estudiante {
        char   nombre[30];
        char   codigo[7];
        int    edad;
        float  promedio;
} estudiantes[MAXIMO];

/* ——————————————————————————————————————————————————————————————————————
Función que dada la cantidad de elementos, realiza la lectura de cada uno de sus
campos (nombre, codigo, edad y promedio), sobre el arreglo estudiantes.
—————————————————————————————————————————————————————————————————————— */
void Leer(int cantidad)
{
        int x;

 for (x=0; x<cantidad; x++)
 {
        printf("\n Teclee el nombre : ");
        gets(estudiantes[x].nombre);
```

```
        printf("Teclee el código : ");
        gets(estudiantes[x].codigo);
        printf("Teclee la edad : ");
        scanf("%d", &estudiantes[x].edad);
        printf("Teclee el promedio : ");
        scanf("%f", &estudiantes[x].promedio); getchar();
 }
}
/* ——————————————————————————————————————————————————— */

/* ———————————————————————————————————————————————————
Función que dada la cantidad de elementos utilizados, realiza un ordenamiento
ascendente sobre el arreglo estudiantes con respecto al campo nota.
——————————————————————————————————————————————————— */
void Ordenar(int cantidad)
{
        int x, y;
        struct estudiante temporal;

        for (x=0; x<cantidad - 1; x++)
                for (y=x+1; y<cantidad; y++)
                        if (estudiantes[x].promedio>estudiantes[y].promedio)
                        {
                                temporal = estudiantes[x];
                                estudiantes[x] = estudiantes[y];
                                estudiantes[y] = temporal;
                        }
}
/* ———————————————————————————————————————————————————*/

/* ———————————————————————————————————————————————————
Función que dada la cantidad de elementos utilizados, muestra aquellos estudiantes
con edad mayor o igual a 18 y cuyo promedio es inferior a 3,5. Adicionalmente,
calcula el promedio total del curso.
———————————————————————————————————————————————————*/
void Mostrar(int cantidad)
{
        int x;
        float suma=0, promediototal;

        printf("\n Estudiantes con edad mayor o igual a 18 y promedio inferior a 3,5 :");
        for (x=0; x<cantidad; x++)
        {
                suma += estudiantes[x].promedio;
                if (estudiantes[x].edad >= 18 && estudiantes[x].promedio < 3.5)
                {
```

```
                    printf("\n Nombre :%s,  Codigo :%s",
                    estudiantes[x].nombre, estudiantes[x].codigo);
                }
        }
        promediototal = suma / cantidad;
        printf("\n El promedio del grupo es : %6.2f", promediototal);
}
/* ------------------------------------------------------------------ */

/* ------------------------------------------------------------------
Función principal que lee la cantidad de estudiantes y se encarga de llamar a las
demás funciones encargadas de realizar las diferentes operaciones del sistema
tales como: leer los datos de los estudiantes, ordenarlos ascendentemente y por
último presentarlos según los criterios requeridos.
------------------------------------------------------------------ */
void main( void )
{
        int  numero;

        printf("Cantidad de estudiantes  ? : ");
        scanf("%d", &numero); getchar();
        Leer(numero);
        Ordenar(numero);
        Mostrar(numero);
}
/* ------------------------------------------------------------------ */
```

9.5 Ejercicios propuestos

1) Explique ¿Cual es la diferencia entre un registro definido con *struct* y otro definido con *typedef struct* ?

2) Especifique, ¿Cómo se puede escribir el contenido de un registro mediante el uso de la función *printf*?

3) Cuando se leen los campos de una estructura y hay cadenas de caracteres, ¿Cómo se debe realizar la lectura?, ¿Requiere del indicador de dirección (&)? Explique su respuesta.

4) Escriba un programa que mediante el uso de registros permita almacenar la información de dos polinomios de cualquier grado, realice la suma y por último evalúe dicha suma para un valor determinado de la variable.

5) Especifique la estructura de un registro para almacenar los datos de un producto para un inventario en un almacén; nombre, código, valor unitario, cantidad, ubicación.

6) Escriba un programa que utilizando el registro definido en el ejercicio anterior permita manejar el inventario de un almacén, dando la posibilidad de realizar las siguientes acciones:

• ingresar, retirar y modificar productos.
• conocer los atributos de un producto.
• conocer la totalidad de productos y su valor total.

7) Defina dos tipos de registros: uno para almacenar tiempos (HH:MM:SS) y el otro para almacenar fechas (DD:MM:AAAA); una vez definidos, realice una función que dados dos instantes en el tiempo con tiempo y fecha, calcule la diferencia entre ellos.

8) Escriba un programa que haciendo uso de registros, lea y almacene la información del conjunto de empleados de una fábrica y permita calcular el valor que se le debe pagar a cada uno, teniendo en cuenta: Datos básicos del empleado, sueldo base, número de horas extras trabajadas, valor de la hora extra, descuentos, etc. Se debe presentar un listado ordenado según el valor a pagar en orden descendente con el nombre, cédula y valor a pagar.

9) Escriba un programa que mediante registros pueda almacenar y luego presentar para los números del 1 al 100, su correspondiente valor en decimal, binario y hexadecimal.

10) Defina el contenido de los registros necesarios para almacenar la información requerida en una biblioteca para los libros allí guardados, de tal manera que se puedan realizar las siguientes operaciones:

• ingresar nuevos libros
• retirar libros
• prestar un libro
• regresar un libro
• conocer el estado de un libro (préstamo, ubicación, número topográfico)
• clasificar los libros según materia

10. FUNCIONES

Todo programa se debe dividir en varios módulos o partes de tal manera que faciliten su entendimiento, desarrollo y posterior mantenimiento. Los módulos se denominan procedimientos o funciones y tienen como primordial fin reutilizar el código escrito de tal manera que un conjunto de instrucciones puedan ser utilizadas una o varias veces, en un mismo programa, teniéndose que escribir una sola vez y dando un mayor orden a la escritura del programa.

10.1. Declaración de funciones

El lenguaje C provee al programador de un mecanismo para agrupar conjuntos de instrucciones independientes, representadas bajo un único identificador siendo posible recibir y regresar valores del medio externo, cuyo fin es el de realizar un determinado proceso, permitiendo su utilización desde distintos puntos del programa con valores diferentes.

La declaración general de una función es:

```
tipo_resultado         nombre_función ( lista de parámetros )
{
    declaraciones
    instrucciones
    return(valor_retorno);
}
```

donde tipo_resultado es el tipo de valor que regresa la función (cuando no regresa ningún valor en el llamado se utiliza el tipo *void*), nombre_función es el identificador o nombre de la función que lo identifica y permite su llamado en el programa (debe ir de acuerdo con el papel o función que desempeña), lista de parámetros es la lista de variables (tipo, nombre de variable) que permiten la entrada y salida de datos e información, declaraciones son aquellas variables locales a la función y que se utilizan únicamente para el desarrollo de la función, instrucciones son las diferentes líneas de código que se realizan en la función y la instrucción *return* es la que finaliza la ejecución de la función retornando el valor dado en su argumento; el valor de retorno tiene que ser del mismo tipo que el tipo_resultado definido para la función. Si el tipo_resultado se define como void o tipo nulo, la instrucción *return* no es necesaria, pero puede ser utilizada sin argumentos.

10.2. Tipos de parámetros

Los parámetros son los elementos que permiten la entrada o salida de información cuando se utilizan funciones; el nombre que se da a un parámetro es independiente del nombre de la variable con la cual se realiza el llamado, pero los tipos deben ser iguales.

Existen parámetros de dos tipos: entrada y salida,

a. Los de entrada denominados por valor son aquellos que se utilizan para ingresar información a las funciones, no siendo útiles para regresar información.

Ej. : Escribir una función que reciba dos números enteros, los sume y presente el resultado.

```
void sumar (int a, int b)
{
        int   suma;

        suma = a + b;
        printf("\n La suma es %d", suma);
}
```

El llamado desde la función principal sería:

```
void main ( void )
{
        int x, y;
        printf("\n Teclee los dos números : ");
```

```
        scanf("%d%d", &x, &y);
        sumar(x, y);
}
```

b. Los parámetros de salida denominados por referencia son aquellos que se comportan como un apuntador y tienen como principal característica que toda modificación que se realice sobre la información allí almacenada, esta se refleja sobre la variable a la cual corresponde el parámetro con el cual fue llamada la función.

A aquellas variables que no correspondan a arreglos (vectores o matrices) y que se requieren por referencia, es necesario que en el llamado de la función se calcule la dirección de almacenamiento o posición que ocupa en la memoria de tal manera que pueda regresar los valores calculados. Esta característica se logra anteponiendo el símbolo (&) al nombre de la variable en su llamado. Para el manejo del parámetro en la función, se debe referenciar la variable anteponiendo el símbolo (*) al nombre de la variable. (ver capítulo XII, sección 12.2. OPERADORES CON APUNTADORES)

La definición de los operadores dirección (&) y contenido (*) es la siguiente:

> &nombre_variable : calcula la posición de memoria o dirección en la cual fue declarada la variable cuyo nombre es nombre_variable.
>
> *dirección_variable : calcula el contenido de la dirección especificada por la variable cuyo nombre es dirección_variable.

Se debe tener en cuenta que los arreglos de todo tipo (vectores, matrices y cadenas de caracteres) se manejan como apuntadores, por lo tanto su comportamiento en las funciones cuando se usa como parámetros siempre será el de variables por referencia; por lo tanto no se le debe anteponer ningún tipo de símbolo en el llamado o en su posterior utilización

Ej. : Sumar dos valores y regresar el resultado en un parámetro.

```
void sumar(int a, int b, int *suma)
{
        *suma = a + b;
}
```

El llamado a la función se realiza de la siguiente manera:

```
void main ( void )
{
        int   x, y, resultado;
        printf("\n Teclee los números : ");
        scanf("%d%d", &x, &y);
        sumar(x, y, &resultado);
        printf("\n La respuesta es %d", resultado);
}
```

10.3. Obtención de resultados

Las funciones regresan valores de tres formas diferentes:

a) En el nombre de la función, especificando el tipo de valor a retornar e incluyendo la instrucción return (expresión) que retorna el valor calculado. El llamado de la función se debe realizar desde una expresión, ya sea desde una asignación, o un llamado de función de tal manera que utilice el valor retornado. Este método devuelve un único valor.

b) Por medio de los parámetros, determinando su uso ya que estos pueden ser de solo entrada, o entrada/salida, siendo posible retornar tantos valores como parámetros se especifiquen en la función.

c) En variables globales, si una función modifica directamente una variable global, al terminar la ejecución de ésta, la variable global modificada se mantiene.

Ej. : Escribir una función que calcule el factorial de un número y lo retorne en su nombre.

```
int factorial( int num )
{
        int     respuesta;

        respuesta = 1;
        while (num > 0)
        {
                respuesta *= num;
                num - -;
        }
        return(respuesta);
}
```

El llamado de la función se haría de la siguiente manera:

```
        valor = factorial(numero);
```

almacenando el resultado de calcular el factorial a la variable numero y almacenándola en la variable valor.

Ej. : Escribir una función que calcule el factorial de un número y lo retorne en uno de sus parámetros.

```
void factorial( int   num,  int   *respuesta )
{
        *respuesta = 1;
        while (num > 0)
```

```
        {
                *respuesta *= num;
                num - -;
        }
 }
```

El llamado de la función se haría de la siguiente manera:

```
factorial(numero, &valor);
```

almacenando el resultado de calcular el factorial a la variable numero y almacenándola en la variable valor.

Ej. : Escribir una función que calcule el factorial de un número y lo retorne en una variable global.

```
int  valor;      // Variable global definida al principio del programa

void factorial( int   num )
{
        int respuesta = 1;
        while (num > 0)
        {
                respuesta *= num;
                num - -;
        }
        valor = respuesta;
}
```

El llamado de la función se haría de la siguiente manera:

```
factorial(numero);
```

almacenando el resultado de calcular el factorial a la variable numero y almacenándola en la variable valor, definida globalmente al comienzo del programa.

Nota: Es importante tener en cuenta que este último método aunque es válido, no es recomendable ya que la posibilidad de confusiones o pérdida de información por previo uso de la variable global, pueden afectar seriamente el funcionamiento del programa.

10.4 Recursión

Método o proceso mediante el cual se define o resuelve un problema en términos del mismo, cumpliendo las siguientes propiedades:

- No debe generar una secuencia infinita de llamados a él mismo.
- Siempre debe existir una condición de escape o salida que permita terminar el ciclo recursivo.
- Cada llamado recursivo de la función debe acercar a la solución del problema, garantizando que se encuentre la solución.

Matemáticamente son muchas las funciones que se definen mediante este método:

a) Calcular el factorial de un número n:

```
        si (n==0)            Resultado :  1
 n! =
        si (n>0)             Resultado :  n * (n-1)!
```

b) Elevar X a la n-ésima potencia:

```
        si (n==0)            Resultado :  1
x^n =
        si (n>0)             Resultado :  x * x^(n-1)
```

c) Calcular el n-ésimo término de la serie de fibonacci:

```
            si (n==0 ó n==1)      Resultado :  1
fibo(n) =
            si (n>1)              Resultado :  fibo(n-1)+fibo(n-2)
```

Computacionalmente, este tipo de soluciones se implementan haciendo uso de funciones y el concepto de recursión aplicado a ellas.

Ej. : Escribir una función que encuentre el factorial de un número determinado

```
int factorial( int numero )
{
        if (numero == 0)
                return (1);
        else
                return (numero * factorial( numero - 1 ));
}
```

Ej. : Escribir una función que eleve un número a una potencia determinada

```
int elevar (int base, int exponente )
{
        if  ( exponente == 0)
```

```
                return (1);
        else
                return ( base * elevar(base, exponente-1));
}
```

Ej. : Escribir una función que calcule el n-ésimo término de la serie de fibonacci

```
int fibonacci( int termino )
{
        if  (termino == 1 || termino == 2)
                return (1);
        else
                return( fibonacci(n-1) + fibonacci(n-2) );
}
```

Ej. : Escribir una función que busque un número determinado entre un arreglo previamente ordenado y determine si este existe o no. (haciendo uso de la búsqueda binaria)

```
int busqueda (int arreglo[ ], int valor, int bajo, int alto)
{
        int medio;
        if (bajo > alto)
                return (FALSO);
        else
        {
                medio = (int) (bajo + alto) / 2;
                if (arreglo[medio] == valor)
                        return (VERDADERO);
                else
                        if (valor < arreglo[medio])
                                return( busqueda( arreglo, valor, bajo, medio - 1 ));
                        else
                                return(busqueda( arreglo, valor, medio + 1, alto ));
        }
 }
```

para esta solución es necesario definir las siguientes constantes:

```
#define  VERDADERO          1       /* Constante que representa el valor verdadero */
#define  FALSO              0       /* Constante que representa el valor falso */
#define  MAXIMO             30      /* Número máximo de elementos del vector */
```

para un vector de N elementos (máximo 30) el segmento del programa principal sería:

```
void main ( void )
{
        int  vector[MAXIMO], cantidad, numero;

        printf("\n Cual es la cantidad de elementos : ");
        scanf("%d", &cantidad);
        for (x=0; x < cantidad ; x++)
        {
                printf("\n Teclee el elemento[%d] : ", x+1);
                scanf("%d", &vector[x]);
        }
        printf("\n Número a buscar : ");
        scanf("%d", &numero);
        if (busqueda(vector, numero, 0, n-1) == VERDADERO)
                printf("\n El número si esta en el vector");
        else
                printf("\n El número no esta en el vector");
}
```

10.5. Ejercicios propuestos

1) Escriba una función que reciba como parámetro un número y que calcule sus factores primos y los regrese en un vector como otro parámetro con su respectivo tamaño.

2) Verifique el siguiente programa el cual consta de una función que tiene como objetivo leer un valor numérico en la mitad de la pantalla y regresarlo mediante un parámetro, y determine si realmente cumple con el objetivo, y de no serlo explique ¿Cuál es el problema?

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

void  LeerNumero(int  *numero)
{
        gotoxy(10,14);
        printf("Teclee un valor numérico : ");
        scanf("%d", numero);
}

void  main(void)
{
        int   x;

        LeerNumero(&x);
        printf("\n El número leído es : %d", x);
}
```

3) Escriba una función que reciba un número entero y regrese dicho número con sus dígitos invertidos.

Ej. : Dado el número 1998, el resultado será el 8991.

4) Escriba una función que reciba dos valores numéricos y resuelva el problema de calcular el combinatorio, mediante el llamado a funciones que hallen el factorial y la división.

5) Escriba una función que recursivamente determine si un valor es par o impar (sin hacer uso de ciclos, del operador residuo, ni divisiones).

