Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Algoritmos y Programación
Guías de Clase

Profa. Yusneyi Y. Carballo Barrera
ND 2011-02

Centro de Enseñanza Asistida por Computador (CENEAC)
Caracas, Junio 2011

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Algoritmos y Programación

Profa. Yusneyi Y. Carballo B.
[yusneyi.carballo@ciens.ucv.ve

Guías de Clase
Recopilación Marzo 2005-Julio 2010

Última actualización: Junio.2011

Contenido

TEMA 1. INTRODUCCIÓN A LA PROGRAMACIÓN

Objetivos:

- Presentación de los alumnos, la docente, la materia y sus condiciones.
- Conversar sobre conceptos elementales asociados a introducción a la programación y enfoques de programación

Contenido:

- Presentación y Verificación de lista de alumnos, Expectativas, Distribución del grupo (Primeros / Repitientes)
- Lectura de la Nota Informativa, Preguntas y Aclaraciones
- Conceptos Básicos sobre el computador, hardware, software, algoritmos, programas, lenguajes de programación, compiladores e intérpretes
- Enfoques para solucionar un problema. Características deseables en un programa. Fases del ciclo de vida del Software.

1.1. Conceptos Básicos sobre el Computador

Computador u Ordenador

Dispositivo electrónico utilizado para procesar datos en forma automática y obtener resultados los cuales pueden a su vez ser organizados y analizados para producir información. El computador está conformado por una parte física llamada **hardware** y por una parte lógica o de programas llamada **software.**

Datos

Es una representación de hechos, conceptos o instrucciones, organizados de manera que se puedan procesar, interpretar o comunicar por medios humanos o automáticas. Los datos, son por ejemplo, representaciones de las características de una persona, objeto, concepto o hecho.

Los datos se pueden introducir en el computador como **entrada** y se procesan para producir **resultados** e información de **salida**.

Hardware

En un computador se refiere a todos los componentes físicos que lo conforman, los aparatos propiamente dichos.

Como ejemplo tenemos los chips de los procesadores (CPU, procesadores matemáticos, procesadores de video), las tarjetas (la tarjeta madre, las tarjetas de memoria como la memoria RAM, las tarjetas de video, red, sonido), las unidades de almacenamiento (disco duro, disquete, cd, dvd, pen drive), los dispositivos periféricos (ratón, teclado, monitor, impresora)

Software

Son los programas que permiten utilizar los recursos del computador. Programación, soporte lógico, parte no-mecánica o no-física de un sistema. Es un conjunto de programas y procedimientos que se incluyen en un computador o equipo con el fin de hacer posible el su uso eficaz de dicha máquina. Son las instrucciones responsables de que el hardware (la máquina) realice su tarea.

Como ejemplo tenemos los **sistemas operativos**, el **software de aplicación**, el **software utilitario** y los **lenguajes de programación.**

Sistema Operativo:

Software básico encargado de controlar diferentes procesos en el computador mediante tres grandes funciones:

- Coordinar y manipular el hardware del computador: como los procesadores, la memoria, las impresoras, las unidades de almacenamiento, el monitor, el teclado o el ratón;
- Organizar los archivos en diversos dispositivos de almacenamiento, como discos flexibles, discos duros, cds;
- Gestionar los errores de hardware y la pérdida de datos.

Software de Aplicación:

Programa informático diseñado para facilitar al usuario la realización de un **determinado tipo** de trabajo. Algunas son aplicaciones desarrolladas **'a la medida'** que ofrecen una gran potencia y soluciones eficientes ya que están exclusivamente diseñadas para resolver un problema específico. Son ejemplos de este tipo de software los programas que realizan tareas concretas como manejo de nómina, análisis de estadísticas, manejo de almacén, etc.

Software Utilitario:

Son programas que facilitan el uso del computador como herramienta para solucionar actividades generales como la edición de textos o la digitalización de materiales. En muchos casos los programas utilitarios son agrupados en **paquetes integrados** de software, por ejemplo el Microsoft Office o el OpenOffice, donde se ofrece soluciones más generales, pero se incluyen varias aplicaciones(procesador de textos, de hoja de cálculo, manejador de base de datos, correo electrónico, visor de imágenes, etc.).

Lenguajes de Programación:

Sirven para escribir programas que permitan la comunicación usuario/máquina y la soluciones de problemas utilizando las ventajas, poder de cálculo, procesamiento y almacenamiento del computador.

Diferencias entre los tipos software

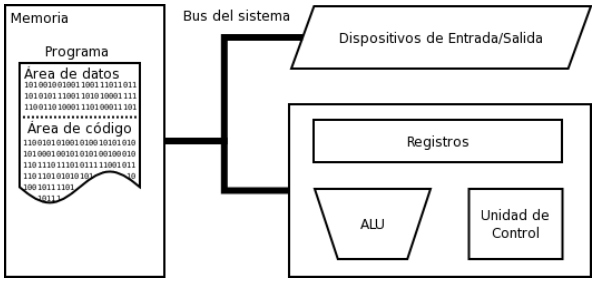
El **software de aplicación** se diferencia de un **sistema operativo** (que hace funcionar al ordenador), de una **utilidad** (que realiza tareas de mantenimiento o de uso general) y de un **lenguaje** (con el cual se crean los programas informáticos), en que suele resultar una solución informática para la automatización de tareas en un área determinada (procesamiento de texto, edición de imágenes, estadística, manejo de correspondencia, etc).

1.2. Estructura Funcional de un Computador (arquitectura Von Neumann)

La **arquitectura Von Neumann** se refiere a las arquitecturas de computadoras que utilizan el mismo dispositivo de almacenamiento para las instrucciones y para los datos (a diferencia de la arquitectura Harvard). El término se acuñó en el año 1945, escrito por el conocido matemático John von Neumann, que propuso el concepto de **programa almacenado**.

Los ordenadores con arquitectura Von Neumann constan de las siguientes partes:

- La **Memoria Principal**,
- La **Unidad Central de Proceso** (CPU o Procesador), su vez formada por:
  - La **Unidad Lógico-Aritmética** (ALU)
  - La **Unidad de Control** (UC)
- Los **Dispositivo de Entrada/Salida**, formados por:
  - El **Bus de datos o unidades de intercambio** que proporciona un medio de transporte de los datos entre las distintas partes.
  - La **Memoria Secundaria**
  - Los **Dispositivos Periféricos**



Memoria Principal

Es el subsistema donde se almacenan temporalmente los datos e instrucciones que son utilizados por el computador. Esta información está representada en una codificación binaria de 0 y 1. La memoria se divide en celdas, donde cada celda tiene una dirección única, de tal forma que el contenido de ellas puede ser buscado, extraído y utilizado.

Unidad Central de Proceso (CPU o Procesador)

Es el subsistema encargado de extraer las instrucciones y los datos en la **memoria principal** para realizar los tratamientos u operaciones correspondientes. Está conformado por la **Unidad de Control** y la **Unidad Lógico-Aritmética**

La **Unidad de Control** (UC) se encarga de:

1. Obtener de la memoria la próxima instrucción a utilizar o ejecutar;
2. Decodificar la instrucción para determinar qué debe hacerse,
3. Según la decodificación hecha, enviar el comando apropiado a la ALU, memoria o controlador de entrada/salida para que realice la tarea

La **Unidad Aritmético-Lógica** (ALU) se encarga de realizar las operaciones aritméticas y comparaciones.

Dispositivos de Entrada/Salida

Es el subsistema que permite al computador interactuar con otros dispositivos, comunicarse con el mundo exterior y almacenar los datos y programas en unidades permanentes, por ejemplo, el disco duro.

Está conformado por:

Bus o unidades de intercambio, que permiten comunicar información entre los componentes del sistema, los periféricos y el mundo exterior.

Memoria Secundaria, permite conservar datos y programas en forma permanente, aún luego de apagar el computador.

Periféricos, dispositivos utilizados para suministrar información entre el computador y el exterior (monitor, teclado, ratón, tarjeta de red, impresoras, tarjetas de memoria, escáner, etc.)

Para Investigar y Meditar

1. ¿Existen otros esquemas de arquitectura u organización del computador? ¿Qué es la **Arquitectura Harvard**?
2. Crea un esquema de los principales **dispositivos de entrada-salida** que conoces, ayúdate clasificándolos en tres (3) categorías: dispositivos de entrada, dispositivos de salida y dispositivos de entrada/salida.

1.3. Conceptos Básicos sobre Construcción de Programas

Algoritmo (*algorithm*)

Es un conjunto bien definido de procedimientos lógicos o matemáticos que se pueden seguir para resolver un problema en un número finito de pasos.

Es una lista finita de pasos que plantean una solución a un problema, preferiblemente pasos los más cortos y simples posibles. Para un mismo problema pueden existir muchos algoritmos que conducen a su solución. La elección del mejor algoritmo está guiada por criterios de eficiencia y eficacia, entre otras características deseables.

Elementos de un algoritmo:

- Datos de entrada
- Proceso o pasos que resuelven el problema
- Datos de salida

Características de un algoritmo:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- El resultado del algoritmo debe estar *definido*. Si se sigue un algoritmo dos veces con los mismos datos de entrada, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento, es decir, se debe tener un número finito de pasos.

Pseudo-código (*pseudo-code*)

En un algoritmo expresado de manera más formal. Se utiliza como una representación intermedia, antes de traducirlo o codificarlo con un lenguaje de programación. En las clases de Algoritmo y Programación utilizaremos el pseudo-código para expresar las soluciones algorítmicas creadas.

Lenguaje de programación (*programming language*)

En computación es cualquier lenguaje artificial que puede utilizarse para definir una secuencia de instrucciones, a fin de que puedan ser procesadas por un computador.

Conjunto de caracteres, reglas, palabras y operaciones con significados previamente asignados y que permiten escribir programas.

La definición de un lenguaje de programación cubre tres aspectos:

1. Léxico: definen los símbolos que sirven para la redacción de un programa y las reglas para la formación de palabras en el lenguaje. Por ejemplo, 10 es un número entero
2. Sintaxis: conjunto de reglas que permiten organizar las palabras del lenguaje en frases, por ejemplo, la operación de división se define como Dividendo/Divisor
3. Semántica: definen las reglas que dan sentido a una frase

Los principales tipos de lenguajes de programación utilizados en la actualidad son:

- Lenguajes de máquina
- Lenguajes de bajo nivel y traductores (lenguaje ensamblador, compiladores, intérpretes)
- Lenguajes de lato nivel (C++, C#, Visual Basic, Java, Turbo Pascal, Prolog, SQL, HTML, JavaScript, VBScript, PHP, VB.Net, Fortran, Delphi, etc.)

