

REPÚBLICA BOLIVARIANA DE VENEZUELA
UNIVERSIDAD PRIVADA DR. RAFAEL BELLOSO CHACÍN
VICERRECTORADO ACADÉMICO
FACULTAD DE INGENIERÍA
ESCUELA DE INFORMÁTICA



Programación I

Primera Edición

María Eugenia Fossi Medina
Maribel del Carmen Medina Parra
María Lourdes Geizzelez Luzardo
Ángel Gerardo Pérez Fernández
Galo Luis Shiera Prieto
Inesmar Carolina Briceño Rivero
Álvaro Alberto Sánchez Colmenares

Fondo Editorial de la Universidad Privada "Dr. Rafael Bellosso Chacín"
Maracaibo - Venezuela
Año 2019

*María Fossi, Maribel Medina, María Geizzelez, Ángel Pérez,
Galo Shiera, Inesmar Briceño, Álvaro Sánchez.*



Programación I
Primera Edición

No está permitida la reproducción total o parcial de este libro, ni su tratamiento o procesamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otro método, así como la distribución de ejemplares mediante alquiler o préstamo público, sin el permiso previo y por escrito de los titulares del copyright.

DERECHOS RESERVADOS 2019

Fondo Editorial de la Universidad Privada "Dr. Rafael Bellosó Chacín"
Circunvalación N° 2, frente a la Plaza de Toros. Maracaibo, Venezuela
Correo Electrónico: fondurbe@urbe.edu
Página Web: www.urbe.edu/investigacion/fondoeditorial

HECHO EL DEPÓSITO DE LEY

ISBN: 978-980-414-020-4

Depósito Legal: LF77820162011968

Montaje del libro: Lcda. Neibeth León.

Ilustración de la portada: Lcda. Neibeth León

Publicación electrónica - Año 2019.

Publicado por el Fondo Editorial en conjunto con la Dirección de Tecnología de la Información de la Universidad Privada Dr. "Rafael Bellosó Chacín".
Maracaibo - Venezuela. Teléfono: +58 261 200-URBE (8723).

Programación I

Primera Edición

Autoridades Universitarias

Dr. Oscar Bellosó Medina
Rector Fundador/Presidente del Consejo Superior

Dr. Oscar Bellosó Vargas
Rector

Dr. Mike González Bermúdez
Vicerrector Académico

MSc. Ángel Alexander Villasmil Rangel
Vicerrector Administrativo

Dr. Humberto Perozo Reyes
Secretario

Dra. Janeth Hernández Corona
Decana de Investigación y Postgrado

Dr. Plácido Martínez Paz
Decano de la Facultad de Ingeniería

Dra. Betty Margarita Galavíz Ramírez
Decana de la Facultad de Ciencias Administrativas

Dra. Lisbeth Fuenmayor Leal
Decana de la Facultad de Ciencias Jurídicas y Políticas

Dra. Marilyn Lescher
Decana de la Facultad de Humanidades y Educación

Dra. Janett del Valle Pirela González
Decana de la Facultad de Ciencias de la Informática

Dra. Adinora Oquendo Garcés
Decana de Extensión

Fondo Editorial

Dr. Roberto Enrique Bozo Acosta
Director

Dra. María Villalobos
Editor

MSc. María Rojas
Editor

Lcda. María José Paloscia
Editor Gráfico

T.S.U. Dianela Prieto
Asistente

DEDICATORIA

*A nuestros padres, sin ellos no existiríamos.
A nuestros hermanos, la unión hace la fuerza.
A nuestros hijos, la razón de luchar por un país y un mundo mejor.*

RECONOCIMIENTOS

*Al M.Sc. Gustavo García y el Fondo Editorial URBE, por el apoyo prestado
en la finalización de nuestro libro.*

A la profesora Yenice Arámbulo, por su ayuda incondicional.

Los autores.

INTRODUCCIÓN GENERAL A LA ASIGNATURA

El proceso de diseñar, codificar, depurar y mantener el software para controlar los equipos computarizados se denomina programación, y su propósito es crear programas que permitan un comportamiento deseado. El proceso para escribir un código, necesita del dominio de varias áreas del saber, lógica, matemáticas, estadística, conocimientos administrativos, entre otros, así como, el dominio de uno o varios lenguajes de programación.

En este sentido, al referirnos al lenguaje de programación, estamos planteando que éste es una herramienta diseñada para describir el conjunto de acciones consecutivas a ejecutar por un componente electrónico, por tanto, es una forma práctica que tienen los humanos para comunicarse con los equipos.

Existen básicamente dos tipos de lenguajes de programación. Los lenguajes de bajo nivel, los cuales son totalmente dependientes de las máquinas, es decir, los programas realizados con ellos, no se pueden migrar o transportar a otra máquina por estar prácticamente diseñados a la medida del hardware. Por otra parte, se tienen los lenguajes de alto nivel, estos están diseñados en función del programador, se trata de un lenguaje independiente al hardware y de fácil migración entre equipos y donde las sentencias son de fácil entendimiento.

Ahora bien, a pesar de no ser aceptado por todos los expertos en el área, muchos programadores hablan de los lenguajes de medio nivel, los cuales se encuentran en un punto medio entre los dos anteriores. Son lenguajes que permiten el acceso a los registros del sistema y trabajar direcciones de memoria, características éstas de los lenguajes de bajo nivel, asimismo, permiten realizar operaciones de alto nivel.

Dentro de estas características entra el Lenguaje C, calificado como la herramienta utilizada por la Universidad privada Dr. Rafael Belloso Chacín, para dictar la cátedra de Programación I. C es un lenguaje estructurado, creado en 1972 por Dennis Ritchie en los laboratorios Bell, fue elaborado inicialmente para la implementación de sistemas operativos, específicamente el UNIX. En la actualidad, se utiliza el estándar C++, el cual es una actualización del C básico, y presenta la posibilidad de redefinir los operadores y de poder crear nuevos tipos de datos que se comporten como tipos fundamentales.

Ante estos planteamientos, se presenta este libro, cuyo propósito es ofrecer las herramientas básicas necesarias para iniciar al estudiante en la utilización de este lenguaje de programación. El texto se encuentra



estructurado en cuatro unidades, la primera unidad denominada Resolución de problemas con herramientas de programación e introducción al lenguaje de programación C++, en esta unidad se busca mostrar al estudiante las sentencias de control básicas y la manera correcta de utilizarlas.

En la segunda unidad, Arreglos, se quiere iniciar al estudiante en las diferentes estructuras de datos que se tienen como herramienta para el manejo de la información. Seguidamente, en la tercera unidad, Funciones en Lenguaje C, se introduce al estudiante en la técnica de programación modular, la cual no es más que dividir el problema en diferentes tareas y cada una de ellas programarlas por separado con la finalidad de reutilizar los códigos ya existentes y optimizar el diseño del programa. Finalmente, en la cuarta unidad, Manejo de Cadenas, se le presenta al estudiante una serie de sentencias que permiten el manejo de cadenas de carácter, así como, su aplicación dentro de un código.

OBJETIVOS DEL PROGRAMA

Objetivo General

Implementar de forma eficiente los términos básicos de la Programación de Nivel Medio asociándola con estructuras de datos simples, utilizando como herramienta el lenguaje de programación C++.

Objetivos Específicos

- Explicar la resolución de problemas con herramientas de programación e introducción al lenguaje de programación C++.
- Describir los Arreglos como estructuras de datos para la realización de diferentes operaciones.
- Explicar las funciones del Lenguaje C como bloques de códigos para la elaboración de tareas específicas.
- Analizar el Manejo de Cadenas de caracteres para la representación de números, letras y símbolos.



ÍNDICE GENERAL

DEDICATORIA.....	5
RECONOCIMIENTOS.....	7
INTRODUCCIÓN GENERAL A LA ASIGNATURA.....	9
OBJETIVOS DEL PROGRAMA.....	11
OBJETIVO GENERAL.....	11
OBJETIVOS ESPECÍFICOS.....	11

UNIDAD I. RESOLUCIÓN DE PROBLEMAS CON HERRAMIENTAS DE PROGRAMACIÓN E INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN C++

1. ALGORITMO.....	17
1.2. Características de los algoritmos.....	19
2. PROGRAMA.....	20
3. DATO.....	23
3.1. Constante.....	25
3.2. Variable.....	25
3.3. Identificador.....	25
4. EXPRESIONES Y OPERADORES.....	26
5. CONTADOR.....	28
6. ACUMULADOR.....	29
7. VALOR CENTINELA.....	29
8. FORMAS DE REPRESENTACIÓN ALGORÍTMICAS.....	30
8.1. Pasos a seguir para elaborar un algoritmo.....	32
9. LENGUAJE C++: DEFINICIONES BÁSICAS.....	34
10. ESTRUCTURA DE UN PROGRAMA EN C++.....	35
11. ELEMENTOS BÁSICOS DE UN PROGRAMA EN C++.....	36
12. COMENTARIOS EN UN PROGRAMA EN C++.....	46
13. PALABRAS CLAVE DEL C++.....	47
14. SENTENCIAS DE CONTROL.....	52
14.1. Toma de decisión simple (if – else).....	52
14.2. Ciclo “Para” (for).....	54
14.3. Ciclo “Hacer – Mientras” (do – while).....	55
14.4. Ciclo “Mientras” (while).....	56
14.5. Toma de decisión múltiple (switch).....	56
15. COMPILACIÓN Y DEPURACIÓN DE UN PROGRAMA.....	58
ACTIVIDADES DE AUTOEVALUACIÓN.....	61



UNIDAD II. ARREGLOS

1. ARREGLOS: DEFINICIÓN.....	65
2. ARREGLOS UNIDIMENSIONALES (VECTORES).....	66
2.1. Tamaño del Arreglo.....	68
2.2. Operaciones con los Vectores.....	69
3. ARREGLOS BIDIMENSIONALES (MATRICES).....	78
3.1. Operaciones con las Matrices.....	79
4. MÉTODOS DE ORDENACIÓN.....	88
4.1. Ordenación por Selección.....	88
4.2. Ordenación por Burbuja.....	94
ACTIVIDADES DE AUTOEVALUACIÓN.....	101

UNIDAD III. FUNCIONES EN LENGUAJE C

1. FUNCIONES: DEFINICIÓN.....	105
2. PROTOTIPO DE UNA FUNCIÓN.....	106
2.1. Tiempo de vida de los datos o tipos de variables.....	107
3. PARÁMETROS.....	110
3.1. Paso de parámetros a una función.....	110
3.2. Los argumentos de la función main ().....	117
4. SOBRECARGA A FUNCIONES.....	117
5. FUNCIONES DE BIBLIOTECAS.....	118
ACTIVIDADES DE AUTOEVALUACIÓN.....	125

UNIDAD IV. MANEJO DE CADENAS

1. DEFINICIÓN.....	131
1.1. Caracteres y literales tipo cadena.....	132
2. LECTURA DE CADENAS.....	134
2.1. Operaciones de entrada y salida.....	134
2.2. Función: scanf.....	135
2.3. Función: sscanf.....	137
2.4. Función: getchar.....	138
2.5. Función: gets.....	138
2.6. Función: cin y cin.getline().....	139
2.7. Función: cin.get.....	140
2.8. Función: cout.put.....	140
3. LA BIBLIOTECA STRING.H.....	140
3.1. Función: memcpy.....	141
3.2. Función: memmove.....	142
3.3. Función: strcpy.....	143



3.4. Función: strncpy.....	143
3.5. Función: strcat.....	144
3.6. Función: strncat.....	145
3.7. Función: memcpy.....	145
3.8. Función: strcmp.....	146
3.9. Función: strncmp.....	147
3.10. Función: strxfrm.....	147
3.11. Función: memchr.....	148
3.12. Función: strchr.....	149
3.13. Función: strcspn.....	150
3.14. Función: strpbrk.....	150
3.15. Función: strrchr.....	151
3.16. Función: strspn.....	152
3.17. Función: strstr.....	152
3.18. Función: strtok.....	153
3.19. Función: memset.....	154
3.20. Función: strerror.....	154
3.21. Función: strlen.....	155
4. ASIGNACIÓN DE CADENAS.....	155
4.1. Inicialización de vectores para almacenar cadenas.....	156
5. LONGITUD Y CONCATENACIÓN DE CADENAS.....	156
5.1. Función: strlen.....	157
5.2. Función: strspn.....	157
5.3. Función: strcspn.....	158
5.4. Función: strcat.....	158
5.5. Función: strncat.....	159
6. OPERACIONES CON LAS CADENAS.....	160
ACTIVIDADES DE AUTOEVALUACIÓN.....	163
BIBLIOGRAFÍA Y OTRAS FUENTES DE CONSULTA.....	165



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

UNIDAD I

RESOLUCIÓN DE PROBLEMAS CON HERRAMIENTAS DE PROGRAMACIÓN E INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN C++



UNIDAD I

RESOLUCIÓN DE PROBLEMAS CON HERRAMIENTAS DE PROGRAMACIÓN E INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN C++

1. ALGORITMO

El origen del término *algoritmo* según Kerwin, N. (2009)<en línea> es atribuido al matemático árabe Abu Ja'far Muhammad Bin Musa al-Khwarizmi en el siglo IX. La palabra se refería originalmente solo a las reglas de la aritmética con números arábigos. No es hasta el siglo XVIII cuando se expande su significado para abarcar la definición dada por Joyanes, L. (2003: 40) quien señala que es un método para resolver problemas mediante una serie de pasos precisos, definidos y finitos, en otras palabras, estos representan una serie de operaciones detalladas y no ambiguas, a ejecutar paso a paso, y que conducen a la resolución de un problema.

Erróneamente, las personas tienden a asociar los algoritmos únicamente a las computadoras y los lenguajes de programación, no obstante, como afirman López, G. y otros (2009: 4), los algoritmos son utilizados en la vida cotidiana cuando cada uno de los individuos ejecuta acciones con la finalidad de obtener un resultado. Por tanto, el simple hecho de preparar una receta de cocina cualquiera, es un algoritmo. Se considera que el primer algoritmo escrito para una computadora, fueron las notas escritas por Ada Byron en 1842, dirigidas al motor analítico de Charles Babbage. Por esta razón, se reconoce a Ada Byron como la primera programadora de la historia, sin embargo, dado que Babbage nunca terminó su motor analítico, el algoritmo jamás llegó a implementarse.

Ahora bien, la utilización de las computadoras en diversas áreas laborales (ingeniería, medicina, gobierno, educación, entre otros), inclusive en la vida diaria, ha obligado a los expertos a crear diversos software para facilitar la utilización de los equipos. La forma de solucionar un problema en la computadora lleva a la escritura y ejecución de un programa, donde la elaboración del mismo es sin duda un proceso básicamente creativo, es importante tener en cuenta una serie de pasos a seguir.

Para Joyanes, L. (2003: 40) estos pasos constituyen el ciclo de vida de todo software, entre ellos:

- **ANÁLISIS:** razonamiento del problema teniendo presente las especificaciones de los requerimientos dados por el cliente y/o usuario.

- *DISEÑO*: luego de analizado el problema, se procede a diseñar una solución que conducirá a un algoritmo que resuelva el problema.
- *CODIFICACIÓN (IMPLEMENTACIÓN)*: la solución se escribe en la sintaxis del lenguaje seleccionado, obteniendo el programa.
- *COMPILACIÓN, EJECUCIÓN Y VERIFICACIÓN*: se ejecuta el programa, comprobando rigurosamente su proceso con la finalidad de eliminar todo tipo de errores presentes.
- *DEPURACIÓN Y MANTENIMIENTO*: el programa se actualiza y se modifica cada vez que sea necesario a manera de satisfacer las necesidades de cambio exigidas por el cliente.
- *DOCUMENTACIÓN*: escritura de las diferentes fases del ciclo de vida del software, primordialmente el análisis, diseño y codificación, unidos a manuales de usuarios y de referencia, así como, reglas para el mantenimiento.

Como se puede observar, las dos primeras fases llevan a un diseño detallado, el cual es escrito en forma de algoritmo y se puede considerar como la fase más importante del desarrollo, debido a ser la fase que permitirá al programador saber a ciencia cierta la manera como será desarrollado el software. Por tanto, es importante definir de forma más específica la palabra Algoritmo, así para López, G. y otros (2009: 4) es: "un conjunto de pasos que, ejecutados de la manera correcta, permite obtener un resultado".

Por otra parte, Joyanes, L. (2003: 41) indica que: "un algoritmo debe producir un resultado en un tiempo finito". Finalmente, Fabelo, R. y Medina, M. (2004: 15) indican que los algoritmos tienden a presentar múltiples soluciones o formas de realizarse, conformados por una serie de sentencias que posteriormente serán escritas en un lenguaje de programación específico.

De acuerdo con los planteamientos expuestos, se puede decir, que los algoritmos se refieren a una serie de operaciones detalladas y no ambiguas a ejecutar paso a paso, y que conducen a la resolución de un problema. En otras palabras, un algoritmo es un conjunto de reglas para resolver una cierta clase de problema o una forma de describir la solución de un problema. A partir de esta configuración conceptual, consideramos que los algoritmos pueden ser básicamente de dos tipos:

- *Algoritmos Cualitativos*: son todos aquellos pasos o instrucciones descritos por medio de palabras que sirven para la obtención de una respuesta o solución de un problema. Ejemplo: la elaboración



de una receta de cocina, el montaje de un caucho o la búsqueda de una dirección.

- *Algoritmos Cuantitativos*: son todos aquellos pasos o instrucciones que involucran cálculos numéricos para llegar a un resultado satisfactorio. Ejemplo: pasos a seguir en la solución de una ecuación de segundo grado, pasos matemáticos para la solución de un factorial o las instrucciones para la liquidación de una nómina.

1.2. Características de los Algoritmos

Al desarrollar un algoritmo para dar solución a un problema determinado, se debe hacer el análisis del problema previo, puesto que el algoritmo es el paso previo a la construcción del programa que ejecutará la computadora. Por este motivo, debe haber coherencia y relación en cada uno de los pasos establecidos. El orden en que se establecen los pasos a seguir en el algoritmo debe ser riguroso y no ambiguo.

El algoritmo es el paso previo al programa, y cuando éste se traslada al lenguaje escogido para representarlo, se debe conservar el orden preestablecido en él, independientemente del lenguaje seleccionado. Es por esto que los errores lógicos que se cometan en la elaboración de éste pasarán al programa y en consecuencia al computador. Por tanto, es importante tener mucho cuidado al momento de elaborarlo.

Al recordar la definición dada por Joyanes, L. (2003: 41), éste indica que los pasos a seguir deben ser precisos, definidos y finitos, éstas son precisamente las tres características básicas que todo algoritmo debe seguir:

- *PRECISO*: un algoritmo debe indicar el orden de realización de cada paso.
- *DEFINIDO*: si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- *FINITO*: si se sigue un algoritmo, se debe terminar en algún momento.

Según Joyanes, L. (2003: 42) estas tres características conducen a los tres elementos fundamentales presentes en todo algoritmo:

- *ENTRADA*: se refiere al o los datos que se necesitan para comenzar a trabajar, en la mayoría de los casos son los datos que el usuario deberá ingresar por teclado, a pesar de que hay ocasiones en las cuales los datos son predeterminados, es decir, son conocidos previamente.

- **PROCESO:** especifica los cálculos, las condiciones, las restricciones y/o instrucciones que se deben seguir para relacionar los datos de entrada y obtener la solución del problema.
- **SALIDA:** es lo que se quiere obtener, se refiere al resultado de ejecutar el proceso.

Los algoritmos son fundamentales para la posterior elaboración de un programa de computadora, asimismo, el analizar y establecer los pasos para la solución de un problema a través de la computadora, no es una tarea fácil, ni se debe tomar a la ligera. Por esta razón, los algoritmos juegan un papel fundamental para la elaboración del producto final, los errores cometidos en estos, se verán reflejados en el programa, de allí, la importancia de saber establecer los tres elementos básicos.

Es común ver como algunas personas, bien por ligereza o bien por descuido, omiten uno o varios pasos y al darse cuenta de su error, ya han perdido tiempo valioso para la implementación del programa. Establecer de manera clara y precisa cada uno de estos elementos, ayudará al buen desempeño del algoritmo y por ende del programa.

2. PROGRAMA

Es bien sabido que los computadores son máquinas y como tales, carecen de inteligencia, capacidad de reflexión y por tanto, no hablan nuestro idioma. Estas máquinas necesitan de un lenguaje determinado, diseñado por el hombre para ellas, el cual requiere continuamente ser descifrado. Estos lenguajes permiten, en primer lugar, escribir las instrucciones necesarias a realizar para resolver el problema de un modo parecido a como se escribiría convencionalmente, y en segundo lugar, se encargan de traducir el algoritmo al lenguaje de máquina con lo que se le confiere al programa la capacidad de ejecutarse en el computador.

En la actualidad, a estos lenguajes se les denominan lenguajes de programación, cada uno de ellos con su propia gramática, su terminología especial y una sintaxis particular. Pero qué es un programa, Corona, M. y Ancona, M. (2011: 3) presentan tres definiciones al respecto:

- Es un algoritmo desarrollado en un determinado lenguaje de programación, para ser utilizado por la computadora, es decir, es una serie de pasos e instrucciones ordenadas y finitas que pueden ser procesadas por una computadora, a fin de permitir resolver una tarea o problema específico.
- Secuencia de instrucciones mediante las cuales se ejecutan diferentes acciones de acuerdo con los datos que se desee procesar en la computadora.



- Expresión de un algoritmo en un lenguaje preciso que puede llegar a entender una computadora.

Para simplificar podríamos decir, que un programa se refiere a un conjunto de instrucciones u órdenes basadas en un lenguaje de programación que una computadora interpreta para resolver un problema o una función específica, en otras palabras, es la secuencia de instrucciones dadas al computador para su funcionamiento.

Es importante resaltar, según lo planteado por Joyanes, L. (2003: 21) que el propósito de un lenguaje de programación, es permitir al hombre comunicarse con la máquina, es decir, habilita a éste para escribir en un lenguaje que sea más apropiado a las características humanas y se pueda traducir al lenguaje de máquina.

Joyanes, L. (2003: 22) plantea, que anteriormente los computadores eran programados en lenguaje máquina pero al ser un lenguaje bastante complejo ocasionaban problemas y se tenían enormes posibilidades de cometer errores, los cuales eran difíciles y complejos de detectar.

En opinión del autor citado, los lenguajes de máquina o lenguajes de bajo nivel, son aquellos que están escritos en lenguajes directamente inteligibles por la máquina, ya que sus instrucciones son cadenas binarias (cadena compuesta por 0 y 1) que especifica una operación. La complejidad de estos lenguajes de máquina orientó el diseño de lenguajes de programación dirigidos a facilitar el trabajo a los programadores, también llamados lenguajes de alto nivel. Estos están diseñados para que las personas escriban y entiendan los programas de un modo más fácil.

Tanto para Joyanes, L. (2003: 22) como para Corona, M. y Ancona, M. (2011: 2), los principales tipos de lenguajes utilizados en la actualidad son:

- **LENGUAJE DE MÁQUINA:** es el único que entiende directamente el computador, debido a que está escrito en lenguaje directamente perceptible por la máquina, es decir, utiliza el alfabeto binario el cual consta solo de dos dígitos (0 y 1), estos reciben el nombre de bits. La ventaja de este lenguaje es la posibilidad de carga en memoria sin necesidad de traducción posterior, lo cual supone una velocidad de ejecución superior a cualquier otro lenguaje de programación. La desventaja radica en la dificultad y lentitud en la codificación, poca fiabilidad, gran dificultad para verificar y poner a punto los programas. En la actualidad, las desventajas superan a las ventajas, por lo tanto, no es recomendable la utilización de los lenguajes de máquina.

- **LENGUAJE DE BAJO NIVEL (ENSAMBLADOR):** fueron diseñados para simplificar el proceso de la programación en lenguaje de máquina, sin embargo, también depende del computador en particular. En este lenguaje, las instrucciones son escritas en códigos alfabéticos conocidos como mnemónicos, los cuales son abreviaturas en inglés (ejemplo: ADD para la suma, DIV para dividir, entre otros). La computadora sigue utilizando el lenguaje máquina para procesar los datos, pero los programas ensambladores traducen antes los símbolos de código de operación especificados a sus equivalentes en el lenguaje máquina.

Estos lenguajes permiten crear programas muy rápidos, pero a menudo, difíciles de aprender. Es importante resaltar el hecho de que los programas escritos de esta manera son altamente concretos de cada procesador. Si se lleva el programa a otra máquina se debe reescribir el programa desde el principio.

- **LENGUAJE DE ALTO NIVEL:** estos lenguajes son los más utilizados por los programadores. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensamblador. Un programa escrito en lenguaje de alto nivel es independiente de la máquina, por lo que estos programas son portables, es decir, pueden ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras. Son lenguajes de programación donde las instrucciones son similares al lenguaje humano, en este sentido, como el ordenador no es capaz de reconocer estas órdenes, es necesario el uso de un intérprete que traduzca el lenguaje de alto nivel a un lenguaje de bajo nivel.

En referencia a los planteamientos expuestos, y realizando una comparación entre los tres lenguajes, es importante resaltar, que al respecto de los lenguajes de máquina es muy difícil que en la actualidad se trabaje directamente con ellos, la codificación inicial muchas veces requiere meses de trabajo, por lo que es muy costosa y tiende a generar muchos errores, revisar las instrucciones para localizar errores es casi tan tedioso como escribirlas por primera vez.

Además, si es necesario modificar posteriormente un programa, el trabajo puede llevarse meses de revisión. Por otro lado, trabajar con los lenguajes de bajo nivel o ensambladores, requieren menos atención a los detalles, los errores que se puedan cometer son menores y más fáciles de localizar.

Asimismo, los programas en lenguaje de bajo nivel son más fáciles de modificar que los de máquina. Sin embargo, la codificación sigue siendo



lenta y la misma está diseñada para una marca y modelo específico de computador, por tanto, al cambiar de equipo lo más probable es que se deba cambiar también el código.

Finalmente, los lenguajes de alto nivel presentan un código fácil de manejar, entender, revisar y depurar, por tanto, es mucho más fácil localizar y corregir los errores, estos lenguajes se pueden utilizar en diferentes marcas y modelos de computadores con pocas o sin modificación alguna. De hecho, son más fáciles de aprender, se pueden escribir más rápidamente, permiten tener una mejor documentación de su codificación, son más fáciles de mantener, entre otras. Estos aspectos llevan a disminuir tanto el costo de producción como el tiempo de trabajo.

3. DATO

El objetivo principal de todo computador, es la manipulación de la información. La información es el conjunto de datos ordenados de forma lógica los cuales brindan un conocimiento, en este sentido, Corona, M. y Ancona, M. (2011: 8) plantean, que los diferentes objetos de información con los que un algoritmo o programa trabaja, se conocen colectivamente como datos.

Sobre el asunto, Joyanes, L. (2003: 90) señala que: *"los datos son los objetos sobre los que opera una computadora"*, esta definición es abalada por Corona M. y Ancona, M. (2011: 8) quienes consideran que los datos se refieren a: *"los diferentes objetos de información con los que un algoritmo o programa trabaja"*.

Por otra parte, López, G. y otros (2009: 26) afirman que son: *"la materia prima con los que se alimenta un programa para generar los resultados, son entidades concretas y mesurables en términos del espacio que ocupan y del valor que representan, como lo pueden ser números, letras o cadenas de caracteres"*.

Considerando las definiciones anteriores, podemos establecer que el dato se refiere a la mínima unidad de información significativa para alguien, es la materia prima para la obtención de información. Sin ellos, los algoritmos y/o programas no podrían funcionar. De hecho, las computadoras actuales pueden trabajar con varios tipos de datos, siendo los algoritmos y/o los programas en sí, los encargados de manipular estos datos. La acción de las instrucciones realizadas por el computador se manifiesta en cambios de los valores en los diferentes datos manipulados, donde los datos de entrada se transforman en datos de procesos y posteriormente en datos de salida.