6) Escriba una función que reciba como parámetro dos números enteros y regrese en el nombre de la misma el Máximo Común Divisor (MCD) entre ellos.

7) Teniendo en cuenta que en el lenguaje C, todo programa se conforma de funciones y que la función principal denominada main, cumple con todas las características de las mismas, escriba un programa que calcule el factorial de un número determinado, el cual se especifica mediante el uso de parámetros de la función main, resultado del llamado desde la línea de comandos. (se deben usar las variables predefinidas *argc, argv*)

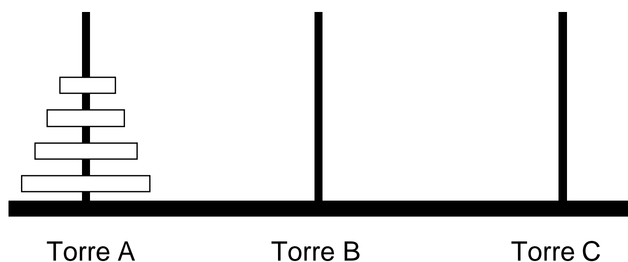
Pudiéndose realizar bajo D.O.S. el siguiente llamado: C:\>factorial 10, obteniendo el resultado solicitado.

8) Escriba una función que reciba como entrada una cadena de caracteres y el tamaño de la misma y de forma recursiva, determine si esta cumple con las características de palíndromo.

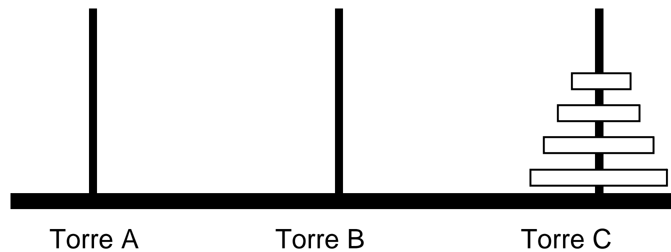
9) Escriba una función que dada una matriz de N x N, calcule el determinante (sugerencia: utilice el método de cofactores)

10) Uno de los problemas clásicos computacionalmente hablando es el de las Torres de Hanoi, el cual dice que se tienen tres torres verticales y en la primera se encuentra un número determinado de discos todos de diferente tamaño, cumpliéndose la siguiente condición: un disco no puede tener sobre él, otro de tamaño mayor. El problema consiste en pasar todos los discos de una torre a otra, utilizando la tercera como apoyo manteniendo la regla de los tamaños.

Ej. : para 4 discos la configuración inicial es:



el resultado



Escriba una función que presente la secuencia de pasos necesarios para trasladar los discos desde la Torre A, hasta la Torre C, para cualquier número de discos.

11. MANEJO DE ARCHIVOS

Los archivos son el mecanismo de almacenamiento en memoria secundaria que proveen los lenguajes de programación con el fin de garantizar la permanencia de la información y permitir el manejo de grandes volúmenes de datos.

11.1. Declaración

Todo archivo se compone de dos identificadores el nombre físico y el nombre lógico; el nombre físico que lo representa ante el sistema operacional, y el nombre lógico o variable que permite su manipulación mediante las instrucciones del lenguaje de programación.

La definición o declaración del nombre o variable lógica, se hace mediante la instrucción:

```
FILE *nombre_logico;
```

el cual se comporta como un apuntador al archivo y es el elemento sobre el cual se realizan la mayor parte de las operaciones con archivos.

11.2. Apertura y cierre de archivos

El proceso de apertura de los archivos se realiza con la asociación entre el nombre físico y el nombre lógico, permitiendo abrir un determinado archivo con su respectivo nombre físico y habilitar su manipulación en el programa mediante una variable lógica. Esta característica se logra mediante la función fopen y tiene la siguiente estructura general:

```
nombre_lógico = fopen(nombre_físico, modo);
```

donde modo es una cadena de caracteres que especifica las características con las cuales se abre el archivo, siendo las más útiles:

MODO	DESCRIPCIÓN
“r”	Abre un archivo existente para lectura únicamente
“w”	Crea y abre un archivo para escritura (si ya existe, será borrado)
“a”	Abre para adición al final del archivo si ya existe, o de lo contrario lo crea
“r+”	Abre un archivo existente para su actualización (lectura y/o escritura)
“w+”	Crea y abre un nuevo archivo para su actualización (lectura y/o escritura)
“a+”	Abre para actualización, permitiendo adicionar si ya existe, o de lo contrario lo crea

Adicionalmente, si los archivos son en modo texto se adiciona al modo la letra "t"; de lo contrario si son en modo binario, se adiciona la letra "b".

Ej. : "rb", "a+b", "wb", "r+t", "a+t".

La función fopen regresa un valor NULL cuando trata de abrir un archivo que no existe utilizando los modos "r" o "r+".

Todo archivo abierto debe ser cerrado para que se realicen las debidas actualizaciones y evitar así que se pierda la información allí almacenada; la instrucción para realizar el cierre de los archivos es *fclose*, la cual presenta la siguiente estructura:

```
fclose(nombre_lógico);
```

donde nombre_lógico es la variable o apuntador que representa el archivo que se va a cerrar.

Existen algunos archivos predefinidos del sistema operativo que representan los diferentes dispositivos del sistema tales como:

- stdin : entradas estándar del sistema (nombre lógico)
- stdout : salidas estándar del sistema (nombre lógico)
- stdprn : impresora estándar del sistema (nombre lógico)
- con : consola (pantalla y teclado) (nombre físico)
- prn : impresora (nombre físico)

Ej. : Abrir el archivo denominado "nomina.txt" para su actualización y poder así manipularlo mediante la variable lógica arch y luego cerrarlo.

```
#include <stdio.h>

FILE   *arch;
void main ( void )
{
```

```
		arch = fopen("nomina.txt", "r+");
		if (arch == NULL)
			printf("\n El archivo no existe");
		else
		{
			printf("\n Archivo abierto y listo para su manipulación y/o actualización");
			fclose(arch);
		}
	}
```

11.3. Escritura en archivos

Existen dos maneras de almacenar información en archivos; en formatos binario y texto, los primeros permiten una mejor utilización del espacio (ocupan menor espacio) y los segundos garantizan mayor transportabilidad de la información entre programas.

11.3.1. Escritura en archivos texto

Para escribir en los archivos tipo texto se utilizan las funciones *fprintf, fputs* y *fputc*.

La función *fprintf* permite almacenar en el archivo variables definidas según los tipos básicos del lenguaje y posee argumentos similares a los del *printf* (estructura de control y lista de argumentos), siendo necesario adicionalmente especificar el nombre lógico del archivo sobre el cual se almacena la información. La estructura general de la función *fprintf* es la siguiente:

```
fprintf(nombre_lógico, control, lista de variables);
```

la cual se define de tipo entero ya que retorna adicionalmente el número de bytes efectivamente escritos en el archivo.

La función *fputs* permite almacenar en el archivo cadenas de caracteres exclusivamente y posee la siguiente estructura general:

```
fputs(cadena, nombre_lógico);
```

fputs escribe la cadena de caracteres especificada (terminada en NULL o fin de cadena) en el archivo especificado bajo nombre_lógico. Caracteres de fin de línea o fin de cadena no se adicionan al archivo; *fputs* retorna el último caracter escrito o en caso de error regresa un valor NULL.

La función *fputc* escribe un caracter en el archivo especificado; la estructura general de la instrucción *fputc* es la siguiente:

```
fputc(caracter, nombre_lógico);
```

donde caracter es el elemento que se escribe en el archivo definido y abierto con el nombre_lógico especificado, regresando el caracter escrito o en caso de error regresa un valor de fin de archivo o la constante EOF que es (-1).

11.3.2. Escritura en archivos binarios

Para escribir en archivos con formato binario se utiliza la función fwrite que permite almacenar cualquier tipo de estructura de información (exceptuando las listas dinámicas) en un archivo. La estructura general de la función *fwrite* es:

fwrite(dirección_variable, tamaño_unidad, número_unidades, nombre_lógico);

donde dirección_variable es un apuntador a la región de memoria donde se encuentra almacenada la información que se desea grabar en el archivo o la dirección de la variable; tamaño_unidad es el número de bytes que ocupa cada elemento que se desea almacenar; número_unidades es la cantidad de elementos que se van a almacenar (cada elemento tiene el tamaño definido en el parámetro anterior) y nombre_lógico es el identificador del archivo, retornando adicionalmente el número de ítems (no bytes) efectivamente escritos en el archivo.

Ej. : Escribir una función que almacene en archivo un vector de N números flotantes.

a) Almacenando uno a uno los elementos del vector

```
void grabar( float vector[ ], int n, char *nombre)
{
        int    x;
        FILE  *archivo;

        archivo = fopen(nombre, "wb");
        for (x=0; x<n; x++)
                fwrite(&vector[x], sizeof(float), 1, archivo);
        fclose(archivo);
}
```

b) Almacenando todo el vector directamente

```
void grabar( float vector[ ], int n, char  *nombre)
{
        FILE  *archivo;

        archivo = fopen(nombre, "wb");
        fwrite(vector, sizeof(float), n, archivo);
        fclose(archivo);
}
```

11.4. Lectura de archivos

Análogamente a la escritura en archivos, existen dos maneras de recuperar información de los archivos; en formatos binario y texto, los primeros permiten una mejor utilización del espacio (ocupan menor espacio) y los segundos garantizan mayor transportabilidad de la información entre programas. Cada vez que se lee de un archivo, el apuntador avanza en la misma proporción, manteniéndose en la siguiente posición del archivo.

11.4.1. Lectura en archivos texto

Para leer de los archivos tipo texto se utilizan las funciones fscanf, fgets y fgetc.

La función *fscanf* posee argumentos similares a los del *scanf*, siendo necesario especificar adicionalmente el nombre lógico del archivo sobre el cual se recupera la información. La estructura general de la instrucción *fscanf* es la siguiente:

```
fscanf(nombre_lógico, control, lista de variables);
```

la función regresa el número de campos efectivamente leídos o EOF (-1) si alcanza el fin de archivo.

La función *fgets* permite la lectura exclusiva de cadenas de caracteres provenientes de un archivo identificado bajo un nombre lógico determinado. La estructura general es la siguiente:

```
fgets(cadena, tamaño, nombre_lógico);
```

donde *fgets* lee caracteres del archivo definido bajo el nombre_lógico, hasta que éste lea una cantidad de caracteres especificada en tamaño, o hasta que encuentre un caracter de fin de línea (\n), almacenándolos en la cadena especificada; la función *fgets* adiciona automáticamente el caracter de fin de cadena a la cadena destino.

La función *fgetc* regresa el siguiente caracter en el archivo especificado bajo la variable nombre_lógico. La estructura general de la instrucción es:

```
caracter =  fgetc(nombre_lógico);
```

en la variable caracter se almacena la información leída, o si se detecta el fin de archivo, fgetc retorna el valor EOF o (-1).

11.4.2. Lectura en archivos binarios

Para leer de los archivos binarios se utiliza la función *fread* que permite recuperar cualquier tipo de estructura de información (exceptuando las listas dinámicas) de un archivo. La estructura general de la función *fread* es:

```
fread(dirección_variable, tamaño_unidad, número_unidades, nombre_lógico);
```

donde dirección_variable es un apuntador a la región de memoria donde se desea almacenar la información que se quiere recuperar del archivo; tamaño_unidad es el número de bytes que ocupa cada elemento que se desea recuperar; número_unidades es la cantidad de elementos que se van a recuperar (cada elemento tiene el tamaño definido en el parámetro anterior) y nombre_lógico es el identificador del archivo. La función regresa adicionalmente el número de ítems efectivamente leídos o cero (0) cuando alcanza el fin de archivo.

Ej. : Escribir un programa que lea el nombre de un archivo y muestre por pantalla su contenido en caracteres.

```
#include <stdio.h>

FILE  *arch;
void main ( void )
{
        char   nombre[30], caracter;

        printf(\n Teclee el nombre del archivo : ");
        gets(nombre);
        arch = fopen(nombre, "r");
        if (arch == NULL)
              printf("\n El archivo con nombre : %s, no existe", nombre);
        else
        {
              fscanf(arch, "%c", &caracter);
              do {
                     putc(caracter);
                     fscanf(arch, "%c", &caracter);
              } while (!feof(arch));
              fclose(arch);
        }
 }
```

11.5. Borrado físico de archivos

Los archivos se pueden borrar físicamente del medio en el cual se encuentran almacenados, haciendo uso de uno de las siguientes instrucciones:

```
unlink(nombre_físico);
```

borra el archivo cuyo identificador ante el sistema operacional sea nombre_físico, en caso de que se borre satisfactoriamente regresa 0, de lo contrario regresa 1.

Adicionalmente existe una macro instrucción definida en el archivo de encabezados *stdio.h*, que realiza la misma acción del *unlink*, con igual sintaxis:

```
remove(nombre_físico);
```

Ej. : Borrar un archivo con nombre "datos.txt"

```
if (unlink("datos.txt") == 0)
        puts("El archivo se borró satisfactoriamente");
else
        puts("El archivo no pudo ser borrado");
```

Se debe tener en cuenta al borrar un archivo que éste se encuentre cerrado, para evitar inconsistencias con el apuntador del mismo.

11.6. Control de los archivos

Existen diferentes instrucciones que se utilizan para el control en el acceso, posicionamiento y manejo del archivo en general.

11.6.1. Movimiento del apuntador del archivo.

El apuntador del archivo se puede mover a voluntad utilizando la función *fseek*, la cual presenta la siguiente estructura general:

```
fseek(archivo, desplazamiento, origen)
```

donde archivo es el manejador lógico del archivo; desplazamiento es la cantidad en bytes que se quiere mover el apuntador del archivo(debe ser una variable tipo *long*), donde valores negativos lo mueven hacia el principio del archivo y valores positivos lo mueven hacia el final del mismo; origen es el lugar a partir del cual se va a realizar el desplazamiento del apuntador del archivo, puede llevar valores de 0, 1 o 2, los cuales se representan mediante las siguientes constantes previamente definidas en el archivo de encabezados *stdio.h*:

CONSTANTE	VALOR	EXPLICACIÓN
SEEK_SET	0	a partir del comienzo del archivo (absoluto)
SEEK_CUR	1	a partir de la posición actual (relativo)
SEEK_END	2	a partir del final del archivo (absoluto desde el final)

La función adicionalmente regresa cero si el apuntador del archivo se pudo mover efectivamente, de lo contrario regresa un valor diferente correspondiente al código del error generado.

También es utilizada la función *rewind*, la cual se utiliza para reposicionar el apuntador del archivo en el comienzo del mismo; la cual presenta la siguiente estructura general:

```
rewind( archivo );
```

la cual es equivalente a:

```
fseek(archivo, 0L, SEEK_SET); // 0L indica que es de tipo long.
```

11.6.2. Determinación de la posición actual del apuntador del archivo

La posición actual del apuntador del archivo se calcula mediante la instrucción *ftell*, la cual presenta la siguiente estructura general:

```
posicion = ftell(archivo);
```

donde archivo es el manejador lógico del archivo. Para el uso de la función ftell es necesario que el archivo esté abierto, regresando un valor de tipo *long* que indica la posición del apuntador medida en bytes con respecto al comienzo del archivo.

11.6.3. Detección del fin de archivo

Se puede determinar si alguna operación de lectura ha alcanzado el fin de archivo, mediante la macro instrucción *feof*, la cual verifica si la señal o bandera de fin de archivo está activa. La estructura general de la macro es:

```
feof(archivo)
```

donde archivo es el manejador lógico del archivo al cual se quiere determinar si su apuntador está en el fin de archivo y regresa un valor de cero si no se detecta, o un número diferente de cero cuando éste detecta el final del archivo.

Es importante tener en cuenta que *feof* es una macro instrucción que verifica el estado de la bandera de fin de archivo controlada por el sistema operativo, y debido a su forma de implementación puede generar errores o confusiones, ya que esta solo se activa cuando se trata de realizar una operación mas allá del final del archivo y no cuando se llega a éste.

Ej. : Leer un archivo de caracteres y escribirlos en la pantalla hasta que se alcance el fin del archivo.

```
#include <stdio.h>

void main ( void )
{
   FILE  *arch;
   char nombre[30], caracter;
```

```
    printf(\n Teclee el nombre del archivo : ");
    gets(nombre);
    arch = fopen(nombre, "r");
    if (arch == NULL)
        printf("\n El archivo con nombre : %s, no existe", nombre);
    else
    {
      while (!feof(arch))
      {
        fscanf(arch, "%c", &caracter);
        putc(caracter);
      }
      fclose(arch);
    }
}
```

El anterior programa está aparentemente correcto, pero cuando se pone en funcionamiento sucede que el último elemento del archivo lo escribe dos veces; esto se debe a que cuando lee el último elemento la función de lectura de archivos fscanf, no detecta el final del archivo (ya que pudo hacer su operación de lectura normalmente) por lo tanto la bandera correspondiente no se modifica, manteniéndose el control del programa dentro del ciclo while, en cuya siguiente iteración trata de leer mas allá del final del archivo, operación que implica la modificación de la bandera correspondiente, pero dicha lectura no se realiza manteniendo en la variable asociada el último valor leído, razón por la cual éste se repite en la escritura mediante la función *putc*.