Programa (*program*)

En Computación, es el conjunto de instrucciones que definen la secuencia de eventos que un computador debe ejecutar para llevar a cabo una tarea, realizando cálculos y suministrando resultados.

Grupo de instrucciones compuestas con la finalidad de resolver un problema específico mediante el uso de un computador. Un programa está codificado en un lenguaje que la máquina es capaz de entender y procesar.

Es la traducción de un **algoritmo** o de un **pseudo-código** utilizando un **lenguaje de programación**.

Programación (*programming*)

Proceso que comprende el análisis del problema, diseño de la solución, escritura o desarrollo del programa, prueba del programa y su corrección.

Es la disciplina de la Computación que trata el desarrollo de programas.

Para Investigar y Meditar

1. Investiga y describe luego con tus palabras lo esencial de cada una de las **características deseables en un programa.**
2. ¿Tu crees que un programa debe poseer necesariamente todas las características deseables?

Aspectos que miden la Calidad en un programa

Algunos aspectos que se consideran para medir la calidad de un programa, también llamados **características deseables en un programa**, son:

- Legibilidad
- Eficiencia
- Reusabilidad del software
- Robustez
- Adaptabilidad
- Eficacia
- Portabilidad

Capacidad de Abstracción

Mecanismo intelectual principal en la actividad de la programación, el cual durante la etapa de análisis del problema permite la separación de los aspectos relevantes de los irrelevantes en el contexto estudiado. Por ejemplo, si el problema consiste en determinar cuál es la persona más alta del salón, lo relevante es la estatura de cada persona, y no color de ojos, talla de calzado, etc.

Enfoques para solucionar un problema

1. Programación **Modular**. Ejemplo, cálculo de la superficie (S = PI*R^2) y longitud (L = 2*PI*R) de la circunferencia
2. Enfoque **Divide y Vencerás**. Ejemplo, sistema de manejo de almacén.
3. Diseño **Descendente (top-down)**. Ejemplo, para el problema de indicar los pasos para ver una película en el cine se podría considerar en un primer nivel los siguientes pasos: 1) Ir al cine, 2) Comprar una entrada, 3) Ver la película 4) Regresar a casa. Luego cada uno de estos pasos pueden subdividirse en otros niveles donde las instrucciones sean cada vez más específicas.

Fases en la resolución de Problemas

Resolver problemas a través de un computador se basa principalmente en analizar, diseñar, escribir y ejecutar un programa con pasos orientados a solucionar el problema. Podemos considerar como **fases de resolución de un problema** las siguientes:

1. Análisis del problema
2. Diseño del algoritmo, utilizando pseudo-código
3. Codificación, traducción del algoritmos a un lenguaje de programación, esto nos permite crear el programa
4. Ejecución del código del programa

5. Verificación del programa
6. Documentación
7. Depuración de errores
8. Mantenimiento y mejora del programa

Ciclo de Vida de Desarrollo del Software y sus fases o etapas más usuales

1. **Análisis**. El problema se analiza teniendo en cuanta los requerimientos o necesidades expresadas por el cliente, la empresa o las personas que utilizaran el programa.
2. **Diseño**. Una vez analizado el problema se diseña una solución que conduce a un algoritmo general que resuelve los elementos más significativos del programa. Este algoritmo puede ser escrito utilizando pseudocódigo.
3. **Codificación (implementación)**. La solución expresada en pseudocódigo se traduce a un programa que el computador pueda procesar utilizando un lenguaje de programación de alto nivel.
4. **Compilación, Ejecución y Verificación**. El programa se ejecuta (corre) y se verifica para eliminar errores de programación o de lógica.
5. **Documentación**. Se agrega al código del programa línea de texto que ayudan al programador y a las personas que a futuro harán mantenimiento al software a entender su estructura y comportamiento. La documentación también incluye escribir informes en donde se describe cómo se realizaron las diferentes fases del ciclo de vida del software (en especial los procesos de análisis, diseño, codificación y prueba), se agregan manuales de usuario y de referencia, así como normas para el mantenimiento.
6. **Depuración y Mantenimiento**. El programa se actualiza y modifica en la medida en que sea necesario de manera que cumpla con las necesidades de los usuarios las cuales también cambian en el tiempo.

1. ¿Piensa y plantea tu algoritmo con pasos estratégicos para pasar Algoritmos?
2. ¿Qué pasos debes contemplar para inscribir el laboratorio de Algoritmos?
3. ¿Cuál sería tu algoritmo para ir al cine y ver una película?

- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. **"Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos"**. Editorial McGraw-Hill, 2003. **Capítulo 2, páginas 40-42**
- DEITEL & DEITEL. **"Cómo Programar en Java"**. Editorial Pearson Prentice Hall, 2004. **Capítulo 1, páginas 5-8**
- **RECOMENDACIÓN:** toma nota y guarda las referencias de los libros o sitios Web que consultas, así ubicarlos fácilmente y usarlos en futuras

TEMA 2. TIPOS DE DATOS

Objetivos:

- Conocer sobre conceptos asociados a tipos de datos elementales, su clasificación, operadores y su precedencia, la conversión implícita y explícita de datos.

Puntos:

- Conceptos de tipo de dato, variables, constantes, precedencia. Clasificaciones de los tipos de datos.
- Tipos de datos elementales (entero, real, caracter, *string*, booleano, sub-rango, enumerado) y sus operaciones.
- Precedencia de operadores. Conversión de tipos, implícita y explícita.

2.1. Conceptos Básicos

Dato

Diferentes entidades u objetos de información con que trabaja un programa. Determina el conjunto de valores que la entidad puede almacenar, los operadores que puede usar y las operaciones definidos sobre ellos.

Tipo de Dato

- Define el conjunto de valores que un elemento o un objeto (una variable, constante, expresión o función) de dicho tipo puede asumir y las operaciones asociadas a tales valores.

- Es un conjunto de entidades o de objetos y las operaciones definidas sobre ellos.

Ejemplos de Clasificaciones para los Tipos de datos

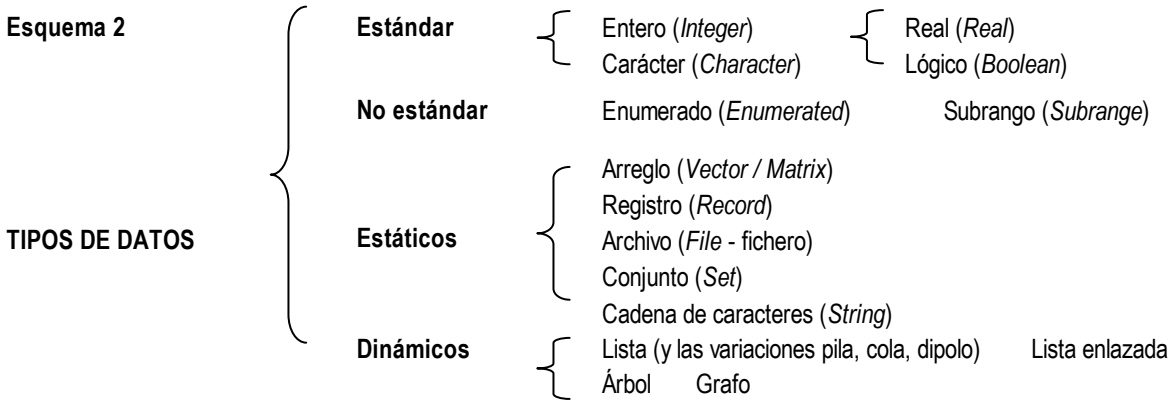
- **Tipos de datos primitivos o elementales**: tipos básicos incluidos en cada lenguaje de programación.
  En el lenguaje de programación Java, también son llamados tipos integrados
- **Tipos de datos estructurados**: tipos basados o construidos a partir de tipos de datos primitivos (por ejemplo, arreglo, registro, archivo, conjunto).
- **Tipos de datos abstractos** (TDA): tipos de datos definidos por el usuario y las operaciones abstractas aplicadas sobre ellos. Los TDA apoyan el concepto de ocultamiento de la información. Esconden los detalles de la representación y permiten el acceso a los objetos sólo a través de sus operaciones, son ejemplos las representaciones de los TDA Lista, Cola, Pila, Árbol y la representación que hace el Enfoque Orientado a Objeto mediante atributos y métodos.

Esquemas de organización de los tipos de datos:

Esquema 1

TIPOS DE DATOS

Número de componentes		**Tipo de Componentes**
1	**Elemental**	Entero, Real (numéricos)
		Carácter, Cadenas de caracteres o *String* (caracteres)
		Lógico o Booleano
		Enumerado
		Subrango
		Tipo de Componentes
n	**Estructurado**	**Homogéneos** (todos los elementos iguales)
		Archivos Arreglos
		Heterogéneos (aceptan elementos distintos)
		Registros



Variable

Nombre asignado a una entidad que puede adquirir **un valor cualquiera** dentro de un conjunto de valores. Es decir, una entidad cuyo valor puede cambiar a lo largo del programa. En un programa de computador se puede asumir que una variable es una posición de memoria donde los valores asignados pueden **ser reemplazados o cambiados por otros valores** durante la ejecución del programa.

Constante

Nombre asignado a una entidad al cual se asigna un valor que mantiene sin cambios durante el programa.

Operaciones de los tipos de datos elementales

- **Operación**: Acción por medio de la cual se obtiene un resultado de un operando. Ejemplos: sumar, dividir, unir, restar.
- **Operando**: número, texto, valor lógico, variable o constante sobre la cual es ejecutada una operación.
- **Operador**: símbolo que indica la operación que se ha de efectuar con el operando, por ejemplo, + / - * > == ≠ ≥ =

Expresiones

Son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones o acciones. Cada expresión toma un valor que se determina evaluando los valores de sus variables, constantes y operadores.

Una expresión consta de *operandos* y *operadores*.

Las expresiones se pueden clasificar en Aritméticas, Lógicas o Carácter. Así tenemos:

- Una expresión aritmética, arroja resultados de tipo numérico (*entero* o *real*)
- Una expresión relacional o una expresión lógica, arrojan resultados de tipo lógico (*booleanos*)
- Una expresión carácter, arroja resultados de tipo carácter (caracteres simples o *string*)

2.2 Precedencia de Operadores

Prioridad definida entre los operadores. Indica en qué orden debe aplicarse diferentes operaciones sobre un conjunto de valores. Permiten aplicar los operadores en el orden correcto.