Al respecto, señala Joyanes, L. (2003: 90) que en el proceso de análisis de un problema, determinar y estructurar los datos a utilizar, es tan importante como el diseño del algoritmo en sí y para el desarrollo del programa. Por esta razón, es importante saber diferenciar los tipos de datos con los que se pueden trabajar, en este sentido, el autor consultado al igual que Corona, M. y Ancona, M. (2011: 9) concuerdan en la siguiente tipología:

- **DATOS NUMÉRICOS:** es el conjunto de los valores numéricos. Se pueden representar de dos formas:
 - *Números Enteros:* corresponde a números completos, no tienen componente decimal o fraccionario y pueden ser positivos o negativos.
 - *Números Reales:* tienen siempre un punto decimal, las fracciones se almacenan en la computadora como números decimales ya que no existe otra forma de almacenar los numeradores y los denominadores de forma separada. Al contrario de los enteros, que suelen tomar valores en un rango determinado, los números reales pueden tomar teóricamente, cualquier valor de la recta numérica real y ser positivos y negativos.
- **DATOS LÓGICOS O BOOLEANOS:** este tipo de dato solo puede tomar dos posibles valores: verdadero (true) o falso (false). Se utiliza para representar las alternativas (sí/no) a determinadas condiciones. En muchos lenguajes se pueden representar con 0 (falso) y 1 (verdadero).
- **DATOS CARÁCTER:** es el conjunto finito y ordenado de caracteres que la computadora reconoce. Los caracteres que reconocen las diferentes computadoras no son estándar, sin embargo, la mayoría reconoce los siguientes caracteres:
 - Caracteres alfabéticos (A, B, C,..., Z) (a, b, c,..., z)
 - Caracteres numéricos (0, 1, 2, 3,..., 9)
 - Caracteres especiales (+, -, *, /, =, <, >, \$, %,)

Dentro de esta tipología, se encuentran las cadenas de carácter (string), lo cual no es más que una sucesión de caracteres que se encuentran delimitados por una comilla simple o comillas dobles, según el tipo de lenguaje de programación. Basados en esta clasificación, en este libro se establecerá la tipología de datos pertenecientes a dos grandes grupos, los datos numéricos y los no numéricos, los mismos se pueden observar en el siguiente gráfico:

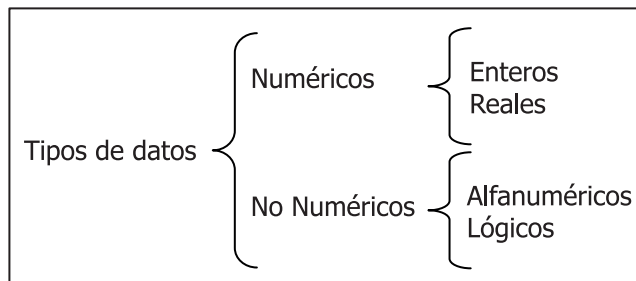


Gráfico 1: Tipos de Datos.

3.1. Constante

Para la matemática, una constante *es una cantidad que tiene un valor fijo* en un determinado cálculo, proceso o ecuación. Esto quiere decir que la constante es un valor permanente que no puede modificarse dentro de cierto contexto, esto lo confirma Corona, M. y Ancona, M. (2011: 11) cuando plantean que se refiere a un dato que permanece sin cambio durante el desarrollo del algoritmo o durante la ejecución del programa.

En este sentido, las constantes en informática y computación, son todos aquellos datos (valores) que no cambian en el transcurso de un algoritmo y son introducidas en el momento de utilizarse o desde el principio del algoritmo. Existen constantes por naturaleza, por ejemplo, el valor de PI (3.1416) o el valor de la gravedad (9.8).

3.2. Variable

Una variable representa aquello que varía o que está sujeto a algún tipo de cambio. Se trata de algo que se caracteriza por ser *inestable, inconstante y mudable*. Por tanto, una variable es un *símbolo*, el cual permite identificar a un elemento no especificado dentro de un determinado grupo. En términos de informática y computación, son todos aquellos valores que pueden o no cambiar en el transcurso de un algoritmo, lo cual es confirmado por Joyanes, L. (2003: 93) cuando señala, que una variable es un objeto o partida de datos cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. La mayoría de los datos son variables por naturaleza, por ejemplo, masa, peso, edad, volumen, entre otros.

3.3. Identificador

Los identificadores permiten representar o darle nombre a los elementos que se utilizan tanto en los algoritmos como en los programas, es decir,

ellos permiten, como su nombre lo indica, identificar una constante, una variable, el nombre de un programa, métodos, funciones, estructuras de datos, entre otros. Así lo indica Corona, M. y Ancona, M. (2011: 8) cuando dicen que es una secuencia de caracteres alfabéticos, numéricos y el guión bajo, con los cuales se le puede dar nombre a variables, constantes, tipo de datos, nombre de funciones o procedimientos, entre otros.

De acuerdo con Joyanes, L. (2003: 792) normalmente los Identificadores deben cumplir las siguientes características:

- Siempre deben comenzar con una letra, bien sea en mayúscula o minúscula.
- El único carácter especial que acepta es el underscore (guión bajo).
- No puede tener espacios en blanco. Si son dos o más palabras, se coloca todo pegado, identificando el comienzo de palabras con mayúsculas o se separa utilizando un underscore.
- Su tamaño depende del lenguaje de programación, sin embargo, por lo general va desde uno (1) a doscientos cincuenta y seis (256).

Ejemplos de Identificadores *bien escritos*:

a	A
b	B121
c	Nombre
d	a1A2
e	PromNota
f	Prom_Nota

Ejemplos de Identificadores *mal escritos*:

a	2b
b	1nombre
c	Promedio de Notas

4. EXPRESIONES Y OPERADORES

Para Joyanes, L. (2003: 94) las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales utilizadas para plantear procesos aritméticos, relacionales y lógicos, por ejemplo:

$P = m * (n + 5) + \text{sqrt}(9)$	→	Expresión Aritmética
$R > K$	→	Expresión Relacional
$(R \geq 10) \text{ AND } (K < 5)$	→	Expresión Lógica



Por otra parte, Corona, M. y Ancona, M. (2011: 19) consideran que las expresiones son el resultado de unir operandos mediante operadores. En esta definición es importante aclarar que los operandos son las variables y las constantes, en cuanto a los operadores, estos son todos aquellos símbolos o signos utilizados para relacionar las variables y/o las constantes.

De esta manera, tomando en consideración lo antes expuesto, para Joyanes, L. (2003: 95) y López, G. y otros (2009: 32) los tipos de expresiones y operadores que se presentan son:

- **ARITMÉTICAS:** se refiere a un conjunto de variables y/o constantes unidas o relacionadas por operadores aritméticos (Ver tabla 1),

Tabla 1.
Operadores Aritméticos

DESCRIPCIÓN	OPERADOR	
	DFD	C++
Exponenciación	^	pow()
Multipliación	*	*
División	/	/
Resta	-	-
Suma	+	+
División Modular	mod()	%
Igualdad	=	=
Incremento	No existe	++
Decremento	No existe	--

- **RELACIONALES:** es un conjunto de variables y/o constantes unidas o relacionadas por operadores relacionales (Ver tabla 2),

Tabla 2.
Operadores Relacionales

DESCRIPCIÓN	OPERADOR	
	DFD	C++
Mayor que	>	>
Mayor o igual que	>=	>=
Menor que	<	<
Menor o igual que	<=	<=
Igual que	=	==
Diferente que	!=	!=

- **LÓGICAS:** es un conjunto de variables y/o constantes unidas o relacionadas por operadores lógicos, cuyos valores pueden ser verdadero o falso (Ver tabla 3),

Tabla 3.
Operadores Lógicos

DESCRIPCIÓN	OPERADOR	
	DFD	C++
Y	And	&&
O	Or	
NO	Not	!

5. CONTADOR

Los contadores son variables que se amplían o se reducen durante la ejecución de un proceso en forma constante. López, G. y otros (2009: 31) consideran que: *"un contador es una variable entera que se incrementa, cuando se ejecuta, en una unidad o en una cantidad constante"*; por otra parte, Corona, M. y Ancona, M. (2011: 80) lo definen como una variable cuyo valor se incrementa o decrementa en una cantidad constante cada vez que se produce un determinado suceso o acción en cada repetición. Finalmente, Fabelo, R. y Medina, M. (2004: 17) puntualizan que el contador es una variable numérica que puede aumentar o disminuir por iteración un valor constante, indicado al momento de su definición.

Todo contador básicamente tiene dos apariciones dentro de un proceso, la primera aparición es la inicialización del contador, en este momento se le asignará al contador el valor inicial y se debe realizar fuera del ciclo donde el contador se ejecutará. Si el enunciado no indica en que valor debe comenzar, entonces se deberá inicializar en un valor que no afecte el proceso, por ejemplo:

Contador = 1

La segunda aparición de un contador es la ejecución, ésta siempre se hace dentro de un ciclo y su sintaxis básica viene dada de la siguiente manera:

Contador = Contador + 1
o
Contador = Contador + Constante



Como se puede observar, el contador en su ejecución depende de sí mismo y de un dato constante. Los mismos son utilizados bien para llevar el conteo del número de repeticiones en un ciclo o para contar el número de apariciones de un procedimiento o suceso específico dentro de un ciclo.

6. ACUMULADOR

Los acumuladores son variables que se incrementan o se decrementan durante la ejecución de un proceso de forma diversa. Para López, G. y otros (2009: 31): “un acumulador es una variable que se incrementa en una cantidad variable”. Básicamente, es una variable que durante la ejecución de un programa cambia de valor y se utiliza para sumar o totalizar un grupo de datos, asimismo, Corona, M. y Ancona, M. (2011: 80) precisan que es una variable que acumula sobre sí misma un conjunto de valores, para de esta manera tener la totalización de los datos.

Al igual que los contadores, todo acumulador tiene básicamente dos apariciones dentro de un proceso, la primera aparición es la inicialización y es igual que en el contador y la segunda es la ejecución, la cual siempre se realiza dentro de un ciclo y su sintaxis básica viene dada de la siguiente manera:

$$\text{Acumulador} = \text{Acumulador} + \text{Variable}$$

Al igual que el contador en su ejecución, el acumulador depende de sí mismo pero ahora de un dato variable y lo que busca es almacenar cantidades resultantes de operaciones sucesivas.

7. VALOR CENTINELA

Es un identificador que durante la ejecución de un algoritmo y/o programa, o, a lo largo del mismo, indica el fin de un proceso en particular. Normalmente, permite controlar hasta cuando se deben ingresar datos, así lo plantean Corona, M. y Ancona, M. (2011: 80) cuando especifican que éste se refiere a una variable que inicia con un valor, luego dentro de un bucle este valor cambia, haciendo falsa la condición del ciclo y por lo tanto, indicando el fin del mismo.

Este valor se utiliza en el desarrollo o implementación de un programa, cuando en el enunciado del problema en cuestión, no aclare de alguna forma, la cantidad de veces a procesar los datos de entrada. En algunos sistemas ya desarrollados, pudiera utilizarse como: Pulse la tecla Esc (Escape) para SALIR.

Es importante destacar, que el proceso controlado por un valor centinela con frecuencia es llamado repetición indefinida, debido a que se desconoce el número de repeticiones que se van a realizar. En este sentido, en un ciclo controlado por un valor centinela, se le deberá indicar al usuario frecuentemente cual es la señal de salida.

8. FORMAS DE REPRESENTACIÓN ALGORÍTMICAS

El Diccionario de la Real Academia Española (2001), citado por López, G. y otros (2009: 4) indica que un algoritmo es un “conjunto ordenado y finito de operaciones que permite hallar la solución de un programa”, en otras palabras, es una manera de resolver un problema de forma clara y precisa mediante una serie de pasos lógicos. Según Joyanes, L. (2003: 43) existen dos maneras básicas de representarlos:

- **DIAGRAMA DE FLUJO:** es la representación gráfica de un algoritmo. Es una representación algorítmica que utiliza para su desarrollo símbolos especiales, que indican la acción respectiva a ejecutar para la resolución de un problema o una situación específica. Los símbolos son:

Tabla 4.
Símbolos Utilizados en los Diagramas de Flujo

SÍMBOLO	DESCRIPCIÓN
	Terminal (Inicio o Fin de un procedimiento)
	Entrada de datos por teclado
	Proceso
	Salida por pantalla
	Conector entre páginas
	Decisión Simple
	Ciclo Para (Lo que se quiere repetir va entre los dos símbolos)

**Continuación...**

	Ciclo Mientras (Lo que se quiere repetir va entre los dos símbolos)
	Indicadores de dirección o líneas de flujo

PSEUDOCÓDIGO: código simulado. Código no refinado gramaticalmente. Representa una herramienta de programación en la que las instrucciones se describen en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura del programa. Es una representación algorítmica que utiliza para su desarrollo, un lenguaje coloquial (POPULAR) y además combina instrucciones o códigos de los lenguajes de programación para la resolución de un problema o una situación específica. Para este texto las sentencias a utilizar son:

Tabla 5.
Sentencias Utilizadas en los Pseudocódigos

PSEUDOCÓDIGO	DESCRIPCIÓN
Inicio / Fin	Terminal (Inicio o Fin de un procedimiento)
Leer()	Entrada de datos por teclado
Se realiza directamente. Ejemplo: $S = a + b$	Proceso
Escribir()	Salida por pantalla
Si(CONDICION) Sentencias Sino Sentencias FinSi	Decisión Simple
Para Inicio hasta Fin, Hacer Sentencias FinPara	Ciclo Para
Mientras(CONDICION) Sentencias FinMientras	Ciclo Mientras

Fuente: Joyanes, L. (2003).

En temas posteriores, se indicará la manera de utilizar tanto la simbología de los diagramas de flujo como la de los pseudocódigos.

8.1. Pasos a seguir para elaborar un algoritmo

Como se estableció anteriormente, los algoritmos permiten buscar la solución de un problema de manera estructural y fácil de entender, ellos son la fase previa para la elaboración de un programa, en este sentido, es importante seguir los siguientes pasos:

1. Leer el enunciado del problema cuantas veces sea necesario, hasta entenderlo completamente.
2. Determinar claramente con que datos de entrada se cuentan para la solución del problema.
3. Aclarar y determinar la información o resultados que se soliciten.
4. Definir qué cálculos y/o comparaciones se necesitan para llegar al resultado final.
5. Tener en cuenta toda clase de condiciones y restricciones para la solución del problema.

Estos pasos son esenciales para la elaboración de un algoritmo, el seguirlos o no, podría significar la diferencia entre el éxito y el fracaso en el diseño. El primer paso indica la importancia de conocer lo que se quiere hacer, el segundo paso es para identificar el o los datos de entrada, luego en el tercer paso se identifica el resultado al cual se debería llegar. Finalmente, los dos últimos pasos, permitirán establecer el proceso, es decir, la manera como los datos de entrada se relacionarán para obtener la salida.

Ejemplo: elaborar un algoritmo que obtenga el salario neto de cada uno de los 20 trabajadores de la Empresa S.A. Se debe tomar en cuenta que se tiene un salario por hora, y la retención es del 10%.

Luego de leer detenidamente el enunciado, se debe hacer el análisis previo del mismo, de la siguiente manera:

- a. Determinar los datos de entrada: se puede observar como el enunciado habla del Salario Hora (SH) y Número Hora (NH).
- b. Determinar la salida: se refiere a lo que se quiere hallar, en este caso es el Salario Neto (SN).
- c. Determinar el proceso a realizar para poder obtener la salida de la siguiente manera:



1. Se pide el salario hora y el número de horas.
 2. Se calcula el salario base, el cual es igual al salario hora por el número de horas.
 3. Se calcula la retención (R) que es el 10% del salario base.
 4. Se calcula el salario neto que es el salario base menos la retención.
 5. Se muestra el salario base, la retención y el salario neto.
- Se repite 20 veces utilizando el ciclo Para

Luego de realizar el análisis previo se debe realizar el diagrama de flujo (Figura 1) y/o el pseudocódigo de la siguiente manera:

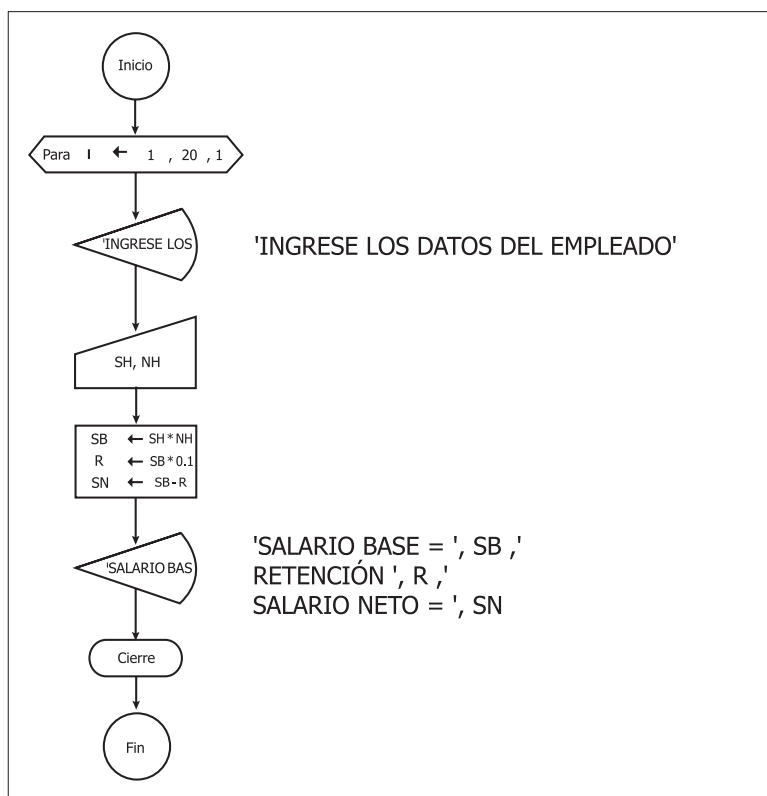


Figura 1. Diagrama de Flujo.

- Pseudocódigo:

"Inicio"

Real: SH,VH,SN,R

Entera: I

Para I = 1 hasta 20, Hacer

 Escribir ("INGRESE LOS DATOS DEL EMPLEADO:")

 Leer (SH, NH)

$SB = SH * NH$

$R = SB * 0.1$

$SN = SB - R$

 Escribir ("SALARIO BASE: ", SB)

 Escribir ("LA RETENCIÓN ES: ", R)

 Escribir ("EL SALARIO NETO ES: ", SN)

FinPara

Fin

9. LENGUAJE C++: DEFINICIONES BÁSICAS

El lenguaje de programación C++ es de propósito general, ya que todo puede programarse en él, desde sistemas operativos, compiladores, aplicaciones de bases de datos, procesadores de texto, juegos y cualquier tipo de aplicaciones.

Ahora bien, Osorio, A. (2006: 10) nos señala que la historia de este lenguaje comienza a principio de los años 70, cuando el programador Dennis Ritchie, quien trabajaba en los laboratorios AT&T Bell, utilizando el BCPL inventado por Martin Richards y el lenguaje B de Ken Thompson, decide diseñar un lenguaje de programación para manejar el hardware de la misma manera que lo hace el lenguaje ensamblador, pero con algo de programación, estructura como los lenguajes de alto nivel.

Así, de acuerdo con el autor citado, nace el lenguaje C en 1972, el mismo fue creado para correr inicialmente con el sistema operativo UNIX, sin embargo, no es hasta que AT&T cede el sistema operativo a varias universidades cuando comienza el auge del Lenguaje C, convirtiéndose en uno de los favoritos para crear aplicaciones.

En tanto que, Bustamante, P. y otros, (2004: 1) especifican, que el Instituto Americano de Normalización (ANSI) en el año 1983, estandarizó el lenguaje C, tarea que duró seis (6) años en completarse, por otra parte, la Organización Internacional de Normalización (ISO) define el C Estándar en 1989. Es a partir de este momento cuando el lenguaje C comienza a evolucionar más rápidamente.



Sin embargo, para 1980 Bjarnes Stroutstrup, quien también trabajaba en los laboratorios AT&T Bell comienza a desarrollar el C++ como una extensión del Lenguaje C que fue denominada **C witch clases**, buscando un lenguaje con las opciones de la programación orientada a objeto.

Bustamante, P. y otros (2004: 1) comentan que en la actualidad el C++ es versátil, potente y general. Su éxito entre los programadores lo ha llevado a ser uno de las primeras herramientas de desarrollo de aplicación. El C++ mantiene las ventajas de C en cuanto a la riqueza de operadores y expresiones, flexibilidad, concisión y eficiencia, eliminando algunas de las limitaciones del C original. Una de las propiedades de C++ es la reutilización del código en forma de librerías de usuario, puesto que genera programas más compactos y rápidos. El código es transportable, es decir, podrá ejecutarse en cualquier máquina y bajo cualquier sistema operativo.

Las teorías expuestas por Bustamante, P. y otros (2004) y Osorio A. (2006), dejan claro que el C++ es una mejoría de C, ya que proporciona capacidades de P.O.O. (Programación Orientada a Objeto) que promete mucho para incrementar la productividad, calidad y reutilización del software. En C++, la unidad de programación es la clase a partir de la cual, los objetos son producidos y manejados en función de la programación orientada al objeto. Las bibliotecas estándares de C++ permiten manipular las capacidades de entrada y salida.

Se pueden especificar entradas y salidas de tipos definidos por el usuario, así como, de tipo estándar. Esta extensibilidad es una de las características más valiosas de este lenguaje de programación. C++ permite un tratamiento común de entradas y salidas de tipos definidos por usuario. Este tipo de estado común, facilita el desarrollo de software en general y de la reutilización de software en particular.

10. ESTRUCTURA DE UN PROGRAMA EN C++

El lenguaje de programación C++ es un lenguaje compilado, el cual contempla tres (3) aspectos importantes durante el desarrollo de programas como son: el código fuente, el código objeto y finalmente el programa ejecutable, tal como puede ser visualizado en la Figura 2 y su descripción específica en la Tabla 6.

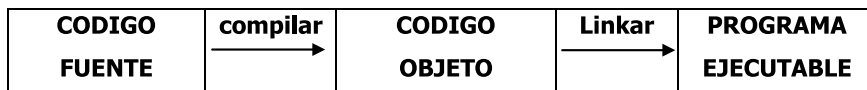


Figura 2. Desarrollo del programa

Tabla 6.
Aspectos del Compilador de C++

ESTRUCTURA	DESCRIPCIÓN
Código Fuente	Programa o conjunto de instrucciones que se editan o se desarrollan en el editor, se guarda con la extensión CPP .
Código Objeto	Programa fuente pero traducido a lenguaje máquina (sucesión de ceros y unos), se almacena con la extensión OBJ .
Programa Ejecutable	Es el programa objeto más las "librerías del C", se graba con la extensión EXE . Es el ejecutable del programa que se ha editado.

Es importante señalar, que todos los programas no pueden ser desarrollados de la misma forma, que no existe una estructura predefinida como esquema para resolver o dar solución al desarrollo de programas, sin embargo, durante la programación del código fuente se recomienda tener en cuenta los siguientes aspectos:

- Documentar lo que se desea programar, mediante el uso de comentarios.
- Declarar o iniciar los archivos de cabeceras o librerías.
- Declarar variables globales y constantes.
- Crear el programa principal.
- Iniciar el programa principal.
- Declarar constantes locales.
- Declarar variables locales.
- Cuerpo del programa o estructuras de control.
- Fin del programa principal.
- Declaración de funciones.
- Inicio de funciones.
- Cuerpo de las funciones.
- Fin de las funciones.

El seguir o no estas recomendaciones podría significar la diferencia entre el éxito o no en la elaboración del programa.

11. ELEMENTOS BÁSICOS DE UN PROGRAMA EN C++

Durante la edición de un programa es necesario tener en cuenta que existen elementos básicos, que son necesarios durante el desarrollo del



código, entre los cuales son destacables: la inicialización de librerías o archivos de cabeceras, el cuerpo principal o clase principal, declaración de variables y constantes, módulos, comentarios, y el conjunto de instrucciones relacionadas a las funciones de flujo de control y ciclos de decisión, entre otros. A continuación, se explicará cada uno de estos elementos:

Cuerpo Principal o Clase Principal: aparece una sola vez en todo el contenido del programa, está representado por la palabra `main` y es el punto donde inicia la ejecución del programa, seguidamente a la palabra `main` la acompañan un grupo de paréntesis indicando que es una función.

```
main ()  
{  
}
```

Archivos de Cabeceras: durante el desarrollo de programas será necesario inicializar librerías o archivos de cabeceras para hacer uso de palabras reservadas, métodos y funciones específicas del C++. Esta inicialización de archivos de cabeceras es conocida en la teoría de compilador como inclusión de archivos, el cual trabaja de forma muy similar a la sustitución de macros, donde el programa busca el archivo e incluye su contenido antes del proceso de compilación del programa. Es importante señalar que los comandos o sintaxis más utilizadas para la inicialización de librerías o archivos de cabecera son:

```
#include <nombre de la librería o archivo>  
#include "nombre de la librería o archivo"  
#define <nombre de la librería o archivo>  
#define "nombre de la librería o archivo"
```

Existe una variación en cuanto al uso de comillas `"..."` y la segunda forma con los símbolos `<...>`, ya que la primera forma busca el archivo en el directorio actual y posteriormente en el directorio estándar de librerías, con respecto a la segunda estructura busca directamente en el directorio estándar de librerías.

```
#include <stdio.h>  
main()  
{  
  
}
```

Es importante señalar que en C++ las librerías o bibliotecas más utilizadas son:

Tabla 7.
Bibliotecas estándar del C++

ENTRADA/SALIDA	
<cstdio>	E/S de la biblioteca de C
<cstdlib>	Funciones de clasificación de caracteres
<wchar>	E/S de caracteres extendidos
<fstream>	Flujos para trabajo con ficheros en disco
<iomanip>	Manipuladores
<ios>	Tipos y funciones básicos de E/S
<iosfwd>	Declaraciones adelantadas de utilidades de E/S
<iostream>	Objetos y operaciones sobre flujos estándar de E/S
<istream>	Objetos y operaciones sobre flujos de entrada
<ostream>	Objetos y operaciones sobre flujos de salida
<sstream>	Flujos para trabajar con cadenas de caracteres
<streambuf>	Búferes de flujos
CADENAS	
<cctype>	Examinar y convertir caracteres
<cstdlib>	Funciones de cadena estilo C
<cstring>	Funciones de cadena estilo C
<wchar>	Funciones de cadena de caracteres extendidos estilo C
cwctype>	Clasificación de caracteres extendidos
<string>	Clases para manipular cadenas de caracteres
CONTENEDORES	
<bitset>	Matriz de bits
<deque>	Cola de dos extremos de elementos de tipo T
<list>	Lista doblemente enlazada de elementos de tipo T
<map>	Matriz asociativa de elementos de tipo T
<queue>	Cola de elementos de tipo T
<set>	Conjunto de elementos de tipo T (contenedor asociativo)
<stack>	Pila de elementos de tipo T
<vector>	Matriz de elementos de tipo T
ITERADORES	
<iterator>	Soporte para iteradores
ALGORITMOS	
<algorithm>	Algoritmos generales (buscar, ordenar, contar, etc.)
<cstdlib>	bsearch y qsort
NÚMEROS	
<cmath>	Funciones matemáticas
<complex>	Operaciones con números complejos
<cstdlib>	Números aleatorios estilo C
<numeric>	Algoritmos numéricos generalizados
<valarray>	Operaciones con matrices numéricas
DIAGNÓSTICOS	
<cassert>	Macro ASSERT
<cerrno>	Tratamiento de errores estilo C
<exception>	Clase base para todas las excepciones
<stdexcept>	Clases estándar utilizadas para manipular excepciones
UTILIDADES GENERALES	
<ctime>	Fecha y hora estilo C
<functional>	Objetos función
<memory>	Funciones para manipular bloques de memoria
<utility>	Manipular pares de objetos
LOCALIZACIÓN	
<locale>	Control estilo C de las diferencias culturales
<locale>	Control de las diferencias culturales
SOPORTE DEL LENGUAJE	
<cmath>	Límites numéricos en coma flotante de estilo C
<climits>	Límites numéricos estilo C
<csetjmp>	Salvar y restaurar el estado de la pila
<csignal>	Establecimiento de manejadores para condiciones
Excepcionales (también conocidos como señales)	
<csdarg>	Lista de parámetros de función de longitud variable
<csddef>	Soporte de la biblioteca al lenguaje C
<csdlib>	Definición de funciones, variables y tipos comunes
<ctime>	Manipulación de la hora y fecha
<exception>	Tratamiento de excepciones
<limits>	Límites numéricos
<new>	Gestión de memoria dinámica
<typeinfo>	Identificadores de tipos durante la ejecución

Fuente: Osorio, A. (2006).