Para corregir esta situación las operaciones con archivos que involucren la macro instrucción *feof* deben hacerse de tal manera que se valide el final del archivo luego de realizar la operación de lectura como se hizo en el ejemplo de la sección 11.4.2.

11.6.4. Grabación de los Buffers

Cada vez que se da la orden de almacenar un valor, variable, mensaje o registro en un archivo, el sistema operacional lo almacena en un buffer, de tal manera que cuando este buffer se llena, se realiza la operación física de almacenamiento en el disco; esto lo hace el sistema con el fin de optimizar los recursos. El lenguaje C ofrece la posibilidad de almacenar la información de los buffers en los archivos en cualquier instante, sin tener que esperar hasta que estos estén llenos haciendo uso de la función *fflush*, la cual obliga que se desocupen los buffers, llevando la información a sus correspondientes archivos. La sintaxis de esta función es la siguiente:

```
fflush(archivo);
```

la función *fflush* regresa un valor de cero (0), si pudo realizar la operación satisfactoriamente o el valor EOF (-1) en caso de algún tipo de error.

Ej. La función *fflush* acompañada del archivo de entradas estándar del sistema, provoca la actualización o limpieza de los buffers del teclado, así:

```
fflush(stdin);
```

También es posible actualizar simultáneamente todos los buffers de los archivos abiertos mediante la función *flushall*, de la siguiente manera:

```
flushall( );
```

adicionalmente, regresa un número entero con la cantidad de buffers actualizados.

11.7. Ejercicios resueltos

1) Escribir un programa que abra el archivo "nomina.txt" para su actualización, permita su manipulación mediante una variable denominada archivo, lea cada uno de sus registros de tipo tiporeg (previamente definido) y luego cierre el archivo.

```
#include <stdio.h>

typedef struct {
        // Lista de campos del registro
} Tiporeg;

void main ( void )
{
        FILE      *archivo;
        Tiporeg  registro;

        archivo = fopen("nomina.txt", "r+b");
        if (archivo == NULL)
              printf("\n El archivo no existe");
        else
        {
              fread(&registro, sizeof(tiporeg), 1, archivo);
              do {
                    /* Escribe el contenido de cada uno de los campos del registro */
                    fread(&registro, sizeof(tiporeg), 1, archivo);
              } while (!feof(archivo));
              fclose(archivo);
        }
 }
```

2) Escribir una función que dado el nombre de un archivo, lea un conjunto de números flotantes (los que encuentre en el archivo), los almacene en un vector y los retorne así como la cantidad de números leídos.

```
void leer(char *nombre, float vector[], int *numero)
{
        FILE  *archivo;
        int     x = 0;

        archivo = fopen(nombre, "rb");
        fread(&vector[x], sizeof(float), 1, archivo);
        do  {
                fread(&vector[++x], sizeof(float), 1, archivo);
        } while (!feof(archivo));
        fclose(archivo);
        *numero = x + 1;
}
```

3) Escribir el programa para leer un archivo de texto por renglones y mostrarlo en la pantalla (cada renglón tiene máximo 80 caracteres):

```
#include <stdio.h>
void main (void)
{
        FILE  *archivo;
        char   renglon[81];
        char   nombre[15];

        printf("\n Nombre del archivo : ");
        gets(nombre);
        archivo=fopen(nombre, "r");
        if (archivo == NULL)
                printf("\n El archivo con nombre : %s no existe", nombre);
        else {
                fgets(renglon, 80, archivo);
                do {
                        printf("\n%s", renglon);
                        fgets(renglon, 80, archivo);
                } while (!feof(archivo));
                fclose(archivo);
        }
}
```

4) Mostrar el contenido de un archivo utilizando la función fgetc, visualizando página por página (parando cada 24 líneas):

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

#define MAX_PANTALLA        24    // Número máximo de filas por pantalla
#define  MAX_NOMBRE         30    // Tamaño máximo del nombre del archivo
#define  FIN_LINEA    '\n'        // Carácter de fin de línea, algunos sistemas utilizan otro

 void main (void)
 {
        FILE  *archivo;
        int    contador = 0;
        char   nombre[MAX_NOMBRE], caracter;

        printf("\n Nombre del archivo: ");
        gets(nombre);
        archivo=fopen(nombre,  "r");
        if (archivo == NULL)
               printf("\n El archivo con nombre: %s no existe", nombre);
        else {
               do {
                      caracter = fgetc(archivo);
                      if (caracter == FIN_LINEA)
                      {
                             contador ++;
                             if (contador == MAX_PANTALLA)
                             {
                                    puts("Presione cualquier tecla para continuar ");
                                    getch();
                                    contador = 0;
                             }
                      }
                      printf("%c", caracter);
               } while (caracter != EOF);
               fclose(archivo);
        }
 }
```

11.8. Ejercicios propuestos

1) Escriba un programa que copie el contenido de un archivo en otro, dados el nombre y camino de los archivos origen y destino.

2) Escriba un programa que dado el nombre de un archivo, contabilice el número de vocales que contiene, separándolas según la vocal.

3) Escriba un programa que dado el nombre de un archivo y una cadena de caracteres, determine si dicha cadena se encuentra en el archivo.

4) Escriba un programa que dado un archivo de tipo texto, cambie todas las ocurrencias de caracteres en minúsculas por su correspondiente en mayúscula, dejando el resultado en el mismo archivo original.

5) Escriba un programa que muestre el contenido de un archivo de texto desde el final hasta el principio. (comando type del DOS en forma invertida)

6) Escriba un programa que permita almacenar en archivos la información del inventario de un almacén con los parámetros descritos en los ejercicios de la sección 9.5. numerales a. y b.

7) Escriba un programa que reciba como entrada el nombre de un archivo y mediante dos opciones permita su encriptación (ocultamiento de la información) y desencriptación.

8) Escriba un programa que sirva para imprimir archivos de texto, de tal manera que reciba el nombre de un archivo y lo imprima organizadamente por páginas dejando las respectivas márgenes y numerando cada una de ellas.

9) Escriba un programa que permita visualizar el contenido de un archivo en la pantalla por páginas, dando la posibilidad de adelantar o regresar una página.

10) Escriba un programa que dado el nombre de un archivo, calcule las siguientes estadísticas:
·número de caracteres

•número de palabras (separador de palabra puede ser: espacio, tabulador o cambio de línea)
•número de renglones (separador un cambio de línea)
•número de párrafos (separador un renglón vacío)

12. APUNTADORES

12.1. Definiciones y declaración de los apuntadores

Teniendo en cuenta que la memoria del computador puede ser manejada de una manera más flexible de lo que permiten las variables estáticas y los arreglos, surgen los apuntadores como elemento integrador y de respaldo para dicho manejo.

Un apuntador es una variable en cuyo contenido se almacena la dirección de una porción de memoria, la cual es utilizada ya sea para utilizar la memoria de forma dinámica, alcanzar posiciones de memoria predefinidas o para el manejo de parámetros por referencia en las funciones.

Una variable guarda un valor y un apuntador se almacena una dirección, la cual se refiere indirectamente a un valor.

Los apuntadores se definen añadiendo el símbolo asterisco (*) a la variable en el momento de su definición.

```
tipo    *nombrevariable;
```

Ej. : Defina una variable de tipo entero y un apuntador a una variable del mismo tipo:

```
int     valor,  *apuntador
```

se definen 2 variables, donde la primera se refiere al lugar o sitio para guardar un valor entero, mientras que la segunda almacena la dirección de una variable que alberga un valor entero.

12.2. Operadores con apuntadores

Existen dos operadores básicos para el uso con los apuntadores; el operador * y el operador &; el primero (*) se utiliza para la declaración de los apuntadores añadiéndolo antes del nombre de la variable y también se utiliza para referirse al contenido o valor de un apuntador. El segundo operador (&) regresa la dirección de la memoria en la cual se encuentra almacenada una variable previamente definida.

Para los apuntadores se cumple la siguiente ecuación:

```
*(&variable) == variable
```

lo que significa que el contenido de la dirección de una variable es la misma variable.

Ej. : Dadas las siguientes declaraciones e instrucciones presentar su evolución.

```
#include <stdio.h>

void main (void)
{
        int     valor;          /* variable tipo entero */
        int     *apuntador;     /* variable tipo apuntador a un número entero */
```

```
        valor = 10;
        apuntador = &valor;
        printf("\n La dirección de la variable valor es : %p", &valor);
        printf("\n El valor de la variable apuntador es : %p", apuntador);
        printf("\n El valor de la variable valor es : %d", valor);
        printf("\n El valor de la variable *apuntador es : %d", *apuntador);
}
```

cuyo resultado puede apreciarse gráficamente de la siguiente manera:

a) Estado inicial de la memoria

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	Indef - 03	No asignado
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

Los valores Indef - 0X, no implican que en dichas posiciones de memoria no existan valores, ya que en la totalidad de la memoria siempre hay valores numéricos comúnmente denominados "basura", lo cual es únicamente información que no tiene ningún tipo de sentido y relevancia para el programa en ejecución.

b) Luego de declarar las dos variables valor y *apuntador, el estado de la memoria es el siguiente:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	Indef - 03	Valor
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	apuntador
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

El sistema operacional asigna posiciones de memoria a las variables según la disponibilidad de esta, no siendo necesario que estas ocupen lugares continuos o aledaños.

c) Luego de realizar la instrucción de asignación sobre la variable valor, valor = 10; , el estado de la memoria es el siguiente:

DIRECCIÓN – RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	10	valor
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	apuntador
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

d) Cuando se asigna a la variable apuntador el contenido o dirección de la variable valor, el estado de la memoria presenta la siguiente disposición:

DIRECCIÓN – RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	10	valor
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	0003	apuntador
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

e) Por último, las variables involucradas en este programa en combinación con los diferentes operadores presentan los siguientes valores:

VARIABLE	VALOR
valor	10
apuntador	0003
*valor	Indef – 10
*apuntador	10
&valor	0003
&apuntador	0006

12.3. Apuntadores en los parámetros por referencia

Cuando se llaman funciones que involucren parámetros y se requiera que regresen un valor en uno de ellos, es necesario utilizar los apuntadores como parte de los parámetros (parámetros por referencia). Esto se debe a que una función para poder regresar un valor requiere el sitio, lugar o dirección de la memoria en el cual debe dejar el resultado requerido, siendo necesario hacer uso de los apuntadores.

Ej. : Si se llama a una función sumar con los dos sumandos y el potencial resultado sin hacer uso de los apuntadores ocurre lo siguiente:

```
#include <stdio.h>

void sumar ( int num1, int num2, int resp)
{
        resp = num1 + num2;
}

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

        a = 10;
        b = 20;
        sumar(a, b, c);
}
```

a) Antes de comenzar el programa se tiene la siguiente configuración de la memoria:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	Indef - 03	No asignado
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

b) Luego de comenzar el programa y declarar las variables a, b, c, el estado de la memoria es:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	a
0002	Indef - 02	b
0003	Indef - 03	c
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

c) Al efectuar las debidas inicializaciones para a y b, se tiene:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

d) Al llamar la función sumar, cada uno de los parámetros se comporta como una variable local independiente, asignándose los valores con los cuales fueron llamados:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	10	num1
0005	20	num2
0006	Indef - 03	resp
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

e) Luego de realizar la operación la suma el estado de la memoria cambia de la siguiente manera:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	10	num1
0005	20	num2
0006	30	resp
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

f) Como no existe ningún tipo de vínculo entre las variables de la función main y los parámetros de la función sumar, el estado de la memoria al terminar la función varía únicamente para liberar el espacio utilizado por las funciones allí utilizadas así:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	10	No asignado
0005	20	No asignado
0006	30	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

g) Por lo tanto el valor de la variable c, no sufre ningún tipo de modificación ante el llamado a la función sumar, siendo necesario hacer uso de los parámetros por referencia mediante los apuntadores.

Ej. : Si se llama a una función sumar con los dos sumandos y el potencial resultado haciendo uso de los apuntadores el funcionamiento del programa es el siguiente:

```
#include <stdio.h>

void sumar ( int num1, int num, int *resp)
{
```

```
        *resp = num1 + num2;
}

void main (void)

{
        int     a, b, c;

        a = 10;
        b = 20;
        sumar(a, b, &c);
}
```

a) Antes de comenzar el programa se tiene la siguiente configuración de la memoria:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	No asignado
0002	Indef - 02	No asignado
0003	Indef - 03	No asignado
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

b) Luego de comenzar el programa y declarar las variables a, b, c, el estado de la memoria es:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	Indef - 01	a
0002	Indef - 02	b
0003	Indef - 03	c
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

c) Al efectuar las debidas inicializaciones para a y b, se tiene:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	Indef - 04	No asignado
0005	Indef - 05	No asignado
0006	Indef - 06	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

d) Al llamar la función sumar, cada uno de los parámetros se comporta como una variable local independiente, asignándose los valores con los cuales fueron llamados:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	Indef - 03	c
0004	10	num1
0005	20	num2
0006	0003	resp
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

e) Luego de realizar la operación de la suma, sobre el contenido de la variable apuntada o referenciada por resp, el estado de la memoria cambia de la siguiente manera:

DIRECCIÓN - RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	30	c
0004	10	num1
0005	20	num2
0006	0003	resp
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

f) Al terminar la función el estado de la memoria varía únicamente para liberar el espacio utilizado por las funciones allí utilizadas así:

DIRECCIÓN – RAM	VALOR	NOMBRE VARIABLE
0001	10	a
0002	20	b
0003	30	c
0004	10	No asignado
0005	20	No asignado
0006	0003	No asignado
0007	Indef - 07	No asignado
0008	Indef - 08	No asignado
0009	Indef - 09	No asignado
0010	Indef - 10	No asignado

g) Por lo tanto el valor de la variable c se modifica automáticamente, ya que la variable res es una referencia o indirección sobre la variable a, de tal manera que al modificar lo apuntado por ella, realmente se modifica la variable c; realizando la operación solicitada.

12.4. Uso de apuntadores en vectores y matrices

Por la forma en que se definen y se ubican en la memoria, los arreglos (vectores y/o matrices) son tratados como apuntadores facilitando así el uso que se pueda dar de ellos. Los nombres de las variables que se definan de tipo arreglo representan la dirección en la memoria en la cual se almacena el primer componente del mismo, razón por la cual los subíndices se explican como desplazamientos relativos a partir de la dirección base o inicial utilizados para el cálculo de la posición referenciada por un determinado subíndice. Es por esta razón que cuando se requiere la lectura de una cadena de caracteres mediante la función scanf, no es necesario utilizar el símbolo de dirección, ya que el nombre representa la dirección.

Un vector puede ser manejado mediante apuntadores, calculando las direcciones relativas de cada uno de sus elementos, basándose en la dirección inicial y el tipo de componente.

Ej. : Definir un vector con 10 números enteros leerlo y posteriormente escribirlo sin utilizar los subíndices:

```
#include <stdio.h>

void main(void)
{
        int     vector[10], x;

        for (x=0; x<10; x++)
        {
```

```
        printf("\n Digite el elemento %d : ", x);
        scanf("%d", (vector+x));
    }
    puts("\n Los elementos leídos son : \n");
    for (x=0; x<10; x++)
    {
        printf("\n El elemento %d : es : %d ", x, *(vector+x));
    }
}
```

Es importante anotar que el valor que se suma al apuntador no es la cantidad de bytes que éste se debe desplazar, sino la cantidad de elementos o posiciones, teniendo en cuenta que el compilador "conoce" el tipo del apuntador (en este caso un entero).

Ej. : Dada una matriz de tamaño 10 x 10, definir una función para que almacene un valor en una posición X, Y, sin hacer uso de subíndices.

Dadas las siguientes definiciones:

```
#define    TAMAÑO      10
```

La función sería:

```
void  almacenar(int  *mat  int posx, int posy, int valor)
{
    *(mat+(posx*TAMAÑO) + posy) = valor;
}
```

El programa principal tendría la siguiente configuración:

```
void main (void)
{
    int      matriz[TAMAÑO][TAMAÑO],x, y, valor = 0, *copia;

    copia = (int *) matriz;
    for (x=0; x<TAMAÑO; x++)
        for (y=0; y<TAMAÑO; y++)
            almacenar(copia, x, y, valor++);

    puts("\n La matriz queda de la siguiente forma : \n");

    for (x=0; x<TAMAÑO; X++)
    {
        for (y=0; y<TAMAÑO; y++)
            printf("%5d", matriz[x][y]);
         putchar('\n');
    }
}
```

12.5. Ejercicios propuestos

1) Defina una variable de tipo entero y establezca un apuntador a la misma, lea un valor y escríbalo en la pantalla utilizando el apuntador y compárelo con la variable.

2) Determine que utilidad o significado puede tener, y si es o no válido realizar la siguiente definición:

```
void  *variable;
```

3) Estudie el siguiente programa que mediante una función lee una cadena de caracteres, determine teniendo en cuenta que cadena es un apuntador, ¿si es válido el uso del modificador const y porqué? :

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

void  LeerCadena(const char  *cadena)
{
        gotoxy(10,14);
        printf("Teclee una cadena : ");
        scanf("%s", cadena);
}

void main(void)
{
        char   nombre[30];

        LeerCadena(nombre);
        printf("\n La cadena leída es : %s", nombre);
}
```

4) Dado que la dirección de memoria de vídeo para una pantalla en modo texto es 0xB8000000, escriba una rutina que permita escribir un carácter en la primer posición de la pantalla; generalícela para que reciba una posición de la pantalla X,Y y una cadena de caracteres y permita escribir dicha cadena a partir de la posición especificada.