Algunos comentarios sobre tipos de datos y precedencia:

- Los lenguajes C, C++ y Java requieren que todas las variables tengan un tipo asociado antes de que puedan ser usadas en un programa. Los lenguajes con esta propiedad son llamados **lenguajes de tipos estrictos**.
- A diferencia de C y C++, los tipos primitivos de Java son portátiles entre todas las plataformas que reconocen a Java. Esto permite a los programadores Java escribir los programas una sola vez sin saber ni preocuparse en cuál plataforma de computadora se ejecutará el programa. Por el contrario, los programadores en C y C++ a menudo tenían que escribir varias versiones de los programas para trabajar con las diversas plataformas de computadoras por que no se garantizaba que los tipos de datos primitivos fueran idéntico o se representaran igual en todas las computadoras.
- Como los diseñadores de Java querían que fuera un lenguaje portátil o transportable, decidieron usar estándares internacionales reconocidos para los formatos de caracteres (*char*, estándar Unicode) y para los números de punto flotante (*flot*, IEEE 754)
- Los operadores de expresiones contenidas dentro de pares de paréntesis () se evalúan primero. Así el programador puede usar paréntesis para obligar a una evaluación en el orden que desee.
- Si una expresión contiene varias operaciones de multiplicación, división o residuo, ó varias operaciones de suma o resta, los operadores se aplican de izquierda a derecha (asociatividad). Entre ellos tienen el mismo nivel de precedencia.

Tabla 1. Prioridad de los operadores en pseudocódigo

De mayor prioridad a menor prioridad. En caso de haber operadores del mismo nivel en una expresión, se evalúan en orden de aparición de izquierda a derecha.

Operadores	Nombres	Orden en que se evalúan
()	Paréntesis ó Corchetes	**Más alta prioridad.** Las expresiones entre paréntesis se evalúan primero. Si los paréntesis están anidados, la expresión más interna se evalúa primero. Si hay varios pares de paréntesis en el mismo nivel, se evalúan de izquierda a derecha.
^	Exponente (equiv. a **)	**Operadores matemáticos** Entre ellos el mismo nivel de precedencia. Si hay varios se evalúan de izquierda a derecha.
– (unario)	Menos unario	
* / div mod	Multiplicación real División real División entera o cociente (div) Residuo o resto entero (mod)	
+ –	Suma, Resta	
DesplazamientoIzq, DesplazamientoDer		**Operadores de desplazamiento de Registro** Desplazamiento a la izquierda o a la derecha en un archivo o en una cadena de caracteres
< ≤ > ≥	Menor, Menor o igual, Mayor, Mayor o igual	**Operadores relacionales.** Si hay varios se evalúan de izquierda a derecha. Entre ellos el mismo nivel de precedencia.
== ≠	Igual, Diferente o distinto de	
No	No lógico (*not*)	**Operadores lógicos**
Y	Y lógico (*and*)	
O	O lógico (*or*)	

Continúa …

+	Concatenación	**Operador de cadena** Permite la concatenación de valores de tipo entero, real o lógico con valores de tipo caracteres o con cadenas de caracteres (string). El resultado de la concatenación es un string. Se utiliza principalmente para crear mensajes.
=	Asignación	**Menor prioridad.** La asignación de valores o resultados a una variable o constante es la última operación que se realiza.

En expresiones donde se combinen operadores se evalúa primero los aritméticos, luego los relacionales y de último los operadores lógicos.

2.3 Conversión de Tipos

La conversión de tipos es el proceso de cambiar un valor de un tipo de dato a otro. Por ejemplo, el string o cadena "1579874" se puede convertir a un número entero, o se puede cambiar un número real a un String o a un Entero.

Las conversiones de tipo pueden ser de **ampliación** o de **restricción**:

- Las conversiones de **ampliación** transforman un valor de un tipo de dato a otro más grande, por ejemplo transforman un valor entero (conjunto más pequeño) a un número real (que es un conjunto más grande). Estas conversiones no producen desbordamiento o pérdida de datos.
- Las conversiones de **restricción** permiten transformar un valor de un conjunto de datos más grande a uno más pequeño, por ejemplo, transformar un número real a un entero o transformar una cadena a carácter. Si transformamos el valor X = 23,14587 a entero obtendríamos como resultado 23, lo cual significa pérdida de información ya que se pierde la precisión de los decimales; transformar el string m = ”casa” a carácter significaría quedarnos solo con el carácter ‘c’ del inicio de la cadena. Estos tipo de transformación son poco convenientes ya que pueden implicar pérdida de información y solo deben ser usados cuando sea estrictamente necesario.

Las conversiones por ampliación o por restricción pueden a su vez ser **explícitas** o **implícitas**.

Las conversiones con pérdida de información tienen lugar cuando el tipo de datos original no tiene un análogo en el tipo de destino de la conversión. Por ejemplo, la cadena "Pedro" no se puede convertir en un número. En estos casos, algunos lenguajes de programación devuelven un valor predeterminado cuando se usa la función de conversión de tipo, por ejemplo el valor **NaN** o el número cero, indicando con estos valores que la conversión de tipos falló.

Algunos tipos de conversiones, como de una cadena a un número, tardan bastante tiempo. Cuantas menos conversiones utilice el programa, más eficaz será.

Conversión implícita

La mayoría de las conversiones, como la asignación de un valor a una variable, se producen automáticamente. El tipo de datos de la variable determina el tipo de datos de destino de la conversión de expresión. En otros casos, la conversión implícita viene dada tanto por los tipos de datos como por los operadores utilizados.

Conversión explícita

Para convertir explícitamente una expresión a un tipo de datos concreto, se utilizan funciones que se asumen predefinidas y disponibles en el pseudocódigo, colocando entre paréntesis el dato, variable o expresión que se va a convertir. Las conversiones explícitas requieren más escritura que las implícitas, pero proporcionan más seguridad con respecto a los resultados. Además, las conversiones explícitas pueden controlar conversiones con pérdida de información.

El comportamiento de la conversión explícita depende del tipo de datos originales y del tipo de datos de destino.

En el ejemplo siguiente se muestra conversiones de datos implícitas y explícitas, usando valores **entero**, **string**, **caracter** y **real**.

```
Entero A;  Real D;  Caracter C;  String pal;      // se declaran las variables indicando su tipo de dato
A = 5;       // a la variable A se le asigna el valor entero 5
D =  i;      // a la variable real D se le asigna el entero 5, el cual cambia por conversión implícita al valor 5,0
C = 'a';     // a la variable C se le asigna el carácter 'a'
pal = "los árboles " + C;        // a pal se le asigna la unión de una cadena y un carácter, resultando, "los árboles c"
                                 // hay conversión implícita de datos por que el carácter C se convierte al concatenar
                                 // en string, por lo tanto estamos concatenando con el + dos cadenas, no sumando
A = aEntero( D + 3,7 );  //  el resultado 8,2 (5,0 + 3,7) es convertido explícitamente de real a entero, asignando
                         //  a la variable A solo la parte entera del resultado, es decir A = 8 y perdiéndose la
                         //  parte decimal o 0,7
C = aCaracter(A);        //  transforma el número entero 8 a carácter, asignando entonces C = '8'
```

Para investigar y meditar

Consulta en guías de tus amigos, en apuntes de semestres pasados o investigando en libros:

1. ¿Cómo puedes representar la conversión de tipos en pseudocódigo?
2. ¿Cuáles son las funciones para conversión explícita de tipos en pseudocódigo?
3. Investiga que operadores utilizan los lenguajes de programación C++, Java y Visual Basic para los operadores aritméticos, lógicos y relacionales usados en la tabla de prioridad de operadores (tabla 1)
4. Verifica si el lenguaje de programación Java utiliza la misma prioridad de operadores que usaremos cuando escribamos algoritmos en pseudocódigo.

TEMA 3. ACCIONES ELEMENTALES

Objetivos:

- Conocer las operaciones elementales de asignación, lectura y escritura.

Puntos:

- Acciones elementales: Declaración, Asignación, Lectura, Escritura.
- Sintaxis y uso de las operaciones.
- Ejercicios.

3.1. Operación de Asignación

Es el modo de darle un valor a una variable, el cual puede ser una constante, otra variable o el resultado de una expresión.

En pseudocódigo la operación de asignación se representa mediante el símbolo u operador **=** para la asignación.

NOTA: en semestres anteriores se ha utilizado el operador ← para indicar la asignación, pero ahora usamos el símbolo =

En el contexto de un lenguaje de programación, a la operación de asignación, se le llama *instrucción* o *sentencia* de asignación.

Comportamiento: Modifica el estado de la variable.

La notación algorítmica (sintaxis) que utilizaremos para la asignación es:

<nombre de variable> = < constante o variable o expresión >;

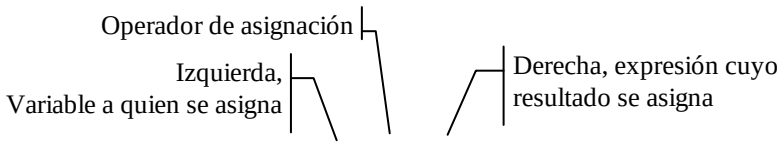
Ejemplo:

Sean las siguientes variables:

```
Caracter C1;        Entero I;        Lógico Enc;
```

// a continuación se le asignaran valores a las variables

```
C1 = 'a';              // a la variable C1 se le asigna o se le da el valor 'a'
Enc = (1 > 500);       // a la variable Enc se le asigna Falso, ya que no se cumple que 1 > 500
I = 5 + 12 div 3;      // a la variable I se le asigna el valor 9 ( I = 5 + 12 div 3= 5 + 4 = 9)
```



Reglas de la asignación: sea la expresión: `b = c * 2,45`

- Una variable en el lado derecho de una sentencia de asignación debe tener un valor antes de ser usado, en el caso de la anterior la variable **c** debe tener un valor inicial antes de evaluar la operación c * 2,45. Por ejemplo, si el valor inicial de **c** = 3, la expresión c * 2,45 arroja como resultado 7,35.
- Por el contrario, si a la variable **c** no le damos un valor antes de usarla, la expresión no puede ser resuelta y por lo tanto no tiene resultado, ya que una de sus variables (**c**) no está inicializada.
- A la izquierda de una sentencia de asignación sólo pueden existir variables o constantes, por lo tanto la expresión Valor – Tasa = Interes * 3,15 **es un error**, ya que a la izquierda no hay una variable sino la expresión Valor –Tasa. La expresión sería correcta rescrita como: Valor = Interes * 3,15 + Tasa;

3.2. Operación de Lectura o Entrada Simple (Leer – *read*)

La operación de **lectura o de entrada de datos** permite leer valores y asignarlos a constantes o a variables.