Función printf (): esta función permite mostrar o visualizar un texto en la pantalla o monitor, se encuentra definida en el archivo de cabecera `cstdio`, es decir, para hacer uso de ella durante el desarrollo de un programa es necesario inicializar la librería antes citada. La función `printf ()` trabaja bajo la estructura de colocar la cadena de texto que se desea mostrar en el monitor entre comillas y dentro del paréntesis que sigue a la palabra `printf`, seguido del punto y coma (;) ya que la mayor parte de las líneas de códigos en C++ finalizan con este símbolo, indicando que la línea de comando está completa.

```
#include <cstdio>
using namespace std;
main ()
{
    printf ("Esta es una línea de texto.");
}
```

Es importante conocer que la función `printf ()` utiliza la barra invertida (\) seguida de caracteres que ejecutan una acción específica, para mostrar textos y datos en pantalla se hace uso de un formato específico, éste se debe colocar al final del texto que se desea mostrar, entre los formatos se encuentran:

Tabla 8.
Formato de Salida

Código formato	Significado
\a	Alerta
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
\\	Barra invertida
\'	Comilla simple
\"	Comillas dobles

Fuente: Bustamante, P. y otros (2004).

Un ejemplo de ello es:

```
#include <cstdio>
#include <conio.h>
using namespace std;
main ()
{
    printf ("Esta es una línea de texto.\n");
    printf ("Y esta es otra \t\n");
    printf ("Línea de texto.\r\n");
    printf ("Esta es una tercera línea.\n\n");
    getch();// en espera de pulsar una tecla
```

En el ejercicio anterior se puede observar que se incluye una nueva librería `#include <conio.h>`, ésta permite activar los comandos de entrada y salida de datos, permitiendo hacer uso de la función `gech()`, la cual consiste en hacer una pausa esperando que se presione cualquier tecla.

Por otra parte, para hacer uso de la función `printf()`, combinada con los tipos de datos para mostrar resultados, es preciso señalar que se debe anteponer el carácter porcentaje (%) seguido de la letra correspondiente, para poder visualizar a la variable que se desee mostrar, entre ellos:

Tabla 9.
Formato de Salida

CÓDIGO FORMATO	SIGNIFICADO
%d	Un entero
%i	Un entero
%c	Un carácter
%s	Una cadena
%f	Un real
%Id	Entero largo
%u	Decimal sin signo
%If	Doble posición
%h	Entero corto
%o	Octal
%x	Hexadecimal
%e	Notación científica
%p	Puntero
%%	Imprime porcentaje

Fuente: Joyanes, L. (2000)



Para demostrar la funcionabilidad de los códigos de formato, cuando se requiere visualizar el resultado de variables, es necesario conocer el tipo de datos con el cual se trabaja y ubicarlos en la tabla anterior a su código correspondiente, un ejemplo es:

```
//programa que permite calcular el área de un cuadrado
#include <stdio.h>
#include <conio.h>
using namespace std;
main ()
{
    float lado1=4.1, lado2=8.3, resultado;
    printf ("El valor del lado 1 es: %f \n",lado1);
    printf ("El valor del lado 2 es: %f \n",lado2);
    resultado=lado1*lado2;
    printf ("El valor del área es: %f \n",resultado);
    getch();
}
```

La salida por pantalla de este programa es:

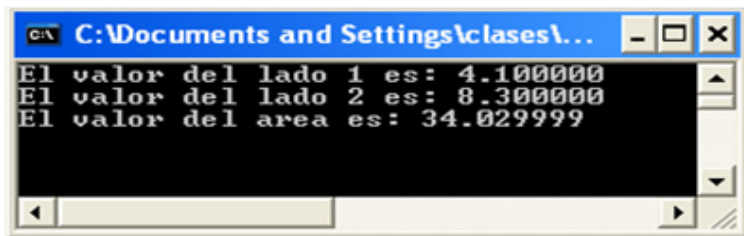


Figura 3. Salida por Pantalla. Área.exe

En el ejercicio anterior se puede observar que los resultados que se visualizan en pantalla muestran seis (6) decimales, por lo general, se requiere dar formato a un (1) o a dos (2) decimales, para lograr esto es necesario anteponer a los códigos de formato el número de decimales seguido de un punto (.), es decir, `%.2f`, en el caso de requerir dos (2) decimales, un ejemplo es:

```
//programa que permite calcular el área de un cuadrado
#include <stdio.h>
#include <conio.h>
using namespace std;
main ()
{
    float lado1=4.1, lado2=8.3, resultado;
```

```
printf ("El valor del lado 1 es: %.2f \n",lado1);
printf ("El valor del lado 2 es: %.2f \n",lado2);
resultado=lado1*lado2;
printf ("El valor del área es: %.2f \n",resultado);
getch();
}
```

La salida por pantalla de este programa es:

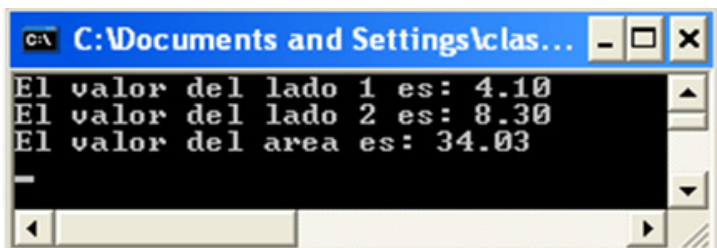


Figura 4. Salida por Pantalla. ÁreaD.exe

Función puts (): imprime una cadena en pantalla seguido de un salto de línea, al igual que la función printf (), aunque a diferencia, esta función solo muestra cadena de caracteres, así como, el contenido de variables del tipo char, veamos un ejemplo de cómo funciona:

```
//programa que permite visualizar un texto y mostrar el contenido de
// una variable del tipo char
#include <stdio>
#include <conio.h>
using namespace std;
main ()
{ char lado1[2]="4", lado2[2]="8"; //la variable char se inicializa
  //con el contenido del valor de la variable y el número
  //de caracteres que contiene la variable
  puts ("PRUEBA DE COMO FUNCIONA PUTS \n");
  puts(lado1);//muestra el contenido de la variable lado1
  puts ("MUESTRA UNA CADENA POR PANTALLA \n");
  puts(lado2);//muestra el contenido de la variable lado2
  getch();
}
```

Función scanf(): permite leer datos por medio del teclado y asignar su contenido a variables de acuerdo al tipo de dato, pertenece a la librería cstdio, en el caso de variables tipo char, no es necesario establecer el



tamaño, el compilador asume el mismo y le agrega un carácter nulo al final indicando el final de la cadena.

-La función `scanf()` maneja dos (2) argumentos, el primero es el signo % acompañado del tipo de dato según el código de formato y encerrado entre comillas, es decir, "%i"; indica que se trata de un dato tipo entero. Seguido por una coma (,) el segundo argumento es el signo & acompañado del nombre de la variable, donde se desea almacenar la lectura desde teclado, un ejemplo sería `scanf("%i",¬a)`,

```
//programa que permite visualizar un texto y permita
//introducir por teclado el contenido de una variable
//de tipo entero y una variable de tipo char
#include <cstdio>
#include <conio.h>
using namespace std;
main ()
{ char NOMBRE[10]; //variable con 10 caracteres
  int nota;
  puts ("INTODUCIR NOTA ");
  scanf("%i",&nota); //leer la variable nota
  puts ("INTODUCIR NOMBRE \n");
  scanf("%s",&NOMBRE); //leer la variable NOMBRE
  puts("LOS DATOS INTRODUCIDOS FUERON \n\n\n");
  printf("LA NOTA ES\n %i",nota);
  puts("\nEL NOMBRE ES ");
  printf(NOMBRE);
  getch();
}
```

Función `getch ()`: lee un carácter por medio del teclado, no es necesario esperar a pulsar la tecla enter, trabaja con la librería `conio.h`, la diferencia entre `getche` y `getch` es que la primera saca por pantalla la tecla que hemos pulsado y la segunda no.

```
#include <cstdio>
#include <conio.h>
using namespace std;
main ()
{
  printf("NOMBRE");
  getch();
}
```

Función `system("cls")`: borra o limpia toda la pantalla de la consola, requiere de la inicialización de la librería `iostream` en la cabecera del programa.

```
//programa que permite visualizar un texto y permita
//introducir por teclado el contenido de una variable
//de tipo entero y una variable de tipo char
#include <cstdio>
#include <conio.h>
#include <iostream> //para poder usar el system
using namespace std;
main ()
{ char NOMBRE[10]; //variable con 10 caracteres
  int nota;
  puts ("INTRODUCIR NOTA ");
  scanf("%i",&nota); //leer la variable nota
  puts ("INTRODUCIR NOMBRE \n");
  scanf("%s",&NOMBRE); //leer la variable NOMBRE
  puts("LOS DATOS INTRODUCIDOS FUERON \n\n\n");
  printf("LA NOTA ES\n %i",nota);
  puts("\nEL NOMBRE ES ");
  printf (NOMBRE);
  getch();
  system("cls"); //limpia la pantalla
  getch();
}
```

Función `system ("PAUSE")`: se encuentra en el archivo de cabecera `iostream` y al igual que el `getch ()` sostiene la pantalla hasta que se presione enter, con la diferencia que ésta muestra el mensaje: "Presione una tecla para continuar..."

Función `cout`: sentencia de salida por consola, se encuentra en el archivo de cabecera `iostream`. Esta sentencia permite mostrar un mensaje en pantalla, para tal fin debe utilizar el operador de salida `<<`, éste operador permite que toda expresión que se encuentre a la derecha salga por el dispositivo que se especifica a la izquierda. Cuando se utiliza esta sentencia, se puede utilizar el `"\n"` para bajar la línea igual como en el `printf` y también se puede utilizar el `"endl"` tal como se muestra en el ejemplo:

```
// Utilizando el getch()
#include <iostream>
#include <conio.h>
using namespace std;
main()
```

```
{  
    cout << "Salida por consola\n";  
    cout << "Otra forma de bajar el cursor" << endl;  
    getch();  
}
```

La salida por consola es:

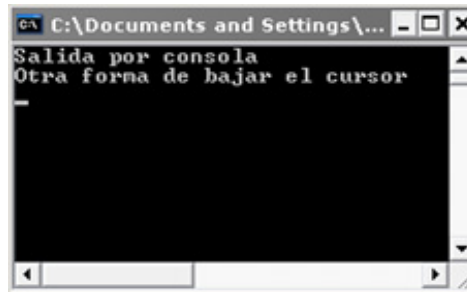


Figura 5. Salida por Pantalla. getch()

```
// Utilizando el system("PAUSE")  
#include <iostream>  
using namespace std;  
main()  
{  
    cout << "Salida por consola\n";  
    cout << "Otra forma de bajar el cursor" << endl;  
    system("PAUSE");  
}
```

La salida por consola es:

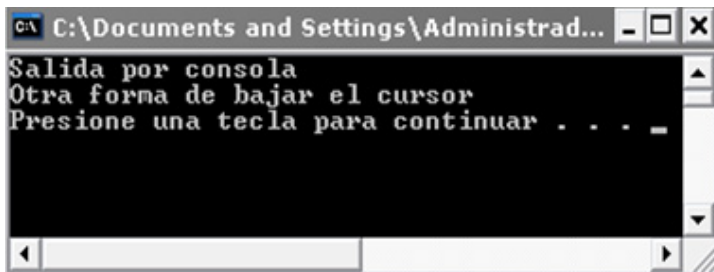


Figura 6. Salida por Pantalla. System("PAUSE");

Función cin: es el sustituto de scanf () en C++, se encuentra en el archivo de cabecera iostream. Representa entrada de dato por consola, específicamente por el teclado, utiliza el operador de entrada >> el cual lee el dispositivo de entrada que está a su izquierda y lo asigna a la variable que está ubicada a la derecha. Con esta función se puede leer una o varias variables a la vez, cuando se leen varios datos al mismo tiempo, se debe tener cuidado el orden en el cual se colocan los datos en la función.

```
#include <iostream>
using namespace std;
main()
{
    int a,b;
    float c;
    cout << "Ingrese el valor de A = ";
    cin >> a;
    cout << "Ingrese los valores de B y C" << endl;
    //forma de leer dos valores a la vez
    cin >> b >> c; //primero se lee un dato int y luego el float
    system("cls");
    cout << "A = " << a << endl;
    cout << "B = " << b << endl;
    cout << "C = " << c << endl;
    system("PAUSE");
}
```

12. COMENTARIOS EN UN PROGRAMA EN C++

Para Joyanes, L. (2000: 38) durante la programación, por lo general, se incorporan comentarios al código fuente de un programa, para hacerlo entendible por el usuario programador, pero carente de significado para el compilador, por lo que se indica al compilador ignorar completamente los comentarios, encerrándolos en caracteres especiales. La combinación de línea diagonal y asterisco - asterisco y diagonal (/* comentario */) y la combinación diagonal – diagonal (//) se utiliza en C++ para delimitar comentarios tanto al inicio como al final del comentario, el cual puede estar distribuido en una línea o en un bloque de varias líneas.

```
# include <cstdio>
/* Éste es un comentario que el compilador ignora. De esta forma, se
pueden colocar varias líneas de comentarios */
int main() // Éste es otro comentario, pero solo toma lo que está a la derecha
{
    printf("Utilizando comentarios "); /* Un comentario está
```



permitido continuar
en otra línea */

```
printf ("en C.");
}
// ....un comentario más...
```

Es importante destacar que existe otra manera de manejar comentarios en C++, la cual consiste en colocar doble diagonal (//), permitiendo hacer comentarios en una línea y posición específica, por ejemplo;

```
# include <stdio.h>
// Este es un comentario que el compilador ignora
int main()
{
    printf("Utilizando comentarios "); //comentario en una línea
    printf ("en C.");
}
```

13. PALABRAS CLAVE DEL C++

En C++, como en cualquier otro lenguaje, existen una serie de palabras claves también llamadas *keywords* que el programador no puede utilizar como identificadores o nombres de variables y/o de funciones. Estas palabras sirven para indicar al computador que realice una tarea determinada.

El C++ es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes de programación. A continuación, se presentan algunas de las palabras clave que maneja el lenguaje y que no pueden ser utilizadas como nombre de variables:

Tabla 9.
Palabras claves (keywords) del C++

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

Fuente: Joyanes, L. (2003).

Al respecto, Joyanes, L. (2000: 565) puntualiza que en C++ se observan dos tipos de datos: *fundamentales y derivados*. Los datos de *tipo fundamental* son los datos básicos que por defecto tiene el lenguaje y que reconoce el compilador. *Los tipos derivados* son aquellos datos que el programador construye a partir de los fundamentales o de otros derivados. Éste es el caso de las estructuras, uniones, clases, arrays y arreglos, entre otros.

En cuanto a los datos de tipo fundamental o básico, Corona, M. y Ancona, M. (2011: 9) establecen que estos se pueden clasificar en enteros y reales, también conocidos como aritméticos, y varían entre 16 y 64 bits de tamaño, donde:

Tabla 10.
Tipo de Datos Fundamentales.

TIPO	TAMAÑO (BYTES)
int (entero)	2
short int (entero corto)	1
long int (entero largo)	4
float (flotante)	4
short float (flotante corto)	2
double (flotante doble precisión)	8
long double (flotante doble precisión largo)	10
char (carácter)	1
void	0

Fuente: Joyanes, L. (2003).

Cada uno de los tipos enteros puede ser utilizado con las palabras clave *signed* y *unsigned*, que indican respectivamente si es un entero con signo o sin él, en este último caso se considera como un valor entero positivo. Por defecto se asume *signed*. Esto significa que, para 16 bits (2 bytes), los valores varían entre 32768 y -32767, ambos inclusive. En cambio, para tipos *unsigned*, el rango es entre 0 y 65535.

Ahora bien, con la finalidad de explicar mejor los tipos de datos básicos de C++, se procederá a explicar cada uno:

Tipo "char" o carácter:

[signed|unsigned] char <identificador>

Es el tipo básico alfanumérico, es decir, que puede contener un carácter, un dígito numérico o un signo. Desde el punto de vista del ordenador, todos esos valores son caracteres. El tamaño de memoria es de 1 byte.

Tipo "int" o entero:



```
[signed|unsigned] [short|long] int <identificador>  
[signed|unsigned] long [int] <identificador>  
[signed|unsigned] short [int] <identificador>
```

Las variables enteras almacenan números enteros dentro de los límites de su tamaño, a su vez, ese tamaño depende de la plataforma del compilador, y del número de bits que use por palabra de memoria: 8, 16, 23, entre otros. No hay reglas fijas para saber el mayor número que podemos almacenar en cada tipo: int, long int o short int, depende del compilador.

Tipo "float" o coma flotante:

```
float <identificador>
```

Las variables de este tipo almacenan números en formato de coma flotante, es decir, son números con decimales. Son aptos para variables de tipo real.

Tipo "bool" o Booleana:

```
bool <identificador>
```

Las variables de este tipo sólo pueden tomar dos valores "true" o "false". Sirven para evaluar expresiones lógicas. Este tipo de variables se puede usar para almacenar respuestas.

Tipo "double" o coma flotante de doble precisión:

```
[long] double <identificador>
```

Las variables de este tipo almacenan números en formato de coma flotante, al igual que float, pero usan mayor precisión. Son aptos para variables de tipo real. Estas variables son recomendadas cuando se trabaja con números grandes.

Tipo "void" o sin tipo:

```
void <identificador>
```

Es un tipo especial que indica la ausencia de tipo. Se usa en funciones que no devuelven ningún valor, también en funciones que no requieren parámetros, se usa en la declaración de punteros.

Tipo "enum" o enumerado:

```
enum [<identificador de tipo>] {<nombre de constante> [= <valor>],  
...} [lista de variables];
```

Se trata de una sintaxis muy elaborada, permite definir conjuntos de constantes, normalmente de tipo int, llamados datos de tipo enumerado. Las variables declaradas de este tipo sólo podrán tomar valores entre los

definidos. El identificador de tipo es opcional, y permitirá declarar más variables del tipo enumerado en otras partes del programa.

```
[enum] <identificador de tipo> <variable1> [,<variable2>[...]];
```

Conocidos los tipos de datos que se pueden manejar en C++, se pueden declarar los identificadores o variables seguidos del tipo de datos y finalizando con punto y coma (;), también se pueden declarar varios identificadores del mismo tipo de dato separados por coma (,), un ejemplo es:

```
int a,b,c;  
float area, altura, base;
```

Al respecto de la inicialización de variables o identificadores pueden hacerse de dos maneras, la primera es inmediatamente al declarar el tipo de dato, se iguala al valor, por ejemplo:

```
int a=5,b=3,c=8;  
float area=1.20, altura=1.5, base=3.2;
```

y la segunda corresponde a igualar el identificador al valor respectivo, es decir;

```
int a,b,c;  
float area, altura, base;  
a=5;  
b=4;  
c=8;  
area=1.20;  
altura=1.5;  
base=3.2;
```

La utilización de una u otra manera de inicializar las variables, es indiferente. Finalmente, si se tiene más de una variable que se deba inicializar con el mismo valor, C++ permite lo siguiente:

```
int a,b,c;  
a = b = c = 0;
```

- *Ámbito de las variables*

Dependiendo de dónde se declaren las variables, podrán o no ser accesibles desde distintas partes del programa. Las variables declaradas dentro de una función, solo serán accesibles desde esa función, convirtiéndose en variables locales o de ámbito local de esa función. Las variables declaradas fuera de las funciones, pueden ser utilizadas desde todas las funciones, o desde cualquier parte del programa. Es decir, serán globales o de ámbito global.



- *Operadores y expresiones*

Un operador es un símbolo que indica cómo deben ser manipulados los datos. Las operaciones asociadas a los datos de tipo fundamentales son conocidas por el compilador y requieren una definición previa de ellas, por ejemplo, la suma de enteros y resta de reales. Estas operaciones deben finalizar con un ";" al final de la línea.

Los operadores están agrupados por orden de precedencia como se puede visualizar en la siguiente Tabla:

Tabla 11.
Operadores Aritméticos y Relacionales

OPERADORES	DESCRIPCIÓN	SINTAXIS
++	post incremento pre incremento	ivalue ++ ++ Ivalue
--	post decremento pre decremento	ivalue -- -- Ivalue
* / %	multiplicación división resto	expr1 * expr2 expr1 / expr2 expr1 % expr2
+ -	suma resta	expr1 + expr2 expr1 - expr2
< <= > >= ==	menor que menor o igual que mayor que mayor o igual que igual que	expr1 < expr2 expr1 <= expr2 expr1 > expr2 expr1 >= expr2 expr1 == expr2

Fuente: Joyanes, L. (2000).

- *Operadores lógicos*

En C++ una expresión es verdadera si devuelve un valor distinto de cero, y falsa en caso contrario. Por lo tanto, el resultado de una operación lógica (AND, OR, NOT) será un valor verdadero (1) o falso (0). Los operadores lógicos en C++ se muestran en la Tabla 12.

Tabla 12.
Operadores Lógicos

OPERADOR	EXPLICACIÓN
Operador && (AND) (expresión && expresión)	<ul style="list-style-type: none">* Da como resultado el valor lógico 1 si ambas expresiones son distintas de cero.* Da como resultado el valor lógico 0 si alguna de las dos expresiones es igual a cero.
Operador (OR) (expresión expresión)	<ul style="list-style-type: none">* Da como resultado el valor lógico 1 si una de las expresiones es distinta de cero. Si lo es la primera, ya no se evalúa la segunda.* Da como resultado el valor lógico 0 si ambas expresiones son cero.
Operador ! (NOT) (expresión !)	<ul style="list-style-type: none">* Da como resultado el valor lógico 1 si la expresión tiene un valor igual a cero.* Da como resultado el valor lógico 0 si la expresión tiene un valor distinto de cero.

Fuente: Joyanes, L. (2000).

14. SENTENCIAS DE CONTROL

En este punto es importante recordar que los diferentes pasos de un algoritmo se expresan en los programas como instrucciones, sentencias o proposiciones. En este sentido, Joyanes, L. (2003: 22) considera que un programa está constituido por una serie de instrucciones, las cuales especifican las operaciones que el computador debe realizar. Asimismo, Bustamante, P. y otros (2004: 30) nos señalan: "las sentencias de un programa en C++ se ejecutan secuencialmente, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última".

Ahora bien, para Joyanes, L. (2000:85) las sentencias de control o estructuras de control, son las encargadas de controlar el flujo de ejecución: secuencia, selección y repetición. A continuación se explicarán cada una de ellas:

14.1. Toma de decisión simple (if – else)

La sentencia if le permite a un programa tomar la decisión para ejecutar una acción u otra, basándose en un resultado de verdadero o falso de la expresión evaluada. La estructura de la sintaxis es la siguiente:

Si la sentencia solo tiene una sentencia asociada no hace falta aperturar llaves:



```

if(condición)
    sentencia;
else
    sentencia;

```

Si la sentencia tiene más de una sentencia asociada, se debe aperturar las llaves para indicar que el grupo de sentencias pertenecen al condicional:

```

if (condición)
{
    sentencia 1;
    sentencia 2;
}
else
{
    sentencia 3;
    sentencia 4;
}

```

Según Joyanes, L. (2000: 86) el propósito de la sentencia if consiste en:

1. Evaluar la condición, la cual puede ser una expresión numérica (ejemplo, $x = 5$).
2. De producirse un resultado verdadero se procede a ejecutar el grupo de sentencias encerradas entre llaves después del if.
3. En el caso de que el resultado de la evaluación de la condición sea falso, no se ejecutan las sentencias 1 y 2 y se pasan a ejecutar las sentencias después de la cláusula else (sentencias 3 y 4).

Un ejemplo de la sentencia if o condicional sería:

```

//PROGRAMA QUE PERMITE LEER LA EDAD
//Y COMPARAR SI ES MAYOR A 20
#include <cstdio>
#include <iostream>
using namespace std;
main()
{
    // declaración variables
    int edad;
    //capturando
    printf("dame edad : ");
    scanf("%d",&edad);
    //comparando
    if( edad>20)
        { cout << "mayor de 20" << endl;

```

```
}  
else  
{ cout << "menor de 20" << endl;  
}  
system("pause");  
}
```

14.2. Ciclo "Para" (for)

La sentencia for permite ejecutar una instrucción, o un conjunto de ellas, cierto número de veces. Joyanes, L. (2000: 107) indica que son ciclos en los que un conjunto de sentencias se ejecutan una vez por cada valor de un rango especificado. La sintaxis de esta sentencia es la siguiente:

```
for (variable = valor; condición ; progresión)  
{  
    sentencia 1;  
    sentencia 2;  
}
```

Al igual que en el if, si el ciclo solo tiene una sentencia asociada, en la sintaxis del for se puede omitir las llaves de la siguiente manera:

```
for (variable = valor; condición ; progresión)  
    sentencia;
```

La ejecución de la sentencia for se puede explicar de la siguiente manera:

- Inicializando la variable.
- Comprobando la condición. Si la condición es verdadera se ejecutan las sentencias y se da un nuevo valor a la variable según lo indicado en la progresión el cual puede ser incremental (variable++) o decremental (variable--).
- Ejecutando la progresión se aumenta o decrementa el valor de la variable en una unidad, comprobando que la condición se cumpla, deteniendo la ejecución de la sentencia for.

Ejemplo:

```
//PROGRAMA QUE IMPRIME EN  
//PANTALLA 10 VECES LA PALABRA PRUEBA  
#include <cstdlib>  
#include <iostream>  
using namespace std;  
main()
```



```
{
// declaración variables
int x;
// instrucción for
for(x=1;x<=10;x++)
    printf("\n PRUEBA");
    system("pause");
}
```

14.3. Ciclo "Hacer – Mientras" (do – while)

La sentencia do-while ejecuta una sentencia (simple o compuesta) una o más veces, dependiendo del valor de una expresión. Joyanes, L. (2000: 113) establece que esta sentencia de control se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Su sintaxis es:

```
do {
    sentencias;
} while (condición);
```

Es importante destacar que la sentencia do-while es la única sentencia de control que finaliza con un ";"

Según el autor citado, esta estructura se realiza de la siguiente forma:

1. Se ejecuta la sentencia.
2. Se evalúa la condición, en el caso de que el resultado sea falso, se pasa el control a la siguiente sentencia del programa. Si el resultado es verdadero el proceso se repite desde el principio.

Ejemplo de la sentencia do - while

```
//PROGRAMA QUE IMPRIME LA PALABRA
//FANATICOS 10 VECES HACIENDO USO DEL
//CICLO DO WHILE
#include <iostream>
using namespace std;
main()
{
// declaración variables
int x=1;
// instrucción do while
do{
    cout << "\n FANATICOS";
    x++;
} while(x<=10);
```

```
system("pause");  
}
```

14.4. Ciclo "Mientras" (while)

La sentencia *while* funciona de forma parecida al bucle *do-while*. La diferencia principal radica en que el bucle *do-while* asegura que, al menos, se ejecuta una vez el código contenido entre las llaves, mientras que con la sentencia *while* siempre depende de la condición lógica. Las sentencias del ciclo se repiten mientras que la condición lógica sea verdadera. Para Joyanes, L. (2000: 102) cuando se evalúa la condición lógica y resulta falsa, se termina y sale del ciclo ejecutándose la siguiente sentencia de programa que se encuentra después del ciclo. Su sintaxis es:

```
while (condición)
```

```
{  
    sentencias;  
}
```

Al igual que en el *if* y en el *for*, si se tiene una sola sentencia asociada se pueden omitir las llaves. A continuación, un ejemplo de la sentencia *while*:

```
//PROGRAMA QUE IMPRIME LA PALABRA  
//INGENIEROS 10 VECES HACIENDO USO DEL  
//CICLO WHILE  
#include <iostream>  
using namespace std;  
main()  
{  
    // Declaración variables  
    int x=1;  
    // instrucción while  
    while(x<=10)  
    {  
        cout << "\n INGENIEROS";  
        x++;  
    }  
    system("pause");  
}
```

14.5. Toma de decisión múltiple (switch)

La sentencia *switch* permite ejecutar una de varias acciones, en función del valor de una expresión. Joyanes, L. (2000: 91) indica que se utiliza para seleccionar una de entre múltiples alternativas. Es una sentencia muy útil



para decisiones múltiples. La sentencia *switch* evalúa la expresión ubicada entre paréntesis y compara su valor con las constantes de cada *case*. La ejecución de las sentencias comienza en el *case*, cuya constante coincida con el valor de la expresión y continúa hasta el final del bloque de la sentencia *switch* o hasta que encuentra una sentencia tipo *break* o *return* que la saque del bloque. La sintaxis de esta sentencia es la siguiente:

```
switch (expresión)
{
    case expresión-constante 1:
        sentencia 1;
        break;
    case expresión-constante 2:
        sentencia 2;
        break;
    case expresión-constante 3:
        sentencia 3;
        break;
    default:
        sentencia 4;
}
```

La expresión evaluada por el *switch* puede ser un valor *int* o un valor *char*. En el caso del primero solo se coloca en el *case* el número (*case 1:*), pero si es *char*, se debe utilizar la comilla simple para especificar el *case* (*case 'a':*). A continuación un ejemplo de la sentencia *switch*:

```
//PROGRAMA QUE PERMITA LEER UNA LETRA Y MOSTRAR
//PALABRAS PARA LA A:AGUILA, B:BANCA, C:CABALLO
// D: DETERMINANTE, E:ESPERANZA
#include <iostream>
#include <conio.h>
using namespace std;
main()
{
    // Declaración variables
    char letra;
    //capturando
    cout << "dame una letra : ";
    letra=getchar();
    getchar();
    //inicia switch()
    switch(letra)
    {
```

```
case 'a': cout << "aguila" << endl;
        break;
case 'b': cout << "banca" << endl;
        break;
case 'c': cout << "caballo " << endl;
        break;
case 'd': cout << "determinante"<< endl;
        break;
case 'e': cout << "esperanza" << endl;
        break;
default:
        cout << "no hay" << endl;
}
system("pause");
}
```

Cabe destacar que se puede omitir el default en el switch, a conveniencia del programador.