5) Escriba una función que con base en los datos del manejo de la memoria de video suministrados en el ejercicio anterior, permita almacenar la información de la pantalla en un archivo de texto.

6) Escriba una función que reciba como entrada dos variables con valores numéricos enteros e intercambie el contenido de dichas variables.

7) Explique que hace y si es válida la siguiente definición, y escriba una equivalente:

```
char *semana[7] = {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"};
```

8) Dada una matriz de dimensiones N x N, escriba los elementos de la diagonal sin hacer uso de subíndices.

9) Escriba una función que reciba como entrada una cadena de caracteres y sin utilizar subíndices, calcule y retorne la cantidad de caracteres que contiene la cadena (calcula la longitud).

10) Escriba una función que reciba dos cadenas de caracteres y sin hacer uso de subíndices, determine si estas cadenas son o no iguales. Si son iguales retorne un valor de uno (1), de lo contrario retorne un valor de cero (0).

13. MEMORIA DINÁMICA / ESTRUCTURAS DE DATOS DINÁMICAS

Al interior de los programas la memoria se maneja mediante variables, las cuales se definen de forma estática desde cuando se escribe el programa, y en el momento de su ejecución la memoria es separada en su totalidad para su uso. El lenguaje C provee mecanismos para la reserva y liberación de la memoria dinámicamente en la medida que los usuarios o los programas lo requieran, dando además la posibilidad de construir estructuras de datos de mayor complejidad tales como: listas, pilas, colas, arboles y grafos entre otras.

Para separar memoria se utiliza la función *malloc* que permite reservar una cantidad determinada de bytes especificados como argumento y regresa un apuntador o la dirección en la cual se pudo separar la memoria requerida. La sintaxis de la función malloc es la siguiente:

```
apuntador = (tipo_apuntador) malloc( numero_bytes );
```

donde tipo_apuntador es el apuntador al tipo de datos que se van a almacenar y numero_bytes es la cantidad de memoria contigua que se requiere. Si la función malloc no puede encontrar la cantidad de memoria requerida, ésta regresa un apuntador a NULL (apuntador nulo).. La dirección de memoria que retorna la función malloc debe ser almacenada en una variable tipo apuntador.

Para liberar la memoria separada mediante la función *malloc*, se utiliza la función free, que recibe como argumento un apuntador a la memoria que se quiere liberar, la cual tiene como objetivo informar al sistema operacional que la memoria referenciada mediante el apuntador está libre y puede ser utilizada para otros fines. La sintaxis de la función free es la siguiente:

```
free( apuntador );
```

donde la cantidad de memoria asignada a apuntador es liberada para que pueda ser destinada posteriormente para otras variables u otros procesos.

Las estructuras de datos dinámicas están conformadas por porciones de memoria definidas en lugares no necesariamente contiguos, unidas mediante apuntadores que indican él o los siguientes elementos que las conforman, hasta encontrar un apuntador cuyo contenido sea NULL (vacío) o se llegue a una marca o apuntador definido para el final de la estructura. Toda estructura dinámica debe tener un apuntador base (denominado cabeza u origen) que indique cual es el primer elemento, el cual se debe manipular de forma diferente a los demás, siendo el identificador o representante de la misma. Debido a la naturaleza de estas estructuras de datos encadenadas toda operación se debe realizar partiendo de la cabeza o primer elemento hasta encontrar la cola o último elemento, el cual se reconoce fácilmente, ya que su apuntador señala a NULL

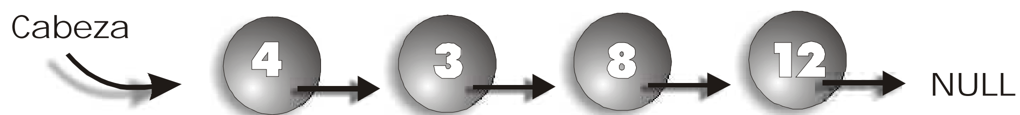
13.1. Listas con encadenamiento simple

El caso más sencillo de estructuras dinámicas es la lista encadenada simple, en la cual se enlaza mediante apuntadores o direcciones un conjunto de porciones de memoria denominadas nodos, los cuales se representan mediante registros compuestos de campos, donde algunos se utilizan para el almacenamiento de la información y uno de ellos es un apuntador destinado para su encadenamiento con el siguiente elemento de la lista, dando la posibilidad de almacenar en forma dinámica la cantidad de información que se requiera, con la característica secuencial que obliga a recorrer la lista desde el principio hasta el fin para realizar las diferentes operaciones sobre dicha estructura.

Las listas encadenadas se deben recorrer desde el primer elemento hasta el último sin la posibilidad de regresar al anterior y sin otro mecanismo de acceso, permitiendo el almacenamiento de grandes volúmenes de información, donde cada nodo se genera en el momento de su requerimiento y se inserta en la estructura según el orden deseado; los diferentes nodos que se incluyen en la lista están definidos en la memoria en lugares no necesariamente contiguos lo que permite un mejor aprovechamiento de la misma, caso contrario ocurre con los arreglos que requieren memoria libre y contigua para su operación.

Una lista encadenada está compuesta por cero o mas nodos enlazados entre sí, donde el último nodo se reconoce fácilmente, ya que su apuntador de encadenamiento señala a NULL.

Gráficamente una lista encadenada simple con los valores numéricos {4, 3, 8 ,12}, se presenta de la siguiente manera:



En el lenguaje C las listas se representan mediante registros o estructuras definidas en forma dinámica, las cuales contienen por lo menos uno de sus campos como un apuntador a un elemento del mismo tipo del registro (definición recursiva), siendo éste el encargado de unir los demás elementos entre sí formando las diferentes estructuras de datos.

La sintaxis de un nodo en una lista encadenada simple se define como un registro cuya forma general se representa de la siguiente manera:

```
struct nombre_del_registro {
    campos de información
    struct nombre_del_registro  *nombres_campos_encadenadores ;
} lista_variables;
```

Es importante notar que el uso del comando typedef para la definición de nuevos tipos también está permitido y es igualmente práctico:

```
typedef struct nombre_del_registro {
    campos de información
    struct nombre_del_registro  *nombres_campos_encadenadores ;
} nuevo_tipo;
```

Para el acceso a cada uno de los campos en este tipo de registros se debe tener en cuenta que las variables son de tipo apuntador, siendo necesario realizar una de las siguientes operaciones:

a) Calcular el contenido de la dirección de la variable y luego se refiere a cada uno de los campos como si fuera un registro estático (forma antigua poco utilizada, pero válida)

```
(*variable).nombre_del_campo
```

b) Mediante el operador de acceso de los apuntadores (->), el cual permite hacer referencia de forma directa a cada uno de los campos de un registro tipo apuntador

```
variable->nombre_del_campo
```

Para manipular una lista encadenada simple, se requiere de funciones que permitan acciones tales como:

- Insertar un elemento por la cabeza
- Insertar un elemento por la cola
- Insertar un elemento de tal manera que quede ordenado entre la lista (ascendente o descendente)
- Recorrer una lista escribiendo el contenido
- Contar los elementos de una lista
- Calcular el mayor y/o menor elemento de una lista
- Ordenar los elementos de una lista una vez que ya han sido ingresados a esta
- Recorrer una lista en orden contrario al encadenamiento que esta presenta escribiendo su contenido
- Sumar los elementos de una lista
- Calcular el promedio o media aritmética de los elementos de una lista
- Eliminar todas las ocurrencias de un elemento en una lista
- Eliminar completamente una lista

- Cambiar un elemento de una lista por otro
- Invertir el sentido de los apuntadores de la lista

13.1.1. Representación de una lista encadenada simple

Es necesario definir un registro o estructura para almacenar la información, el cual deberá tener tantos campos como información se desee guardar, teniendo en cuenta el tipo de valores de cada una de las variables y un apuntador para realizar el encadenamiento de la lista.

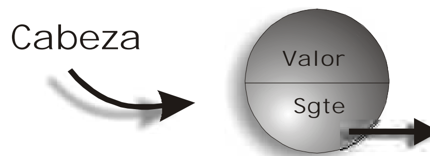
Ej. : Defina una estructura para representar una lista encadenada simple y que almacene un valor de tipo entero.

```
/* ___________________________________________________________________
Estructura de datos sobre la cual se define la lista encadenada simple (uso typedef).
___________________________________________________________________*/
typedef struct nodo {
        int             valor;      /* Información a guardar en la lista */
        struct nodo     *sgte;      /* Apuntador a un elemento del mismo tipo */
} Nodo;

Nodo  *cabeza;
```

o su equivalente sin el uso de *typedef*:

```
/* ___________________________________________________________________
Estructura de datos sobre la cual se define la lista encadenada simple.
___________________________________________________________________ */
struct nodo {
        int             valor;      /* Información a guardar en la lista */
        struct nodo     *sgte;      /* Apuntador a un elemento del mismo tipo */
} *cabeza;
```

El apuntador cabeza de tipo struct nodo o Nodo se representa gráficamente de la siguiente manera (una vez que se le reserva la memoria):



donde los campos de este registro se referencian de la siguiente manera:

para el campo valor de tipo entero.
cabeza->valor

para el campo sgte de tipo apuntador.
cabeza->sgte

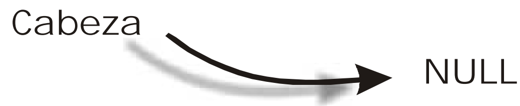
13.1.2. Insertar un elemento

Existen tres formas básicas de insertar un elemento en una lista encadenada simple: insertar por la cabeza, insertar por la cola o insertar en forma ordenada.

Ej. : Para insertar un elemento por la cabeza se procede de la siguiente manera:

a) inicialmente se debe tener una lista vacía (la cabeza debe apuntar a NULL).

```
cabeza = NULL;
```



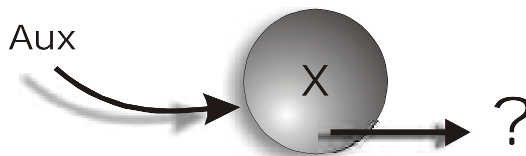
b) se debe utilizar una variable del tipo apuntador para generar un nuevo nodo (separar y asignar la memoria requerida para almacenar la información)

```
Nodo   *aux;         // es equivalente a:   struct nodo *aux;
aux = (struct nodo *) malloc (sizeof(struct nodo));
```

c) se verifica si la operación de reserva de memoria pudo realizarse satisfactoriamente, de lo contrario el apuntador aux señalará a NULL, indicando que no se encontraba disponible la cantidad de memoria requerida, siendo necesario evitar que el proceso continúe y presentar el mensaje apropiado.

```
if (aux == NULL)
        printf("\n No hay memoria suficiente ");
```

el nuevo nodo que se crea tiene un valor indefinido para su campo dato (información) y un apuntador sgte que no tiene especificado el sitio o lugar al cual apunta. (debe inicializarse)

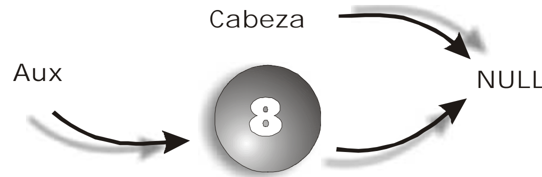


d) se almacena en el nuevo nodo marcado como aux el valor que se desea insertar, utilizando el campo destinado para tal fin (valor).

```
aux->valor = dato;
```

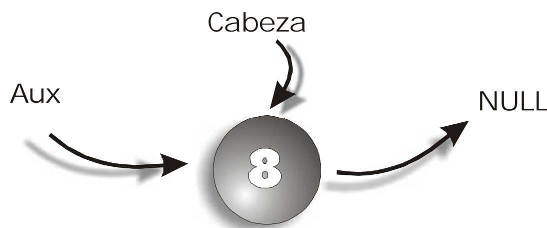
e) se redirige el apuntador sgte del nuevo nodo (aux) hacia el lugar donde apunta actualmente la variable cabeza (primer elemento), para así insertar el nodo en la lista.

```
aux->sgte = cabeza;
```

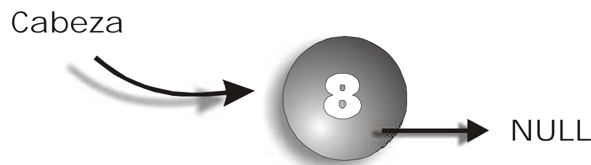


f) se redirecciona el apuntador que marca el primer elemento de la lista (cabeza), hacia el nuevo nodo creado, terminando así el proceso de inserción.

```
cabeza = aux;
```



g) reorganizando los apuntadores y el gráfico, la nueva lista queda:



el apuntador aux por ser de carácter local, deja de aparecer, pero sus acciones se mantienen mientras que dure el programa o dicha memoria no sea liberada.

La función completa que inserta un nodo a una lista por la cabeza de la misma se escribe de la siguiente manera:

```
/* ___________________________________________________________________
adiciona un número insertando un nuevo nodo en la cabeza de la lista
___________________________________________________________________*/
void insertarcab(int dato)
{
        struct nodo *aux;

        aux = (struct nodo *) malloc (sizeof(struct nodo));
         if (aux == NULL)
                printf("\n No hay memoria suficiente ");
```

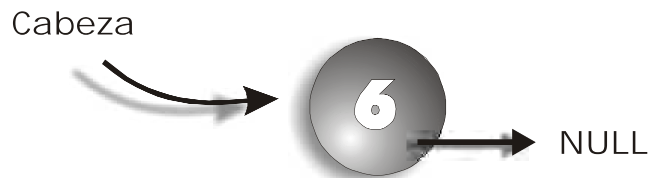
```
        else
        {
                aux->valor = dato;
                aux->sgte = cabeza;
                cabeza = aux;
        }
 }
```

Análogamente se realiza el proceso de inserción por la parte final de la lista (comúnmente denominada la cola de la lista), siendo necesario realizar algunas operaciones adicionales, tales como validar si la lista está vacía o contiene un único elemento, de lo contrario se debe desplazar un apuntador hasta el final de la misma y realizar allí el proceso de inserción.

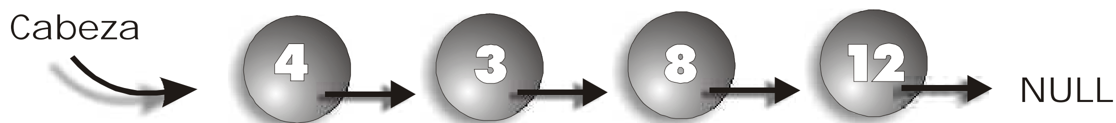
Ej. : insertando un nodo con valor 6.

a) si la lista está vacía, el proceso es igual al de insertar por la cabeza.

obteniendo:

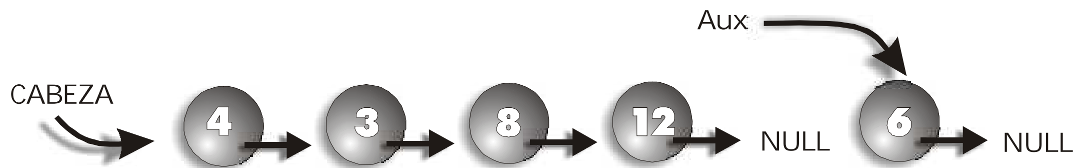


b) si la lista posee por lo menos un elemento, su representación gráfica es la siguiente:



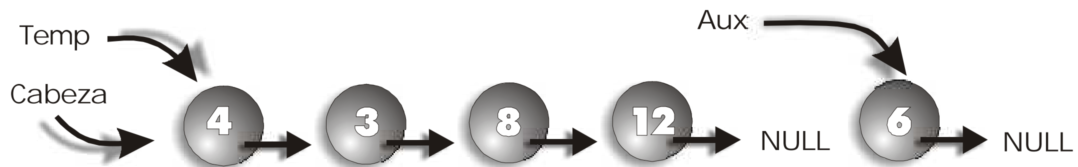
siendo necesario crear un nuevo nodo mediante la función malloc, mediante las siguientes instrucciones (requiere las debidas validaciones):

```
aux = (struct nodo *) malloc (sizeof(struct nodo));
aux->valor = dato;
aux->sgte = NULL;
```



c) se utiliza una variable temporal (temp) de tipo apuntador (*struct* nodo *temp), que se fija en el primer elemento de la lista (cabeza), la cual servirá para su recorrido y posterior enlace, justo hasta alcanzar el último elemento (lugar de la inserción).