Almacena en una variable un valor extraído desde un dispositivo externo, del cual hacemos abstracción aunque generalmente es el teclado cuando se está programando. En pseudocódigo usamos la acción Leer para obtener los datos que nos suministra el usuario del algoritmo, datos necesarios para el procesamiento o cálculo posterior.

Los datos de entrada se introducen en el computador mediante dispositivos de entrada (teclado, pantalla, unidades de disco, escáneres, entre otros). Los datos de salida pueden aparecer en un dispositivo de salida (pantalla, impresora, cornetas, entre otros).

Comportamiento: La acción elemental Leer cambia el valor en la variable o variables que se está usando en la instrucción de lectura. Luego de leer un valor, el **valor de la variable cambia** en forma similar a si se hiciera una asignación.

La notación algorítmica (sintaxis) que utilizaremos para la asignación es:

Leer(<Nombre de variable>);

Ejemplo: sean las variables:

```
Entero I;        Real X, Y        Caracter c;
```

Acciones de lectura:

```
Leer(I);         Leer(X, Y);      Leer(c);
```

NOTA: Cuando vamos a leer datos suministrados por el usuario, la acción Leer suele trabajar en conjunto con la acción Escribir, ya que primero le tenemos que informar al usuario que datos necesita el algoritmo o que datos le estamos solicitando, para luego leerlos, por ejemplo:

```
Escribir("Suministre la edad de los 2 estudiantes");  Leer(edad1, edad2);
```

Respetando la secuencia lógica de acciones, primero le decimos al usuario que datos estamos necesitando y luego los leemos.

3.3. Operación de Escritura o Salida Simple (Escribir – *write*)

Permite mostrar el valor de una variable, constante o expresión. Cuando estamos programando la acción Escribir trasmite un valor a un dispositivo externo, por ejemplo lo muestra por la pantalla del computador. En pseudocódigo la acción Escribir nos permite mostrar mensajes o mostrar los resultados al usuario del algoritmo.

Comportamiento: muestra mensajes y resultados almacenados en variables, constantes o expresiones.

La notación algorítmica (sintaxis) que utilizaremos para la asignación es:

Escribir(< Nombre Variable, Constante o Expresión>);

Ejemplo:

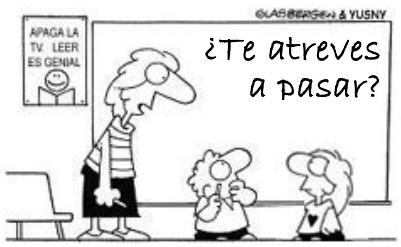
```
Entero i;   // se declara la variable
i = 4;      // a la variable i se le asigna el valor 4
Escribir("El valor de la variable i es: " + i );      // este escribir muestra un mensaje equivalente a:
                                                      //          El valor de la variable i es: 4
```

Ejercicios para Consolidar:

1. Diseña algoritmos que resuelvan los siguientes problemas:
   1.1. Calcular el promedio de 4 temperaturas medidas en ºC (Celsius) y mostrar su equivalente en ºF (Fahrenhei) y considerando:

$$^{o}C = \frac{^{o}F - 32}{1{,}8}$$

   1.2. Calcular un descuento del 25% para el precio de un curso y mostrarle al usuario cuanto se ahorra y cuanto va a pagar
   1.3. Convertir una cantidad de segundos, suministrados como un valor entero positivo, a minutos y a horas.



2. Traduce las siguientes expresiones matemáticas en forma de expresiones algorítmicas:

1. Área_Círculo = Π r ^{2} donde Π o PI tiene un valor de 3,1416
2. $r = \sqrt{(x-a)^2 + (y-b)^2}$
3. $x^2 + y^2 - 2ax - 2by + a^2 + b^2 - r^2 = 0$
4. $\sqrt{(x+c)^2 + (y-0)^2} + \sqrt{(x-c)^2 + (y-0)^2} = 2a$
5. $\frac{(x-p)^2}{a^2} + \frac{(y-q)^2}{b^2} = 1$
6. $A = \left(\frac{b1 + b2}{2}\right)h$

- AGUILAR, Luis Joyanes. **"Programación en C++. Algoritmos, Estructuras de Datos y Objetos"**. Editorial McGraw-Hill, 2000. **Capítulo 3, Prioridad y Expresiones en C++. Páginas 65-79**
- DEITEL & DEITEL. **"Cómo Programar en Java"**. Editorial Pearson Prentice Hall, 2004. **Anexos A y B, , páginas 1191-1193**
- **RECOMENDACIÓN:** toma nota y guarda las referencias de los libros o sitios Web que consultas, así ubicarlos fácilmente y usarlos en futuras consultas.

TEMA 4. ESTRUCTURAS DE CONTROL CONDICIONAL SI Y SELECCIÓN

Objetivos:

- Conocer la sintaxis, semántica y uso de las principales estructuras de control básicas:
  - Secuenciamiento de Instrucciones,
  - Condicional simple, Condicional doble, Condicional anidado.
  - Selección múltiple.

Puntos:

Las estructuras de control o formas de composición de acciones, son los mecanismos mediante los cuales es posible construir nuevas acciones a partir de otras. En este curso estudiaremos las siguientes:

- Secuenciamiento
- Condicional simple, Condicional compuesto, Condicional anidado y Selección múltiple

4.1. Secuenciamiento

Consiste en presentar un conjunto de instrucciones en un orden. Las instrucciones se realizan en el mismo orden en que son escritas. El símbolo **;** se utiliza para separar una instrucción de otra.

Notaciones algorítmicas (sintaxis):

Notación 1:

```
<instrucción 1>;
< instrucción 2>;
        …
< instrucción n>;
```

Notación 2:

```
< instrucción 1>;  < instrucción 2>;  …  < instrucción n>;
```

Comportamiento: representa las acciones del algoritmo y el orden en que se realizarán

Las instrucciones (<instrucciones 1> a la < instrucciones n>) no solo tienen que ser operaciones básicas, también pueden ser llamadas a acciones o funciones que realizan un procedimiento, estructuras de control de programa (Si, Selección) o estructuras iterativas (Para, Mientras, Repetir), las cuales serán explicadas en próximos temas.

Ejemplo 1:

```
AlgoritmoPrincipal
    // Secuencia de instrucciones que inicializa un número, le suma 3 y le calcula la raíz cuadrada
        Entero i;
        i = 1;                          // la variable i toma el valor de 1
        i = i + 3;                      // la variable i toma el valor de 4 (1 + 3)
        i = i ^ (1/2);                  // la variable i toma el valor de 2 (raíz cuadrada de 4)
        Escribir("El valor fin de la variable i es:  " + i);        // se escribe el mensaje de resultados

FinAlgoritmoPrincipal;    // Fin de la secuencia de instrucciones
```

4.2. Condicional Simple: Si – Entonces – Fsi (*If – Then – EndIf*)

El condicional simple **Si – Entonces – Fsi** ejecuta un conjunto de instrucciones si cumple la condición evaluada en el Si.

Sintaxis y Comportamiento:

En la instrucción **Si** <condición> **Entonces** <instrucciones S1>; **FSi;** se evalúa la <condición> y:

- Si la condición es verdadera, ejecutan o realizan las instrucciones del bloque S1
- Si la condición es falsa, no se realiza ninguna instrucción del Si

Recomendación: Cuando escribas las instrucciones en los algoritmos utiliza indentación o sangría para mostrar los niveles de anidación en las instrucciones. En este caso agregar un sangrado a la derecha para el bloque de acciones a fin de diferenciarlo de las instrucciones **Si - Entonces - FSi**

Ejemplo 2: Escribir un algoritmo que aumenta Bs. 150 al sueldo de las personas que ganan hasta Bs. 605

```
AlgoritmoPrincipal
    // Procedimiento para calcular el monto de un aumento de sueldo
        Real S;
        Escribir("Suministre el del sueldo de la persona"); // se pide el valor del sueldo al usuario
        Leer(S);                                  // lee en S el sueldo actual del empleado
        Si  S ≥ 0  y  S ≤ 605  Entonces           // el condicional compara el sueldo del empleado
Sangría →   S = S + 150;                          // aumenta el sueldo del empleado
            Escribir( "El nuevo sueldo es: " + S );   // mensaje con el resultado
        Fsi;
FinAlgoritmoPrincipal;   // Fin de la secuencia de instrucciones
```

4.3. Condicional Compuesto: Si – Entonces – Sino – Fsi (*If – Then – Else – EndIf*)

Permite elegir entre dos opciones o alternativas posibles, en función de que se cumpla o no la condición expresada en el Si.

Sintaxis y Comportamiento:

En la instrucción **Si** <condición> **Entonces** <instrucciones S1>; **Sino** <instrucciones S2>; **FSi;** se evalúa la <condición> y:

- Si la condición es verdadera, ejecutan o realizan las instrucciones del bloque S1
- Si la condición es falsa, se ejecutan o realizan las instrucciones del bloque S2

Recomendación: utilizar indentación o sangría en los algoritmos y códigos.

Ejemplo 3: Escribe una secuencia de instrucciones que te permitan aumentar Bs. 150 al sueldo de las personas que ganan Bs. 605 o menos, y aumentar en Bs. 100 a las personas que ganan más de Bs. 605

```
AlgoritmoPrincipal
    // Procedimiento para calcular un aumento de acuerdo al sueldo del empleado
        Real S;
        Escribir("Suministre el monto del sueldo de la persona"); Leer(S);
        Si  S ≤ 605 Entonces   S = S + 150;
        Sino
            S = S + 100;
```

```
        FSi;
        Escribir("El nuevo sueldo del empleado es: " + S);        // se escribe mensaje y el nuevo sueldo
FinAlgoritmoPrincipal;   // Fin de la secuencia de instrucciones
```

4.4. Condicional Anidado

Permite incluir dentro del cuerpo de la instrucción Si a otras instrucciones Si simples o compuestas. Se utiliza para elegir entre varias opciones o alternativas en función del cumplimiento o no de las diferentes condiciones que se van verificando en cada instrucción Si.

Comportamiento: en cada condicional se cumple el mismo comportamiento que se ha indicado para el condicional simple o anidado.

Recomendación: utilizar indentación o sangría en los algoritmos y códigos. En este caso agregar sangría para diferenciar el nivel de las instrucciones de cada bloque SI.