15. COMPILACIÓN Y DEPURACIÓN DE UN PROGRAMA

El compilador es el elemento más característico del lenguaje C++, su misión consiste en traducir del lenguaje fuente a lenguaje de máquina. Joyanes, L. (2000: 44) indica que cada lenguaje de programación tiene unas reglas especiales para la construcción del programa, el cual se denomina sintaxis. El compilador lee el programa del archivo de texto (código fuente) creado, y comprueba que el programa sigue las reglas de sintaxis del lenguaje de programación.

El compilador es capaz de detectar errores durante el proceso de compilación, enviando al usuario el correspondiente mensaje de error. Los errores de programación pueden clasificarse en varios tipos, dependiendo de la fase en que se presenten. A continuación, se explican los errores más comunes detectados por el compilador:

ERRORES DE SINTAXIS: son errores en el programa fuente. Están asociados a palabras reservadas mal escritas, expresiones erróneas o incompletas y variables que no existen.

AVISOS: además de errores, el compilador puede dar también avisos (warnings). Los avisos son errores, pero no lo suficientemente graves como para impedir la generación del código objeto.

ERRORES DE ENLAZADO: el programa enlazador puede encontrar errores. Se refieren a funciones que no están definidas en ninguno de los



ficheros objetos ni en las librerías. Esto ocurre cuando falta incluir alguna librería, fichero objeto, o que falte definir alguna función o variable.

ERRORES DE EJECUCIÓN: después de obtener un archivo ejecutable, es posible que se produzcan errores. En el caso de los errores de ejecución normalmente no obtendremos mensajes de error, sino que simplemente el programa finalizará bruscamente.

ERRORES DE DISEÑO: son los errores más difíciles de corregir y prevenir. Están relacionados con el diseño del algoritmo.

Luego de corregir los errores por parte del programador, es cuando el compilador hace la traducción a código de máquina o código objeto para su posterior ejecución. Para efectos del presente libro se hace uso del editor y compilador Dev C++ versión 4.9.9.2 (ver figura 7), cuya licencia es de uso público y gratuito:



Figura 7. Acerca de VevC++

Fuente: Editor DevC++. Versión 4.9.9.2 (2009) <en línea>

Cruz, N. (2004: 4) <en línea> señala, que el Dev-C++ es un entorno de desarrollo integrado (IDE) para el lenguaje de programación C/C++ el cual permite crear:

- Programas ejecutables para Win32.
- Programas ejecutables para consola.
- Construcción de DLL's y bibliotecas estáticas.

- Además, se puede utilizar en combinación con otros compiladores basados en GCC.

En cuanto al entorno de programación es fácil de usar ya que presenta un ambiente amigable, con un menú de opciones para personalizar el proceso de ejecución y compilación (Ver Figura 8). En Dev C++, los comentarios se distinguen por el color azul, la definición de las librerías en verde, y las palabras reservadas en negritas. Para revisar o depurar los errores es necesario luego de escribir el código fuente, pulsar las teclas <Control + F9> y para compilarlo se deben emplear la combinación de <Control + F9>, asimismo, para depurar y compilar solo basta con presionar la tecla <F9>.

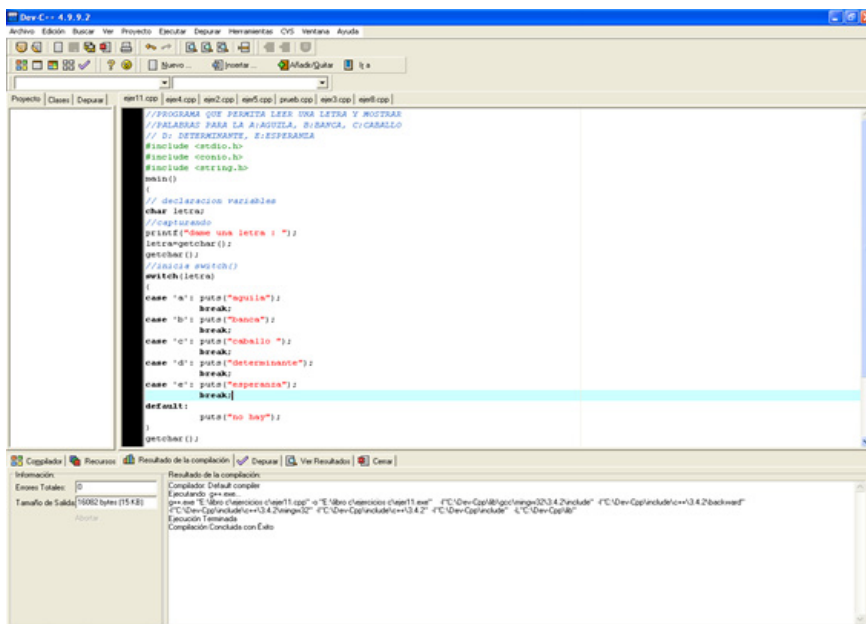


Figura 8. Entorno del Editor

Fuente: Editor DevC++. Version 4.9.9.2 (2009) <en línea>

ACTIVIDADES DE AUTOEVALUACIÓN



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

Nombre: _____

Cédula: _____ Sección: _____

Seguidamente, se presentan una serie de términos responda de manera clara y precisa:

1. Defina con sus propias palabras:

• Variable: _____

• Algoritmo: _____

• Valor Centinela: _____

• Identificador: _____

• Pseudocódigo: _____

2. Explique las características de los Algoritmos:

3. Realizar los siguientes programas en C++:

- a. Dados cuatro números diferentes, ordenarlos de manera ascendente y descendente.
- b. Calcular el salario neto semanal de N empleados teniendo el número de horas trabajadas y el salario por hora. Cada empleado recibe una asignación del 7% de su salario base, finalmente si el salario base supera los 39.000 Bs, a éste se le debe hacer una deducción del 1.5% por motivo de impuesto.
- c. Suponiendo que el operador de multiplicación no está disponible, hacer un programa que calcule A por B utilizando sumas sucesivas.
- d. Dados N números ingresados por teclado, hallar cuantos números pares hay, la sumatoria de los números pares y finalmente el promedio de los números impares.
- e. Se desea realizar una estadística de las edades de los estudiantes en un curso de programación basados en la siguiente tabla:
 - Alumnos de menos de 18 años.
 - Alumnos entre 18 y 20 años.
 - Alumnos de más de 20 años.

Hallar el número de estudiantes en cada uno de los diferentes parámetros, así como, el promedio de las edades. Se debe tener en cuenta que lo único que se leen son las edades.



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

UNIDAD II

ARREGLOS



UNIDAD II ARREGLOS

1. ARREGLOS: DEFINICIÓN

Un arreglo es un conjunto finito y fijo de elementos del mismo tipo, que se hallan almacenados en posiciones contiguas y consecutivas de memoria. Para Osorio, A. (2006: 63) los arreglos son una serie de localidades consecutivas de memoria que están asignadas a un mismo nombre y a un mismo tipo de dato, por otra parte, Joyanes, L. (2003: 249) señala que es un conjunto finito y ordenado de elementos homogéneos, finalmente, Bustamante, P. y otros (2004: 38) indican que es un modo de manejar una gran cantidad de datos del mismo tipo bajo el mismo nombre o identificador.

Con base en las teorías expuestas, un arreglo es un conjunto de valores o cantidades parecidas, que por sus cualidades se comportan de idéntica forma o deben ser tratados en forma similar. Por tanto, se pueden definir los arreglos como una estructura de datos lineales que presentan una o más dimensiones (subíndices), permitiendo almacenar y manipular varios datos de un mismo tipo. De hecho, un arreglo presenta un único nombre para todas sus posiciones de memoria, y estas posiciones se van a identificar por un subíndice particular el cual determina su posición relativa en la memoria.

Por otra parte, Corona, M. y Ancona, M. (2011: 136) puntualizan que el nombre que se le asigne al arreglo deberá presentar las mismas características de las variables, y en cuanto al subíndice, éste es un número entero positivo que dependiendo del lenguaje de programación, comenzará por cero o por uno (en el caso específico de C++, el subíndice comienza en cero). Es necesario destacar, que la cantidad de índices que tenga el arreglo, indicará el tipo de arreglo que se está utilizando. Por ejemplo: cuando el arreglo tiene un solo subíndice, éste es *unidimensional*, cuando tiene dos subíndices, *bidimensional*, tres subíndices o más, *multidimensional*.

Otro elemento que se debe tomar en cuenta con respecto a los arreglos, es la *longitud* o el *tamaño* del mismo. En este sentido, se puede decir que es el número de elementos que puede manejar o manipular el arreglo. Cuando el arreglo tiene un solo subíndice, simplemente es el último valor del mismo, pero cuando se tiene más de un subíndice, la longitud es el producto de los valores máximos de cada subíndice.

En este sentido, Corona, M. y Ancona, M. (2011: 136) precisan que los arreglos cumplen con las siguientes características básicas:

1. Ser una lista de un número finito de n elementos de un mismo tipo.
2. Almacenar los elementos del arreglo de memoria contigua.
3. Tener un único nombre de variable que representa a todos los elementos y éstos se diferencian por un índice o subíndice.
4. Acceder de manera directa o aleatoria a los elementos individuales del arreglo, por el nombre del arreglo y el índice o subíndice.

En cuanto a su utilización, los arreglos pueden aplicarse para varios propósitos ya que permiten agrupar variables relacionadas. Por ejemplo, se puede utilizar un arreglo para almacenar las notas de cada alumno de una sección o para almacenar el precio de cada libro de una librería, entre otras. La ventaja principal de este tipo de estructura, es que permite organizar y clasificar los datos de forma tal que su manipulación sea mucho más fácil.

2. ARREGLOS UNIDIMENSIONALES (VECTORES)

Muchas veces se necesita trabajar con un conjunto de valores en forma de lista, una serie de temperaturas, una lista de nombres, entre otros, y manipular estos datos con variables, se hace tarea imposible por la gran cantidad de valores. En estos casos específicos, la utilización de Arreglos lineales o unidimensionales podría significar la diferencia en el manejo de los datos y facilitaría su manipulación.

Según Joyanes, L. (2003: 249), un arreglo unidimensional es un tipo de dato estructurado compuesto de un número de elementos finitos, tamaño fijo y elementos homogéneos. *Finito* indica que hay un último elemento, *Tamaño fijo*, significa que el tamaño del arreglo debe ser conocido en tiempo de compilación, *Homogéneo*, representa que todos los elementos son del mismo tipo. Los elementos del arreglo se almacenan en posiciones contiguas de memoria, a cada una de las cuales se puede acceder directamente.

Ejemplo: suponiendo que se quiere almacenar las temperaturas relacionadas a 50 observaciones de un experimento científico, en este caso, se puede crear un arreglo unidimensional de 50 elementos llamado "*Temperatura*", quedando la estructura de la manera como se muestra en la figura 9.

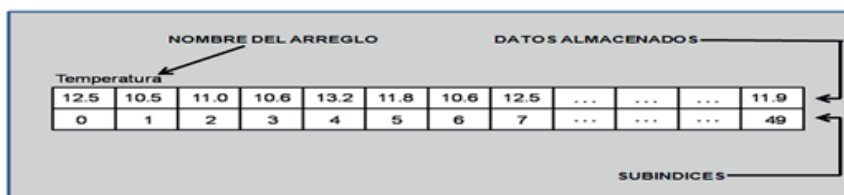


Figura 9. Arreglo Unidimensional

Se puede observar en esta estructura llamada *Temperatura*, el almacenamiento de 50 elementos de valores diferentes (en algunos casos se pueden repetir valores) y un único tipo de dato (en este caso específico, real). Por otra parte, como se mencionó anteriormente, los subíndices en C++ comienzan desde cero (0), por tanto, el último subíndice es el 49. Del cero al cuarenta y nueve (0 – 49) se tienen las cincuenta (50) temperaturas.

Ahora bien, para declarar un arreglo unidimensional en C++, se debe realizar de la siguiente manera:

`tipo nombre_arreglo[tamaño];`

Donde:

tipo Declara el tipo de dato básico del arreglo, el cual determina el tipo de dato de cada elemento contenido en el arreglo.

Nombre_arreglo Es el nombre que se le dará al arreglo, el cual debe tener las mismas características del nombre que se le da a las variables.

[] Indica que se está declarando si es un arreglo unidimensional.

tamaño Es el número de elementos que tendrá el arreglo.

Por ejemplo, si se quiere declarar un arreglo unidimensional llamado "lista" y que permita guardar un máximo de 10 elementos, se debe hacer:

`int lista[10];`

En este caso se está declarando un arreglo unidimensional tipo entero, de nombre "*lista*", el cual presenta 10 elementos (Ver figura 10). Es importante destacar que en C++ el subíndice comienza en cero y termina en $n - 1$, es decir, en este caso específico, el último subíndice es 9.

Ahora bien, si se quiere trabajar un elemento específico del arreglo, se deberá colocar el nombre del arreglo más un juego de corchete, y dentro de estos el subíndice del elemento, es decir, al referirse al tercer elemento de la lista la nomenclatura serial " *lista[2]* " (recuerde que el primer elemento de la lista es el de subíndice cero). Por ejemplo, si se deseara calcular el 25% del quinto elemento de la lista, la línea de código sería la siguiente:

$$\text{Porc} = \text{lista}[4] * 0.25;$$

En este caso, el resultado sería 8, debido a que el elemento que se encuentra en la celda indicada tiene un valor de 32.

lista										
Dato	23	43	12	34	32	56	12	23	34	76
Subíndice	0	1	2	3	4	5	6	7	8	9

Figura 10. Arreglo Unidimensional. Subíndice.

2.1. Tamaño del Arreglo

De esta manera, tomando en consideración lo antes expuesto, los arreglos son estructuras estáticas, lo cual significa que luego de establecer su tamaño, éste no puede ser modificado. El tamaño de un arreglo está relacionado con la cantidad de memoria que éste necesita para poder funcionar.

Para López, G. y otros (2009: 91) una estructura estática es cuando el tamaño (número y tipo de elementos) se define en tiempo de compilación y no cambia durante la ejecución del programa. Por tanto, es importante tener en cuenta la memoria requerida por un arreglo para su funcionamiento, ésta se encuentra directamente relacionada con el tipo de dato del arreglo, así como, el tamaño del mismo. Es decir, suponiendo un arreglo *int* (un dato tipo *int* tiene 2 bytes de memoria) de 10 elementos, el total de bytes de memoria requeridos es:

Total Bytes = 2Bytes * 10
Total Bytes = 20 Bytes

En este sentido, si se desea calcular el tamaño de la memoria que se requiere para trabajar con un arreglo específico durante la ejecución de un programa, entonces se puede utilizar la función *sizeof()*, ésta devuelve el número de bytes reservados para el arreglo completo.



Por ejemplo, el arreglo lista declarado anteriormente, es entero de 10 elementos, al aplicar la siguiente línea de código,

```
n = sizeof(lista);
```

n tomará el valor de 20, el cual es el número de bytes requeridos para el arreglo lista. Ahora bien, si se desea solicitar el tamaño de un elemento de forma individual, se debe aplicar la siguiente línea de código,

```
n = sizeof(lista[6]);
```

En este caso, n guardará el número 2, el cual es el número de bytes de un dato tipo *int*.

Por otra parte, se debe tener cuidado con los límites de los arreglos, debido al hecho de que el compilador de C++ revisa el código más no la lógica. Por ejemplo, observemos este fragmento de código:

```
int lista[10], i;  
  
for(i = 0; i < 100; i++)  
    lista[i] = i;
```

Aunque sintácticamente hablando, el código está bien escrito, al momento de ejecutar el programa se produciría un error, debido a que el arreglo *lista* se declaró con 10 elementos y en el ciclo utilizado para hacer el recorrido del mismo se tiene un máximo de 100. En este caso, el programador debe diseñar un mecanismo de control con la finalidad de evitar este problema. El código adecuado sería:

```
Int lista[10], i;  
  
for(i = 0; i < 10; i++)  
    lista[i] = i;
```

2.2. Operaciones con los Vectores

Al hablar de operaciones con los vectores (arreglos unidimensionales), se refiere a los siguientes procesos básicos: *asignación*, *lectura/escritura*, *recorrido*, *actualización*, *ordenación* (éste último se explicará más adelante en el tema correspondiente a Métodos de Ordenación) y búsqueda. Seguidamente, Joyanes, L. (2003: 253) detalla cada uno de ellos:

ASIGNACIÓN: no es más que guardar el dato en una celda específica del arreglo. Si se desea asignar valores a todos los elementos de un vector, se debe recurrir a estructuras de control repetitivas (*while / do – while / for*). Por ejemplo:

a.- Si se desea asignar el número 10 a un vector de nombre edad en la celda número 2 (subíndice), se realizaría de la siguiente manera:

`edad[2] = 10;`

b.- Si se desea asignar de manera predeterminada todos los elementos de un vector de 10 elementos tipo int, se procedería de la siguiente manera:

`int Vec[10] = {1,2,3,4,5,6,7,8,9,0};`

En este caso, cada elemento se separa por coma y el primer elemento (1) se ubicaría en la celda 0 (subíndice), el segundo en la celda 1 y así sucesivamente.

c.- Si se desea asignar en un vector de 10 el valor 5 en todas las celdas, una manera de realizar el proceso sería:

```
for(i=0; i<10; i++)  
    Vec[i] = 5;
```

LECTURA / ESCRITURA: es el proceso mediante el cual los valores del arreglo son ingresados por teclado. Para esto es necesaria la utilización de sentencias repetitivas de control. Por ejemplo:

```
#include <iostream>  
using namespace std;  
main()  
{  
    int i;  
    int lista[10];  
    cout << "INGRESE LOS DATOS: " << endl;  
    for(i=0; i<10; i++)  
    {  
        cout << "INGRESE EL DATO ["<<(i+1)<<"] = ";  
        cin >> lista[i];  
    }  
}
```

RECORRIDO: es el tratamiento secuencial del vector, mediante el cual se accede sucesiva y consecutivamente a los contenidos de cada uno de los elementos del mismo. Este proceso se realiza cada vez que se va a trabajar con el arreglo.

Por ejemplo: si se desea sumar una serie de números en un vector precisamente ingresado, el código en C++ sería:

```

#include <iostream>
using namespace std;
main()
{
    int i, suma=0;
    int lista[10];
    cout << "INGRESE LOS DATOS: " << endl;
    for(i=0; i<10; i++)
    {
        cout << "INGRESE EL DATO ["<<(i+1)<<"] = ";
        cin >> lista[i];
    }
    // Recorrido del vector para hacer la sumatoria de sus elementos
    for(i=0; i<10; i++)
        suma += lista[i];    // suma = suma + lista[i];
    system("cls");
    cout << "LA SUMA DE LOS ELEMENTOS = " << suma << endl;
    system("pause");
}

```

ACTUALIZACIÓN: esta operación se clasifica a su vez en tres operaciones elementales: añadir, insertar y borrar.

- Añadir: es la operación de añadir un nuevo elemento al final del vector. La única condición necesaria para esta operación consiste en la comprobación de espacio de memoria suficiente para el nuevo elemento. Es decir, que el arreglo no tenga todas las celdas ocupadas. Por ejemplo:

```

#include <iostream>
using namespace std;
main()
{ int i, vec[10];
  cout << "INGRESE LOS PRIMEROS CINCO ELEMENTOS" << endl;
  for(i=0; i<5; i++)
  { cout << "ELEMENTO["<< (i+1) <<"] = ";
    cin >> vec[i];
  }
  // Como el vector no se llenó en su totalidad, se puede hacer esto:
  Vec[5] = 5;
}

```

- Inserción: consiste en introducir en el arreglo el valor de un elemento, bien sea en una celda que haya quedado vacía o para sustituir el valor de una celda ocupada. Por ejemplo:

```
#include <iostream>
using namespace std;
main()
{ int i,n;
  int lista[10];
  cout << "INGRESE LOS DATOS:"<< endl;
  for(i=0; i<10; i++)
  {
    cout << "INGRESE EL DATO ["<< (i+1) <<"] = ";
    cin >> lista[i];
  }
  do{ cout << "ING. LA POSIC. DE LA CELDA A GUARDAR = ";
      cin >> n;
  }while((n<0) || (n>=10));
  cout << "ING. EL VALOR A GUARDAR: ";
  cin >> lista[n];
  system("PAUSE");
}
```

- Borrado: consiste en eliminar del arreglo el valor de un elemento. El borrado se puede realizar de dos formas: a) introduciendo, en la celda de memoria correspondiente, una señal de borrado en lugar del dato almacenado; y b) eliminando totalmente su contenido, mediante el desplazamiento de los elementos del vector para ocupar el espacio borrado, si el elemento en cuestión no es el último del vector, el último elemento del vector queda con una señal de borrado. El código en C++ suponiendo que la señal de borrado sea el cero (0) y que se quiere borrar el elemento número 6 sería:

```
#include <iostream>
using namespace std;
main()
{
  int i,n;
  int lista[10];
  cout << "INGRESE LOS DATOS:" << endl;
  for(i=0; i<10; i++)
  {
```




```

        cout << "INGRESE EL DATO ["<< (i+1) <<"] = ";
        cin >> lista[i];
    }
    do{ cout << "ING. LA POSIC. DE LA CELDA A BORRAR = ";
        cin >> n;
    }while((n<0) || (n>=25));
    lista[n] = 0;
    system("PAUSE");
}

```

El código en C++ suponiendo que se va a eliminar el mismo elemento desplazando los que están por encima de él y la señal de borrado sea cero, sería:

```

#include <iostream>
using namespace std;

main()
{
    int i,valor,c;
    int lista[10];
    c=0;
    cout << "INGRESE LOS DATOS:" << endl;
    for(i=0; i<10; i++)
    {
        cout << "INGRESE EL DATO ["<< (i+1) <<"] = ";
        cin >> lista[i];
    }
    system("cls");
    cout << "ING. EL VALOR A BUSCAR: ";
    cin >> valor;
    system("cls");
    for(i=0; i<10; i++)
    {
        if(lista[i] == valor)
        {
            c++;
            cout << valor << " SE ENCUENTRA EN LA CELDA " << i <<
endl;
        }
    }
    if(c == 0)
        cout << "NO EXISTE NINGUN ELEM. CON ESE VALOR" << endl;
}

```

```

else
    cout << "EXISTEN " << c << " CON ESE VALOR" << endl;
system("PAUSE");
}

```

BÚSQUEDA: consiste en realizar un recorrido del vector, comenzando de su posición más baja de memoria (en este caso desde cero), a fin de localizar en el arreglo un dato determinado. En este fin, se puede dar el caso de que no exista el dato o en caso contrario que esté más de uno. Un ejemplo del código sería:

```

#include <iostream>
using namespace std;

main()
{
    int i, ndp;
    float prom, sum, nota[25];
    ndp = 0;
    sum = 0;
    cout << "INGRESE LAS NOTAS DE LOS ALUMNOS:" << endl;
    for(i=0; i<25; i++)
    {
        do{ cout << "INGRESE LA NOTA " << (i+1) << " = ";
            cin >> nota[i];
        }while((nota[i]<0) || (nota[i]>20));
        sum += nota[i];
    }
    prom = sum / 25;
    for(i=0; i<25; i++)
    {
        if( nota[i] < prom)
            ndp++;
    }
    system("CLS");
    cout << "EL PROMEDIO DE LAS 25 NOTAS = " << prom << endl;
    cout << "EXISTEN " << ndp << " MENORES AL PROMEDIO" <<
endl;
    system("PAUSE");
}

```

Para comprender como se utilizan los arreglos en C++, se va a realizar el siguiente ejemplo explicándolo paso a paso con comentarios.



Calcular el promedio de 25 notas e indicar cuantas están por debajo del mismo.

/* Lo primero que se debe hacer en todos los programas en C++, es aperturar las librerías que se van a utilizar, así como, la apertura del método main.

```

*/
#include <iostream>
using namespace std;
main()
{
    //Se declaran las variables a utilizar y el arreglo nota
    int i,ndp;
    float prom, sum, nota[25];

    //Se inicializan las variables auxiliares
    ndp = 0;    //variable para contar el número de notas por debajo del
    prom.
    sum = 0;    //variable para acumular las notas

    //Se pide las notas de los alumnos, utilizando un ciclo "for"
    cout << "INGRESE LAS NOTAS DE LOS ALUMNOS:" << endl;
    for(i=0; i<25; i++)
    { do{ cout << "INGRESE LA NOTA " << (i+1) << " = ";
        cin >> nota[i];
    }while((nota[i]<0) || (nota[i]>20));
        sum += nota[i];
    }
    //Cálculo del promedio
    prom = sum / 25;

    // Para buscar cuantas notas están por debajo del promedio, se debe
    hacer
    // un recorrido del arreglo e ir comparando la nota con el promedio.
    for(i=0; i<25; i++)
    { if( nota[i] < prom)
        ndp++;
    }

    // Se muestran los resultados.
    system("CLS");

```

```
cout << "EL PROMEDIO DE LAS 25 NOTAS = " << prom << endl;  
cout << "EXISTEN " << ndp << " MENORES AL PROMEDIO" <<  
endl;  
system("PAUSE");  
}
```

Observemos otro ejemplo: en un almacén, al final del día se registraron un total de 15 ventas en uno de sus departamentos, los montos de cada venta fueron: 1250.00 – 987.25 – 1874.50 – 784.25 – 2745.00 – 1873.95 – 1254.85 – 759.10 – 1524.25 – 987.25 – 1425.60 – 1035.85 – 854.25 – 2457.10 – 1325.45. Calcular el monto total obtenido por las ventas, así como, la ganancia del almacén, suponiendo que ésta es solo el 30% del monto total.