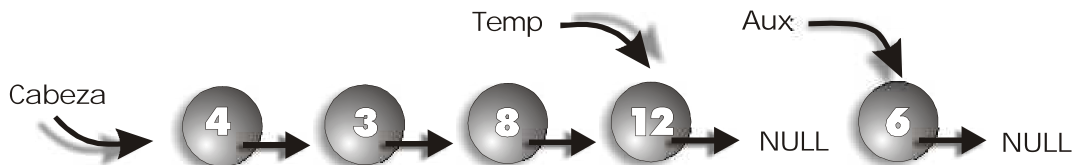
```
temp = cabeza;
```



d) se desplaza el apuntador temp hasta llegar al último elemento, mediante un ciclo, verificando que el siguiente elemento sea diferente de NULL (final de la lista).

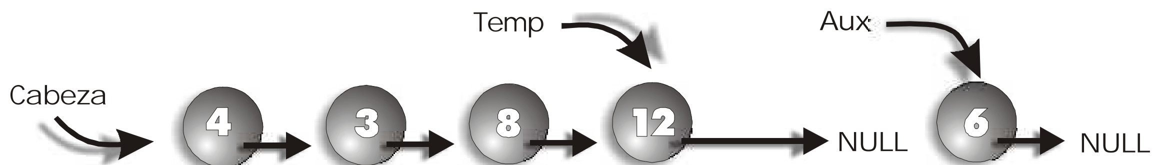
```
while (temp->sgte != NULL)
        temp = temp->sgte;
```

obteniendo al final del ciclo:

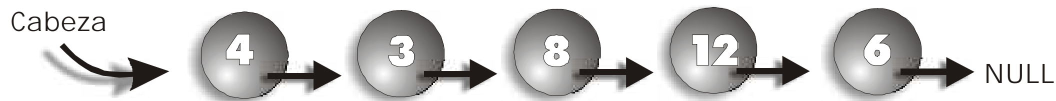


e) una vez ubicado el apuntador temp en el último nodo o elemento de la lista, se procede a su encadenamiento con el nodo aux que apunta al elemento a insertar. Se modifica el apuntador sgte del nodo temp para que apunte al nuevo nodo aux.

```
temp->sgte = aux;
```



f) reorganizando la representación gráfica y eliminando aquellos apuntadores temporales, la lista queda de la siguiente manera:



La función completa que inserta un nodo a una lista por la cola de la misma se escribe de la siguiente manera:

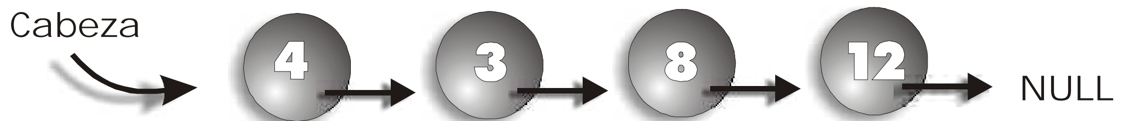
```
/* ______________________________________________________________________
adiciona un número insertando un nuevo nodo al final de la lista (cola de la lista)
______________________________________________________________________*/

void insertarcol(int dato)
{
        struct nodo *aux, *temp;

        aux = (struct nodo *) malloc (sizeof(struct nodo));
        if (aux == NULL)
                printf("\n No hay memoria suficiente ");
        else
        {
                aux->valor = dato;
                aux->sgte = NULL;
                if (cabeza == NULL)
                        cabeza = aux;
                else
                {
                        temp = cabeza;
                        while (temp->sgte != NULL)
                                temp = temp->sgte;
                        temp->sgte = aux;
                }
        }
}
```

13.1.3. Eliminar un elemento

Para eliminar un nodo de una lista se deben tener en cuenta dos casos diferentes; primero si el nodo a eliminar es la cabeza, en el cual se debe modificar físicamente la cabeza o identificador de la lista, y segundo cuando es otro nodo diferente de la cabeza.

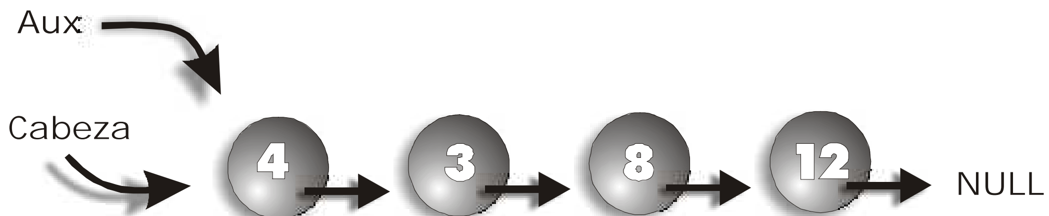
Para el primer caso se verifica si el valor buscado es el almacenado en la cabeza, siendo necesario disponer de un apuntador adicional para guardar el contenido del nodo a eliminar y así poder liberar la memoria utilizada.

Dada la siguiente lista se desea eliminar el nodo con información 4 :



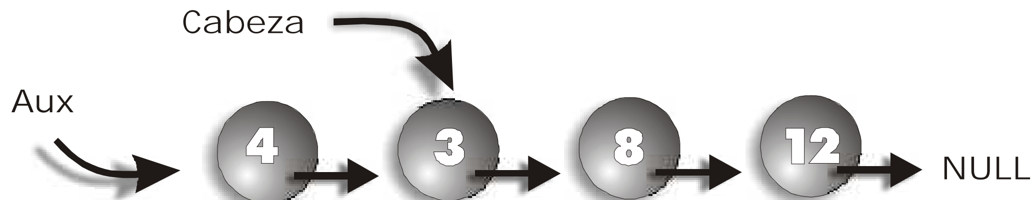
a) se posiciona un apuntador aux en el nodo a eliminar (en este caso la cabeza)

```
aux = cabeza;
```



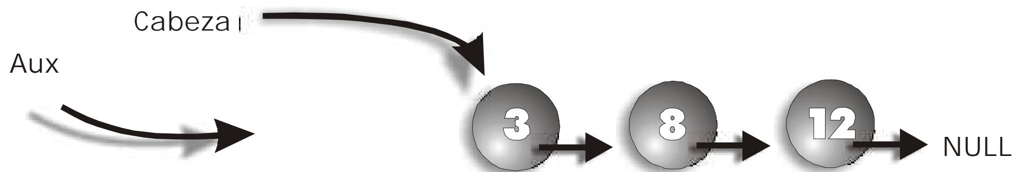
b) se adelanta el apuntador de la cabeza al siguiente nodo de la lista.

```
cabeza = cabeza -> sgte;
```



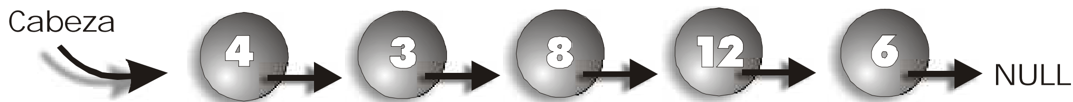
c) se libera el nodo correspondiente y la lista queda:

```
free(aux);
```



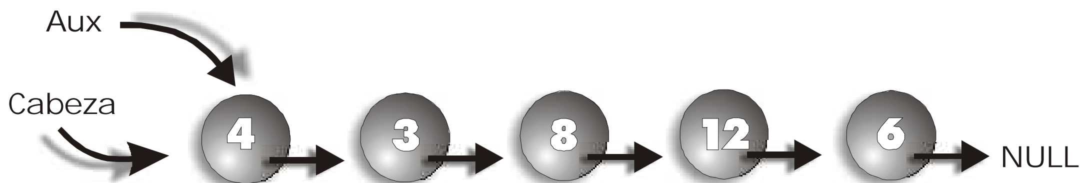
Para eliminar un nodo diferente de la cabeza, es necesario recorrer la lista mediante un apuntador auxiliar (aux), preguntando por el contenido del siguiente nodo hasta encontrar el valor buscado o que se termine la lista (el valor no está en la lista); se debe tener en cuenta que la pregunta se realiza desde el nodo anterior debido a que es necesario estar un nodo atrás del que se desea eliminar con el fin de poder realizar el debido reencadenamiento de los nodos.

Dada la siguiente lista, se desea eliminar el nodo cuya información es el número 12:



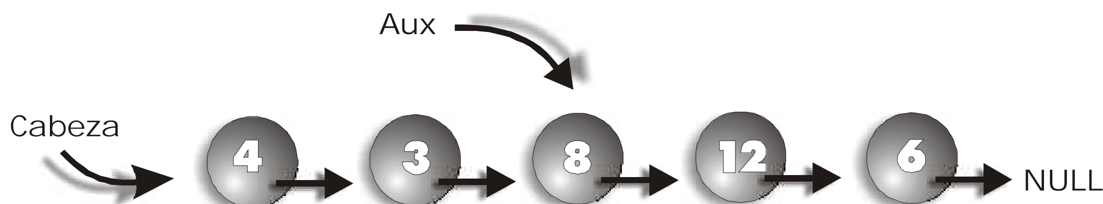
a) se posiciona un apuntador en la cabeza (para ser utilizado como temporal o auxiliar)

```
aux = cabeza;
```



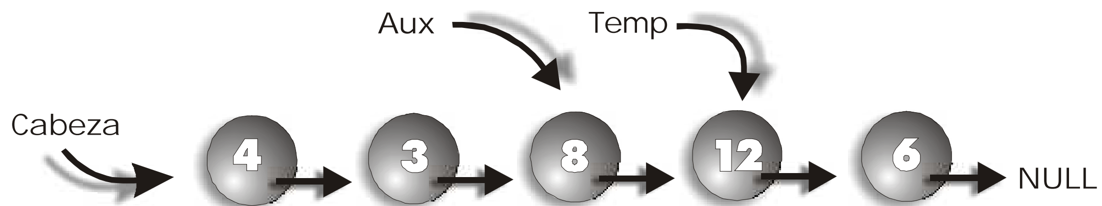
b) se desplaza el apuntador aux hasta que el nodo siguiente contenga el valor buscado o se llegue al último nodo (lo que significa que el valor no está en la lista)

```
while (aux->sgte != NULL && aux->sgte->valor != dato)
        aux = aux -> sgte;
```



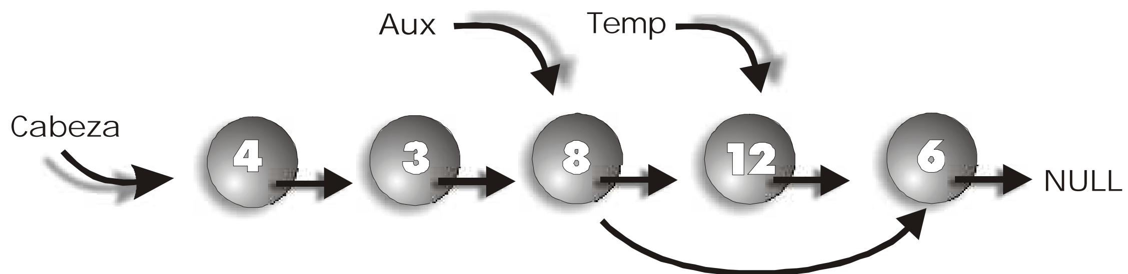
c) una vez terminado el ciclo de búsqueda hay dos posibilidades : el valor fue o no encontrado en la lista. Para determinar si el valor efectivamente está en la lista, basta con verificar si el campo siguiente del apuntador aux es NULL, de lo contrario el valor fue encontrado. Una vez determinada la existencia del dato, en el caso afirmativo se procede a posicionar un apuntador temporal (temp) en el nodo a eliminar con el fin de poder liberar la memoria por él ocupada.

```
temp = aux -> sgte;
```



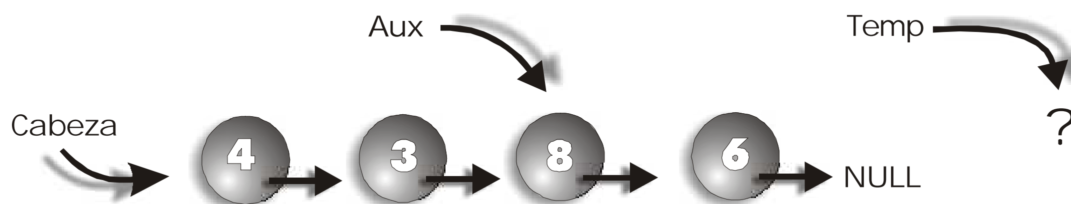
d) se adelanta el apuntador sgte del nodo anterior al que se quiere eliminar (aux), hasta el siguiente del que se desea borrar con el fin de reencadenar la lista.

```
aux -> sgte = temp -> sgte;
```



e) por último se libera la memoria del nodo eliminado, quedando la lista reencadenada y con los valores requeridos.

```
free(temp);
```



La función completa para eliminar un nodo de una lista es:

```
/* ---------------------------------------------------------------------------
    Elimina la primer ocurrencia de un dato en una lista
--------------------------------------------------------------------------- */
void eliminar ( int dato )
{
        struct nodo *aux, *temp;

        if (cabeza != NULL)
        {
                if (cabeza->valor == dato)
                {
                        aux = cabeza;
                        cabeza = cabeza->sgte;
                        free(aux);
                }
                else
                {
                        aux = cabeza;
                        while(aux->sgte != NULL && aux->sgte->valor != dato)
                                aux = aux -> sgte;
                        if (aux->sgte->valor == dato)
                        {
                                temp = aux->sgte;
                                aux->sgte = temp->sgte;
                                free(temp);
                        }
                }
        }
}
```

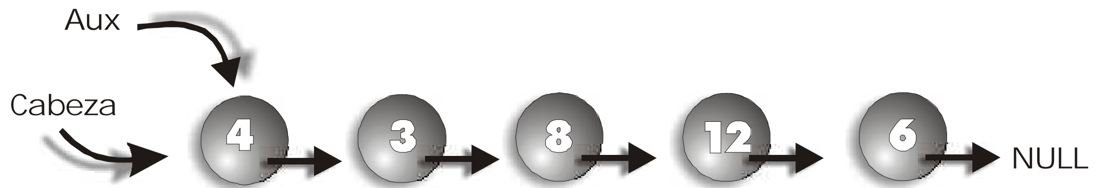
13.1.4. Recorrer una lista

Una lista encadenada simple se puede recorrer básicamente de dos maneras: desde la cabeza hacia la cola o de la cola hacia la cabeza.

Para recorrer una lista desde la cabeza hasta la cola (final de la lista), basta con posicionar un apuntador auxiliar en la cabeza y se desplaza hacia su siguiente hasta que dicho apuntador llegue al final de la misma.

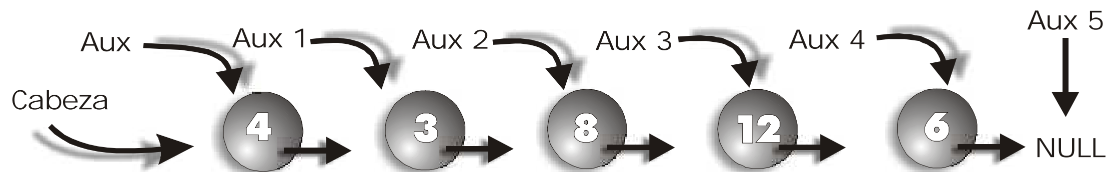
a) se posiciona un apuntador aux en la cabeza de la lista:

```
aux = cabeza;
```



b) mientras que el apuntador no llegue al final de la lista, se realiza la operación requerida con la información del nodo y se adelanta el apuntador aux hasta su siguiente:

```
while (aux != NULL)
{
        /* Operación que se quiere realizar (escribir, sumar, contar, buscar mayor, etc.) */
        aux = aux -> sgte;
}
```



aux 1, aux 2, ..., aux 5, indican cada una de las iteraciones del ciclo.