Ejemplo 4: Crea un algoritmo que calcule el monto de un aumento a partir del sueldo del empleado y su antigüedad (cantidad de años trabajados en la empresa). Para las personas que ganan hasta Bs. 500, se les aumenta 15% si tienen menos de 5 años de antigüedad en la empresa y 20% si tienen 5 o más años. Para los que ganan más de Bs. 500 y hasta 2.500 se les aumenta 5% por cada 5 años de antigüedad.

```
AlgoritmoPrincipal
    // Procedimiento para aumentar el sueldo según el monto del sueldo actual y los años de servicio
        Real S, Aumento;   Entero Años, Aux;
        Escribir("Suministre el sueldo y los años de antigüedad"); Leer(S, Años); //solicitan y leen datos

        Si  S < 0  o Años < 0  Entonces                  // los valores de entrada son inválidos
            Escribir ("Verifique el sueldo o los años de servicio, ambos deben ser valores positivos");

        Sino
            Si S ≤ 500 Entonces                          // contempla sueldo menores o iguales a 500

                // este SI anidado contempla la antigüedad para los que ganan hasta Bs. 500
                Si Años < 5  Entonces                    // trabajadores con menos de 5 años
                    Aumento = S * 0,15;                  // se calcula el monto del aumento, usando 15%
                Sino
                    Aumento = S * 0,20;                  // se calcula el monto del aumento, usando 20%
                Fsi;
            Sino          // contempla los que ganan más de 500

                Si S > 500  y  S≤  2.500 Entonces  // ganan más de 500 y hasta 2.500
                    Aux = Años Div 5;     // calcula cuantas veces se le aplicará el aumento del 5%
                    Aumento = S * (Aux * 5) / 100;                  // se calcula el monto del aumento
                Sino
                    Aumento = 0;        // no se contempla aumento para los que ganan más de 2.500
                    FSi;
            FSi;
        FSi; // cerrando si de la validación
        Si Aumento == 0 Entonces
            Escribir("Sueldo actual: " + S + " gana más de 2.500, no le aplica aumento");
```

```
        Sino
            Escribir("Sueldo anterior: " + S  + " y tiene una antigüedad de: " + Años );
            Escribir("Recibe aumento de: " + Aumento  +  " y su nuevo sueldo es: " +
(Sueldo+Aumento) );
        FSi;
FinAlgoritmo Principal;  // Fin de la secuencia de instrucciones
```

4.5. Selección Múltiple: Selección – Sino – FSelección (*Select – End Select*)

La selección múltiple permite evaluar una condición o expresión que puede tomar muchos valores distintos. Se ejecutaran las instrucciones correspondientes al caso que se cumple. La principal ventaja de la estructura Selección es que permite crear algoritmos que sean legibles y evitar la confusión creada por el anidamiento de muchos bloques SI.

Comportamiento: se evalúa cada condición de la selección y se realiza el bloque de instrucciones correspondientes a la condición que se cumple.

Ejemplo 5: Una librería ofrece descuentos en libros con fecha de publicación del año 2009 o anterior. Para aquellos libros entre 2009 y 2005 se da un descuento del 10%, para los que están entre 2004 y 2000 descuento del 22%, para los que están entre 1999 y 1994 descuento del 35,5%, para los que están entre 1993 y 1985 descuento del 50%, para los anteriores a 1985 se da 70% de descuento.

Crea una secuencia de instrucciones que a partir del año de publicación y el precio del libro permita saber el monto del descuento y el precio final de venta. Todos los libros pagan un IVA de 1%. Considera también que los precios no pueden ser negativos y que la librería no tiene libros anteriores a 1975 (estos casos serían considerados errores).

```
Algoritmo Principal;
    // Procedimiento para calcular el descuento aplicado a un libro según el año de publicación

        String Nombre;     // guardará el nombre del libro
        Real Precio, PorcentajeDesc, MontoDesc, MontoIVA, PrecioFinal;  // variables para el precio y
        Entero AñoPub;     //  año de publicación del libro;

        Escribir("Suministre el nombre del libro, su precio y año de publicación");
        Leer(Nombre, Precio, AñoPub);    // se solicitaron y ahora leyeron los datos de entrada

        Si Nombre == "" o  Precio < 0  o  AñoPub < 1975  o  AñoPub > 2009 Entonces
            // se validan los datos de entrada
            Escribir("Error en datos de entrada, el nombre, precio o año de publicación son inválidos");
        Sino
            // se determina el porcentaje de descuento para los libros publicados entre 1975 y el 2009
            Selección
                AñoPub > 2009:    PorcentajeDesc = 0%;
                AñoPub ≥ 2009  y  AñoPub ≤ 2005:    PorcentajeDesc = 10%;
                AñoPub ≥ 2004  y  AñoPub ≤ 2000:    PorcentajeDesc = 22%;
                AñoPub ≥ 1999  y  AñoPub ≤ 1994:    PorcentajeDesc = 35,5%;
                AñoPub ≥ 1993  y  AñoPub ≤ 1985:    PorcentajeDesc = 50%;
                AñoPub ≥ 1975  y  AñoPub < 1985:    PorcentajeDesc = 70%;
            FSelección;

            // se calculan el descuento, el impuesto y el precio final
```

```
        MontoDesc = Precio *  PorcentajeDesc;
        MontoIVA = ( Precio -  MontoDesc ) * 1%;
        PrecioFinal = ( Precio -  MontoDesc ) + MontoIVA;

        // se muestran los resultados o datos de salida del algoritmo
        Escribir(“Libro vendido: ” + Nombre  );
        Escribir(“El precio inicial del libro es Bs.: ” + Precio + “ y es del año: ” + AñoPub  );
        Escribir(“Porcentaje de descuento: ” + PorcentajeDesc );
        Escribir(“Monto descuento Bs.: ” + MontoDesc );
        Escribir(“Monto del IVA Bs.: ” + MontoIVA );
        Escribir(“Precio final del libro Bs.: ” + PrecioFinal );

    FSi;

FinAlgoritmo Principal;
```

Ejercicios para consolidar

Crea algoritmos que resuelvan los siguientes problemas:

1. Para un valor entero positivo Seg que representa una cantidad en segundos, indicar su equivalente en minutos, horas y días
2. Solicitar un valor entero que representa un año e indicar si se trata de un año bisiesto
3. Dada una letra cualquiera indicar si es consonante, vocal o un dígito
4. Determinar si un dígito es par o impar
5. Determinar el máximo de 2 valores
6. Calcular el valor absoluto de un número N
7. Determinar el signo de la suma de dos números, sin calcularla
8. Solicitar un número entero de 4 dígitos significativo y descomponerlo para mostrar unidades de mil, centenas, decenas y unidades
9. Indicar si un alumno aprobó o no una materia conformada por 2 parciales (que representan el 50% de la definitiva), 2 quices (30% de la definitiva) y 2 proyectos (el primero 10% y el segundo 20%). En el caso de que su definitiva sea mayor o igual a 9 pero menor a 9,5, indicarle que debe asistir a actividades de recuperación. Si la nota es menor a 9 el alumno está reprobado.

IMPORTANTE: recuerda incluir todas las validaciones que consideras necesarias sobre los datos de entrada y siempre mostrar los resultados del algoritmo

TEMA 5. ESTRUCTURAS DE CONTROL ITERATIVAS

Objetivos:

- Conocer la sintaxis, comportamiento y utilidad de las principales estructuras iterativas utilizadas en programación.

Puntos:

- Bucles o ciclos
- Para
- Repetir
- Mientras

5.1. Bucles o Ciclos

Un **bucle, lazo, ciclo o *loop*** (en inglés) es un segmento de algoritmo o programa (una secuencia de instrucciones) que se repiten un determinado número de veces mientras se cumple una determinada condición, en otras palabras, un bucle o ciclo es un conjunto de instrucciones que se repiten mientras una condición es verdadera o existe.

A cada repetición del conjunto de acciones se denomina **iteración**.

Para que un bucle no se repita indefinidamente debe tener una condición de **parada o de fin**. Esta **condición de parada** o de fin se verifica cada vez que se hace una iteración. El ciclo o *loop* llega a su fin cuando la condición de parada se hace verdadera.

La condición de parada puede estar al principio de la estructura repetitiva o al final.

Al igual que las estructuras de selección simple o compuesta (los bloques si – entonces – sino – fsi), en un algoritmo pueden utilizarse varios ciclos. Estos **ciclos pueden ser independientes** (una a continuación de otro) **o anidados** (ciclos dentro de ciclos).

Para representar los bucles o lazos utilizaremos en el curso las estructuras de control **Para, Repetir y Mientras**.

¿Para qué me sirve esto?

Los ciclos o procesos repetitivos de instrucciones son procedimientos fundamentales en el uso de las computadoras y por lo tanto en muchos algoritmos y programas.

Vamos a utilizar ciclos cuando:

- Necesitemos **REPETIR INSTRUCCIONES** un determinado número de veces, mientras se cumpla una condición, mientras un hecho sea verdadero o hasta cuando se alcance un determinado valor o condición.
- Cuando necesitemos **CONTAR** un determinado número de elementos o de acciones, por ejemplo contar las sílabas de un texto, los elementos de una secuencia que verifican una determinada condición o contar el número de personas que cumplen ciertas características. En este caso se incluirán **contadores** dentro del bucle.

  Los **contadores son variables** (generalmente de tipo **Entero**) que tienen un valor inicial y que se incrementan o decrementan en un valor constante cada vez que ocurre una iteración. Cuando los contadores se decrementan se habla de descontar, en lugar de contar.
- También usaremos ciclos cuando necesitemos **ACUMULAR** o **TOTALIZAR** terminados valores cada vez que se realiza una iteración. Los **acumuladores también son variables** (generalmente de tipo **Entero**, **Real** o **String**), que almacenan valores variables resultantes de las operaciones que se realizan en cada ciclo.

  Por ejemplo, podemos usar para ir sumando los precios de varios vehículos y luego calcular el precio

promedio general (con una variable acumulador de tipo **real**), para calcular la potencia o el factorial de un número a través de multiplicaciones sucesivas (un acumulador de tipo **entero** o **real**) o para ir agregando a una cadena de caracteres, letras o sílabas que construirán un mensaje (una variable acumulador del tipo **string**).

- ¿Recuerdas los ejercicios donde hemos necesitado repetir un grupo de acciones y no podíamos? Ahora los ciclos te permitirán repetir esos bloques de instrucciones.

5.2. Estructura Iterativa Para ... FinPara *(For ... Next ... End For)*

Es una estructura iterativa que es controlada por una variable (llamada también **variable índice**), la cual se incrementa o decrementa hasta llegar a un valor límite o valor final que representa la condición de parada.

La estructura **Para** comienzan con un valor inicial de la variable índice, las acciones especificadas para el ciclo se ejecutan un número determinado de veces, a menos, que el valor inicial de la variable índice sea mayor que el valor límite que se quiere alcanzar.