En este caso, el enunciado no solo muestra el tamaño del arreglo (15 elementos), si no también, está aportando los valores que se deben guardar en cada una de las posiciones (celdas) del mismo. En este caso, se dice que los datos del arreglo son pre-determinados y la forma de declarar es la siguiente:

tipo nombre_arreglo[tamaño] = {V1, V2, ... Vn};

Donde dentro del juego de llaves, se colocan los valores a guardar en el arreglo. Por tanto, el código sería:

```
#include <iostream>  
  
//Forma de definir las constantes  
  
#define N 15      //Tamaño del arreglo  
#define P 0.3     //Porcentaje de ganancia  
using namespace std;  
  
main()  
{  
    int i,aux;  
    double mt, gana;  
  
    //Declaración del arreglo pre-determinado  
  
    double venta[N] = {1250.00, 987.25, 1874.50, 784.25, 2745.00,  
                        1873.95, 1254.85, 759.10, 1524.25, 987.25,  
                        1425.60, 1035.85, 854.25, 2457.10, 1325.45};  
  
    //Se inicializan en cero las variables auxiliares  
  
    mt = gana = aux = 0;
```

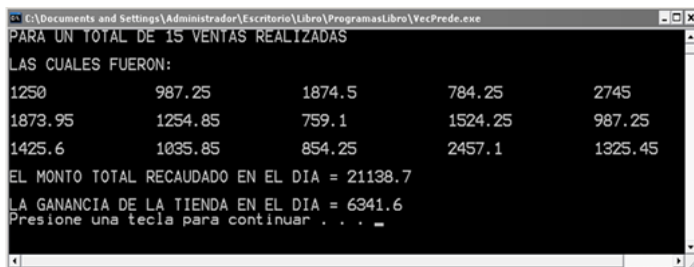
```

//Se calcula el monto total de la venta
for(i=0; i<N; i++)
    mt += venta[i];
//Se calcula la ganancia de la tienda
gana = mt * P;
//Se muestran los datos por pantalla
cout << "PARA UN TOTAL DE " << N << " VENTAS REALIZADAS"
<< endl;
cout << "\nLAS CUALES FUERON: " << endl << endl;
//Para mostrar el vector se hace:
for(i=0; i<N; i++)
{ cout << venta[i] << "\t\t";
  aux++;
  if(aux == 5)
  { cout << endl;
    aux = 0;
  }
}

//ahora se muestra el monto total y la ganancia
cout << "EL MONTO TOTAL RECAUDADO EN EL DIA = " << mt <<
endl;
cout << endl;
cout << "LA GANANCIA DE LA TIENDA EN EL DIA = " << gana <<
endl;
system("PAUSE");
}

```

La salida de este programa sería:



```

C:\Documents and Settings\Administrador\Escritorio\Libro\ProgramasLibro\VecPrede.exe
PARA UN TOTAL DE 15 VENTAS REALIZADAS
LAS CUALES FUERON:
1250      987.25      1874.5      784.25      2745
1873.95    1254.85      759.1      1524.25     987.25
1425.6     1035.85      854.25     2457.1      1325.45
EL MONTO TOTAL RECAUDADO EN EL DIA = 21138.7
LA GANANCIA DE LA TIENDA EN EL DIA = 6341.6
Presione una tecla para continuar . . . _

```

Figura 11. Salida por Pantalla. VecPrede.exe

3. ARREGLOS BIDIMENSIONALES (MATRICES)

Otra forma de representar los arreglos, son los bidimensionales o matrices, según Joyanes, L. (2003: 259) es un vector de vectores. Por tanto, es un conjunto de elementos del mismo tipo en el que el orden de los componentes es significativo y en el que se necesitan especificar dos subíndices para poder identificar a cada elemento del arreglo. Asimismo, Corona, M. y Ancona, M. (2011: 156) lo definen como un conjunto de n elementos del mismo tipo, almacenados en memoria contigua en una matriz o tabla.

Analizando, las definiciones anteriores, podemos plantear que un arreglo bidimensional es una estructura que permite el manejo de varios datos de un mismo tipo a la vez. El mismo se representa por medio de filas y columnas, por tal motivo, se necesita el manejo de dos subíndices (uno para la fila y el otro para la columna), lo cual hace la idea de que se esté trabajando con una matriz.

Por ejemplo, dado un arreglo bidimensional llamado "*matriz*" de cuatro filas y tres columnas, es decir, `matriz[4][3]` (siempre se coloca primero las filas y luego las columnas), la representación gráfica del mismo sería la expresada en la figura 12.

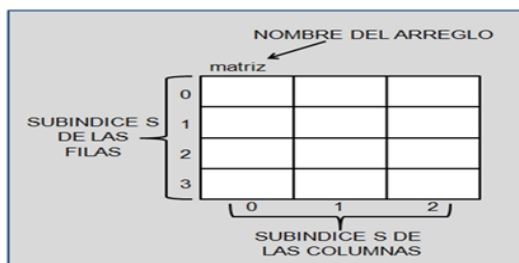


Figura 12. Arreglo Bidimensional.

En este caso, se puede observar como los subíndices de la filas van del cero al tres (0 al 3) y por otra parte, los subíndices de las columnas van del cero al dos (0 al 2), es importante resaltar que al igual que en los arreglos unidimensionales, los subíndices comienzan en cero y terminan en $n-1$. Por tanto, si se desea trabajar un elemento específico del arreglo, se debe colocar el nombre del mismo más el subíndice de la fila dentro de un juego de corchetes y el subíndice de las columnas dentro de otro juego de corchetes, es decir, "`matriz[1][2]`"; de esta manera, se estaría refiriendo al elemento que se encuentra en la segunda fila tercera columna del arreglo:

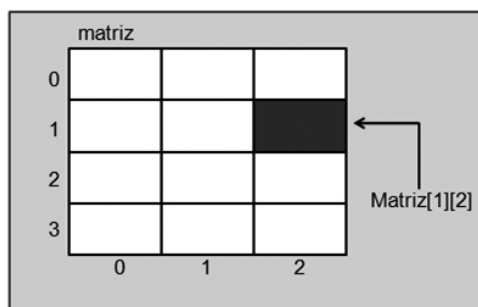


Figura 13. Arreglo Bidimensional. Subíndice.

Si se desea saber cuál es el número total de elementos que el arreglo puede almacenar, se deberá multiplicar el número de filas por el número de columnas. En este sentido, se puede decir que el arreglo *matriz* trabaja con un total de 12 elementos (4 filas por 3 columnas = 12 elementos).

Ahora bien, para declarar un arreglo bidimensional en C++, se debe realizar de la siguiente manera:

tipo nombre_arreglo [tamaño_f] [tamaño_c];

Donde:

Tipo	Declara el tipo básico del arreglo, el cual determina el tipo de cada elemento contenido en el arreglo.
Nombre_arreglo	Es el nombre que se le dará al arreglo, el cual debe tener las mismas características del nombre que se le da a las variables.
[][]	Indica que se está declarando que es un arreglo bidimensional.
Tamaño_f	Es el número de filas que tendrá el arreglo.
Tamaño_c	Es el número de columnas que tendrá el arreglo.

Por ejemplo: supongamos que se quiere declarar un arreglo bidimensional tipo entero llamado tabla de 4 filas y 3 columnas:

int tabla [4][3];

3.1. Operaciones con las Matrices

Al hablar de operaciones con las matrices (arreglos bidimensionales), podemos hablar de los mismos procesos expuestos por Joyanes, L. (2003:

253) en los vectores, pero adaptados a éstas. Recordemos que estos son: asignación, lectura/escritura, recorrido, actualización, y búsqueda. A continuación se explicarán cada uno de ellos:

ASIGNACIÓN: igual que en los vectores, no es más que guardar el dato en una celda específica del arreglo. Si se desea asignar valores a todos los elementos de un vector, se debe recurrir a estructuras de control repetitivas (while / do – while / for), pero ahora enlazadas. Por ejemplo:

a.- Si se desea asignar el número 10 a una matriz de nombre mat en la celda ubicada en la fila 2 columna 3, se realizaría de la siguiente manera:

```
mat[2][3] = 10;
```

b.- Si se desea asignar de manera predeterminada todos los elementos de una matriz de 3x3 elementos tipo int, se haría de la siguiente manera:

```
int Mat[3] [3] = {{1,2,3},  
                  {4,5,6},  
                  {7,8,9}};
```

En este caso, se deben colocar todos los datos dentro de una llave ({ }), en éstas se colocarán tantos grupos de llaves con los valores separados por coma (,), como filas tenga la matriz.

c.- Si se desea asignar en una matriz de 3x4 el valor 5 en todas las celdas, una manera de realizar el proceso sería:

```
for(i=0; i<3; i++)  
    for(j=0; j<4; j++)  
        vec[i] = 5;
```

De esta manera, el primer ciclo for permitirá movernos entre las filas y el segundo entre las columnas.

LECTURA / ESCRITURA: es el proceso mediante el cual los valores de la matriz son ingresados por teclado. Para esto es necesaria la utilización de sentencias repetitivas de control enlazadas. Por ejemplo:

```
#include <iostream>  
using namespace std;  
main()  
{  
    int i, j;  
    int matriz[3] [3];  
    cout << "INGRESE LOS DATOS DE LA MATRIZ: " << endl;  
    for(i=0; i<3; i++)
```




```

        for(j=0; j<3; j++)
        {
            cout << "INGRESE EL DATO ["<<(i+1)<<"] ["<<(j+1)<<"] = ";
            cin >> matriz[i] [j];
        }
    }
}

```

RECORRIDO: se utiliza para acceder a los elementos del arreglo. Sin embargo, es importante resaltar que el recorrido se realiza por filas, es decir, primero se recorre la fila cero (0), luego la fila (1) y así sucesivamente,

```

#include <iostream>
using namespace std;
main()
{
    int i,j;
    int tabla[4][3];
    cout << "INGRESE LOS DATOS:" << endl;

    //Recorrido del arreglo
    for(i=0; i<4; i++)
    {
        for(j=0; j<3; j++)
        { cout << "INGRESE EL DATO ["<<(i+1)<<"] ["<<(j+1)<<"] = ";
          cin >> tabla[i][j];
        }
    }
    system("PAUSE");
}

```

En el ejemplo podemos observar la utilización de dos ciclos *for* contiguos, el primero controla el movimiento de las filas y el segundo el de las columnas. Como el segundo *for* se encuentra dentro del primero, el recorrido se realiza fila por fila, es decir, cuando el primer *for* se ejecuta por primera vez, se ingresan todos los elementos de esa fila y así sucesivamente.

ACTUALIZACIÓN: al igual que en los vectores, esta operación se clasifica a su vez en tres operaciones elementales: añadir, insertar y borrar.

- Añadir: es la operación de añadir un nuevo elemento a la matriz. La única condición necesaria para esta operación consiste en la comprobación de espacio de memoria suficiente para el nuevo

elemento. Es decir, que el arreglo no tenga todas las celdas ocupadas. Por ejemplo:

```
#include <iostream>
using namespace std;
main()
{ int i, j, mat[3][4];
  cout << "INGRESE LAS DOS PRIMERAS FILAS" << endl;
  for(i=0; i<2; i++)
    for(j=0; j<4; j++)
    { cout << "ELEMENTO[" << (i+1) << "]" << "[" << (j+1) << "]" = ";
      cin >> mat[i][j];
    }
  // Como la matriz no se llenó en su totalidad, se puede hacer esto:
  cout << "INGRESE LOS ELEMENTOS DE LA TERCERA FILA:" << endl;
  for(i=0; i<4; i++)
  {   cout << "ELEMENTO[3][" << (i+1) << "]" = ";
      cin >> mat[2][i];
  } }
```

- Inserción: igual que en los vectores, la idea consiste en introducir en la matriz el valor de un elemento cualquiera, bien sea en una celda que haya quedado vacía o para sustituir el valor de una celda ocupada.

```
#include <iostream>
using namespace std;
main()
{
  int i,j,ni,nj;
  int tabla[4][3];
  cout << "INGRESE LOS DATOS:" << endl;
  for(i=0; i<4; i++)
    for(j=0; j<3; j++)
    { cout << "INGRESE EL DATO [" << (i+1) << "]" << "[" << (j+1) << "]" = ";
      cin >> tabla[i][j];
    }
  system("CLS");

  do{ cout << "ING. LA FILA DE LA CELDA A GUARDAR = ";
      cin >> ni;
  }while((ni<0) || (ni>=4));
```



```

do{ cout << "ING. LA COLUM. DE LA CELDA A GUARDAR = ";
    cin >> nj;
}while((nj<0) || (nj>=3));

cout << "ING. EL VALOR A GUARDAR = ";
cin >> tabla[ni][nj];
system("PAUSE");
}

```

- Borrado: consiste en eliminar del arreglo el valor de un elemento, bien sea introduciendo en la celda correspondiente, una señal de borrado en lugar del dato almacenado o eliminando totalmente su contenido.

```

#include <iostream>
using namespace std;

main()
{
    int i,j,ni,nj;
    int tabla[4][3];
    cout << "INGRESE LOS DATOS:" << endl;
    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
        { cout << "INGRESE EL DATO ["<<(i+1)<<"]["<<(j+1)<<"] = ";
          cin >> tabla[i][j];
        }
    system("CLS");
    do{ cout << "ING. LA FILA DE LA CELDA A GUARDAR = ";
        cin >> ni;
    }while((ni<0) || (ni>=4));
    do{ cout << "ING. LA COLUM. DE LA CELDA A GUARDAR = ";
        cin >> nj;
    }while((nj<0) || (nj>=3));

    tabla[ni][nj] = 0; //Eliminación del elemento

    system("PAUSE");
}

```

BÚSQUEDA: consiste en realizar el recorrido de la matriz a fin de localizar en ella un dato determinado. Si se da el caso de no existir, el dato se debe indicar mediante un mensaje o en caso contrario que esté más de uno, los mismos deben ser identificados, diciendo cuantos existen o mostrando las posiciones (subíndices) de cada uno.

```
#include <iostream>
using namespace std;
main()
{
    int i,j,valor,c=0;
    int tabla[4][3];
    cout << "INGRESE LOS DATOS:" << endl;
    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
        { cout << "INGRESE EL DATO ["<<(i+1)<<"]["<<(j+1)<<"] = ";
          cin >> tabla[i][j];
        }
    system("CLS");
    cout << "ING. EL VALOR A BUSCAR = ";
    cin >> valor;
    system("CLS");

    // Proceso de búsqueda

    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
        { if(tabla[i][j] == valor)
          { c++;
            cout << valor << " ESTA EN LA CELDA ["<<(i+1)<<"]["<<(j+1)<<"]" << endl;
          }
        }
    if(c == 0)
        cout << "NO EXISTE NINGUN ELEM. CON ESE VALOR" << endl;
    else
        cout << "EXISTEN ["<< c <<"] ELEMENTOS CON ESE VALOR" <<
endl;
    system("PAUSE");
}
```

A continuación, se explicará paso a paso el siguiente ejemplo utilizando los comentarios dentro del código:

Elaborar un programa en C++ que lea dos matrices de 5*4 y posteriormente realice la suma de ellas.

```
#include <iostream>
using namespace std;
```



```

main()
{
    // Declaración de las variables y los arreglos a utilizar
    int i,j;
    int matriz1[5][4], matriz2[5][4], matrizS[5][4];
    // Se ingresan los elementos de la primera matriz
    cout << "INGRESE LOS DATOS DE LA MATRIZ 1:" << endl;
    for(i=0; i<5; i++)
        for(j=0; j<4; j++)
            { cout << "INGRESE EL DATO ["<<(i+1)<<"]["<<(j+1)<<"] = ";
              cin >> matriz1[i][j];
            }
    system("CLS");
    // Se ingresan los elementos de la segunda matriz
    cout << "INGRESE LOS DATOS DE LA MATRIZ 2:" << endl;
    for(i=0; i<5; i++)
        for(j=0; j<4; j++)
            { cout << "INGRESE EL DATO ["<<(i+1)<<"]["<<(j+1)<<"] = ";
              cin >> matriz2[i][j];
            }
    system("CLS");
    // Se realiza la suma de las matrices, mostrando resultado por pantalla
    cout << "MATRIZ SUMA ES:" << endl << endl;
    for(i=0; i<5; i++)
        { for(j=0; j<4; j++)
            { matrizS[i][j] = matriz1[i][j] + matriz2[i][j];
              cout << "\t" << matrizS[i][j] << "\t";
            }
          cout << endl;
        }
    system("PAUSE");
}

```

Observemos ahora otro ejemplo: Dadas las notas de 10 alumnos (ver tabla 13), calcular el promedio de cada alumno, así como, el promedio del curso, indicar cuantos alumnos aprobaron el curso y cuantos reprobaron.

Tabla 13.
Notas de los alumnos

	Nota 1	Nota 2	Nota 3
Alumno 1	15	12	17
Alumno 2	8	10	9
Alumno 3	14	10	7
Alumno 4	14	18	15
Alumno 5	8	6	9
Alumno 6	14	12	10
Alumno 7	9	5	10
Alumno 8	18	14	13
Alumno 9	20	18	20
Alumno 10	17	20	18

En el ejemplo anterior, los datos del arreglo bidimensional se producen con anticipación, por tanto, al momento de declarar el vector se deberá usar la siguiente forma:

```
tipo nombre_arreglo [tamaño_f] [tamaño_c] = { {elementos fila 0},  
                                                {elementos fila 1},  
                                                .....  
                                                {elementos fila n-1}  
                                                };
```

Donde los elementos de cada fila deben ir separados por coma. Por tanto, el programa sería de la siguiente manera:

```
#include <iostream>  
using namespace std;  
main()  
{  
    //Se declara el arreglo predeterminado  
    int nota[10][3] = { {15, 12, 17},  
                        {8, 10, 9},  
                        {14, 10, 7},  
                        {14, 18, 15},  
                        {8, 6, 9},  
                        {4, 12, 10},
```



```

        {9, 5, 10},
        {18, 14, 13},
        {20, 18, 20},
        {17, 20, 18} };

//Se declaran las variables y el arreglo auxiliar a utilizar
int i,j;      // para los subíndices
int sum;      // para sumar las notas por alumnos
int prom[10]; // para el cálculo del promedio por alumno
int sumCur;   // acumulador para sumar los promedios por alumnos
int ProCur;  // para hallar el promedio del curso
int aluApr;   // contador para la cantidad de aprobados
int aluRep;   // contador para la cantidad de reprobados
//Se inicializan el acumulador y los contadores en cero
sumCur = aluApr = aluRep = 0;
//Se apertura el recorrido del arreglo para comenzar los cálculos
for(i=0; i<10; i++)
{
    sum = 0;
    for(j=0; j<3; j++)
        sum += nota[i][j];
    prom[i] = sum / 3; // Cálculo del promedio por alumno
    // Se pregunta si el promedio es >= 10
    if(prom[i] >= 10)
        aluApr++; // Alumnos aprobados
    else
        aluRep++; // Alumnos reprobados
    sumCur += prom[i]; // Sumatoria de todos los promedios
}

//Cálculo del promedio del curso
ProCur = sumCur / 10;
//Se muestran los datos en pantalla
cout << "LA NOTA DE CADA ALUMNO ES:" << endl << endl;
cout << "\tN_1\tN_2\tN_2\tPRO" <<endl;
for(i=0; i<10; i++)
{
    cout << "Alu[" << i+1 << "]\t";
    for(j=0; j<3; j++)
        cout << nota[i][j] << "\t";
    cout << prom[i];
    cout << endl;
}
cout << endl;
cout << "    PROMEDIO DEL CURSO = " << ProCur << endl;

```

```
cout << " CANTIDAD DE APROBADOS = " << aluApr << endl;
cout << "CANTIDAD DE REPROBADOS = " << aluRep << endl;
system("PAUSE");
```

La salida de este programa sería:



```
LA NOTA DE CADA ALUMNO ES:
Alu[1]  N_1  N_2  N_2  PRO
Alu[2]  15  12  17  14
Alu[3]  8  10  9  9
Alu[4]  14  18  7  10
Alu[5]  8  6  9  10
Alu[6]  14  12  10  12
Alu[7]  9  5  10  8
Alu[8]  18  14  13  10
Alu[9]  20  14  20  15
Alu[10] 17  20  18  18

PROMEDIO DEL CURSO = 12
CANTIDAD DE APROBADOS = 7
CANTIDAD DE REPROBADOS = 3
Presione una tecla para continuar...
```

Figura 14. Salida por Pantalla. Notas.exe

4. MÉTODOS DE ORDENACIÓN

Los métodos o algoritmos de ordenación, permiten como su nombre lo indica, ordenar los datos tanto en vectores como en matrices. Para Corona, M. y Ancona, M. (2011: 266), su finalidad es clasificar datos en un orden ascendente o descendente mediante un criterio. Por otra parte, Joyanes, L. y otros (2007: 212) indican, que la ordenación o clasificación de datos es una operación consistente en disponer un conjunto de datos en algún determinado orden con respecto a uno de los campos de los elementos del conjunto. En este sentido, existen distintos métodos con distintas características y complejidad, desde el método más simple, como el Bubblesort (Método Burbuja), el cual se refiere a simples iteraciones, hasta el Quicksort (Método Rápido), que al estar optimizado usando recursividad, el tiempo de ejecución es menor y es más efectivo.

Esta unidad se centrará en los métodos más populares, básicos y sencillos, analizando la cantidad de comparaciones que suceden y revisando el código, escrito en C++.

4.1. Ordenación por Selección

Para Joyanes, L. (2003: 370) y Corona, M. y Ancona, M. (2011: 268) la idea básica de la ordenación por selección es encontrar el elemento más pequeño de la lista y transferirlo a la primera posición, luego encontrar el segundo elemento más pequeño y llevarlo a la segunda posición, y así sucesivamente. En este sentido, Joyanes, L. (2003: 370) nos da los siguientes pasos:



1. Seleccionar el elemento menor del vector de n elementos.
2. Intercambiar dicho elemento con el primero.
3. Repetir estas operaciones con los $n-1$ elementos restantes, seleccionando el segundo elemento, continuar con los $n-2$ elementos restantes hasta que solo quede el mayor.

Basados en lo antes expuesto, se puede deducir que el número de veces a recorrer el arreglo es de $N-1$ veces y el número de intercambios máximo a realizar también es de $N-1$ intercambios. Es un método bastante sencillo de codificar, sin embargo, el tiempo de ejecución no es muy óptimo pero es apropiado utilizarlo con arreglos pequeños.

Por ejemplo, si se tiene la siguiente lista:

Figura 15. Ordenación. Selección 01

Para ordenar el primer valor:

1. Se localiza el elemento más pequeño de la lista realizando un recorrido en el arreglo (Figura 16).

Figura 16. Ordenación. Selección 02

2. Se intercambia el elemento más pequeño con el que se encuentra en la primera posición (Figura 17).

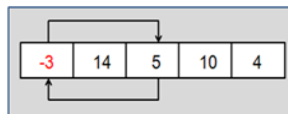
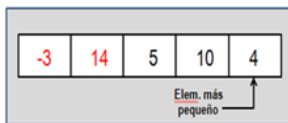


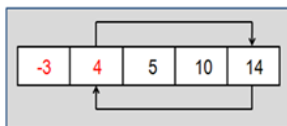
Figura 17. Ordenación. Selección 03

Para ordenar el segundo valor:

1. Se localiza el segundo elemento más pequeño de la lista realizando un recorrido en el arreglo (Figura 18).


Figura 18. Ordenación. Selección 04

2. Se intercambia el segundo elemento más pequeño con el que se encuentra en la segunda posición (Figura 19).


Figura 19. Ordenación. Selección 05

Para ordenar el tercer valor:

1. Se localiza el tercer elemento más pequeño de la lista realizando un recorrido en el arreglo (Figura 20).

■

Figura 20. Ordenación. Selección 06

Como se puede observar, el resto de los elementos del arreglo ya se encuentran ordenados, por tanto, aunque los recorridos se seguirán realizando hasta completar el número de iteraciones ($n-1$ recorridos), ya no se realizará ningún intercambio quedando de esta manera el arreglo ordenado.

El código que se utiliza para el método de selección es:

```
for( i = 0; i < (n-1); i++)
{
    Ind = i;
    aux = lista[i];
    for( j = (i+1); j < n; j++)
    {
        if(lista[j] < aux)
        {
            Ind = j;
            aux = lista[j];
        }
    }
}
```



```

    }
    lista[Ind] = lista[i];
    lista[i] = aux;
}

```

Donde el primer ciclo for cumple dos funciones: primero el controla el número de veces que se debe recorrer el arreglo (debemos recordar que es N-1 veces), y la segunda, es la de inicializar las dos variables auxiliares. El segundo ciclo for, es el encargado de realizar el recorrido del arreglo, observemos que el valor de inicio viene dado por i+1, esto es debido a que después que un elemento es colocado en la posición que le corresponde, ese elemento no se vuelve a comparar con ningún otro.

Por otra parte, se observan dos variables auxiliares, la primera es Ind, esta variable se utilizará para guardar la posición del elemento más pequeño, mientras que la variable aux, guardará el valor del elemento más pequeño. Por ejemplo, si se quiere realizar la ordenación de 10 elementos ingresados por teclado, el código sería:

```

#include <iostream>
using namespace std;

main()
{ int i,j,Ind,aux,lista[10];

//Se pide ingresar los elementos del arreglo
cout << "INGRESE LOS DATOS:" << endl;
for(i=0; i<10; i++)
{ cout << "INGRESE EL DATO [" << (i+1) << "] = ";
  cin >> lista[i];
}

//Se aplica el método de ordenación

for( i = 0; i < (10-1); i++)
{ Ind = i;
  aux = lista[i];
  for( j = (i+1); j < 10; j++)
  { if(lista[j] < aux)
    { Ind = j;
      aux = lista[j];
    }
  }
  lista[Ind] = lista[i];
}
}

```

```

        lista[i] = aux;
    }
    //Se muestra el arreglo ordenado
    system("CLS");
    cout << "LOS DATOS ORDENADOS SON:" << endl;
    for(i=0; i<10; i++)
        cout << "ELEMENTO[" << (i+1) << "] = " << lista[i] << endl;
    system("PAUSE");
}

```

Ahora bien, si lo que se quiere es ordenar los elementos en un arreglo bidimensional, se debe tomar en cuenta en primer lugar, que la ordenación se realizará bien sea por una fila específica o por una columna específica, y en el momento de realizar el intercambio se deben cambiar todos los elementos de la columna o todos los elementos de la fila. Por ejemplo, supongamos el arreglo bidimensional de la figura 21.

Figura 21. Arreglo Bidimensional 01

Donde la ordenación se debe realizar tomando como referencia los elementos de la segunda columna (figura 22),

5	8	3	7
2	7	1	3
9	3	6	2
7	3	2	9

Figura 22. Arreglo Bidimensional 02

El código a aplicar sería en este caso:



```
#include <iostream>

//FORMA DE DEFINIR CONSTANTE
#define n 4
#define m 4

using namespace std;

main()
{
    int mat[n][m] = { {5,8,3,7},
                      {2,7,1,3},
                      {9,3,6,2},
                      {7,3,2,9} };

    int i,j,k;           //Auxiliares de los ciclos
    int vecAux[m];       //Vector auxiliar para realizar el intercambio
    int ind;             //Auxiliar posición más pequeña
    int aj;              //Auxiliar columna seleccionada

    /*Se establece el auxiliar de la columna en la posición seleccionada
    para la búsqueda */

    aj = 1;

    //Se apertura el ciclo para el proceso de ordenación
    for(i=0; i<(n-1); i++)
    { ind = i;

        //Se guardan en el vector los datos de la fila base
        for(k=0; k<m; k++)
            vecAux[k] = mat[i][k];
        for(j=(i+1); j<n; j++)
        { if(mat[j][aj] < vecAux[aj])
          { ind = j;

              //Se guardan en el vector los elementos de la fila
              //con el elemento más pequeño

              for(k=0; k<m; k++)
                  vecAux[k] = mat[j][k];
          }
        }

        //Se guardan los elementos de la fila i en la fila ind
        for(k=0; k<m; k++)
            mat[ind][k] = mat[i][k];

        //Se guardan los elementos del vector auxiliar en la fila
```

```
        for(k=0; k<m; k++)
            mat[i][k] = vecAux[k];
    }

    //Mostrar matriz ordenada
    system("CLS");
    cout << "La matriz ordenada por la segunda columna es:" << endl
<< endl;
    for(i=0; i<n; i++)
    { cout << "\t";
      for(j=0; j<m; j++)
          cout << mat[i][j] << "\t";
      cout << endl;
    }
    cout << endl;
    system("PAUSE");
}
```

La salida del programa sería:

■

Figura 23. Salida por pantalla. OrdMat.exe

4.2. Ordenación por Burbuja

El método de burbuja o método de intercambio directo, es uno de los más conocidos por su sencillez y facilidad de implementación. La idea básica es comparar elementos consecutivos en cada paso a lo largo del arreglo. Cada vez que se realiza una comparación, los elementos se intercambian entre sí, en caso de no estar en orden. Funciona revisando cada elemento del arreglo con el siguiente, intercambiándolos de posición si están en el orden equivocado. Por tanto, es necesario recorrer varias veces todo el arreglo hasta que no se necesiten más intercambios, lo cual significa que la ordenación está completa.



En este sentido, Joyanes, L. y otros (2007: 220) plantean, que la técnica utilizada se denomina *ordenación por burbuja u ordenación por hundimiento*, debido a que los valores más pequeños “burbujean” gradualmente hacia la cima o parte superior del arreglo, de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del arreglo.