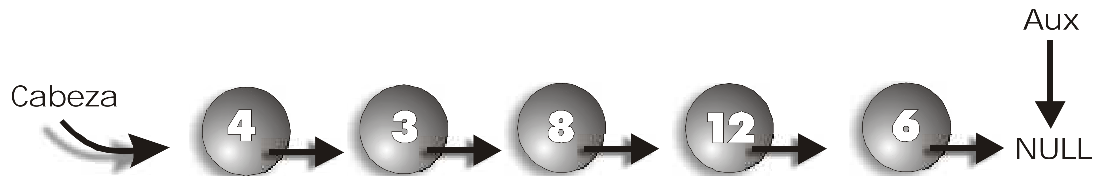
La función completa que recorre una lista desde la cabeza hasta la cola contabilizando el número de nodos allí almacenados es la siguiente:

```
/* ____________________________________________________________
Cuenta el número de elementos y/o nodos de una lista
_____________________________________________________________*/
 int contar(void)
 {
        struct nodo   *aux;
        int           cantidad = 0;

        aux = cabeza;
        while(aux != NULL)
        {
               cantidad ++;
               aux = aux->sgte;
        }
        return (cantidad);
 }
```

Es posible recorrer una lista desde la cola hasta la cabeza, teniendo en cuenta que al no existir un enlace entre un elemento y su anterior, es necesario utilizar una marca que indique el último elemento procesado, el cual se debe actualizar cada vez que se visita un nodo hasta llegar a la cabeza.

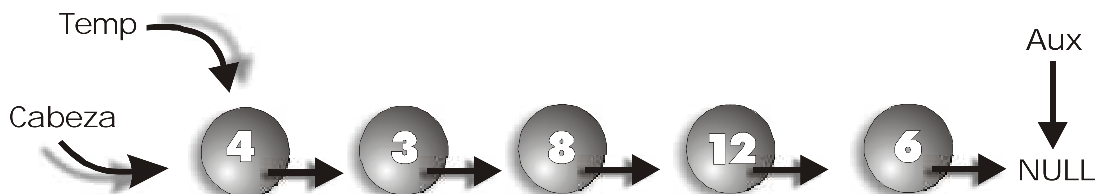
a) primero se posiciona una marca (apuntador aux) en el final de la lista (NULL),

```
aux = NULL;
```



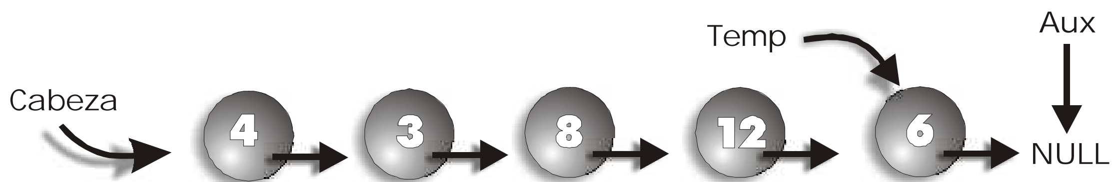
b) se posiciona un apuntador temporal en el primer elemento de la lista (cabeza):

```
temp = cabeza;
```



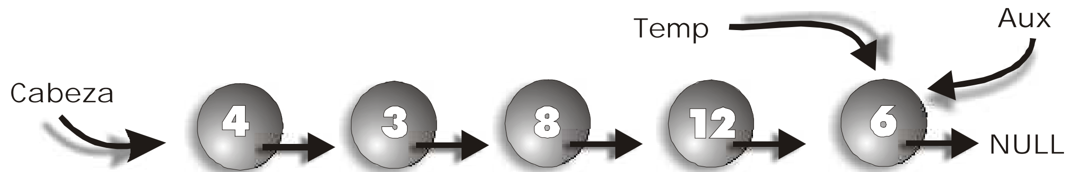
c) se recorre la lista con el apuntador temp hasta que su campo siguiente (sgte) coincida con la marca, permitiendo así realizar la operación requerida.

```
while (temp->sgte != NULL)
        temp = temp -> sgte;
```



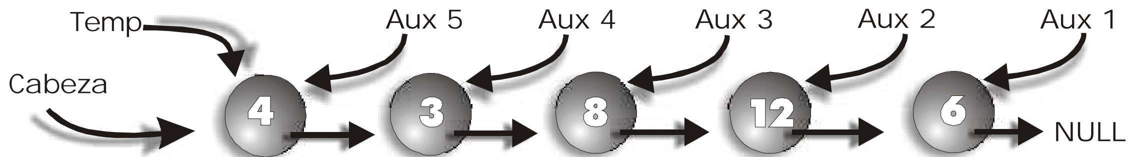
d) una vez realizada la operación requerida (escribir, contabilizar, verificar, etc.) se actualiza la posición del apuntador aux, llevándola a temp (nodo anterior).

```
aux = temp;
```



e) se repiten los pasos b, c, d, hasta que el apuntador aux sea igual a la cabeza de la lista.

```
while (cabeza != aux)
{
        temp = cabeza;
        while (temp->sgte != aux)
                temp = temp -> sgte;
        /* operaciones  con el nodo  temp */
        aux = temp;
}
```



aux 1, aux 2, ..., aux 5, indican cada una de las iteraciones del ciclo.

Una implementación completa de la función que escribe el contenido de una lista en orden inverso es la siguiente:

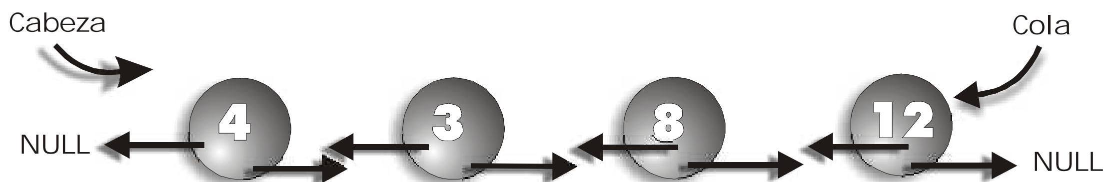
```
/* ______________________________________________________________
Escribe una lista desde la cola hasta la cabeza (forma invertida)
cabeza es una variable global.
______________________________________________________________*/

void escribirinv(void)
 {
        struct nodo *aux, *marca;

        marca = NULL;
        printf("\n Los elementos de la lista son : \n");
        while(cabeza != marca)
        {
                aux = cabeza;
                while (aux->sgte != marca)
```

```
                    aux = aux->sgte;
               printf("%d ",aux->valor);
               marca = aux;
          }
     }
```

13.2. Listas con encadenamiento doble

Son estructuras de datos conformadas por nodos unidos mediante apuntadores que indican el siguiente y anterior elemento que conforman la lista, se encuentran delimitadas por apuntadores NULL al principio y al final de la misma. Toda lista debe tener un apuntador base (denominado cabeza de la lista) que indique cual es la cabeza o primer elemento, el cual se debe manipular de forma diferente a los demás, siendo el identificador o representante de la lista. Adicionalmente se puede disponer de otro apuntador que indique la cola o último elemento de la lista (denominado cola de la lista). Debido a la naturaleza de las listas con encadenamiento doble, toda operación se debe realizar partiendo de la cabeza o de la cola hasta encontrar el elemento nulo, el cual se reconoce fácilmente, ya que su apuntador señala a NULL.

Gráficamente una lista doblemente encadenada con los elementos {4, 3, 8, 12}, se presenta de la siguiente manera:



13.2.1. Representación de una lista encadenada doble

Las listas doblemente encadenadas se representan mediante registros o estructuras recursivas, similares a las encadenadas simples, las cuales presentan dos de sus campos como apuntadores a elementos del mismo tipo, siendo estos los encargados de encadenar los demás elementos entre sí, formando la estructura de datos denominada lista doblemente encadenada.

La sintaxis se define de la siguiente manera:

```
struct nombre_del_registro {
     /* campos de información  */
     struct nombre_del_registro  *nombres_campos_encadenadores;
} lista_de_variables;
```

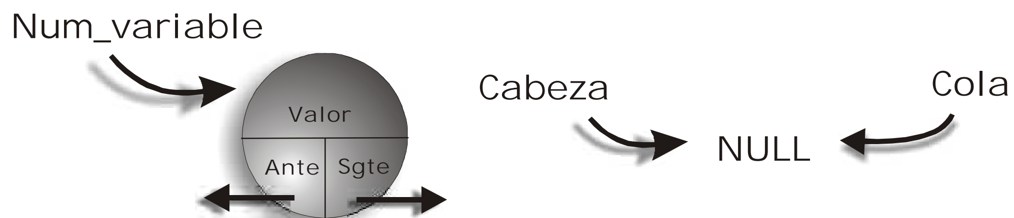
Las operaciones que se pueden realizar sobre una lista doblemente encadenada son las mismas que en listas encadenadas simples, teniendo la ventaja de que debido al doble encadenamiento y la presencia de dos apuntadores de base (cabeza y cola) es mucho más fácil realizar las diferentes operaciones.

```
/* ___________________________________________________________________________
Estructura de datos sobre la cual se define una lista doblemente encadenada.
____________________________________________________________________________*/
typedef struct nodo {
        int             valor;          /* Información a guardar en la lista */
        struct nodo     *ante, *sgte;/* Apuntadores a elementos del mismo tipo */
} Nodo;

Nodo *cabeza, *cola;
```

o su equivalente sin el uso de *typedef:*

```
/* ___________________________________________________________________________
Estructura de datos sobre la cual se define una lista doblemente encadenada.
____________________________________________________________________________*/
struct nodo {
        int             valor;          /* Información a guardar en la lista */
        struct nodo     *ante, *sgte;/* Apuntadores a elementos del mismo tipo */
} *cabeza, *cola;
```

Un nodo de tipo *struct* nodo presenta la siguiente configuración:



13.2.2. Insertar un elemento

Existen las mismas tres formas básicas de insertar un elemento en una lista encadenada doble que en una lista encadenada simple: insertar por la cabeza, insertar por la cola o insertar en forma ordenada.

Para la inserción de un elemento por la cabeza se deben seguir los siguientes pasos:

a) inicialmente se debe tener una lista vacía (cabeza y cola deben apuntar a NULL)

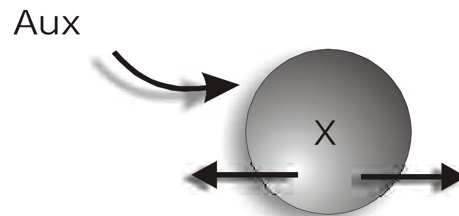
```
cabeza = cola = NULL;
```

b) se debe utilizar una variable del tipo apuntador para generar un nuevo nodo (separar y asignar la memoria requerida para almacenar la información)

```
aux = (struct nodo *) malloc (sizeof(struct nodo));
```

c) se verifica si la operación de reserva de memoria pudo realizarse satisfactoriamente, de lo contrario el apuntador aux apuntará a NULL, indicando que no se encontraba disponible la cantidad de memoria requerida, siendo necesario evitar que el proceso continúe y presentar el mensaje apropiado.

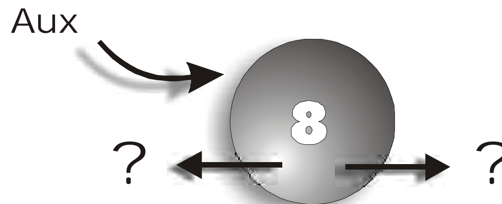
```
if (aux == NULL)
        printf("\n No hay memoria suficiente ");
```



el nuevo nodo que se crea tiene un valor indefinido para su campo dato (información) y dos apuntadores ante y sgte que no tienen especificado el sitio o lugar al cual apuntan. (deben inicializarse)

d) se almacena en el nodo el dato que se desea insertar, utilizando el campo destinado para tal fin (valor).

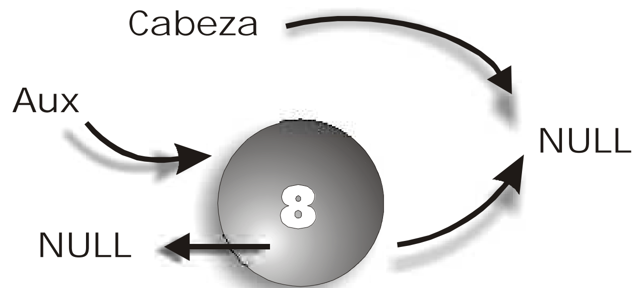
```
aux->valor = dato;
```



e) se redirige el apuntador sgte del nuevo nodo (aux) hacia el lugar donde apunta cabeza, para así

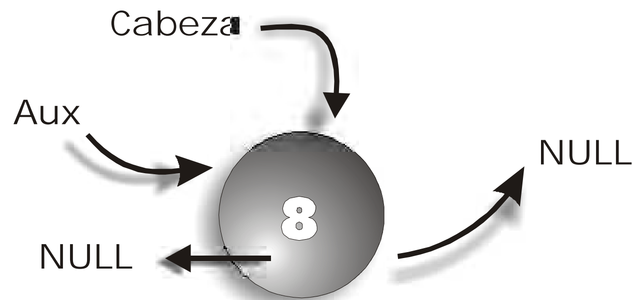
insertar el nodo en la lista y el apuntador ante se dirige a NULL, para indicar que no hay elementos antes del primero.

```
aux->sgte = cabeza;
aux->ante = NULL;
```



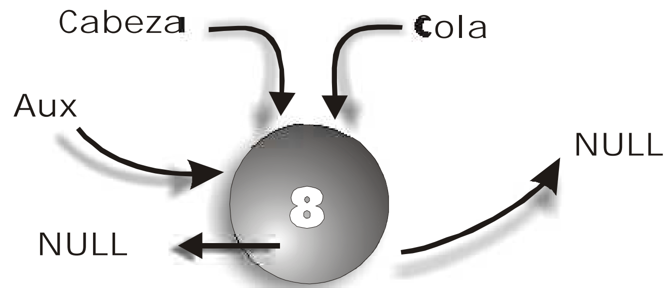
f) se redirecciona el apuntador que marca la cabeza de la lista (cabeza), hacia el nuevo nodo creado, terminando así el proceso de inserción.

```
cabeza = aux;
```

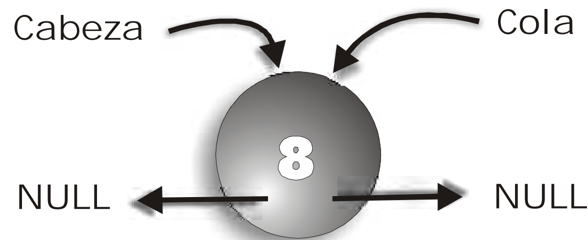


g) se verifica si el apuntador del último elemento (cola) está en NULL (la lista estaba vacía), y de ser así se modifica la dirección de dicho apuntador con el nuevo nodo, de lo contrario se enlaza el anterior de la cabeza con el nuevo nodo y se actualiza la cabeza de la lista.

```
If (cabeza == NULL || cola == NULL)
        cabeza = cola = aux;
else
{
        cabeza->ante = aux;
        cabeza = aux;
}
```



h) reorganizando los apuntadores, la lista queda:



el apuntador aux por ser de carácter local, deja de aparecer, pero sus acciones se mantienen mientras que dure el programa o dicha memoria no sea liberada.

La función completa que inserta un nodo en una lista por la cabeza se escribe de la siguiente manera:

```
/* ------------------------------------------------------------------------
   insertando un nuevo nodo en la cabeza de la lista, dado un valor determinado
------------------------------------------------------------------------ */
void insertarcab(int dato)
{
        struct nodo *aux;

        aux = (struct nodo *) malloc(sizeof(struct nodo));
        if (aux == NULL)
                printf("\n No hay memoria suficiente ");
        else
        {
                aux->valor = dato;
                aux->sgte = cabeza;
                aux->ante = NULL;
                 if (cabeza == NULL || cola == NULL)
                        cabeza = cola = aux;
                else
                {
```

```
                    cabeza->ante = aux;
                    cabeza = aux;
               }
          }
     }
```

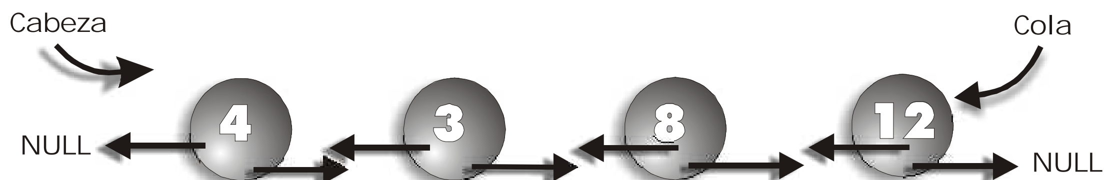
13.2.3. Eliminar un elemento

Para eliminar un nodo de una lista doblemente encadenada es necesario tener en cuenta tres casos diferentes; primero, si el elemento a eliminar se encuentra en la cabeza; segundo, si el elemento a eliminar se encuentra en la cola, y tercero, si el elemento a eliminar está entre la cabeza y la cola.