🛠 **SE RECOMIENDA USARLO:** la estructura **Para** es recomendada cuando se conoce el número de veces que se deben ejecutar las instrucciones del ciclo, es decir, en los casos en que el número de iteraciones es fijo y conocido.

El incremento o decremento de la variable_índice suele ser de 1 en 1, salvo cuando se indica lo contrario. La variable índice suele ser de tipo **Entero** y se utilizan comúnmente nombres como i, j o k (no importa si en mayúsculas o minúsculas)

✎ **SINTAXIS del Para:**

Para variable_índice = valor1 **hasta** valor2 **en** inc/dec **hacer** <inst 1> … <inst n> **FinPara;**	En este ciclo la *variable_índice* se inicializa con *valor1*. El ciclo continuará hasta que la *variable_índice* sobrepase a *valor2*. *inc* es la unidad en que se incrementa o decrementa la *variable_índice* (generalmente el incremente es en 1)

5.3. Estructura Iterativa Repetir - Hasta (*Repeat – Until*)

Ejecuta un bloque de instrucciones varias veces hasta que se cumple la condición que es verificada al final del bucle.

Las instrucciones dentro del ciclo **Repetir** se van a realizar mientras la condición de parada evaluada al final sea falsa. Dicho de otro modo, el ciclo se va a detener cuando la condición de parada se haga verdadera.

✎ **SINTAXIS del Repetir:**

```
Repetir
    < instrucción 1>;
    ...
    < instrucción n>;
    <actualización variable(s) usada(s) en la cond. parada>;
Hasta <expresión_condición_de_parada>;
```

🛠 **SE RECOMIENDA USARLO:** la estructura **Repetir** es recomendada cuando las instrucciones del ciclo se pueden

realizar **al menos 1 vez antes** de comprobar la condición de parada.

5.4. Estructura Iterativa Mientras – Hacer – FinMientas (*While ... Do ... EndWhile*)

Es una estructura iterativa que permite verificar la condición de **entrada** al ciclo **antes** del cuerpo de instrucciones a repetir.

Como la evaluación de la condición de entrada se realiza al **inicio** del bloque **Mientras**, puede ocurrir que las instrucciones del ciclo no se realicen ni siquiera 1 vez, a diferencia del **Repetir**, donde el bloque de instrucciones se realiza al menos 1 vez porque la condición de parada se verifica al final. Las instrucciones del Mientras se pueden realizar 0 o más veces antes de que se cumpla la condición de terminar el ciclo.

El conjunto de instrucciones dentro del **Mientras – FinMientras** se ejecuta cuando la condición de entrada del principio se cumple (es verdadera). Dicho de otro modo, el ciclo de instrucciones dentro del **Mientras** se va a detener cuando la condición se haga falsa.

SINTAXIS del Mientras:

Mientras <expresión_condición_de_entrada> **hacer**

<instrucción 1>;

....

< instrucción n>;

<actualización variable(s) usada(s) en la cond. entrada>;

FinMientras;

SE RECOMIENDA USARLO: la estructura **Mientras** es recomendada cuando tienes que verificar la condición de entrada al inicio y si se cumple, entonces, entrar al ciclo y realizar sus instrucciones.

IMPORTANTE ⊙j⊙

- Observa que para el **Repetir** -**también para el Mientras**- debes incluir en el cuerpo de instrucciones una acción donde se actualice la o las variables que usas en la condición de parada. Por el contrario, en el **Para** no es necesario incluir esta instrucción ya que en el encabezado de la instrucción (Para i==1 hasta X en 1 hacer) ya se incluye el incremento.

¿CUÁL USAR? CON EJEMPLOS:

1. Si la condición de parada del ciclo depende de que hayas repetido 30 veces el ciclo:
   - Usas un bloque **Para** con la sintaxis: Para i = 1 hasta 30 en 1 hacer

     No necesitas una instrucción dentro del **Para** que incremente el valor de la variable i, ya que la instrucción "en 1" del encabezado del **Para** se encarga de incrementar y actualizar esta variable índice.
2. Si la condición de parada depende de que hayas repetido 30 veces el ciclo o de que leas la palabra "detener":
   - No deberías usar un **Para**, porque no necesariamente vas a repetir el ciclo 30 veces. Si el usuario te suministra en alguna repetición del ciclo la palabra "detener" saldrías del ciclo aunque no lo hayas repetido 30 veces.

     Además en la sintaxis del **Para** NO PUEDES verificar que la palabra leída sea la "detener"
   - En el Para **NO PUEDES COLOCAR** algo como Para i = 1 (hasta 30 o palabra ="detener") en 1 hacer
   - Usas un bloque **Repetir** o un bloque **Mientras** que **sí te permiten verificar en una expresión** las condiciones de que se haya repetido 30 veces el ciclo o que la palabra suministrada sea "detener"

- Cuando uses un Mientras o un Repetir, debes incluir instrucciones dentro del ciclo que actualicen las variables de la

condición de parada, para este ejemplo, debería tener una instrucción que aumente el valor de i (i = i + 1) y otra que te permita leer un nuevo valor para la variable palabra (Escribir("¿desea detenerse?") Leer(palabra)).
Tendrías instrucciones equivalentes a:

```
AlgoritmoPrincipal
// verifica si se suministra una 'a' antes de 30 intentos
    Entero cont;
    Caracter letra;
    cont = 0;   // inicializo en 0 el contador de letras

    Repetir
        Escribir("suministre un valor para la letra");
        Leer(letra);
        cont = cont  + 1;
        Escribir("estamos en la iteración " + cont);
    Hasta letra == 'a'  o   cont == 30;

    Si letra == 'a' Entonces
        Escribir("encontré la letra a en la iteración " + cont);
    Sino
        Escribir("repetí 30 veces el ciclo y no leí una A");
    Fsi;
// fin de las instrucciones
FinAlgoritmoPrincipal
```

```
AlgoritmoPrincipal
// verifica si se suministra una 'a' antes
// de 30 intentos
    Entero cont;   Carácter letra;
    cont = 1;
    // solicitamos y leemos la primera letra
    Escribir("suministre la letra inicial");  Leer(letra);

    // verificará la condición a la entrada del mientras
    Mientras letra ≠ 'a' y cont ≤ 30 hacer
        Escribir("estamos en la iteración " + cont);
        Escribir("suministre nuevo valor para letra");
        Leer(letra);
        cont == cont + 1;
    FMientras;

    Si letra == 'a' Entonces
        Escribir("encontré la  A  en la iteración " +
cont);
    Sino
        Escribir("repetí 30 veces el ciclo, no hay A ");
    Fsi;
// fin de las instrucciones
AlgoritmoPrincipal
```

EJEMPLOS

1. Se desea sumar 35 números (pueden ser enteros o reales) suministrados por teclado y calcular su promedio

```
Algoritmo Principal
// instrucciones para SumaryPromediar
    Real suma, promedio, num;     Entero j;
    suma = 0;

    Para j = 1 hasta 35 en 1 hacer
        Escribir("Suministre el nuevo número ");
        Leer(num);
        suma = suma + num;
    FPara;
    promedio = suma / ( j - 1);
    Escribir("La suma de los números es: " + suma);
    Escribir("El promedio de los números es: " + promedio);
FinAlgoritmo Principal;
```

2. Se desea conocer cuantas veces se repiten en un texto que el usuario suministra letra por letra y que termina con **‘*’**

```
Algoritmo Principal
// instrucciones para contar las vocales en una
// secuencia de letras

    Entero cantA, cantE, cantI, cantO, cantU;
    Caracter letra;

    // se inicializan las variables contador
    cantA=0; cantE=0; cantI=0; cantO=0; cantU=0;

    Escribir (“suministre la primera letra”);   Leer(letra);

    Mientras letra ≠ ‘*’ hacer
        Selección
            letra =’a’: cantA = cantA + 1;
            letra=’e’: cantE = cantE + 1;
            letra=’i’: cantI = cantI + 1;
            letra=’o’: cantO = cantO + 1;
            letra=’u’: cantU = cantU + 1;
        fSelección;

        // actualizo la variable de la condición de parada
        Escribir (“suministre próxima letra”); Leer(letra);

    FMientras;

    Escribir(“La cantidad de vocales A es: ” + cantA);
    Escribir(“La cantidad de vocales E es: ” + cantE);
    ...
    Escribir(“La cantidad de vocales U es: ” + cantU);

FinAlgoritmo Principal;
```

3. Calcule el promedio general de notas de un grupo de alumnos para una materia. Se sabe que la materia tiene al menos un alumno. El usuario le indicará cuando terminó de suministrar todas las notas.

```
Algoritmo Principal
// instrucciones para calcular el promedio
// general de notas de un grupo de alumnos sin
// conocer a priori cuantas notas serán

    Real nota, promedio;
    Entero numAlumnos;
    String continuar;
    promedio = 0;  numAlumnos = 0;
    continuar = “Si”;

    Repetir

        // solicitamos y leemos la nota hasta que
        // sea válida, es decir, esté entre 0 y 20 puntos
        Repetir
          Escribir(“Suministre la nota del alumno”);
          Leer(nota);
        Hasta nota >= 0  y  nota <= 20;

        promedio = (promedio + nota);
        numAlumnos = numAlumnos + 1;

        // actualizamos la variable que nos indica
        // si hay que detener el ciclo
        Escribir(“¿Desea suministrar otra nota?
                  Indique Si ó No”);
        Leer(continuar);

    Hasta  continuar == “No”;

    promedio = promedio / numAlumnos;

    Escribir(“El promedio general es: ”  + promedio );

FinAlgoritmo Principal;
```

TEMA 6. PROCEDIMIENTOS

Objetivos:

- Conocer la sintaxis utilizada para la representación de acciones, funciones y parámetros.
- Conocer los tipos de pase de parámetro y sus implicaciones.
- Conocer el alcance de las variables que se declaran dentro el bloque de instrucciones de un procedimiento.

Puntos:

- Procedimientos • Acciones • Funciones • Pase de Parámetros • Alcance de Variables

6.1. Procedimientos

La definición de procedimientos permite asociar un nombre a un bloque de instrucciones. Luego podemos usar ese nombre para indicar en algún punto de un algoritmo que vamos a utilizar ese bloque de instrucciones, pero sin tener la necesidad de repetirlas, sólo **invocando** al procedimiento por su nombre.