Como se puede observar, este método obtiene su nombre debido a la forma de subir los elementos por el arreglo durante los intercambios, es decir, como si fueran pequeñas “burbujas”. Por otra parte, puesto que solo utiliza comparaciones para operar elementos, se le considera un algoritmo de comparación, siendo el más sencillo de implementar. Los pasos a realizar según Joyanes, L. (2003: 362) en el método son:

1. Compara el primer elemento del arreglo con el segundo, si están en orden, estos se mantienen tal cual, en caso contrario se intercambian entre sí.
2. Luego compara el segundo elemento con el tercero, de nuevo se intercambian si es necesario.
3. El proceso continúa hasta que cada elemento del arreglo ha sido comparado con sus elementos adyacentes y se han realizado los intercambios necesarios.

Si se toma la misma lista de valores del método anterior, es decir:

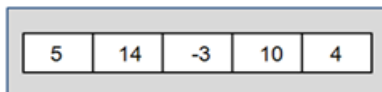


Figura 24. Ordenación. Burbuja 01

La forma de hacer el ordenamiento sería, para el primer recorrido:

1. Compara el primer elemento con el segundo, como el primero es menor, los deja igual (figura 25).

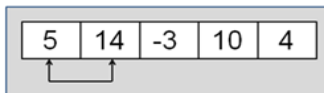


Figura 25. Ordenación. Burbuja 02

2. Luego, compara el segundo elemento con el tercero, como el tercero es menor al segundo, se hace el intercambio (Figura 26).

Figura 26. Ordenación. Burbuja 03

3. Posteriormente, compara el tercer elemento con el cuarto, como el cuarto es menor al tercero, se hace el intercambio (figura 27).

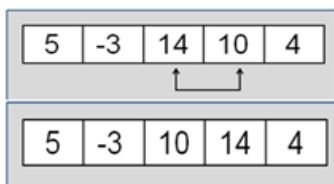


Figura 27. Ordenación. Burbuja 04

4. Por último, compara el cuarto elemento con el quinto, como el quinto es menor al cuarto, se hace el intercambio (figura 28).

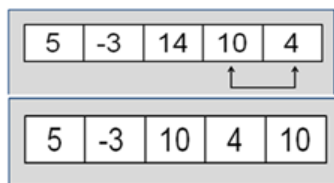


Figura 28. Ordenación. Burbuja 05

Como se puede observar, con un solo recorrido del arreglo no se tiene la ordenación completa, hace falta realizar $n - 1$ recorridos para que el proceso de ordenación esté completo en su totalidad. Por tanto, el algoritmo que se debe realizar para el proceso es:

```
for(i = 0; i < (10-1); i++)
    for(j = 0; j < (10-1); j++)
    { if(lista[j] > lista[j + 1])
        { aux = lista[j];
          lista[j] = lista[j+1];
          lista[j+1] = aux;
        }
    }
}
```




Al igual que el método anterior, en éste también se pueden observar dos ciclos *for*, el primero es simplemente para establecer el número de recorridos que se deben realizar, el segundo *for* es el encargado de realizar los recorridos, obsérvese que dentro de esta sentencia, se establece la comparación de los elementos y se utiliza la variable auxiliar para hacer el intercambio.

Por tanto, si se desea hacer el proceso de ordenación para 10 elementos ingresados por teclado, el código a utilizar sería:

```
#include <iostream>

using namespace std;

main()
{ int i,j,aux,lista[10];
  cout << "INGRESE LOS DATOS:" << endl;

  //Proceso para ingresar los datos al arreglo
  for(i=0; i<10; i++)
  { cout << "INGRESE EL DATO [" << (i+1) << "] = ";
    cin >> lista[i];
  }

  //Proceso de ordenación
  for(i = 0; i < (10-1); i++)
    for(j = 0; j < (10-1); j++)
      { if(lista[j] > lista[j + 1])
        { aux = lista[j];
          lista[j] = lista[j+1];
          lista[j+1] = aux;
        }
      }

  //Proceso para mostrar el arreglo ordenado

  system("CLS");
  cout << "LOS DATOS ORDENADOS SON:" << endl;
  for(i=0; i<10; i++)
    cout << "ELEMENTO[" << (i+1) << "] = " << lista[i] << endl;
  system("PAUSE");
}
```

Ahora bien, si la ordenación se quiere realizar en la misma matriz del método anterior, tomando como base la misma columna (figura 29):

5	8	3	7
2	7	1	3
9	3	6	2
7	3	2	9

Figura 29. Ordenación. Burbuja Matriz

El código que se aplicaría sería:

```
#include <iostream>
//FORMA DE DEFINIR CONSTANTE
#define n 4
#define m 4

using namespace std;

main()
{
    int mat[n][m] = { {5,8,3,7},
                      {2,7,1,3},
                      {9,3,6,2},
                      {7,3,2,9} };

    int i,j,k;           //Auxiliares de los ciclos
    int vecAux[m];       //Vector auxiliar para realizar el intercambio
    int aj;              //Auxiliar columna seleccionada

    /*Se establece el auxiliar de la columna en la posición seleccionada
    para la búsqueda */

    aj = 1;

    //Se realiza el proceso de ordenación.

    for(i=0; i<(n-1); i++)
        for(j=0; j<(n-1); j++)
            if(mat[j][aj] > mat[j+1][aj])
```



```
{
    //Se pasa los elementos de la fila j al arreglo auxiliar

    for(k=0; k<m; k++)
        vecAux[k] = mat[j][k];

    //Se pasa los elementos de la fila j+1 a la fila j

    for(k=0; k<m; k++)
        mat[j][k] = mat[j+1][k];

    //Se pasa los elementos del arreglo auxiliar a la fila j+1

    for(k=0; k<m; k++)
        mat[j+1][k] = vecAux[k];
}

//Mostrar matriz ordenada
system("CLS");
cout << "La matriz ordenada por la segunda columna es:" << endl
<< endl;
for(i=0; i<n; i++)
{ cout << "\t";
  for(j=0; j<m; j++)
    cout << mat[i][j] << "\t";
  cout << endl;
}
cout << endl;
system("PAUSE");
```


ACTIVIDADES DE AUTOEVALUACIÓN



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

Nombre: _____

Cédula: _____ Sección: _____

1. Elabore el código en C++ para resolver los siguientes enunciados:
 - a. Dada una lista de N elementos, determinar la suma de los números pares y los números impares de forma independiente.
 - b. Se tienen N empleados de una compañía, en un arreglo unidimensional llamado "Suel" se tienen almacenados los sueldos de los empleados, en un arreglo llamado "Asig" las asignaciones totales de cada empleado y un arreglo llamado "Dedu" las deducciones de cada uno. Crear un arreglo llamado "Total" que contenga el neto a pagar a cada empleado.
2. Empleando una lista de N números, elaborar un programa que determine cuántos y cuáles de ellos son múltiplos de un número X ingresado por teclado.
3. Se tienen los promedios de N alumnos de una clase, hallar el promedio de la sección, número de alumnos aprobados y reprobados, alumnos que están por encima del promedio e indicar cuántos alumnos se encuentran eximidos (para eximir, la nota debe ser ≥ 18).
4. Utilizando una matriz cuadrada de orden N, elabore un programa que transforme los términos de dicha matriz de acuerdo a las siguientes reglas: los elementos de la diagonal principal se elevan al cuadrado, los restantes elementos se multiplicarán por un escalar ingresado por el usuario y al resultado se le deberá sumar 4.
5. Diseñe un programa que determine la transpuesta de una matriz.
6. Desarrollar un algoritmo que cumpla con lo siguiente:
 - a. Dada una matriz cuadrada de $N \times N$
 - b. Hallar el vector columna cuyos elementos están constituidos por la suma de los elementos de cada fila de la matriz.
7. En un almacén X, se venden tres tipos de producto, el producto A tiene un precio establecido de 1380 Bs., el precio del producto B es de 1450 Bs., y el precio del producto C es de 1410 Bs. Teniendo en cuenta el supuesto de que las cantidades vendidas de cada uno de ellos se suministran mediante una tabla, desarrollar un programa para determinar: el ingreso diario y el ingreso total durante una semana.



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

UNIDAD III

FUNCIONES EN LENGUAJE C



UNIDAD III FUNCIONES

1. FUNCIONES: DEFINICIÓN

Las funciones, según Bustamente, P. y otros (2004: 46), son una parte de código independiente del programa principal y de otras funciones, que pueden ser llamadas enviándole unos datos o no para que realice una determinada tarea y/o proporcione unos resultados. Por su parte, Osorio, R. (2006: 56) establece que una función es un bloque de instrucciones a las que se le asigna un nombre. Entonces, cada vez que se necesite la misma será invocada. En el mismo orden de ideas, Corona, M. y Ancona, M. (2011: 173) indican que se trata de un subprograma que realiza una tarea específica que puede o no recibir valores.

Tomando como base las consideraciones expuestas por los autores citados, podemos inferir que las funciones son bloques de códigos utilizados para dividir un programa en partes más pequeñas, cada una de las cuales tendrá una tarea determinada, ésta se ejecutará cada vez que sea llamada a través de su nombre. Atendiendo a este hecho, el programador puede distribuir la solución de un problema utilizando en un mismo programa funciones que permiten: la entrada de los datos, proceso(s) y salida de la información. En síntesis, para construir una función, en primer lugar debe realizarse la definición o declaración cumpliendo con las normas de sintaxis que el compilador del lenguaje exige.

Su **sintaxis** es:

tipo _ función nombre _ función (tipo de dato nombre de argumentos)

```
{  
  declaración de variables locales  
  bloque de sentencias o cuerpo de la función  
  return  
}
```

De la sintaxis anterior, se tiene lo siguiente:

tipo _ función: puede ser de cualquier tipo de dato conocido. El valor devuelto por la función será de este tipo. En otras palabras, si la función es desarrollada para la suma de un número entero más un número real, el resultado de esta operación es un número con decimales. Por lo tanto, el tipo de dato que retornará la función será de tipo float (número real). Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor

de tipo entero (**int**). Si no se necesita que retorne ningún valor se debe indicar el tipo vacío (**void**).

- *Variables locales*: son variables que se declaran dentro de la función y se utilizan en ésta, no pueden salir de las mismas porque su valor se destruye. Son opcionales, el programador decide si las usa o no.
- *Nombre _ función*: es el nombre con el que se denomina a la función.

Tipo y nombre de argumentos: son los parámetros que reciben la función. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se transfiere a la función al hacer la llamada a la misma. Pueden existir funciones que no reciban argumentos. Entonces son llamadas funciones sin parámetros.

Bloque de sentencias: es el conjunto de instrucciones que serán ejecutadas cuando se realice la llamada a la función. Finalmente, en este bloque el programador coloca todo lo que se necesita para resolver el problema, es decir, fórmulas, condiciones, ciclos, entre otros. También se le conoce como el cuerpo de la función.

Es importante señalar, que las funciones pueden ser llamadas desde la función **main()** o desde otras funciones. Nunca se debe llamar a la función **main()** desde otro lugar del programa. No obstante, es necesario recalcar, que los argumentos o parámetros de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

return: **permite** devolver a la función **main()** el valor generado por la función. Es una sentencia opcional, pueden crearse funciones en donde no se utilice.

2. PROTOTIPO DE UNA FUNCIÓN

Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Corona, M. y Ancona, M. (2011: 174) indican que el prototipo es el encabezado de una función, es decir, la primera línea de la función. Para que un programa en C/C++ sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones. Estos pueden escribirse antes de la función **main** o bien en otro archivo. En este último caso, se le indica al compilador mediante la directiva **#include**.

Para Corona, M. y Ancona, M. (2011: 174), la ventaja de utilizar prototipos es que las funciones pueden estar en cualquier lugar y en cualquier orden, y pueden ser llamadas desde cualquier punto del programa principal o de otra función. Asimismo, Deitel, P. y Deitel, H. (2009: 211) establecen que



un prototipo de una función es también conocido como declaración de una función e indica al compilador el nombre de una función, el tipo de dato devuelto por la función, el número de parámetros que la función espera recibir, los tipos de esos parámetros y el orden en el que estos se esperan.

En síntesis, las funciones corresponden a bloques de códigos que realizan una tarea específica, y ésta es llamada o invocada a resolver el problema para el cual fue diseñada, permitiendo obtener una mejor legibilidad y facilidad de cambio en el programa general.

2.1. Tiempo de vida de los datos o tipos de variables

Al igual que en el `main()`, en las funciones secundarias o auxiliares se pueden declarar variables (declaración local), sin embargo, las funciones también pueden utilizar variables que se hayan declarado de manera global. En este sentido, López, G. y otros (2009: 62) plantean, que las declaraciones locales pueden emplearse en cualquier punto dentro de la función, su existencia y alcance están limitados por la función que la contiene, mientras que, las declaraciones globales se realizan fuera de cualquier función y su alcance es de todas las funciones del programa. Considerando lo expuesto, es importante explicar la diferencia entre una declaración global y una local según Joyanes, L. (2003: 220).

Globales: las variables globales son las que están declaradas para el programa o algoritmo principal, del que dependen todos los subprogramas. En este sentido, las variables globales permanecen activas durante todo el programa. Se crean al iniciarse éste y se destruyen de la memoria, al finalizar. Pueden ser utilizadas en cualquier función, por lo que, las variables declaradas normalmente antes de definir la función `main()` son accesibles por todas las funciones, es decir, son de ámbito general.

Locales: estas variables están declaradas y definidas dentro de un subprograma, por lo tanto, es importante resaltar que las variables declaradas dentro de una función, solo serán accesibles desde la función donde fue creada. Para Joyanes, L. (2000: 139) se deben tomar en cuenta las siguientes reglas para las variables locales:

- En el interior de una función, a menos que explícitamente cambie un valor de una variable, no se puede cambiar esa variable por ninguna sentencia externa a la función.
- Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir la misma variable. Cada variable es distinta y pertenece a su función específica.
- Las variables locales de las funciones no existen en memoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden

compartir la misma memoria para sus variables locales (pero no al mismo tiempo).

Si dos variables, una global y una local, tienen el mismo nombre, la local prevalecerá sobre la global dentro de la función en la cual ha sido declarada.

Dos variables locales pueden tener el mismo nombre siempre que estén declaradas en funciones diferentes.

A continuación se presenta un ejemplo, donde se utiliza una variable global y una local:

```
/* Variables globales y locales. */  
#include <cstdlib>  
#include <iostream>  
int num1=1;  
using namespace std ;  
main()  
{  
    int num2=10;  
    cout<<endl ;  
    cout<<num1<< endl;  
    cout<<num2<< endl;  
    system("Pause");  
}
```

Al ejecutar este programa se generará lo siguiente:

1

10

Presione una tecla para continuar. . .

Asimismo, se presenta otro ejemplo, donde se ilustra el uso de una variable local y global:

```
#include <iostream>  
#include <cstdlib>  
using namespace std ;  
  
int variable_global=13;  
int main(int)  
{  
    int variable_local=20;  
    cout<<"\nCodigo Fuente que muestra los usos de variables "\n"  
    "Globales y Locales en Lenguaje C++\n"<<endl;
```



```
cout<<"La Variable Global tiene asignado un: "\
<<variable_global<<endl;
cout<<"\nLa Variable Local tiene asignado un: "\
<<variable_local<<endl;

system("Pause") ;
}
```

Los resultados que genera este programa al ejecutarse son :

Código Fuente que muestra los usos de variables Globales y Locales en Lenguaje C++

```
La Variable Global tiene asignado un: 13
La Variable Local tiene asignado un: 20
```

Presione una tecla para continuar. . .

En el siguiente ejemplo, se puede observar la declaración de una función (prototipo). Al no recibir ni retornar ningún valor, está declarada como *void* en ambos lados. También podemos observar que existe una *variable global* llamada *num*. Esta variable es reconocible en todas las funciones del programa. Ya en la función *main()* se encuentra una *variable local* llamada *num*. Al ser una variable local, ésta tendrá preferencia sobre la global. Por tanto, la función escribirá los números 10 y 5.

```
/* Declaración de funciones. */
#include <cstdlib>
#include <iostream>
using namespace std ;
void funcion(void); /* prototipo */
int num=5; /*Variable Global*/
main()
{
int num=10; /*Variable Local*/
cout<<"\nLa Variable tiene asignado un: "\
<<num<<endl;
funcion(); /* llamada */
system("Pause") ;
}
void funcion(void)
{
cout<<"\nLa Variable tiene asignado un: " <<num<<endl;
}
```

3. PARÁMETROS

Los parámetros se conocen también como argumentos de funciones y no es más que la o las diferentes variables que se pasan a una función cuando es llamada. Joyanes, L. (2003: 223) puntualiza: “cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. Se comportan como otras variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir”.

Basado en lo anterior, podemos decir que los parámetros son el medio a partir del cual podemos difundir el perímetro de la o las variables locales de una función a otra, además, son quienes nos permiten establecer comunicaciones entre las funciones.

3.1. Paso de parámetros a una función

En C/C++ las funciones pueden o no recibir parámetros, cuando la función no recibe parámetros, ésta puede trabajar con las variables globales previamente declaradas o las variables locales declaradas dentro de la función. Ahora bien, cuando al momento de llamar a la función ésta recibe una o varias variables, es cuando se está haciendo un pase de parámetros. En este sentido, Corona, M. y Ancona, M. (2011: 189) indican, que en cada llamada a la función se genera una copia de los valores de los parámetros actuales, los cuales se almacenan en variables temporales en la pila mientras dure la ejecución de la función. De igual manera, las funciones pueden retornar un valor. Esto se hace mediante la instrucción *return*, que finaliza la ejecución de la función, devolviendo o no un valor. La sintaxis es:

`return (valor o expresión);`

El valor devuelto por la función debe asignarse a una variable. De lo contrario, el valor se perderá. Seguidamente, los parámetros o argumentos de una función no son más que variables locales que reciben un valor. Este valor es enviado a la misma al llamar a la función. Pueden existir funciones que no reciban argumentos. De esta manera, tomando en consideración lo antes expuesto, el siguiente ejemplo ilustra el paso por parámetros:

```
//Paso por parámetros
# include <cstdlib>
# include <iostream>
using namespace std ;
int suma(int,int);//Declaración de la función prototipo
```



```

main()
{
    int a,b,result; // Declaración de Variables Locales
    cout<<"Primer valor : " <<endl; //Ingreso de los valores en las
variables a y b
    cin>>a;
    cout<<"Segundo valor : " <<endl;
    cin>>b;
    //Se llama la función, y el resultado que retorna la función se guarda
en result
    result= suma(a,b) ;
    //Salida por pantalla de la suma de dos valores
    cout<<"La suma de los dos números es : " <<endl;
    cout<<result <<endl; //
    system("Pause");
}
int suma(int a,int b) //Definición de la función
{
    return(a+b) ;//Cuerpo de la función
}

```

En primer término, este programa se compone de dos funciones. La función `main()` y la función creada por el usuario llamada `suma()` que utiliza dos parámetros de tipo entero. En segundo término, la función prototipo construida por el usuario calcula la suma de dos números y devuelve el resultado mediante la sentencia `return`. Por lo tanto, existen dos formas de enviar parámetros a una función: por valor y por referencia.

- Por valor:

Significa que una copia del argumento se pasa al parámetro de la función. En tal sentido, todo lo que se realice en el código de la función no va a alterar el valor del argumento usado para llamar a la función. Como resultado de esto, cualquier cambio que se realice dentro de la función en el argumento enviado, **NO** afectará al valor original de las variables utilizadas en la llamada. Es como trabajar con una copia, no con el original. No es posible enviar por valor **arreglos**, debe hacerse por referencia.

Por ejemplo:

```

#include <iostream>
#include <cstdlib>
using namespace std ;
double AreaCírculo (double); //declaración
int main(int)

```

```
{
double r=10.0;
double a;
a = AreaCirculo(r); //llamada a la función
cout <<"El área del círculo de radio "<< endl ;
cout << a << endl;
system("Pause");
}
//definición de la función
double AreaCirculo( double radio)
{
radio = 3.14 * radio * radio;
return radio;
}
```

Primeramente, se declara la función llamada `ÁreaCirculo()` con un parámetro. Seguidamente, la función `main ()` presenta la declaración de una variable `r` que se inicializa en 10.0, también puede ingresarse por teclado. Por otra parte, la variable `a` es declarada para almacenar lo que retorna la función cuando es llamada a través de la sintaxis `a= AreaCirculo(r)`. Finalmente, la variable `r` le copia el valor al parámetro `radio` (paso de argumento por valor) para que la fórmula que corresponde al cuerpo de la misma pueda resolverse y retornar el `radio` a la función `main()` en donde se copia en la variable `a`.

- Por referencia:

Consiste esta fase en enviar a la función, la dirección de memoria donde se encuentra la variable o dato. Cualquier modificación **SI** afectará a las variables utilizadas en la llamada.

Por ejemplo:

```
#include <iostream>
#include <cstdlib>
using namespace std ;
void Cambio( int &a, int &b);
int main(int)
{
int x=25, y=35;
Cambio(x,y);
cout << x <<endl;
cout << y <<endl;
system("pause");
```




```
//return Exit_SUCCESS;
}  
void Cambio( int &a, int &b)  
{  
    int temporal = a;  
    a = b;  
    b = temporal;  
}
```

La salida de este programa es:

35

25

Cabe destacar, que para enviar un valor por referencia se utiliza el símbolo **&** (ampersand) delante de la variable enviada. Esto le indica al compilador que la función que se ejecutará tendrá que obtener la dirección de memoria donde se encuentra la variable. Es necesario recalcar, que esta forma de pasar argumentos mediante las direcciones en lugar de valores, debe utilizarse cuando la función deba devolver parámetros modificados. Un caso de particular interés es el paso de arreglos (vectores, matrices y cadenas de caracteres).

En el siguiente ejemplo, podemos observar nuevamente como las variables intercambian su valor luego de la llamada de la función (*paso por referencia*).

```
# include <cstdlib>  
# include <iostream>  
  
using namespace std ;  
void intercambio(int &primero,int &segundo);  
  
int main(int argc,char * argv[])  
{  
    int a,b;  
    cout<<"Primer valor : " <<endl;  
    cin>>a;  
    cout<<"Segundo valor : " <<endl;  
    cin>>b;  
  
    cout<<"El orden original es : " <<endl;  
    cout<<a <<endl;  
    cout<<b <<endl;  
  
    //Función Intercambio  
    intercambio(a,b);
```

```
    cout<<endl;

    cout<<"Los valores Invertidos son :" <<endl;
    cout<<a <<endl;
    cout<<b <<endl;
        system("Pause");
    // return Exit_SUCCESS ;
}
void intercambio(int &primero,int &segundo)
{
    int tercero;
    tercero=primero;
    primero=segundo;
    segundo=tercero;
    return;
}
```

Las variables con un * son conocidas como *punteros*, el único dato en 'C' que puede almacenar una dirección de memoria.

```
# include<cstdlib>
#include <iostream>
using namespace std ;
void intercambio(int *,int *);
int main(int) /* Intercambio de valores */
{
    int a=20,b=13;
    cout<<" NÚMEROS EN ORDEN ORIGINAL"<<endl;
    cout << "A: " << a << endl;
    cout << "B: " << b << endl;
    intercambio(&a,&b); /* llamada */
    cout<<" NÚMEROS CAMBIADOS"<<endl;
    cout << "A: " << a << endl;
    cout << "B: " << b << endl;

    getchar();
}
void intercambio (int *x,int *y)
{
    int aux;
    aux=*x;
    *x=*y;
    *y=aux;
    //printf(" MUESTRO NÚMEROS INTERCAMBIADOS \n");
}
```



```
//printf("a=%d y b=%d",*x,*y);  
}
```

En efecto, la salida generada será:

NÚMEROS EN ORDEN ORIGINAL

A: 20

B: 13

NÚMEROS CAMBIADOS

A: 13

B: 20

Por otra parte, un arreglo puede pasarse como argumento a una función. Para pasar el arreglo a la función el nombre de éste deberá aparecer solo, sin corchetes ni índices, como argumento real en la llamada a la función. El correspondiente argumento formal o parámetro por valor tiene que ser declarado como un arreglo en la declaración de los parámetros. Para lograr declarar un vector o arreglo unidimensional, se escribirá con un par de corchetes vacíos. El tamaño o dimensión no se especifica en la declaración de los argumentos formales.

Finalmente, este código fuente determina el promedio de tres notas e ilustra lo anteriormente expuesto:

```
#include <iostream>  
#include <cstdlib> //para rand()  
using namespace std;  
float promedio(float x[]); //Declaración  
int main(int)  
{  
    float prome;  
    float lista[3];  
    cout<<"Ingrese tres notas :"<<endl;  
    for(int i=0;i<3;i++)  
        cin>>lista[i];  
    prome= promedio(lista);  
    cout<<"El Promedio es :"<<prome<<endl;  
    system("pause") ;  
}  
float promedio(float x[]) //Declaración  
{  
    float suma=0;  
    for(int i=0;i<3;i++)  
        suma=suma+ x[i];  
}
```

```
    return(suma/3);  
}
```

Como resultado, se muestra:

Ingrese tres notas :

15

16

12

El Promedio es: 14.3333

Presione una tecla para continuar . . .

Como complemento se presenta otro ejemplo del uso de parámetros tipo arreglo en una función.

```
#include <iostream>  
#include <cstdlib> //para rand()  
using namespace std;  
void ImprimeDatos(float*, int); //Declaración  
int main(int)  
{  
    int Num=10;  
    float *dat;  
    dat = new float[Num];  
    for (int i=0;i<Num;i++) dat[i] = (float)rand()/RAND_MAX;  
    ImprimeDatos(dat,Num);  
    system("Pause");  
}  
//void ImprimeDatos( float datos[], int Num )  
void ImprimeDatos( float* datos, int Num ) //Definición  
{  
    int i ;  
    for (i=0;i<Num;i++)  
        cout<<datos[i]  
        <<endl ;  
}
```

En efecto, la salida que genera este programa es:

```
0.00125126  
0.563585  
0.193304  
0.80874  
0.585009  
0.479873  
0.350291
```



0.895962

0.82284

0.746605

Presione una tecla para continuar . . .

3.2. Los argumentos de la función main()

Como cualquier función, la función principal (main) también puede recibir parámetros, seguidamente, Osorio, A. (2006: 58) explica tanto su sintaxis como lo que significa cada parámetro:

Sintaxis: `int main (int argc, char *argv[], char *envp[])`

- El parámetro *argc*, es un entero que contiene el número de argumentos pasados a la función desde la línea de comandos. Es decir, almacena el número de argumentos en la tabla *argv*.
- El parámetro *argv*, es una tabla de cadenas de caracteres (se verá más adelante), cada cadena de caracteres contiene un argumento pasado en la línea de comandos, la primera cadena contiene el nombre con el que fue invocado y las siguientes son los demás argumentos.
- El parámetro *envp*, es una tabla de cadenas, que pasa información sobre una variable de entorno del sistema operativo.

En este ejemplo se observa un programa que escribirá un saludo por pantalla.

```
/* Argumentos de la función main(). */
#include <cstdlib>
#include <iostream>
using namespace std ;
main(int argc,char *argv[]) /* argumentos */
{
    cout << "Lenguaje C" << endl;
    cout << "Programa Ejemplo" << endl;

    if (argc<2)
    {
        cout << "Función main() con argumentos" << endl;
        getchar() ;
    }
    cout << "Hola" << endl;
    system("pause") ;
}
```

4. SOBRECARGA A FUNCIONES

Muchos lenguajes de programación, permiten la sobrecarga de métodos o funciones, éste se refiere a que dos funciones puedan tener el mismo nombre siempre y cuando reciba parámetros diferentes. En este sentido, Joyanes, L. (2000: 582) señala que las funciones sobrecargadas se diferencian en el número y tipo de argumentos, o en el tipo que devuelven las funciones, y su cuerpo son diferentes en cada una de ellas. Por Ejemplo:

```
# include<cstdlib>
#include <iostream>
using namespace std ;
void función1(int a);
void función1(double a);
int main(int)
{
double a=19.50;
int b=8;
función1(a);
función1(b);
system("pause");
}
void función1(int a)
{
cout << "Valor entero: " << a << endl;
}
void función1(double a)
{
cout << "Valor real: " << a << endl;
}
```

Por lo tanto, lo que se muestra por pantalla es:

Valor real: 19.5

Valor entero: 8

Presione una tecla para continuar. . .