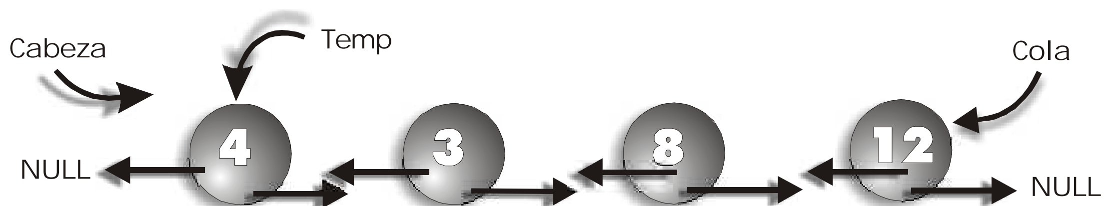
El primer y segundo casos son similares a eliminar en listas simplemente encadenadas, siendo necesario cambiar la cabeza o la cola según sea el caso y modificar el apuntador sgte o ante, llevándolo a NULL. Para el tercer caso es necesario recorrer la lista con un apuntador temporal (temp) hasta encontrar el nodo a eliminar, una vez encontrado se procede a reencadenar los diferentes nodos teniendo en cuenta que es necesario que el apuntador sgte del nodo anterior al nodo a eliminar, apunte al nodo siguiente del nodo a eliminar, y que el apuntador ante del nodo siguiente al nodo a eliminar apunte al nodo anterior del nodo a eliminar; por último, se libera la memoria utilizada por el nodo.

Dada una lista doblemente encadenada con los elementos {4, 3, 8, 12}, eliminar el nodo cuyo contenido es el número 8.

La situación inicial es:

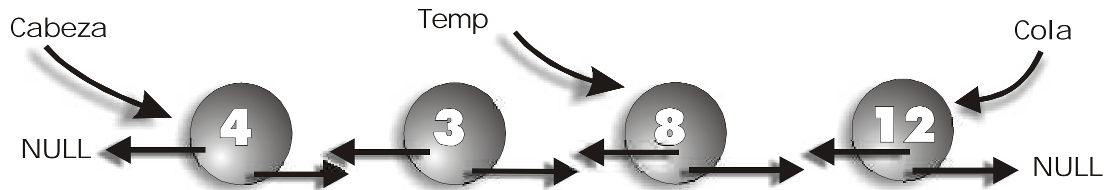


a) se posiciona el apuntador temporal (temp) en la cabeza de la lista:



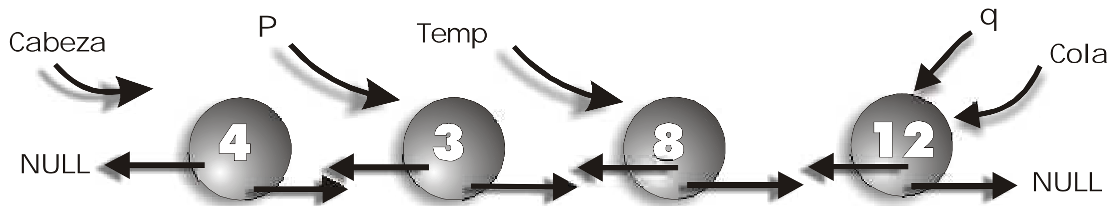
c) se recorre la lista hasta encontrar el elemento a eliminar o que se termine la lista.

```
while (temp != NULL && temp->valor !=dato)
        temp = temp->sgte;
```



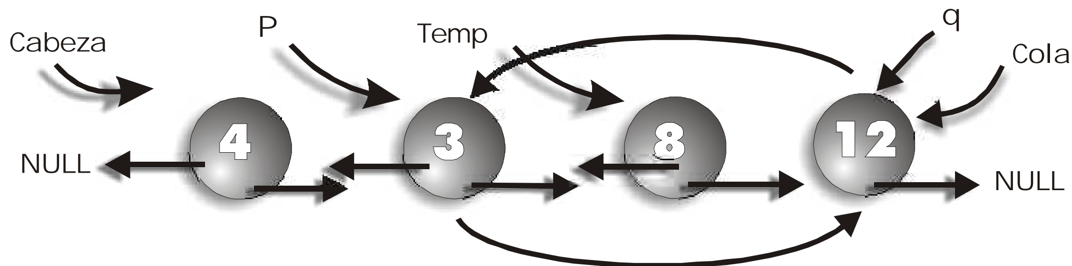
d) se procede a reencadenar los nodos, siendo necesario encadenar el anterior con el siguiente y viceversa. (para mayor comodidad se utilizan dos apuntadores adicionales en el anterior y el siguiente del auxiliar)

```
p = temp->ante;
q = temp->sgte;
```



El anterior del nodo a eliminar (p) debe apuntar al siguiente del nodo a eliminar (q), y el apuntador ante del siguiente del nodo a eliminar (q), siempre y cuando éste no apunte a NULL, debe apuntar al anterior al nodo a eliminar (p).

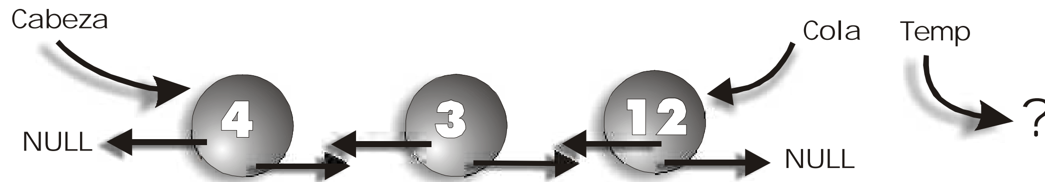
```
p->sgte = q;
q->ante = p;
```



e) se libera la memoria utilizada por el nodo temp

free(temp);

una vez reorganizados los apuntadores, la representación gráfica queda así:




```
/* ________________________________________________________________________
Eliminando la primer ocurrencia de un valor determinado, comenzando desde la cabeza
__________________________________________________________________________*/
void eliminar( int dato )

{
	struct nodo *temp, *p, *q;

	if (cabeza == NULL || cola == NULL) // La lista esta vacía
	{
		puts("\nLa lista esta vacía");
		return;
	}
	if (cabeza->valor == dato)	// El elemento a eliminar está en la cabeza
	{
		temp = cabeza;
		cabeza = cabeza->sgte;
		 if (cabeza == NULL)	// La lista tenía un solo elemento
			cola = NULL;
		else
			cabeza->ante = NULL;
		free(temp);		// Se libera la memoria
	}
	else
	{
		temp = cabeza;
		while (temp != NULL && temp->valor != dato) // Se recorre hasta encontrarlo o
		terminar la lista
			temp = temp->sgte;
		if (temp != NULL)
		{
			p = temp->ante;
```

```
                q = temp->sgte;
                p->sgte = q;
                if (q != NULL)      // Se valida que no sea el último elemento de la lista
                        q->ante = p;
                else
                        cola = p;
                free(temp);         // Se libera la memoria
            }
            else
            puts("\nEl elemento no está en la lista");
        }
    }
```

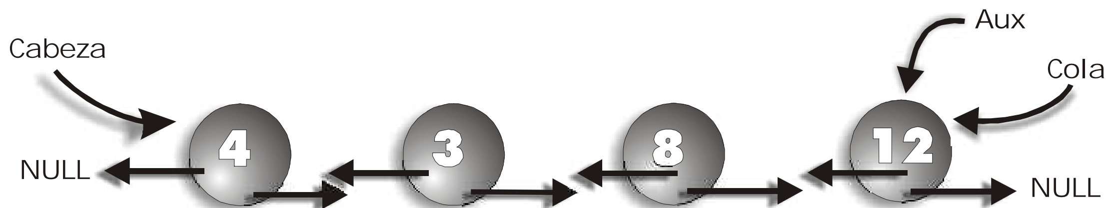
13.2.4. Recorrer una lista

Una lista encadenada doble se puede recorrer de dos maneras diferentes: desde la cabeza hasta la cola, o desde la cola hasta la cabeza.

Para recorrer una lista desde la cola hasta la cabeza (el de la cabeza a la cola es similar al de las listas encadenadas simples), basta con ubicar un apuntador auxiliar en la cola y se mueve hacia su nodo anterior mediante el apuntador ante, hasta que dicho apuntador llegue al comienzo de la lista (NULL)

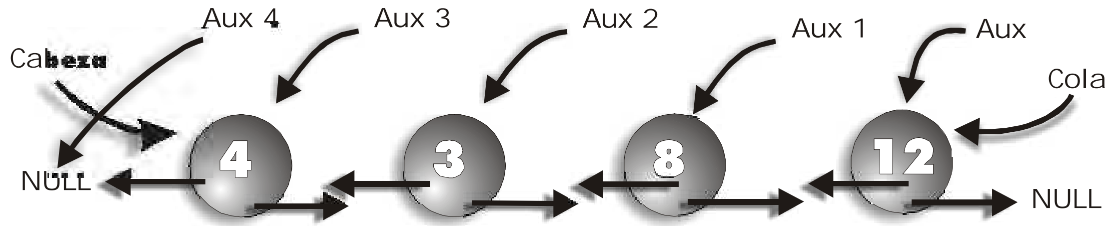
a) se posiciona un apuntador aux en la cola de la lista:

aux = cola;



b) mientras que el apuntador aux no llegue hasta el comienzo de la lista, se realiza la operación requerida con la información del nodo y se mueve el apuntador hacia su campo ante.

```
        while (aux != NULL)
        {
                /* Se realizan las operaciones requeridas con la información, tales como:
                escribir, sumar, buscar mayor, menor, etc.  */
                aux = aux -> ante;
        }
```



aux 1, aux 2, aux 3 y aux 4, indican cada una de las posiciones del apuntador aux, según las iteraciones del ciclo.

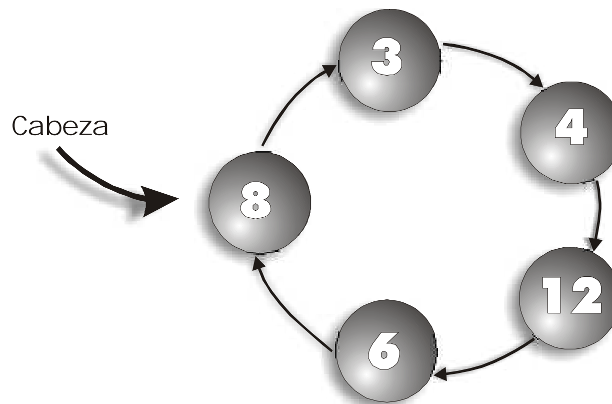
13.3. Estructuras de datos avanzadas

Adicionalmente, se puede construir innumerables estructuras de datos haciendo uso de los apuntadores, entre las cuales vale la pena mencionar: listas circulares, pilas, colas, árboles, grafos, matrices, entre otras.

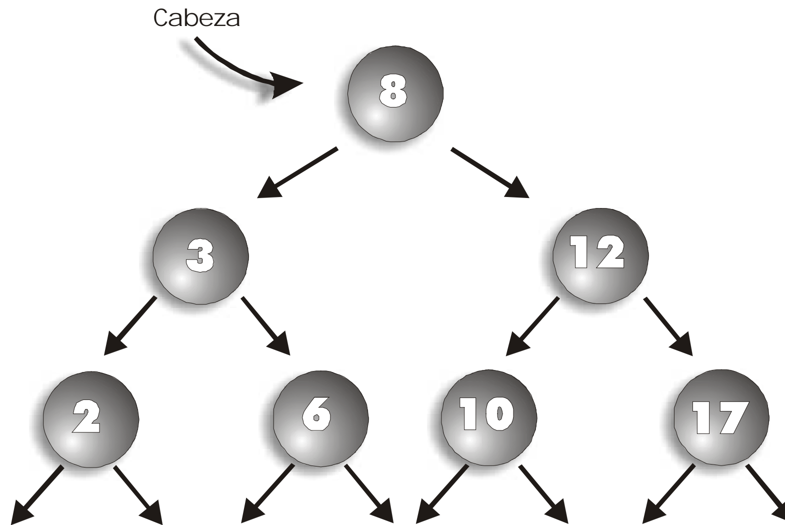
Las estructuras de datos avanzadas solo serán presentadas como ejemplo del potencial que proveen lenguajes de programación como el C, no siendo este el objetivo del libro, debido al grado de profundización requerido para su completo entendimiento y utilización. Para una mayor profundización en el tema se recomienda las referencias [SAN88], [TEN93] y [VILL 95].

La descripción y representación gráfica de algunas de las principales estructuras de datos, es la siguiente:

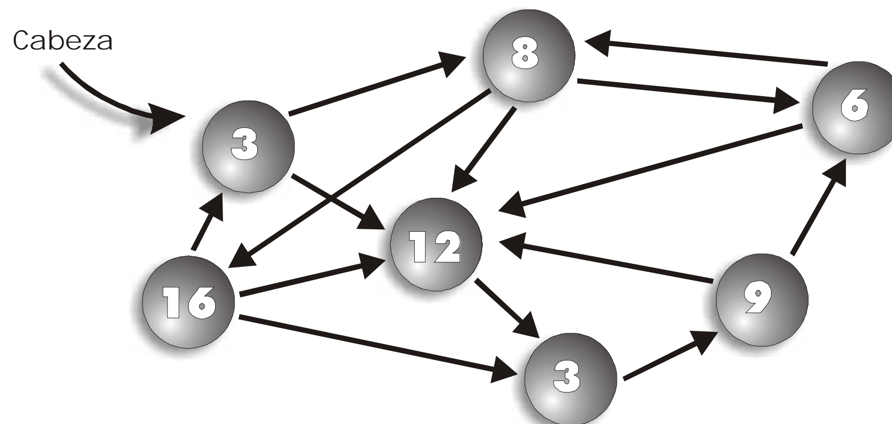
a) Listas circulares: Son estructuras tipo lista sin comienzo y fin definido, las cuales se caracterizan por se cíclicas, de tal manera que son aptas para la representación de información que requiera estar en constante movimiento, tal como la utilización del procesador de un computador en sistemas operacionales multitarea, simular manejo de rondas, etc.



b) Arboles: Son estructuras de datos jerárquicas, las cuales por su representación gráfica se asemejan a un árbol invertido el cual que se caracteriza por que cada nodo tiene dos o más enlaces con elementos del mismo tipo (el árbol binario tiene dos enlaces por nodo); los árboles están conformados por un primer nodo denominado raíz el cual representa la estructura de datos y es el único medio de acceso. Los nodos que se encuentran bajo otro se denominan hijos o descendientes. Un nodo sin hijos se conoce como nodo terminal o como hoja. La principal utilidad de los árboles radica en la posibilidad de almacenar información que implique jerarquías o niveles de información, tal como las estructuras de almacenamiento en los discos de los computadores (directorios o carpetas), representación de organigramas institucionales, entre otras.



c) Grafos: Son estructuras de datos que se caracterizan por las múltiples relaciones entre los nodos, dando la posibilidad de modelar y representar estructuras con encadenamientos heterogéneos en cuanto a la cantidad y disponibilidad, tales como: sistemas viales, oleoductos, acueductos, representación relacional en las bases de datos, diagramas de flujo, entre otras.



13.4. Ejercicios propuestos

1) Escriba una función que escriba una lista encadenada simple en orden inverso, haciendo uso de la recursión.

2) Invierta el contenido de una lista doblemente encadenada, modificando directamente los apuntadores.

3) Dada una lista simplemente encadenada, determine si el contenido es palíndromo o capicúa.

4) Ordene ascendentemente una lista encadenada simple de valores numéricos, cambiando los valores almacenados en cada uno de los nodos.

5) Defina una lista encadenada para simular el manejo de un vector y escriba las funciones que permitan su manipulación tales como:

- guardar(vector, posición, valor): almacena en la lista vector en la posición dada el valor especificado.
- conocer(vector, posición) : dada una lista y una posición entre la lista, regresa el valor almacenado en dicha posición.
- cantidad(vector): dada una lista representada por vector, regresa la cantidad de elementos almacenados.

6) Ordene en forma descendente una lista doblemente encadenada de valores numéricos, cambiando físicamente los apuntadores.

7) Implemente una matriz mediante el uso de apuntadores y calcule su determinante.

8) Escriba un programa que permita definir la suma de dos matrices implementadas por apuntadores, mostrando la respectiva respuesta.