Los procedimientos pueden ser clasificados en acciones o funciones. Las **acciones** se caracterizan por **no retornar** valores al algoritmo que las llama, mientras que las **funciones retornan un** valor. Sin embargo, aunque las acciones no retornan valores, si pueden informar al algoritmo que las llamó (a veces llamado **algoritmo principal**) de cambios realizados por sus instrucciones en algunos valores a través de una herramienta que se llama **pase de parámetros**. Los **parámetros** permiten utilizar la misma secuencia de instrucciones con diferentes datos de entrada. Utilizar parámetros es opcional.

Cuando entre las instrucciones de un algoritmo vemos el nombre de un procedimiento (acción o función), decimos que estamos **llamando o invocando** al procedimiento.

Los procedimientos facilitan la programación modular, es decir, tener bloques de instrucciones que escribimos una vez pero que podemos llamar y utilizar muchas veces en varios algoritmos. Una vez terminada la ejecución de un procedimiento (acción o función), se retorna el control al punto de algoritmo donde se hizo la llamada, para continuar sus instrucciones.

En el próximo tema, Programación Orientada a Objetos, veremos que los procedimientos son denominados **Métodos**.

6.2. Acciones

Conjunto de instrucciones con un nombre que pueden ser llamadas a ejecución cuando sea necesario. No retornan valores.

Sintaxis de la definición formal de la Acción

```
Acción <Nombre> [ (lista parámetros formales) ]
        // comentario sobre lo que la acción hace
        < definición de variables locales >
        < instrucción 1 de la acción >
        …
        < instrucción n de la acción >
FAcción <Nombre>;
```

Donde:

Acción … FAcción delimitan el inicio y fin de la acción.

<Nombre> es el identificar de la acción.

[<lista de parámetros formales>] son objetos o variables que utilizan las instrucciones dentro de la acción, pero que no tienen valor hasta que la acción no es llamada y utilizada. Esta lista es opcional por eso aparece entre corchetes.

<definición de las variables locales> es el conjunto de valores que se usan dentro de la acción.

<instrucciones> a veces también llamadas simplemente acciones, es la secuencia de instrucciones a ser ejecutadas por la acción.

Ejemplo 1: Acción sin pase de parámetros

Acción CalcularPrecio
// a partir de un precio neto suministrado por el usuario, calcula e informa el monto del IVA y el precio final
String Nombre; Real Precio, MontoIva, PrecioFinal;
Escribir("indique el nombre y el precio neto del artículo"); Leer(Nombre, Precio);
MontoIva = Precio * 0,14; PrecioFinal = Precio + MontoIva;
Escribir("Artículo: " + Nombre);
Escribir("Precio Bs. " + Precio + " IVA Bs. " + MontoIva + " Precio Final Bs. " + PrecioFinal);
FAcción CalcularPrecio;

6.3. Funciones

Al igual que las acciones con conjuntos de instrucciones con un nombre, pero se caracterizan por **retornar** (enviar o devolver) **un valor** al algoritmo que la llama.

Como el resultado de la función es retornado al algoritmo principal, debe usarse una variable para almacenar este resultado, es decir, en una variable del algoritmo principal se "captura" el valor retornado por la función. Luego el valor almacenado en la variable puede ser utilizado por el algoritmo que llama a la función.

Sintaxis de la definición formal de la Función

Función <Nombre> [(lista parámetros formales)] **:** <tipo dato retorno>
// comentario sobre lo que la función hace
< definición de variables locales >;
< instrucciones de la función >;
retornar (<variable, constante o expresión compatible
con el tipo de retorno >);
FFunción <Nombre>;

Donde:

Función … FFunción delimitan el inicio y fin de la función.

<Nombre> es un identificador (nombre) válido

: operador usado para expresar que la función retornará un valor del tipo de dato que se indica luego.

Esta lista es opcional por eso aparece entre corchetes.

<tipo de dato de retorno> indica el tipo de dato del valor que la función retornará al algoritmo que la llamó (por ejemplo, podría retornar un entero, ó un string, ó un objeto, etc.)

retornar instrucción predefinida utilizada para indicar el lugar y el valor a retornar por parte de la función.

Ejemplo 2: Función sin pase de parámetros

Función CantidadPalabrasLos : Entero
// solicita al usuario palabras, verifica si se suministra la palabra "los" y en ese caso se cuenta.
// el algoritmo termina cuando se lea la palabra "fin"
String palabra; Entero CantLos;
CantLos = 0;
Escribir("A continuación se le solicitarán palabras. Suministre la palabra fin, para terminar el algoritmo");

```
Repetir
    Escribir("Suministre una palabra");  Leer(palabra);
    Si palabra == "los" Entonces
     CantLos = CantLos + 1;
    FSi;
Hasta palabra == "fin";

Retornar(CantLos);
    // esta instrucción retorna o devuelve la cantidad de palabras al algoritmo que llamo a esta función
FinFunción CantidadPalabrasLos;
```

6.4. Pase de Parámetros y Tipos de Parámetros

Parámetros actuales

Son los valores indicados en la llamada a la acción o función en el algoritmo principal. Son los valores que se desean pasar desde el algoritmo principal a las instrucciones de la acción o función.

Parámetros formales

Son los nombres dados a los parámetros en la definición formal de la acción o función. Con estos nombres se conocerán a los valores de los parámetros dentro de la acción o función y funcionan como variables locales dentro de ellos.

Sintaxis de los parámetros formales en la cabecera de la acción o función:

```
Acción <Nombre> ([Var] TipodeDato NombreParámetro; …)        // para las acciones

Variable_retorno = Función <Nombre> ([Var] TipodeDato NombreParámetro; …)  // para las funciones
```

En la definición formal debe especificarse por los parámetros el tipo de sustitución (por valor o por referencia), su tipo de dato y su nombre. Si el tipo de sustitución es por **referencia** se indica con la palabra **Ref** (también se estila en algunos lenguajes usar **Var** en lugar de **Ref**) antes del tipo de dato, si es por **valor**, no se coloca nada. Si varios parámetros tienen el mismo tipo de dato y todos son pasados por valor puede escribirse el Tipo de dato una sola vez y luego la lista de los nombres de los parámetros separadas por coma, por ejemplo, Entero x, y, z

Pase de parámetros por valor

El parámetro actual no es modificado si se modifica el parámetro formal dentro del procedimiento, ya que **ambos parámetros ocupan posiciones diferentes en memoria**. Esto se debe a que el parámetro actual se evalúa y el resultado se copia en el correspondiente **parámetro formal, que ocupa otra posición en memoria**. Por ello, cuando se regresa de la acción o función al algoritmo desde donde se hace la llamada los parámetros actuales mantienen su valor original.

Pase de parámetros por referencia

El parámetro actual sufre los mismos cambios que el parámetro formal. El parámetro actual no puede ser ni una constante ni una expresión. **Ambos parámetros ocupan la misma posición en memoria**. Por ello, cuando se regresa de la acción/función al algoritmo desde donde se hace la llamada los parámetros actuales han cambiado.

6.5. Llamada (o invocación) de las acciones o funciones

La llamada a una acción o función es una instrucción que permite la ejecución de la secuencia de sus instrucciones. Consiste en indicar el nombre y los parámetros actuales que van a ser utilizados.

Los parámetros actuales que se indican en la llamada de la acción (por ejemplo en el algoritmo principal), deben corresponderse con los parámetros formales de la definición de la acción o /función. Por ello, la cantidad de parámetros actuales debe ser igual a la cantidad de parámetros formales y del mismo tipo de dato.

Sintaxis de la llamada a la acción o función:

instrucciones_algoritmoPrincipal
...
<Nombre_Acción> ([lista de valores de los parámetros actuales]) // llamada para las acciones
Variable = <Nombre_Función> ([lista de valores de los parámetros actuales]) // llamada para las funciones

Ejemplos

Ejemplo 3: Acciones y funciones con pase de parámetros

Acción Principal

```
// Solicita una fecha de nacimiento y llama a acciones y funciones que hacen varios cálculos con ella
Entero diaNac, mesNac, añoNac, edad;   String Nombre;   Lógico esCorrecta, esBis, esMenorEdad;

Escribir("suministra tu nombre");   Leer(Nombre);

Repetir
    Escribir("suministra el día, mes y año de tu fecha de nacimiento");
    Leer(diaNac, mesNac, añoNac);
    EsCorrecta = ValidarFecha(diaNac, mesNac, añoNac);   // se llama a una función que verifica la fecha
Hasta esCorrecta == Verdadero;

esBis = ValidarBisieto(añoNac);   // se llama a una función que valida si el año de nacimiento fue bisiesto
Si esBis == Falso   Entonces
    Escribir("no naciste en un año Bisiesto")
Sino
    Escribir("Naciste en un año Bisiesto")
FSi;
Cumpleaños(nombre, diaNac, mesNac, añoNac);      // calcula cuando cumple años
edad = 0; esMenorEdad = Falso;
CalcularEdad(añoNac, edad, esMenorEdad);   // calcula la edad y si es o no menor de edad
Escribir(nombre + " tu tienes " + edad + " años");
```

FinAcción Principal;

// declaración formal de las acciones y funciones utilizadas en el algoritmo principal
// en muchos lenguajes de programación estas declaraciones formales deben estar ANTES del algoritmo que las llama.

Función ValidarFecha(Entero d, m, a) : Lógico
// verifica si el día (d), mes (m) y año (a) conforman una fecha válida
Lógico correcta;
correcta = Verdadero;

// completa las instrucciones faltantes

retornar(correcta);
FinFunción ValidarFecha**;**

Función ValidarBisiesto (Entero a) : Lógico
// verifica si el año (a) es bisiesto. Son bisiestos todos los años divisibles por 4, excluyendo los que sean
// divisibles por 100, pero no los que sean divisibles por 400.
retornar ((a mod 4 == 0) y ((a mod 100 ≠ 0) o (a mod 400 == 0));

FinFunción ValidarBisiesto**;**

Acción Cumpleaños(String N; Entero dia, mes, año)
// verifica si ya la persona cumplió años. Este algoritmo compara contra la fecha 04-06-2007

String Mensaje; Entero mesesFaltan;