5. FUNCIONES DE BIBLIOTECAS

En la programación en C/C++ es posible utilizar funciones que no estén incluidas en el propio programa, y esto permite la inclusión de archivos. Para ello se emplea la directiva *#include*, que permite añadir librerías o funciones que se encuentran en otros archivos al programa, bien sea de la biblioteca estándar del C/C++ (ver la Tabla 7 en la primera unidad) o



previamente elaboradas por el programador. En este sentido, Joyanes, L. (2000: 141) indica que se pueden incluir tantos archivos de cabecera como sean necesarios, incluyendo los suyos propios que definen sus funciones.

Para indicar al compilador que van a incluirse archivos externos, se procede de dos maneras (siempre antes de las declaraciones):

1. Indicándole al compilador la ruta donde se encuentra el archivo.

```
#include "misfunc.h"  
#include "c:\includes\misfunc.h"
```

2. Indicando que se encuentran en el directorio por defecto del compilador.

```
#include <misfunc.h>
```

A continuación se darán algunas funciones más utilizadas junto con los archivos de cabecera donde se encuentran ubicadas (tomado de diferentes autores: Bustamante y otros 2004, Deitel y otro 2003, Joyanes L. y otros 2007 y Osorio A. 2006).

- Librería stdio.h (C) o cstdio (C++)

`printf()`

Función: escribe en la salida estándar con formato.

Sintaxis: `printf(formato , arg1 , ...);`

`scanf()`

Función: lee de la salida estándar con formato.

Sintaxis: `scanf(formato , arg1 , ...);`

`puts()`

Función: escribe una cadena y salto de línea.

Sintaxis: `puts(cadena);`

`gets()`

Función: lee y guarda una cadena introducida por teclado.

Sintaxis: `gets(cadena);`

`fopen()`

Función: abre un archivo en el modo indicado.

Sintaxis: `pf=fopen(archivo , modo);`

`fclose()`

Función: cierra un archivo cuyo puntero se le indica.

Sintaxis: `fclose(pf);`



`fprintf()`

Función: escribe con formato en un archivo.

Sintaxis: `fprintf(pf , formato , arg1 , ...);`

`fgets()`

Función: lee una cadena de un archivo.

Sintaxis: `fgets(cadena , longitud , pf);`

- Librería `stdlib.h`

`atof()`

Función: convierte una cadena de texto en un valor de tipo float.

Sintaxis: `numflo=atof(cadena);`

`atoi()`

Función: convierte una cadena de texto en un valor de tipo entero.

Sintaxis: `nument=atoi(cadena);`

`itoa()`

Función: convierte un valor numérico entero en una cadena de texto.

La base generalmente será 10, aunque se puede indicar otra distinta.

Sintaxis: `itoa(número , cadena , base);`

`exit()`

Función: termina la ejecución y abandona el programa.

Sintaxis: `exit(estado); /* Normalmente el estado será 0 */`

- Librería `conio.h`

`clrscr()`

Función: borra la pantalla.

Sintaxis: `clrscr();`

`clreol()`

Función: borra desde la posición del cursor hasta el final de la línea.

Sintaxis: `clreol();`

`gotoxy()`

Función: cambia la posición del cursor a las coordenadas indicadas.

Sintaxis: `gotoxy(columna , fila);`

`textcolor()`

Función: selecciona el color de texto (0 - 15).

Sintaxis: `textcolor(color);`

`textbackground()`

Función: selecciona el color de fondo (0 - 7).

Sintaxis: `textbackground(color);`



`wherex()`

Función: retorna la columna en la que se encuentra el cursor.

Sintaxis: `col=wherex();`

`wherey()`

Función: retorna la fila en la que se encuentra el cursor.

Sintaxis: `fila=wherey();`

`getch()`

Función: lee y retorna un único carácter introducido mediante el teclado por el usuario. No muestra el carácter por la pantalla.

Sintaxis: `letra=getch();`

`getche()`

Función: lee y retorna un único carácter introducido mediante el teclado por el usuario. Muestra el carácter por la pantalla.

Sintaxis: `letra=getche();`

- Librería `string.h`

`strlen()`

Función: calcula la longitud de una cadena.

Sintaxis: `longitud=strlen(cadena);`

`strcpy()`

Función: copia el contenido de una cadena sobre otra.

Sintaxis: `strcpy(copia , original);`

`strcat()`

Función: concatena dos cadenas.

Sintaxis: `strcat(cadena1 , cadena2);`

`strcmp()`

Función: compara el contenido de dos cadenas. Si `cadena1 < cadena2` retorna un número negativo. Si `cadena1 > cadena2`, un número positivo, y si `cadena1` es igual que `cadena2` retorna 0 (o NULL).

Sintaxis: `valor=strcmp(cadena1 , cadena2);`

- Librería `graphics.h`

Además de las que se utilizaron al explicar la programación gráfica existen otras funciones. A continuación se mencionan algunas de ellas.

`getmaxcolor()`

Función: retorna el valor más alto de color disponible.

Sintaxis: `mcolor=getmaxcolor();`

setactivepage()

Función: en modos de video con varias páginas, selecciona la que recibirá todas las operaciones y dibujos que se realizan.

Sintaxis: setactivepage(página); /* En modo VGA página = 0 ó 1 */

setvisualpage()

Función: en modos de video con varias páginas, selecciona la que se visualizará por pantalla.

Sintaxis: setvisualpage(página);

- Librería dir.h

En esta librería se encuentran una serie de rutinas que permitirán realizar operaciones básicas con directorios y unidades de disco.

chdir()

Función: cambia el directorio actual.

Sintaxis: chdir(ruta); /* Podemos indicar la unidad: chdir("a:\\DATOS");

*/

getcwd()

Función: lee del sistema el nombre del directorio de trabajo.

Sintaxis: getcwd(directorio,tamañocad) /* Lee el directorio y la unidad

*/

getdisk()

Función: lee del sistema la unidad actual.

Sintaxis: disk=getdisk() + 'A'; /* Retorna un entero: 0 = A: , 1 = B:

... */

mkdir()

Función: crea un directorio.

Sintaxis: mkdir(nombre);

fflush(stdin)

Función: limpia el buffer de teclado.

Sintaxis: fflush(stdin);

Prototipo: stdio.h

sizeof()

Función: operador que retorna el tamaño en bytes de una variable.

Sintaxis: tamaño=sizeof(variable);

cprintf()

Función: funciona como el printf pero escribe en el color que se haya activado con la función textcolor sobre el color activado con textbackground.

Sintaxis: cprintf(formato , arg1 , ...);

Prototipo: conio.h



`kbhit()`

Función: espera la pulsación de una tecla para continuar la ejecución.

Sintaxis: `while (!kbhit()) /* Mientras no pulsemos una tecla... */`

Prototipo: `conio.h`

`random()`

Función: retorna un valor aleatorio entre 0 y `num-1`.

Sintaxis: `valor=random(num); /* También necesitamos randomize */`

Prototipo: `stdlib.h`

`randomize()`

Función: inicializa el generador de números aleatorios. Debe ser llamado al inicio de la función donde se utilice el `random`. También se debe utilizar el `include time.h`, ya que `randomize` hace una llamada a la función `time`, incluida en este último archivo.

Sintaxis: `randomize();`

Prototipo: `stdio.h`

`System()`

Función: ejecuta el comando indicado. Esto incluye tanto los comandos del sistema operativo, como cualquier programa que se le indique. Al acabar la ejecución del comando, volverá a la línea de código situada a continuación de la sentencia `system`.

Sintaxis: `system(comando); /* p.ej: system("arj a programa"); */`

Prototipo: `stdlib.h`

En programación existe el enfoque de *divide y vencerás*, y esto permite que el programa sea más fácil de manipular y de corregir. Además, la reutilización de código o reutilización del software que es el uso de funciones existentes, como bloques constructivos para crear nuevos programas son fundamentos de la programación actual. Por lo tanto, una función permite dividir un programa en varios módulos, y cada uno de estos realizará una tarea específica. En tal sentido, el programador podrá elaborar programas más eficientes, así como también, evitar repetir código.

En programas que contengan muchas funciones, la estructura de la función deberá ser organizada como un grupo de llamadas a funciones para que se ejecuten y realicen la tarea para la cual fueron diseñadas. Además, se pueden utilizar funciones de biblioteca del lenguaje que simplifican una diversidad de tareas al programador.

ACTIVIDADES DE AUTOEVALUACIÓN



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

Nombre: _____

Cédula: _____ Sección: _____

1. Realice una función en C++ que permita calcular el factorial de número.
2. Suponiendo que no se tenga habilitada la función pow() para el cálculo del exponencial de un número, realice usted una función que permita calcular N elevado a la M utilizando multiplicaciones sucesivas.
3. Elabore en C++ los siguientes ejercicios utilizando funciones:
 - a. Dado un vector X compuesto de N elementos, calcular la desviación estándar aplicando la siguiente fórmula:

$$C = \frac{\sum_{i=1}^N (X_i - \text{MedX})^2}{N - 1}$$

$$\text{MedX} = \frac{X_1 + X_2 + X_3 + \dots + X_N}{N}$$

- b. Dados dos arreglos unidimensionales (A y B) de N elementos cada uno. Calcular:

$$C = \frac{\sum_{i=1}^N (A_i - B_i)^2}{N}$$

- c. En un campeonato de ciclismo infantil, se tienen 6 equipos. Cada uno de ellos, participan en 6 rondas de juegos, los puntajes obtenidos se observan en la tabla 14. Se desea saber, cuál fue el puntaje total obtenido por cada equipo y cuál fue el equipo ganador.

Tabla 14
Datos del Ejercicio 3 - c

EQUIPO	R_1	R_2	R_3	R_4	R_5	R_6
A	0	1	3	1	1	1
B	1	1	0	1	1	1
C	3	3	3	1	1	1
D	0	0	0	0	3	1
E	3	3	3	1	0	1
F	1	0	0	3	1	1

- d. Las inscripciones del congreso de informática a realizarse en el mes de julio, fueron durante el mes de mayo. En el proceso se generó una tabla de datos de 5 filas (días de la semana laboral) y 4 columnas (semanas que duró el proceso de inscripción) en donde se guardó el número de inscritos en cada día de la semana. Al finalizar este proceso, los organizadores del evento desean saber:
- El número de personas inscritas en el mes.
 - El número de personas inscritas en cada una de las semanas del mes.
 - Y por último, desean determinar cuál de los días de inscripción resultó más efectivo, es decir, reportó mayor cantidad de inscritos.
- e. La empresa "VIVO", se dedica a realizar ventas a domicilio, para ello cuenta con N vendedores repartidos entre diferentes zonas de trabajo en la ciudad. La zona norte tiene un total de 5 vendedores y al final del año pasado, la cantidad de productos vendidos por cada uno de los vendedores en cada mes se pueden ver en la tabla 15. Elaborar un programa que permita calcular: el total de las ventas realizadas en el año por cada uno de los vendedores, y el total de ventas realizadas en la zona norte. Por otra parte, al final se desea ver la tabla ordenada descendientemente por las ventas totales de cada vendedor.



Tabla 15
Datos del Ejercicio 3 - e

N_V	ENE	FEB	MAR	ABR	MAY	JUN	JUL	AGO	SEP	OCT	NOV	DIC
1	120	130	198	138	102	106	102	106	198	152	106	168
2	132	172	100	196	168	195	143	147	105	152	187	201
3	142	168	165	187	179	176	106	153	144	164	146	245
4	102	149	106	105	106	168	184	174	165	109	102	204
5	132	145	197	136	146	102	130	100	165	132	185	201



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSO CHACÍN

UNIDAD IV

MANEJO DE CADENAS



UNIDAD IV MANEJO DE CADENAS

1. DEFINICIÓN

Las variables que pueden almacenar valores no numéricos más grandes que un carácter sencillo son conocidas como cadenas. Una cadena de caracteres (string) es un conjunto de caracteres (incluido el blanco) que se almacenan en localidades contiguas de memoria. Bustamante, P. y otros (2004: 19) definen las cadenas de carácter como una secuencia de caracteres delimitados por comillas, dentro de ellas pueden aparecer espacios en blanco y donde el compilador siempre sitúa un byte nulo (\0) al final de cada cadena para señalar el final de ésta. Las mismas son representadas como un vector de caracteres, donde cada elemento del vector representa un carácter de la cadena, en este sentido, Joyanes, L. (2000: 229) confirma, que una cadena es un tipo de dato compuesto, un arreglo de caracteres (char), terminado por un carácter nulo (' \0 '), NULL (ver figura 30).

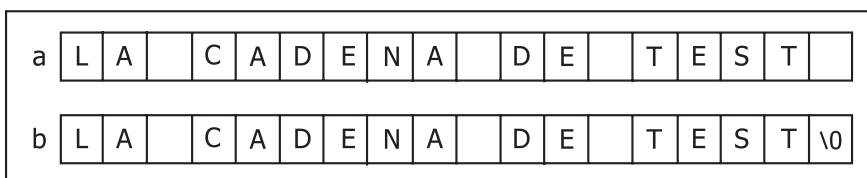


Figura 30. Manejo de Cadenas.

Leyenda: a.- Arreglo de Cadena / b.- Cadena de Caracteres.

Fuente: Joyanes, L. (2000)

Las librerías del lenguaje C++ proveen soporte para el manejo de cadenas a través de la clase estándar String. Éste no es un tipo de dato fundamental, pero se comporta de manera similar en su uso básico. Una primera diferencia con los tipos de datos básicos es que para declarar y usar objetos (variables) de este tipo, se requiere incluir un encabezado de archivo adicional en el código fuente <string> y tener acceso al espacio de nombres std.

```
// Mi primer cadena
#include <iostream>
#include <string>
using namespace std;
```

```
int main ()
{
    string micadena = "Esto es una cadena";
    cout << micadena;
    return 0;
}
```

Como se puede ver en el ejemplo previo, las variables de este tipo pueden ser inicializadas con cualquier literal alfanumérico válido, de manera similar a como las variables de tipo numérico pueden ser inicializadas con un literal numérico. Los siguientes ejemplos representan alternativas válidas de inicialización:

```
string micadena = " Esto es una cadena";
string micadena ("Esto es una cadena");
```

Las cadenas también pueden ejecutar todas las operaciones básicas que realizan los otros tipos de datos fundamentales, como ser declaradas sin un valor inicial y luego recibir valores durante la ejecución del programa:

```
// mi primer cadena
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string MC;
    MC = "TEXTO INICIAL DE LA CADENA DE CARACTER";
    cout << MC << endl;
    MC = " TEXTO CAMBIADO DE LA CADENA DE CARACTER ";
    cout << MC << endl;
    return 0;
}
```

1.1. Caracteres y literales tipo cadena

También existen constantes no-numéricas como:

```
'z'
'p'
"Hola mundo"
"¿Cómo está ud?"
```



Las dos primeras expresiones representan constantes simples tipo carácter y las dos siguientes representan literales compuestos de varios caracteres. Es de notar que, para representar un simple carácter, el mismo debe ser encerrado entre apóstrofes ('), mientras que para una cadena (la cual consiste de más de un carácter) se haga uso de comillas (") para delimitarla. Cuando se escriben caracteres y literales tipo cadena, es necesario colocar los delimitadores correspondientes para distinguirlos de posibles variables o palabras reservadas. Nótese la diferencia entre las dos expresiones siguientes:

x
'x'

x puede referirse a una variable cuyo identificador es dicha letra, mientras que 'x' se refiere a la constante 'x'.

El literal tipo cadena y los caracteres tienen ciertas peculiaridades, como las secuencias de escape. Estos son caracteres especiales que son difíciles o imposibles de expresar en el código fuente de un programa, como el retorno del carro (\n) o el tabulador (\t). Todos ellos están precedidos de una barra invertida (\). A continuación, se presenta una lista de secuencias de escape:

Tabla 16.
Secuencias de Escape.

\n	Bajar una línea
\r	Retorno de carro
\t	Tabulador horizontal
\v	Tabulador vertical
\b	Retroceso (backspace)
\f	Nueva página (page feed)
\a	Alerta sonora
\'	Apostrofe
\"	Comilla
\?	Signo de interrogación
\\	Barra Invertida

Fuente: Bustamante, P. y otros (2004).

Las cadenas pueden extenderse más allá de una línea de código, colocando una barra inversa al final de cada línea para señalar la continuación a la siguiente línea.

"cadena expresada en \
dos líneas"

También, se puede concatenar varias constantes de cadenas, separándolas por uno o varios espacios en blanco, tabuladores o cualquier otro carácter válido no imprimible:

"esto forma" "una sencilla" "cadena" "de caracteres"

Finalmente, si se desea que la cadena de literales esté explícitamente constituida de caracteres amplios (tipo `wchar_t`), en vez de caracteres normales (tipo `char`), podemos preceder la cadena de literales con el prefijo `L`, como lo indica el siguiente ejemplo:

`L"Esta es una cadena de caracteres anchos"`

Los caracteres anchos se usan principalmente para representar caracteres exóticos o que no se encuentran en el lenguaje español.

2. LECTURA DE CADENAS

Una cadena suele ser representada entre comillas (ej.: "palabra"), mientras que un carácter de esa cadena (`char` en inglés) suele ser representado entre comillas simples o apóstrofes (`'p'`). Por ejemplo, en C:

```
char c = 'a';  
char str[5] = "hola";
```

Generalmente, para acceder a un carácter en una posición determinada, se suele usar la forma variable `[posición]`, como cuando se accede a un vector. Para poder mostrar una comilla (") dentro de la cadena y no tener problemas con las comillas que la delimitan, se usan secuencias de escape. Esto se aplica a otros caracteres reservados o no imprimibles como el retorno de carro. No obstante, las expresiones para producir estas secuencias de escape dependen del lenguaje de programación que se esté usando. Una forma común, en muchos lenguajes de escapar un carácter es anteponiéndole un `«\»` (sin comillas), p. e.: `«\»` (sin comillas).

2.1. Operaciones de entrada y salida

Las operaciones de lectura y escritura (e/s) pueden ser ejecutadas en C++ usando las librerías estándar de Entrada y Salida: `cstdio` (conocida como `stdio.h` en el lenguaje C). y la `iostream`. En este sentido, Corona, M. y Ancona, M. (2011: 286) indican, que el lenguaje C no tiene palabras reservadas para la e/s estándar del sistema, éstas se realizan mediante funciones de biblioteca que se encuentra en la `stdio.h`. Por otra parte, Bustamante, P. y otros (2004: 57) dicen que el C++ no dispone de



sentencias de e/s, en su lugar se utilizan operadores contenidos en las librerías estándar y que forman parte integrante del lenguaje.

Es importante resaltar, que estas librerías usan lo que se conoce como flujos para operar con dispositivos físicos (hardware: teclados, impresoras, terminales y cualquier tipo de archivos soportados por el sistema). Los flujos son una abstracción que permite interactuar con dichos dispositivos de una manera uniforme, todos los flujos tienen propiedades similares, independientemente de las características individuales de los medios físicos con los cuales están asociados. En este sentido, Joyanes, L. (2000: 343) define un flujo como una abstracción que se refiere a un flujo o corriente de datos que fluyen entre un origen o fuente y un destino o suministro. En esencia, un flujo es una abstracción que se refiere a una interfaz común a diferentes dispositivos de entrada y salida.

Ahora bien, basados en las teorías de autores como Joyanes, L. (2000), Joyanes, L. y otros (2007) y Corona, M. y Ancona, M. (2011) se procederá a explicar algunas sentencias de e/s para cadenas de carácter:

2.2. Función: scanf

- *Formato general:*

```
int scanf ( const char * formato, ... );
```

Descripción: lee datos con formato de stdin (teclado) y los almacena de acuerdo a los parámetros de configuración en las localidades apuntadas por los argumentos adicionales. Los especificadores para formato siguen el prototipo:

`%[*][ancho][modificadores]tipo`

Tabla 17.

Especificadores de formato para lectura.

*	Opcional: indica que el dato va a ser recuperado de stdin, pero será ignorado, es decir, no será almacenado en el argumento correspondiente.
Ancho	Especifica el máximo número de caracteres a ser leído en la operación de lectura actual.
Modificadores	Especifica un tamaño diferente en el tipo int (casos: d, i y n), unsigned int (casos: o, u y x) o float (casos: e, f y g) para los datos apuntados por los correspondientes argumentos adicionales. h : short int (para d, i y n), o unsigned short int (para o, u y x) l : long int (para d, i y n), o unsigned long int (para o, u y x), o double (para e, f y g) L : long double (para e, f y g)
Tipo	Un carácter que especifica el tipo de dato a ser leído y cómo debe ser leído. Ver tabla siguiente.

Fuente: Joyanes, L. (2000) y Corona, M. y Ancona, M.(2011).

- *Especificadores de tipo:*

Tabla 18.
Especificadores de Tipo.

Tipo	Descripción	Tipo de argumento
C	Carácter sencillo: lee el próximo carácter. Si un valor distinto de 1 se especifica para ancho, la función scanf() lee la cantidad de caracteres especificada por ancho y los almacena en el arreglo que se ha pasado como argumento.	char*
D	Entero decimal: número opcionalmente precedido de un signo + o -.	int*
e,E,f,g,G	Punto flotante: número que contiene un punto decimal, opcionalmente precedido por un signo + o - y opcionalmente seguido por la letra e E y un número decimal. Ejemplos de entradas válidas: 732.103 y 7.12e4	float *
O	Entero Octal	int *
S	Cadena de caracteres: lee una secuencia de caracteres hasta alcanzar un carácter de terminación (espacio en blanco, tabulador o nueva línea).	char *
U	Entero decimal sin signo.	unsigned int *
x,X	Entero hexadecimal.	int *

Fuente: Joyanes, L. (2000) y Corona, M. y Ancona, M.(2011).

Ejemplo:

```
/* Ejemplo de scanf */
#include <stdio>
#include <iostream>

using namespace std;

int main ()
{
    char str [80];
    int i;

    printf ("Ingrese su apellido: ");
    scanf ("%s",str);
    printf ("Ingrese su edad: ");
    scanf ("%d",&i);
    printf ("Sr. %s , %d años de edad.\n",str,i);
    printf ("Ingrese un número hexadecimal: ");
```




```
scanf ("%x",&i);
printf ("Ud. ha ingresado %#x (%d).\n",i,i);
system("pause");
}
```

Este ejemplo muestra alguno de los tipos de datos que pueden ser leídos con scanf:

```

Ingrese su apellido: Paredes
Ingrese su edad: 29
Sr. Paredes, 29 años de edad.
Ingrese un número hexadecimal: ff
Ud. ha ingresado 0xff (255).
```

2.3. Función: sscanf

- *Formato general:*

```
int sscanf ( const char * cad, const char * format, ...);
```

Descripción: lee datos con formato de una cadena de entrada, y los almacena de acuerdo a los parámetros de configuración en las localidades apuntadas por los argumentos de entrada. Las localidades apuntadas por cada argumento adicional son llenadas con su tipo de valor correspondiente, especificado en el formato de la cadena. El prototipo y los modificadores de tipo son los mismos de la función scanf(). Ejemplo:

```
/* Ejemplo de sscanf */
#include <cstdio>
#include <iostream>

using namespace std;

int main ()
{
    char sentence []="Rodolfo tiene 12 años de edad";
    char str [20];
    int i;

    sscanf (sentence,"%s %*s %d",str,&i);
    printf ("%s -> %d\n",str,i);

    system("pause");
}
```

Salida:

Rodolfo -> 12

2.4. Función: getchar

- *Formato general:*

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

Descripción: obtiene un carácter de stdin (teclado). Es equivalente a `getc`, con `stdin` como su argumento. Ejemplo:

```
/* Ejemplo de getchar */
#include <stdio>
#include <iostream>

using namespace std;

int main ()
{
    char c;
    puts ("Ingrese texto e incluya un punto ('.') en el texto para terminar:");
    do {
        c=getchar();
        putchar (c);
    } while (c != '.');
    System("pause");
}
```

Resultado:

Ésta es una prueba.

2.5. Función: gets

- *Formato general:*

```
char * gets ( char * cad );
```

Descripción: lee una secuencia de caracteres desde `stdin`(teclado) y los almacena como una cadena en `cad`. Un carácter nulo (`'\0'`) es automáticamente agregado después del último carácter copiado a `cad` para señalar el final de la cadena. Es conveniente señalar, que la función `gets()` no permite especificar un límite sobre cuántos caracteres van a ser leídos, por lo que hay que tener cuidado con el tamaño del arreglo apuntado por `cad` para evitar desborde de almacenamiento (buffer overflows). Ejemplo:

```
/*Ejemplo de gets */
#include <stdio>
#include <iostream>

using namespace std;
```



```
int main()
{
    char string [256];
    printf ("Ingrese su dirección completa: ");
    gets (string);
    printf ("Su direction es: %s\n",string);
    system("pause");
}
```

2.6. Función: cin y cin.getline()

- *Formato general:*

```
cin >> cad;
cin.getline(variable, tamaño);
```

Descripción: lee una palabra ingresada por teclado y la guarda en la variable cad. Cuando se lee con esta función, no se puede colocar espacios en blanco. Ahora bien, si se desea leer oraciones con espacios en blanco intercalados, se deberá utilizar el cin.getline(). Ejemplo:

```
/*Ejemplo de cin */
#include <iostream>
using namespace std;
int main()
{
    string palabra;
    cout << "Ingrese una palabra: ";
    cin >> palabra;
    cout << "LA PALABRA FUE = " << palabra;
    system("pause");
}

/*Ejemplo de cin.getline */
#include <iostream>
using namespace std;
int main()
{
    char oración[100];
    cout << "Ingrese una oración: ";
    cin.getline(oracion, 100);
    cout << "LA ORACION FUE = " << oracion;
    system("pause");
}
```

2.7. Función: cin.get

- *Formato general:*

```
cin.get(carácter);
```

Descripción: lee un carácter a la vez. Ejemplo:

```
/*Ejemplo de cin.get */
#include <iostream>
using namespace std;
int main()
{
    char letra;
    cout << "Ingrese una letra: ";
    cin.get(letra);
    cout << "LA LETRA FUE = " << letra;
    system("pause");
}
```

2.8. Función: cout.put

- *Formato general:*

```
cout.put(caracter);
```

Descripción: muestra un carácter a la vez. Ejemplo:

```
/*Ejemplo de cout.put */
#include <iostream>
using namespace std;
int main()
{
    char letra;
    cout << "INGRESE UNA LETRA = ";
    cin.get(letra);
    cout << "LA LETRA ES = ";
    cout.put(letra);
    system("pause");
}
```

3. LA BIBLIOTECA STRING.H

El manejo de las cadenas de caracteres se realiza a través de funciones específicas, incluidas en la librería cstring (string.h, en el lenguaje C). Para



poder hacer uso de las mismas, se debe incluir al principio del programa la directiva include y especificar el encabezador de archivo (header file) string.h, de la forma siguiente:

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

A continuación, se presenta una tabla resumen con la lista de funciones disponibles:

Tabla 19.
Lista de funciones disponibles.

Operación	Nombre	Descripción
Copiar	memcpy	Copia bloques de memoria
	memmove	Mueve un bloque de memoria
	Strcpy	Copia una cadena
	strncpy	Copia caracteres de una cadena
Concatenación	Strcat	Concatena cadenas
	strncat	Agrega caracteres al final de una cadena
Comparación	memcmp	Compara dos bloques de memoria
	strcmp	Compara dos cadenas
	strcoll	Compara dos cadenas usando símbolos locales
	strncmp	Compara caracteres de dos cadenas
	Strxfrm	Transforma una cadena usando símbolos locales
Búsqueda	memchr	Ubica un carácter en un bloque de memoria
	Strchr	Ubica la primera ocurrencia de un carácter en una cadena
	Strcspn	Get span until character in string
	Strpbrk	Ubica un carácter en una cadena
	Strrchr	Ubica la última ocurrencia de un carácter en una cadena
	Strspn	Get span of character set in string
	Strstr	Ubica una subcadena
	Strtok	Divide una cadena usando símbolos separadores
Otras	memset	Rellena un bloque de memoria
	Strerror	Get pointer to error message string
	Strlen	Obtiene la longitud de una cadena

Fuente: <http://www.cplusplus.com/reference/cstring/> (2015).