9) Una pila es una estructura de datos en la cual se guarda información de tal manera que el último valor guardado es el primero en salir (LIFO o UEPS – último en entrar, primero en salir), implemente mediante listas la estructura de datos pila con las siguientes funciones:

- inicializar (pila): inicializa la estructura de datos para la pila.
- insertar (pila, valor): inserta el valor en la pila.
- vacía (pila): detecta si la pila está o no vacía.
- mirar (pila): regresa el último valor insertado en la pila (sin retirarlo)
- retirar (pila): retira el último valor de la pila.

10) Utilizando la estructura de datos pila definida en el ejercicio anterior, diseñe una función que lea una serie de valores y los escriba en el orden inverso a su lectura.

BIBLIOGRAFÍA

[ALB96] Albarracín, Mario., Alcalde, Eduardo., García, Miguel., "Introducción a la Informática", Mc Graw Hill, 1996.

[BOR93] Bores, R. Rosales, R., "Computación: Metodología, Lógica Computacional y Programación", Mc Graw Hill, Mexico, 1993.

[BRA97] Brassard, G. Bratley, T., "Fundamentos de Algoritmia", Prentice Hall, 1997.

[DEI95] Deitel, H. M. , Deitel, P. J., "Como Programar en C / C++", Segunda Edición, Prentice Hall, 1995.

[GLA93] Glassey, R., "Numerical Computation Using C", Academic Press Inc, 1993.

[GRI81] Gries, D., "The Science of Programming", Springer-Verlag, 1981.

[JOH97] Johnsonbaugh, R., Kalin, M. "C for Scientists and Engineers", Prentice Hall, 1997.

[KEL89] Kelley, Al., Pohl, Ira., "A Book on C", Benjamin Cummings, 1989.

[KER88] Kernighan B. W., Ritchie D. M., "The C Programming Language", Segunda Edición, Prentice Hall, 1988.

[RAM95] Ramírez, Luz A. "Análisis de Sistemas y Programación de Computadores", Universidad Autónoma de Manizales. Facultad de Ingeniería de Sistemas. Dirección de Tecnologías, 1995.

[SAN88] SANTAMARIA, C., "Estructuras de Datos en C.", Santafé de Bogotá, Impresores Ltda., 1988.

[TAM96] Tamayo, A. "Programación Estructurada un Enfoque Algorítmico", Universidad Nacional sede Manizales, 1996.

[TEN93] TENENBAUM, A., LANGSAM, Y., AUGENSTEIN, M. "Estructuras de Datos en C", Prentice Hall, 1993.

[TRE82] Tremblay, J. P., Bunt, R. B., "Introducción a la Ciencia de las Computadoras, Enfoque Algorítmico ", Mcgraw Hill, Mexico, 1982.

[VILL95] Villalobos, J., "Diseño y Manejo de Estructuras de Datos en C", Departamento de Ingeniería de Sistemas, Universidad de los Andes, 1995.

ANEXOS

ANEXO 1: TABLA DE CARACTERES ASCII

	0	1	2	3	4	5	6	7	8	9
0	**nul**	**soh**	**stx**	**etx**	**eot**	**enq**	**ack**	**bel**	**bs**	**ht**
1	**lf**	**vt**	**np**	**cr**	**so**	**si**	**dle**	**dc1**	**dc2**	**dc3**
2	**dc4**	**nak**	**syn**	**etb**	**can**	**em**	**sub**	**esc**	**fs**	**gs**
3	**rs**	**us**	**sp**	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}		**del**		

Significado de las abreviaturas:

ack	acknowledge	reconocimiento de transmisión
bel	bell	campanilla audible
bs	back space	retroceso
can	cancel	cancelar
cr	carriage return	retorno de carro
dc1	device control 1	control de dispositivo 1
dc2	device control 2	control de dispositivo 2
dc3	device control 3	control de dispositivo 3
dc4	device control 4	control de dispositivo 4
del	delete	Borrar un caracter
dle	data link escape	escape de enlace de datos
em	end of medium	final del medio
enq	enquiry	petición de transmisión
eot	end of transmission	fin de transmisión
esc	escape	escape / salida
etx	end of text	final de texto
ff	form feed	avance de página (impresora)
fs	file separator	separador de archivos

gs	group separator	separador de grupos
ht	horizontal tabulation	tabulador horizontal
lf	line feed	avance de línea
nak	negative acknowledge	transmisión / reconocimiento negativo
nul	null	caracter nulo
rs	record separator	separador de registros
si	shift in	poner desplazador de bits
so	shift out	quitar desplazador de bits
soh	start of heading	marca de comienzo de cabecera
stx	start of text	marca de comienzo de texto
sub	substitution	sustitución
syn	synchrounous idle	espera sincrónica
us	unit separator	separador de unidades
vt	vertical tabulation	tabulador vertical

Se puede hacer un pequeño programa en lenguaje C, que permite conocer los diferentes caracteres del código ascii, así:

```
#include <stdio.h>

void main ( void )
{
        int   x;

        for (x=1; x<= 255; x++)
        {
                printf("\n Codigo : %d, Caracter : %c", x, x);
                 /* Para visualizarlos en grupos de 24 caracteres */
                if ( x%24 == 0)
                {
                        printf("\n Teclee <Return> para continuar");
                        getchar();
                }
        }
}
```

ANEXO 2

FUNCIONES MATEMÁTICAS BÁSICAS (BORLAND C 3.1)

A. Operadores matemáticos:

+	suma entre dos cantidades numéricas
-	diferencia o resta entre dos cantidades numéricas
*	multiplicación entre dos cantidades numéricas
/	división entre dos cantidades numéricas
%	residuo de la división entre dos cantidades numéricas

B. Funciones matemáticas definidas en el archivo de encabezados math.h

double acos (double)	arco coseno de un ángulo en radianes
double asin (double)	arco seno de un ángulo en radianes
double atan (double)	arco tangente de un ángulo en radianes
double atan2 (double, double)	arco tangente de y / x
double ceil (double)	redondea por encima un número tipo *double*
double cos (double)	coseno de un ángulo en radianes
double cosh (double)	coseno hiperbólico de un ángulo en radianes
double exp (double)	exponencial de un número tipo *double*
double fabs (double)	valor absoluto de un número tipo *float*
double floor (double)	redondea por debajo un número tipo *double*
double fmod (double, double)	modulo o residuo entre dos números de tipo *double*
double frexp (double, int)	divide un número en mantisa y exponente
double ldexp (double, int)	calcula x * 2**exp, x es tipo *double* y exp es tipo *int*
double log (double)	logaritmo neperiano de un número tipo *double*
double log10 (double)	logaritmo en base 10 de un número tipo *double*
double modf (double, double)	parte un número en su entero y decimal respectivo
double pow (double, double)	exponenciación entre dos números tipo *double*
double sin (double)	seno de un ángulo en radianes
double sinh (double)	seno hiperbólico de un ángulo en radianes
double sqrt (double)	raíz cuadrada de un número
double tan (double)	tangente de un ángulo en radianes
double tanh (double)	tangente hiperbólica de un ángulo en radianes
long double acosl (long double)	arcocoseno de un ángulo en alta precisión
long double asinl (long double)	arcoseno de un ángulo en alta precisión
long double atan2l (long double , long double)	arcotangente del cociente Y/X en alta precisión
long double atanl (long double)	arcotangente de un ángulo en alta precisión
long double ceill (long double)	redondea por encima un valor tipo *double*
long double coshl (long double)	coseno hiperbólico de un ángulo en alta precisión

long double cosl (long double)	coseno de un ángulo en alta precisión
long double expl (long double)	encuentra e elevado a una potencia en alta precisión
long double fabsl (long double)	valor absoluto de un número de pto. flotante
long double floorl (long double)	redondea por debajo un valor tipo *double*
long double fmodl (long double , long double)	calcula el modulo de la división en un tipo *long double*
long double frexpl (long double , int)	parte un valor tipo *long* en mantisa y exponente
long double ldexpl (long double , int)	calcula (x * (2 ** exponente))
long double log10l (long double)	calcula el logaritmo en base 10 de un valor tipo *double*
long double logl (long double)	calcula el logaritmo en base e de un valor tipo *double*
long double modfl (long double , long double)	parte un valor tipo *double* en entero y decimal
long double powl (long double , long double)	eleva un número a un exponente que son de tipo *double*
long double sinhl (long double)	calcula el seno hiperbólico de un valor tipo *double*
long double sinl (long double)	calcula el seno hiperbólico de un valor tipo *double*
long double sqrtl (long double)	calcula la raíz cuadrada de un número tipo *double*
long double tanhl (long double)	calcula la tangente hiperbólica de un número tipo *double*
long double tanl (long double)	calcula la tangente de un número tipo *double*
int abs (int)	calcula el valor absoluto de un número de tipo *int*
double atof (const char *)	calcula el número tipo *float* equivalente a una cadena
double hypot (double , double)	calcula la hipotenusa de un triángulo rectángulo
long labs (long)	calcula el valor absoluto de un número entero largo
double poly (double , int, double *)	genera un polígono dados los coeficientes
double pow10 (int)	calcula 10 elevado a un valor tipo *int*
long double pow10l (int)	calcula 10 elevado a un valor tipo *int* con alta precisión
long double atol (const char *)	convierte una cadena en un valor tipo *long*
long double hypotl (long double , long double)	calcula la hipotenusa de un triángulo recto
long double polyl (long double, int, long double *)	genera un polígono dados los coeficientes de tipo *double*

C. Constantes numéricas básicas definidas en el archivo de encabezados math.h:

#define M_E	2.71828182845904523536	número de Euler (e)
#define M_LOG2E	1.44269504088896340736	logaritmo de M_E
#define M_LOG10E	0.434294481903251827651	logaritmo en base 10 de M_E
#define M_LN2	0.693147180559945309417	logaritmo natural de 2
#define M_LN10	2.30258509299404568402	logaritmo natural de 10
#define M_PI	3.14159265358979323846	número PI
#define M_PI_2	1.57079632679489661923	PI / 2
#define M_PI_4	0.785398163397448309616	PI / 4
#define M_1_PI	0.318309886183790671538	1 / PI
#define M_2_PI	0.636619772367581343076	2 / PI
#define M_1_SQRTPI	0.564189583547756286948	1 / √PI
#define M_2_SQRTPI	1.12837916709551257390	2 / √PI
#define M_SQRT21.	41421356237309504880	raíz cuadrada de 2 (√2)
#define M_SQRT_2	0.707106781186547524401	raíz de 2 dividida por 2

ANEXO 3

ERRORES DE EJECUCIÓN FRECUENTES

A continuación se presenta un conjunto de errores semánticos frecuentes a la hora de escribir un programa en el computador haciendo uso del lenguaje C, que por su naturaleza no son detectados por el compilador, haciendo que el programa funcione pero no produzca los resultados esperados. Estos errores normalmente no son fáciles de detectar por lo cual se sugiere verificar la sintaxis que aquí se presenta.

a. Problema: Condicionales que siempre se evalúan como ciertos o verdaderos

```
if (variable1 = variable2) {
        instrucciones
}
```

Planteamiento : La sentencia if se comporta como una asignación en la cual el valor de la variable2 se lleva a la variable1, dando como resultado el valor de la asignación. Se debe tener en cuenta que un condicional se evalúa como verdadero para cualquier valor diferente de cero (0); razón por la cual este condicional solo se evaluará como falso cuando la variable2 tenga como valor cero (0).

Solución : La comparación de igualdad para dos cantidades numéricas siempre se debe realizar con doble símbolo igual (==)

```
if (variable1 == variable2) {
        instrucciones
}
```

b. Problema: Un ciclo for que independientemente de las condiciones o valores que se especifiquen solo se realiza una vez.

```
for (x=0; x<5; x++);
        printf("%d", x);
```

Se esperaría que escribiera los valores entre 0 y 4, pero en realidad únicamente escribe el valor 5.

Planteamiento: El punto y coma (;) ubicado al final de la línea en que se encuentra la instrucción for, se comporta como una instrucción nula o vacía, lo cual tiene como resultado que cierra el dominio del ciclo, ejecutando cinco veces la instrucción nula; el valor cinco lo escribe una vez terminada la totalidad del ciclo, siendo esta la siguiente instrucción a ejecutar en la jerarquía del programa.

Solución: El programa se corrige eliminando el punto y coma (;) al final de la instrucción for.

```
for (x=0; x<5; x++)
```

```
        printf("%d", x);
```

c. Problema: Un ciclo while que a pesar de tener las condiciones de terminación apropiadas para su terminación, nunca termina.

```
x=0;
while (x<5);
{
        printf("\n%d", x);
        x++;
}
```

Planteamiento : El punto y coma puesto al final de la instrucción while cierra el ciclo de ejecución del mismo (tal como se planteó en el problema anterior), evitando que las condiciones de avance hacia la terminación se realicen (incrementar la variable x) al igual que instrucción de salida printf, razón por la cual la condición de salida del mismo nunca se alcanzará quedando el programa en un ciclo indefinido.

Solución: El problema se corrige eliminando el punto y coma (;) mencionado así:

```
x=0;
while (x<5)
{
        printf("\n%d", x);
        x++;
}
```

d. Problema : Función gets que no lee la información (se salta la instrucción sin ejecutarla)

Dado el siguiente programa, la instrucción gets(nombre); señalada no se ejecuta; el compilador parece no tenerla en cuenta.

```
#include<stdio.h>

void main(void)
{
        int     numero, x;
        float   nota;
        char    nombre[30];

        printf("\nNumero de estudiantes : ");
scanf("%d", &numero);
for (x=0; x<numero; x++)
{
```

```
                printf("\nTeclee el nombre : ");
⇒               gets(nombre);
                printf("\nTeclee la nota : ");
                scanf("%f", &nota);
        }
 }
```

Planteamiento : Este problema ocurre debido a que cada vez que la función scanf lee un dato proveniente del teclado, únicamente toma de éste aquellos valores que le interesan; al final de toda lectura utilizando la función *scanf* se debe digitar la tecla <enter>, la cual tiene asociado un caracter del código ascii, el cual no es leído por la función scanf permaneciendo en el buffer, razón por la cual si posterior a dicha lectura se utiliza la función gets, la cual lee caracteres del teclado hasta encontrar un caracter de fin de línea o <enter>, tomará dicho caracter almacenado en el buffer como la cadena esperada para la lectura.

Solución: Cada vez que se realice una lectura mediante la instrucción scanf es necesario desocupar el buffer con el fin de retirar el caracter de fin de línea o <enter> y evitar así que posteriores llamados a la función gets puedan "fallar". Esto se logra añadiendo un llamado a la macrofunción getchar( ) luego de cada llamado a la función *scanf*.

```
#include<stdio.h>

void main(void)
{
        int   numero, x;
        float   nota;
        char  nombre[30];

        printf("\nNumero de estudiantes : ");
        scanf("%d", &numero); getchar( )
        for (x=0; x<numero; x++)
        {
                printf("\nTeclee el nombre : ");
                gets(nombre);
                printf("\nTeclee la nota : ");
                scanf("%f", &nota); getchar( );
        }
 }
```

e. Problema: División de dos valores numéricos que no regresa la respuesta correcta.

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

```
void main(void)
{
        int      dividendo, divisor;
        float   resultado;

        printf("\nValores a dividir :  (dividendo, divisor) : ");
        scanf("%d%d", &dividendo, &divisor);
        resultado = dividendo / divisor;
        printf("\El resultado es  : %f");
}
```

Ejecutando el anterior programa con valores de 10 y 3 para dividendo y divisor respectivamente, el resultado es 3.0000, donde el resultado esperado sería de 3.3333.

Planteamiento : Este problema ocurre cuando se requiere efectuar operaciones aritméticas entre números de un determinado tipo (ej. enteros) y el resultado de su operación es de un tipo de mayor capacidad o rango (ej. flotantes).

Solución : Los operadores aritméticos regresan resultados según el mayor tipo entre sus operadores. De esta manera la división entre dos números enteros dará como resultado un número entero, así su resultado no lo sea; es necesario obligar que uno de los valores que intervienen en la operación sea de un tipo mayor, acción que se logra mediante el casting de valores, anteponiendo el tipo a uno de los operadores, de tal manera que el mayor tipo de la operación sea el requerido en la respuesta.

```
        #include<stdio.h>

        void main(void)
        {
                int      dividendo, divisor;
                float   resultado;

                printf("\nValores a dividir :  (dividendo, divisor) : ");
                scanf("%d%d", &dividendo, &divisor);
⇒              resultado = (float) dividendo / divisor;
                printf("\El resultado es  : %f");
        }
```

El tipo *int* de la variable dividendo es evaluado en la división como un *float*, debido al casting o cambio de tipo, operando un valor tipo *float* y un valor tipo *int*, dando como resultado un valor tipo *float*.

Impreso en el Centro de Publicaciones
de la Universidad Nacional de Colombia Sede Manizales

Apoyo Académico
Informática