3.1. Función: memcpy

- *Formato general:*

```
void * memcpy ( void * destino, const void * fuente, size_t num );
```

Descripción: copia los valores de num bytes, desde la localidad apuntada por fuente, directamente al bloque de memoria apuntado por destino. Ejemplo:

```
/* Ejemplo con memcpy */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[]="Cadena muestra";
    char str2[40];
    char str3[40];
    memcpy (str2,str1,strlen(str1)+1);
    memcpy (str3,"copia exitosa",16);
    printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
    return 0;
}
```

Resultado:

```
str1: Cadena muestra
str2: Cadena muestra
str3: copia exitosa
```

3.2. Función: memmove

- *Formato general:*

```
void * memmove ( void * destination, const void * source, size_t num);
```

Descripción: copia los valores de num bytes desde la localidad apuntada por fuente, directamente al bloque de memoria apuntado por destino. La acción de copiar tiene lugar si se usa un almacén intermedio (buffer), permitiendo el solapamiento entre fuente y destino. Ejemplo:

```
/* Ejemplo con memmove */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "memmove puede ser muy útil.....";
    memmove (str+20,str+15,11);
    puts (str);
    return 0;
}
```



```
}
```

Resultado:

Memmove puede ser muy útil

3.3. Función: strcpy

- *Formato general:*

```
char * strcpy ( char * destino, const char * fuente );
```

Descripción: copia la cadena apuntada por fuente en el arreglo apuntado por destino, incluyendo el caracter de terminación de cadena (null). Ejemplo:

```
/* Ejemplo con strcpy */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[]="Cadena de muestra";
    char str2[40];
    char str3[40];
    strcpy (str2,str1);
    strcpy (str3,"copia exitosa");
    printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
    return 0;
}
```

Resultados:

Str1: Cadena de muestra

Str2: Cadena de muestra

Str3: copia exitosa

3.4. Función: strncpy

- *Formato general:*

```
char * strncpy ( char * destino, const char * fuente, size_t num );
```

Descripción: copia los primeros num caracteres desde fuente a destino. Si el final de la cadena fuente (indicado con el caracter nulo) se encuentra antes que los num caracteres hayan sido copiados, destino se rellena con ceros, hasta que el total de caracteres haya sido escrito. Ejemplo:

```
/* Ejemplo con strncpy */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[] = "Ser o no ser";
    char str2[4];
    strncpy (str2, str1, 3);
    str2[5] = '\0';
    puts (str2);
    return 0;
}
```

Resultado:

Ser

3.5. Función: strcat

- *Formato general:*

```
char * strcat ( char * destination, const char * source );
```

Descripción: agrega una copia de la cadena fuente a la cadena destino. El caracter de terminación en la cadena destino es remplazado por la cadena que se ha agregado al final de aquella. Ejemplo:

```
/* Ejemplo con strcat */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[80];
    strcpy (str, "Estas ");
    strcat (str, "cadenas ");
    strcat (str, "estan");
    strcat (str, "concatenadas.");
    puts (str);
    return 0;
}
```

Resultado:

Estas cadenas están concatenadas



3.6. Función: **strncat**

- *Formato general:*

`char * strncat (char * destination, char * source, size_t num);`

Descripción: agrega los primeros num caracteres de la cadena fuente a la de destino, incluyendo el caracter nulo de terminación. Ejemplo:

```
/* Ejemplo de strncat */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[20];
    char str2[20];
    strcpy (str1,"Ser ");
    strcpy (str2,"o no ser");
    strncat (str1, str2, 6);
    puts (str1);
    return 0;
}
```

Resultado:

Ser o no s

3.7. Función: **memcmp**

- *Formato general:*

`int memcmp (const void * puntero1, const void * puntero2, size_t num);`

Descripción: compara los primeros num bytes del bloque de memoria apuntado por puntero1, con los primeros num bytes del bloque de memoria apuntado por puntero2, retornando 0 si son iguales o un valor diferente de 0 para indicar cuál es más grande, en caso de que difieran. Ejemplo:

```
/* memcmp example */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[256];
    char str2[256];
    int n;
```

```
printf ("Ingrese una palabra: "); gets(str1);
printf ("Ingrese otra palabra: "); gets(str2);
n=memcmp ( str1, str2, 256 );
if (n>0) printf ("%s' es más grande que '%s'.\n",str1,str2);
else if (n<0) printf ("%s' es menor que '%s'.\n",str1,str2);
else printf ("%s' es igual que '%s'.\n",str1,str2);
return 0;
}
```

Resultado:

```
Ingresa una palabra: edificio
Ingresa otra palabra: libro
edificio es más grande que libro
```

3.8. Función: strcmp

- *Formato general:*

```
int strcmp ( const char * cad1, const char * cad2 );
```

Descripción: compara la cadena cad1 con cad2. El proceso se realiza carácter por carácter, hasta alcanzar el carácter nulo de terminación. El resultado es un valor entero: si es cero, las dos cadenas son iguales, si es positivo indica que el primer carácter que difiere en cad1, es mayor que el correspondiente en cad2, si es negativo indica lo opuesto. Ejemplo:

```
/* Ejemplo de strcmp */
#include <stdio.h>
#include <string.h>

int main ()
{
    char szKey[] = "manzana";
    char szInput[80];
    do {
        printf ("Cuál es mi fruta favorita? ");
        gets (szInput);
    } while (strcmp (szKey,szInput) != 0);
    puts ("Respuesta correcta!");
    return 0;
}
```

Resultado:

```
¿Cuál es mi fruta favorita? Naranja
```



¿Cuál es mi fruta favorita? Manzana
Respuesta correcta!

3.9. Función: **strncmp**

- *Formato general:*

```
int strncmp ( const char * cad1, const char * cad2, size_t num );
```

Descripción: compara hasta num caracteres de la cadena cad1 con los de cad2. El proceso se realiza carácter por carácter, hasta alcanzar el carácter nulo de terminación o que los caracteres difieran, o se complete el número de caracteres a comparar, lo que suceda primero. El resultado es un valor entero: si es cero, significa que los caracteres comparados en las dos cadenas son iguales, si es positivo indica que el primer carácter que difiere en cad1 es mayor que el correspondiente en cad2, si es negativo indica lo opuesto. Ejemplo:

```
/*Ejemplo de strncmp */
#include <stdio.h>
#include <string.h>
int main ()
{
    char str[][5] = { "R2D2" , "C3PO" , "R2A6" };
    int n;
    puts ("Buscando los robots cuyo nombre comienza con R2...");
    for (n=0 ; n<3 ; n++)
        if (strncmp (str[n],"R2xx",2) == 0)
        {
            printf ("Hallado %s\n",str[n]);
        }
    return 0;
}
```

Resultado:

```
Buscando los robots cuyo nombre comienza con R2...
Hallado R2D2
Hallado R2A6
```

3.10. Función: **strxfrm**

- *Formato general:*

```
size_t strxfrm ( char * destino, const char * fuente, size_t num );
```

Descripción: transforma la cadena fuente de acuerdo a los símbolos locales y copia los primeros num caracteres de la cadena transformada en la cadena destino, retornando su longitud. Si sólo se requiere la longitud, se especifica un apuntador nulo para la cadena destino y 0 (cero) en el parámetro num. Ejemplo:

```
/*Ejemplo de strxfrm */
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1, buffer[80];
    int length;

    printf("Ingrese una cadena de caracteres.\n ");
    string1 = gets(buffer);
    length = strxfrm(NULL, string1, 0);
    printf("Se requiere un arreglo de %d elementos para almacenar la
cadena\n",length);
    printf("\n\n%s\n\n transformada de acuerdo",string1);
    printf(" a los símbolos locales de este programa. \n");
}
```

3.11. Función: memchr

- *Formato general:*

```
const void * memchr ( const void * ptr, int valor, size_t num );
void * memchr ( void * ptr, int valor, size_t num );
```

Descripción: busca, dentro de los primeros num bytes del bloque de memoria apuntado por ptr, la primera ocurrencia de valor (interpretado como un carácter sin signo), and retorna un apuntador a dicho valor. Ejemplo:

```
/* Ejemplo de memchr */
#include <stdio.h>
#include <string.h>

int main ()
{
    char * pch;
    char str[] = "Ejemplo de cadena";
    pch = (char*) memchr (str, 'p', strlen(str));
    if (pch!=NULL)
```



```
printf ("p' encontrada en la posicion %d.\n", pch-str+1);
else
    printf ("no se encontró 'p'.\n");
return 0;
}
```

Resultado:

'p' encontrada en la posición 5

3.12. Función: strchr

- *Formato general:*

```
const char * strchr ( const cad * str, int caracter );
char * strchr (      cad * str, int caracter );
```

Descripción: retorna un apuntador a la primera ocurrencia de carácter en la cadena cad. Ejemplo:

```
/* Ejemplo de strchr */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "Esta es una cadena de muestra";
    char * pch;
    printf ("Buscando el caracter 'a' en \"%s\"...\n",str);
    pch=strchr(str,'a');
    while (pch!=NULL)
    {
        printf ("Hallado en %d\n",pch-str+1);
        pch=strchr(pch+1,'a');
    }
    return 0;
}
```

Resultado:

```
Buscando el carácter 'a' en "Ésta es una cadena de muestra"
Hallado en 4
Hallado en 11
Hallado en 14
Hallado en 18
Hallado en 29
```

3.13. Función: **strcspn**

- *Formato general:*

```
size_t strcspn ( const char * cad1, const char * cad2 );
```

Descripción: busca en cad1 la primera ocurrencia de cualquiera de los caracteres que son parte de cad2, retornando el número de caracteres de cad1 que fueron leídos antes de esta ocurrencia. Ejemplo:

```
/* Ejemplo de strcspn */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "fcba73";
    char keys[] = "1234567890";
    int i;
    i = strcspn (str,keys);
    printf ("El primer número en la cadena está en la posición %d.\n",i+1);
    return 0;
}
```

Resultado:

El primer número en la cadena está en la posición 5

3.14. Función: **strpbrk**

- *Formato general:*

```
const char * strpbrk ( const char * cad1, const char * cad2 );
char * strpbrk (      char * cad1, const char * cad2 );
```

Descripción: retorna un apuntador a la primera ocurrencia en cad1 de cualquiera de los caracteres que son parte de cad2, o un apuntador nulo en caso contrario. Ejemplo:

```
/* Ejemplo de strpbrk */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = " ésta es una cadena de muestra ";
    char key[] = "aeiou";
```



```
char * pch;
printf ("Vocales en '%s': ",str);
pch = strpbrk (str, key);
while (pch != NULL)
{
    printf ("%c ", *pch);
    pch = strpbrk (pch+1,key);
}
printf ("\n");
return 0;
}
```

Resultado:

Vocales en 'ésta es una cadena de muestra': e a e u a a e a e u e a

3.15. Función: strrchr

- *Formato general:*

```
const char * strrchr ( const char * cad, int caracter );
char * strrchr (      char * cad, int caracter );
```

Descripción: retorna un apuntador a la última ocurrencia de caracter en la cadena cad. Ejemplo:

```
/* strrchr example */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = " ésta es una cadena de muestra ";
    char str[] = " ésta es una cadena de muestra ";
    char * pch;
    pch=strrchr(str,'s');
    printf ("Última ocurrencia de 's' hallada en %d \n",pch-str+1);
    return 0;
}
```

Resultado:

Última ocurrencia de 's' hallada en 26

3.16. Función: strstr

- *Formato general:*

```
size_t strstr ( const char * cad1, const char * cad2 );
```

Descripción: retorna la longitud de la porción inicial de cad1, que consiste solamente de los caracteres que son parte de cad2. Ejemplo:

```
/* Ejemplo de strstr */
#include <stdio.h>
#include <string.h>

int main ()
{
    int i;
    char strtext[] = "129th";
    char cset[] = "1234567890";

    i = strstr (strtext,cset);
    printf ("La longitud del número inicial es %d.\n",i);
    return 0;
}
```

Resultado:

La longitud del número inicial es 3

3.17. Función: strstr

- *Formato general:*

```
const char * strstr ( const char * cad1, const char * cad2 );
char * strstr ( char * cad1, const char * cad2 );
```

Descripción: localiza una subcadena, retornando un apuntador a la primera ocurrencia de cad2 en cad1, o un apuntador nulo si cad2 no es parte de cad1. Ejemplo:

```
/* Ejemplo de strstr */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "Ésta es una cadena simple";
    char * pch;
```




```
pch = strstr (str,"simple");
strncpy (pch,"muestra",7);
puts (str);
return 0;
}
```

Resultado (en este ejemplo, se reemplaza simple por muestra en la cadena original):

Ésta es una cadena muestra

3.18. Función: strtok

- *Formato general:*

```
char * strtok ( char * cad, const char * delimitadores );
```

Descripción: divide la cadena en componentes léxicos o "tokens": palabras clave (if, else, while, int,...), identificadores, números, signos, o un operador de varios caracteres, (por ejemplo, :=). Estos componentes se encuentran separados por cualquiera de los símbolos que sean parte de delimitadores. Ejemplo:

```
/* Ejemplo de strtok */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "- Ésta, una cadena muestra.";
    char * pch;
    printf ("Dividiendo la cadena \"%s\" en los componentes:\n",str);
    pch = strtok (str, ",.-");
    while (pch != NULL)
    {
        printf ("%s\n",pch);
        pch = strtok (NULL, ",.-");
    }
    return 0;
}
```

Resultado:

Dividiendo la cadena "-Ésta, una cadena muestra." En los components:
 Esta
 una

cadena
muestra

3.19. Función: memset

- *Formato general:*

```
void * memset ( void * puntero, int valor, size_t num );
```

Descripción: rellenar un bloque de memoria con un valor específico. Coloca los primeros num bytes del bloque de memoria apuntado por punter al valor especificado (el cual es un carácter sin signo). Ejemplo:

```
/*Ejemplo de memset */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "casi todo programador debería conocer memset!";
    memset (str,' ',4);
    puts (str);
    return 0;
}
```

Resultado:

----todo programador debería conocer memset!

3.20. Función: strerror

- *Formato general:*

```
char * strerror ( int errnum );
```

Descripción: obtiene el apuntador a un mensaje de error. Interpreta el valor de errnum generando una cadena que describe el error correspondiente a dicho valor numérico. La cadena de error producida por strerror depende de la plataforma de desarrollo y compilador. Ejemplo:

```
/* Ejemplo de strerror: lista de errores */
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main ()
{
```



```
FILE * pFile;
pFile = fopen ("inexist.ent","r");
if (pFile == NULL)
    printf ("Error abriendo archivo inexist.ent: %s\n",strerror(errno));
return 0;
}
```

Resultado:

Error abriendo archivo inexist.ent: No such file or directory

3.21. Función: strlen

- Formato general:

```
size_t strlen ( const char * cad );
```

Descripción: retorna la longitud de la cadena cad. Ejemplo:

```
/* Ejemplo de strlen */
#include <stdio.h>
#include <string.h>

int main ()
{
    char szInput[256];
    printf ("Ingrese una frase: ");
    gets (szInput);
    printf ("La frase ingresada tiene %u caracteres de longitud.\n",
(unsigned)strlen(szInput));
    return 0;
}
```

Resultado:

Ingrese una frase: solo para probar
La frase ingresada tiene 16 caracteres de longitud

4. ASIGNACIÓN DE CADENAS

La asignación directa de cadenas sólo está permitida cuando se hace junto con la declaración. El siguiente ejemplo producirá un error en el compilador, ya que una cadena definida de este modo se considera una constante.

```
char Saludo[5];
Saludo = "HOLA"
```

La manera correcta de asignar una cadena es:

```
char Saludo[5];  
Saludo[0] = 'H';  
Saludo[1] = 'O';  
Saludo[2] = 'L';  
Saludo[3] = 'A';  
Saludo[4] = 0;
```

En el ejemplo anterior, las cuatro primeras posiciones se usan para almacenar los caracteres "HOLA" y la posición extra, para el carácter nulo.

O bien:

```
char Saludo[5] = "HOLA";
```

4.1. Inicialización de vectores para almacenar cadenas

Si al declarar un vector de caracteres deseamos darle un valor inicial de tipo cadena, tenemos dos posibilidades:

1. Especificar la cadena entre comillas dobles, omitiendo el '\0', aunque manteniendo el tamaño del vector como si estuviera:

```
char cadena[5]="Hola";
```

2. Especificar la cadena entre comillas dobles, omitiendo el '\0', y sin especificar el tamaño para el vector. El compilador tomará el tamaño adecuado, reservando también espacio para el '\0':

```
char cadena[]="Hola";
```

Aunque la segunda forma es la más recomendable, puesto que nos evita tener que contar los caracteres de la cadena con la cual estamos inicializando, la primera será necesaria cuando queramos que el vector tenga un tamaño mayor que el de la cadena inicial (si por ejemplo pensamos, que más adelante en el programa se pueda almacenar una cadena de más caracteres en el mismo vector).

5. LONGITUD Y CONCATENACIÓN DE CADENAS

Concatenación: operación de unir dos o más cadenas de caracteres. El operador a usar es el signo +. Ej.: sea la variable pareja y las constantes "Luisa", " y ", "Carmen"; la instrucción siguiente:

```
pareja = "Luisa" + " y " + "Carmen"; # en C++ y Java con la clase String.
```



Va a producir como resultado una nueva cadena "Luisa y Carmen" que se va a almacenar en la variable *pareja*.

Para determinar la longitud, existen las siguientes funciones: `strlen`, `strspn`, `strcspn`.

5.1. Función: `strlen`

- *Formato general:*

```
size_t strlen ( const char * cad );
```

Descripción: retorna la longitud de la cadena *cad*. Ejemplo:

```
/* Ejemplo de strlen */
#include <stdio.h>
#include <string.h>

int main ()
{
    char szInput[256];
    printf ("Ingrese una frase: ");
    gets (szInput);
    printf ("La frase ingresada tiene %u caracteres de longitud.\n",
(unsigned)strlen(szInput));
    return 0;
}
```

Resultado:

```
      Ingrese una frase: solo para probar
      La frase ingresada tiene 16 caracteres de longitud
```

5.2. Función: `strspn`

- *Formato general:*

```
size_t strspn ( const char * cad1, const char * cad2 );
```

Descripción: retorna la longitud de la porción inicial de *cad1*, que consiste solamente de los caracteres que son parte de *cad2*. Ejemplo:

```
/* Ejemplo de strspn */
#include <stdio.h>
#include <string.h>

int main ()
{
    int i;
```



```
char strtext[] = "129th";
char cset[] = "1234567890";

i = strspn (strtext,cset);
printf ("La longitud del número inicial es %d.\n",i);
return 0;
}
```

Resultado:

La longitud del número inicial es 3

5.3. Función: **strcspn**

- *Formato general:*

```
size_t strcspn ( const char * cad1, const char * cad2 );
```

Descripción: busca en cad1 la primera ocurrencia de cualquiera de los caracteres que son parte de cad2, retornando el número de caracteres de cad1 que fueron leídos antes de esta ocurrencia. Ejemplo:

```
/* Ejemplo de strcspn */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "fcba73";
    char keys[] = "1234567890";
    int i;
    i = strcspn (str,keys);
    printf ("El primer número en la cadena está en la posición %d.\n",i+1);
    return 0;
}
```

Resultado:

El primer número en la cadena está en la posición 5

Para realizar la operación de concatenación, existen las siguientes funciones: `strcat` , `strncat`.

5.4. Función: **strcat**

- *Formato general:*

```
char * strcat ( char * destination, const char * source );
```

Descripción: agrega una copia de la cadena fuente a la cadena destino. El carácter de terminación en la cadena destino, es remplazado por la



cadena que se ha agregado al final de aquella. Ejemplo:

```
/* Ejemplo con strcat */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[80];
    strcpy (str,"Estas ");
    strcat (str,"cadenas ");
    strcat (str,"están");
    strcat (str,"concatenadas.");
    puts (str);
    return 0;
}
```

Resultado:

Estas cadenas están concatenadas.

5.5. Función: **strncat**

- *Formato general:*

```
char * strncat ( char * destination, char * source, size_t num );
```

Descripción: agrega los primeros num caracteres de la cadena fuente a la de destino, incluyendo el carácter nulo de terminación. Ejemplo:

```
/* Ejemplo de strncat */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[20];
    char str2[20];
    strcpy (str1,"Ser ");
    strcpy (str2,"o no ser");
    strncat (str1, str2, 6);
    puts (str1);
    return 0;
}
```

Resultado:

Ser o no s

6. OPERACIONES CON LAS CADENAS

Al considerar las cadenas como un tipo de datos, hay que definir (o conocer) cuáles son las operaciones que se pueden realizar con ellas, por lo que a continuación se presentan en forma resumida:

- **Asignación:** consiste en asignarle una cadena a otra.
- **Concatenación:** implica unir dos cadenas o más (o una cadena con un carácter) para formar una cadena de mayor tamaño.
- **Búsqueda:** se fundamenta en localizar dentro de una cadena una subcadena más pequeña o un carácter.
- **Extracción:** se trata de sacar fuera de una cadena una porción de la misma según su posición dentro de ella.
- **Comparación:** se utiliza para comparar dos cadenas y determinar si son iguales, entre otros resultados.

Dichas operaciones se encuentran resumidas en la siguiente tabla:

Tabla 20.
Operaciones con Cadenas.

Operación	Nombre	Descripción
Copiar	memcpy	Copia bloques de memoria
	memmove	Mueve un bloque de memoria
	Strcpy	Copia una cadena
	Strncpy	Copia caracteres de una cadena
Concatenación	Strcat	Concatena cadenas
	Strncat	Agrega caracteres al final de una cadena
Comparación	memcmp	Compara dos bloques de memoria
	Strcmp	Compara dos cadenas
	Strcoll	Compara dos cadenas usando símbolos locales
	strncmp	Compara caracteres de dos cadenas
	Strxfrm	Transforma una cadena usando símbolos locales
Búsqueda	memchr	Ubica un carácter en un bloque de memoria
	Strchr	Ubica la primera ocurrencia de un carácter en una cadena
	Strcspn	Get span until character in string
	Strpbrk	Ubica un carácter en una cadena
	Strrchr	Ubica la última ocurrencia de un carácter en una cadena
	Strspn	Get span of character set in string
	Strstr	Ubica una subcadena
	Strtok	Divide una cadena usando símbolos separadores
Otras	memset	Rellena un bloque de memoria
	strerror	Get pointer to error message string
	Strlen	Obtiene la longitud de una cadena

Fuente: Joyanes L (2000) y Joyanes, L. y otros (2007).



En síntesis, en un lenguaje como C++, las cadenas son arreglos de caracteres terminados en caracteres nulos (`\0`). Dichas cadenas se pueden usar para representar números, letras y todos aquellos símbolos que conforman un alfabeto, ya sea individual o colectivamente, y se pueden manipular utilizando técnicas estándares de procesamiento, elemento por elemento. Por tanto, las librerías del lenguaje C++ proveen soporte para el manejo de cadenas a través de la clase estándar `String`. Éste no es un tipo de dato fundamental, pero se comporta de manera similar en su uso básico.

Por otra parte, existen funciones de biblioteca para procesamiento de cadenas como una unidad completa. Algunas de las funciones de cadenas típicas son: longitud de la cadena, comparar cadenas, insertar cadenas, copiar cadenas, concatenar cadenas, entre otros. Hoy es cada vez más frecuente el uso de los computadores para procesar problemas orientados al manejo de datos de tipo alfanumérico o de tipo texto, por lo que el estudio y manejo de las cadenas de caracteres es de gran importancia.

ACTIVIDADES DE AUTOEVALUACIÓN



UNIVERSIDAD
Privada
DR. RAFAEL BELLOSILLO CHACÍN

Nombre: _____

Cédula: _____ Sección: _____

1. Elaborar una función en C++ similar a `strlen` que pueda manejar cadenas sin terminador. Tip: se necesitará conocer y pasar la longitud de la cadena.
2. Elaborar una función en C++ que regrese verdad, si una cadena de entrada es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha, o de derecha a izquierda. Por ejemplo, AREPERA.
3. Elaborar una función en C++ que permita una posible implementación de la función `strtok()`:
 - a. Usando otras funciones de manejo de cadenas.
 - b. Desde los principios de apuntadores.
4. Elaborar una función en C++ que convierta todos los caracteres de una cadena a mayúsculas.
5. Elaborar una función en C++ que invierta el contenido de la memoria en bytes. Es decir, si se tienen n bytes, el byte de memoria n se invierte con el byte 0, el byte $n-1$ se intercambia con el byte 1, etc.
6. ¿Es correcta la siguiente instrucción?
`char cadena[]="Hola";`

SI		NO	
----	--	----	--

Porque: _____

7. La unión de dos o más cadenas, recibe el nombre de: _____

8. ¿Cuál es el resultado de ejecutar las siguientes instrucciones?

```
char str[80];  
strcpy (str,"En la");  
strcat (str," union ");  
strcat (str," está la ");  
strcat (str," fuerza");
```

9. ¿Cuál de los flujos está asociado con la operación de lectura de datos?

10. Indique cuál de las siguientes expresiones no es válida:

```
"z"  
'p'  
'Hola mundo'  
"¿Cómo está ud?"
```



BIBLIOGRAFIA Y OTRAS FUENTES DE CONSULTA

- Badenas, Jorge; Llopis, José Luis y otros. (2001). **"Curso práctico de programación en C y C++"**. Valencia - España: publicaciones de la Universitat Jaume I.
- Bronson, Gary. (2000). **"C++ para ingeniería y ciencias"**. D.F. - México: International Thomson.
- Bustamante, Paul; Aguinaga, Iker; Aybar, Miguel; Olaizola, Luis y Lazacano, Iñigo. (2004). **"Aprender C++ Básico como si estuviera en primero"**. <http://www.tecnun.es> (Campo tecnológico de la Universidad de Navarra). Consultada:11/09/2015.
- Cruz, Nelson. (2004). **Manual de Ayuda al desarrollo en el entorno Dev C++**. Universidad Nacional de Colombia. <http://dis.unal.edu.co/~programacion/language/c/DevC++.pdf>. Consultada: 11/05/2015
- Corona, María y Ancona, María. (2011). **"Diseño de Algoritmos y su Codificación en Lenguaje C"**. D.F. - México: McGraw-Hill.
- Deitel, Harvey y Deitel, Paul. (2009). **"Cómo programar en C++ y Java"**. D. F. – México: Pearson Prentice-Hall. <http://www.FreeLibros.com>
- Downey, Allen. (2012). **"How to think like a computer scientist"**. <http://www.greenteapress.com/thinkcpp/thinkCScpp.pdf> Consultada:
- Fabelo, Ricardo y Medina, Maribel. (2004). **"Algoritmo y Programación I y II"**. Maracaibo - Venezuela: Fondo Editorial URBE.
- Joyanes, Luis. (2003). **"Fundamentos de Programación. Algoritmos, Estructuras de Datos y Objetos"**. D. F. - México: McGraw-Hill.
- Joyanes, Luis. (2000). **"Programación en C++. Algoritmos, Estructuras de datos y Objetos"**. D. F –México: McGraw-Hill.
- Joyanes, Luis; Castillo, Andrés; Sánchez, Lucas y Zahonero, Ignacio. (2005). **"C. Algoritmos, programación y estructuras de datos"**. D.F. - México: McGraw-Hill.

Joyanes, Luis; Sánchez, Lucas y Zahonero, Ignacio. (2007). **"Estructuras de datos en C++"**. D. F. - México: McGraw-Hill.

Joyanes, Luis y Zahonero, Ignacio. (2005). **"Programación en C: Metodología, algoritmos y estructura de datos"**. D.F. - México: McGraw-Hill.

Kerwin, Norma. (2009). **El Origen de la palabra Algoritmo en castellano**. <http://normakerwin.blogspot.com/2009/02/el-origen-de-la-palabra-algoritmo-en.html>. Consultada: 11/09/2015.

López, Gustavo; Jeder, Ismael y Vega, Augusto. (2009). **"Análisis y Diseño de Algoritmos"**. Buenos Aires – Argentina: editorial Alfaomega.

Meyers, Scott. (2005). **"Effective C++ 55 Specific Ways to Improve Your Programs and Designs"**. Boston – E.E.U.U. Pearson Education.

Osorio, Alan. (2006). **"C++ Manual Teórico – Práctico"**. <http://slent.iespana.es/programacion/index.html>. Consultada: 11/09/2015.

Oviedo, Efraín. (2004). **"Lógica de Programación"**. Bogotá – Colombia: Ecoe Ediciones.

Savitch, Walter. (2007). **"Resolución de problemas con C++"**. D.F. – México: Pearson Educación.