Si año == añoActual() Entonces // 1er si, para la fecha de ejemplo añoActual() = 2007
Mensaje = "todavía no has cumplido el primer añito";
Sino
Si mes < mesActual() Entonces // ya cumplió años, para la fecha de ejemplo mesActual() = 6
Mensaje = " ya cumpliste años, felicitaciones atrasadas";
Sino
Si mes > mesActual() Entonces // ya cumplió años, 3er si
faltan = mes mod 12 + 1;
Mensaje = " faltan aproximadamente " + faltan + " para tu cumpleaños";
Sino
Si dia < diaActual() Entonces // ya cumplió años, 4to si, diaActual() = 4
Mensaje = " ya cumpliste años, felicitaciones atrasadas";
Sino
Si dia > diaActual() Entonces // cumple años muy pronto, 5to si
faltan = dia - 22;
Mensaje = " alégrate sólo quedan " + faltan + " días para tu cumple";

```
                    Sino
                        Mensaje = " cumpleaños Feliz !!! muchos deseos de salud y prosperidad en tu día";
                    FSi;  // cerrando si 5
                Fsi;  // cerrando si 4
            Fsi;  // cerrando si 3
        Fsi;  // cerrando si 2
    Fsi;  // cerrando si 1
    Escribir(N + Mensaje);
FinAcción Cumpleaños;

Acción CalcularEdad(Entero año; Ref Entero edad; Lógico menor)
    // Calcula la cantidad aproximada de años que tiene una persona
    edad = 2006 – añoNac;     menor == (edad < 18);
    Escribir("¿La persona es menor de edad? "  + menor);
FinAcción CalcularEdad;
```

Ejercicios Asignados

1. Para el ejemplo 3 completa:
   - La función ValidarFecha
2. Acción que muestre las tablas (del 0 al 10). Debe preguntar al usuario que tipo(s) de tabla que quiere generar (suma , resta, multiplicación o división) para que número (p.e., la tabla de sumar del 9, o la tabla de dividir del 2). El algoritmo se detiene cuando el usuario suministra el valor –99
3. Función que obtenga la suma de los números pares que hay entre dos números leídos por teclado. Utiliza esta acción para calcular la suma entre 10 pares valores que te dará el usuario en un algoritmo principal.
4. Escribe un acción que solicite las notas de un alumno, calcule su promedio, escriba esa nota y si aprobó o no.
5. Utiliza la acción anterior para calcular, en una sección de 15 alumnos, el promedio de nota de cada uno y si aprobó.

6.6. Ámbito o Alcance de Identificadores (Global, Local y No Local)

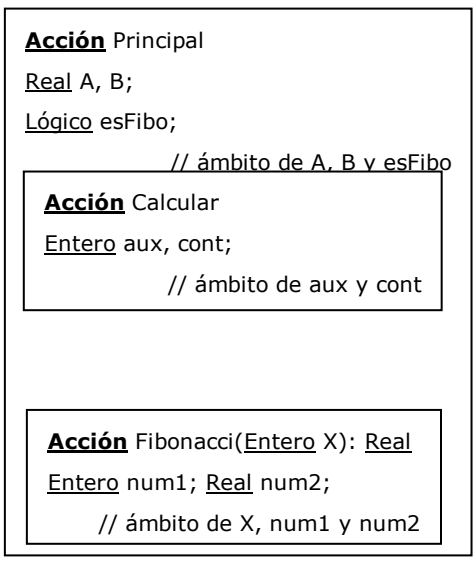
Ámbito o Alcance

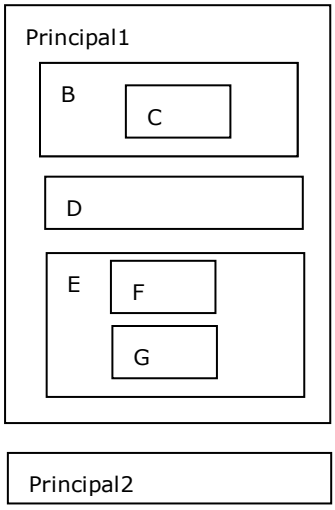
El ámbito o alcance de un identificador, es decir, el alcance de una variable, una constante, un parámetro o un procedimiento (sea acción o función) es el área dentro de un algoritmo o dentro de un programa donde ese identificador es conocido y puede ser usado. Clásicamente los identificadores se han clasificado según su alcance en **locales** y **globales**.

Veamos estas ideas a través de ejemplos:

- Si en un algoritmo o acción **Principal** definimos una acción (o función) llamada **Calcular** y dentro de esta acción (o función) definimos una variables con nombre **aux** y **cont**, se dice entonces, que las variables **aux** y **cont**, son locales o internas a la acción **Calcular**.
- Si en el mismo algoritmo definimos otra acción (o función) llamada **Fibonacci** que usa las variables **num1, num2** y tiene un parámetro llamado **X,** tenemos que éstos tres (3) identificadores sólo serán conocidos o pueden ser usados dentro del procedimiento **Fibonacci**.
- Las variables **num1** y **num2**, y el parámetro **X** son locales a Fibonacci por lo tanto no pueden ser usadas dentro de **Calcular**, igualmente, las variables **aux** y **cont** pueden ser usadas dentro de Calcular, pero no pueden ser alcanzadas desde Fibonacci.
- Si en la acción **Principal** se declaran las variables **A, B y esFibo**, estos identificadores con considerados variables globales. En esencia las variables globales con aquellas declaradas en la acción Principal del que dependen todos los subprocedimientos (en este caso Calcular y Fibonacci). Las variables A, B y esFibo están dentro del alcance o pueden ser usadas dentro de Calcular y dentro de Fibonacci.
- También podemos decir que las variables A, B, esFibo y los procedimientos Calcular y Fibonacci son locales o internos a la acción Principal.

Gráficamente podemos ver a continuación los límites del alcance de variables





Variables definidas en	Accesibles desde
Principal1 (*)	B, C, D, E, F, G
B	B, C
C	C
D	D
E	E, F, G
F	F
G	G
Principal2(*)	Conocidas dentro de

Variables Locales

Son los variables, constantes, parámetros declarados y definidos dentro de un procedimiento (acción o función). Incluso son locales los procedimientos definidos dentro de otro procedimiento. Las variables locales son conocidas dentro del procedimiento en que fueron definidas, por lo tanto, en otro procedimiento pueden definirse variables con los mismos nombres y son consideradas variables distintas. Aunque se llamen iguales por el hecho de ser definidas dentro de procedimientos distintos ocupan posiciones de memoria distintas. Una variable (constante o parámetro) local a un subprograma **A** no tiene ningún significado en otro subprograma definido fuera de **A**, el valor que se le asigne a la variable no podrá ser usado en los otros subprogramas.

También se utiliza la idea de que una variable es local al procedimiento donde ella es visible (o alcanzable).

✓¿Por qué usar variable locales?

Es conveniente usar variables locales porque ayuda a que los programas sean **independientes y modulares**. La comunicación entre la acción Principal y los procedimientos definidos dentro de ella se facilita, ya que esta comunicación se realiza a través de la lista de parámetros.

El uso de variables locales o internas apoya también la reutilización de código, la legibilidad, la modificación y la actualización de los algoritmos y programas, ya que para comunicar un procedimiento con otro, sólo necesitamos tener conocimiento de que hace ese programa y cuáles son sus parámetros, no necesitamos conocer cómo está programado internamente.

Al dividir un algoritmo en acciones y funciones que resuelvan partes más pequeñas del problema, y que a su vez usen variables locales hacemos posible que un proyecto grande sea dividido en partes en donde varios programadores trabajan en forma independiente y con mayor eficiencia. El uso de variables locales también permite utilizar el mismo identificador o nombre de la variable en diferentes subprogramas.

Variables Globales

Son las variables declaradas para el programa o **algoritmo principal**, del que dependen todos los subprogramas.

Un identificador declarado como global en un procedimiento principal, es alcanzable o puede ser usada por todos los procedimientos declarados internos o locales a esa acción, salvo que sea redefinida esta variable dentro de uno de los procedimientos.

✗¿Por qué no debemos abusar de las variables globales?

Las variables globales tiene la ventaja de compartir información entre diferentes subprogramas sin necesidad de agregar una entrada correspondiente a la variable en la lista de parámetros del procedimiento. Sin embargo su uso es más perjudicial que beneficioso, ya que no apoyan la modularización, ni la reutilización del código, ya que si se modifica el identificador de una variable global o es eliminada, tiene que revisarse en código de todos y cada uno de los procedimientos que la utilizan para reflejar estas modificaciones.

Al utilizar variables globales en forma indiscriminada destruimos los mecanismos que aporta la notación pseudoformal y los lenguajes de programación para mantener la modularización y que los procedimientos sean independientes entre sí. Usar variables puede ser desastroso en especial en programas largos, que tienen muchos procedimientos creados por el usuario. En vista de que todos los identificadores (variables, constantes, parámetros y procedimientos) deben ser declarados, crear procedimientos que usen variables globales requiere recordar todas las declaraciones de variables que se hicieron al inicio del procedimiento principal.

Aún más difícil será rastrear o ubicar un error en un programa extenso que utiliza variables globales, ya que el error podrá estar ubicado en cualquiera de los procedimientos internos que usa esta variable global.

Variables No Locales

Son los identificadores (variables que están definidos en otro procedimiento, incluso en el algoritmo principal, pero que tienen alcance (o son visibles) en el procedimiento actual.

Una forma sencilla de identificar las variables no locales de un procedimiento B, es considerando como tales a todas las variables locales y no locales del procedimiento A que lo invoca o llama, siempre y cuando dentro de B no se redefinan estas variables. Por definición, todo procedimiento es no local así mismo.

A continuación se muestra un ejemplo de un algoritmo y la identificación de variables globales, locales y no locales:

```
Acción CalcCorriente
// calcula voltajes según fórmulas físicas
Entero voltios, resistencia;
Real corriente, factorResist;

Acción CalcularResistencia(Real FR)
    // declaración de la acción

    Función valRes(Real FR): Lógico
      Lógico esValido;
      esValido = (FR >= 0);
      Retornar(esValido);
    Función valRes;

   // se invoca a la función valRes
    Si valRes(FR) == falso Entonces
        Escribir("el factor de resistencia debe ser
        positivo");
     FSi;
    … // otras instrucciones de CalcularResistencia

FinAcción CalcularResistencia;

Función ValorCorriente(Entero V, R): Real
    // declaración de la función
    Entero corriente; Real cor;
    corriente = V^2 * R;
    cor = corriente ^(1/5) * 3,1416;
    … // otras instrucciones
    Retonar(cor);
FinFunción ValorCorriente;

… // otras instrucciones del procedimiento principal
```

- Globales: voltios, resistencia, corriente, factorResist

- Locales y No Locales:

CalcCorriente	voltios, resistencia, corriente, factorResist (1) CalcularResistencia, ValorCorriente (2)	CalcCorriente(*)
CalcularResistencia	FR, valRes	voltios, resistencia, corriente, factorResist, CalcularResistencia, ValorCorriente(*)
valRes	FR (de valRes), esValido	voltios, resistencia, corriente, factorResist, CalcularResistencia, ValorCorriente, valRes(*)
ValorCorriente	V, R, cor, corriente (de ValorCorriente)	voltios, resistencia, factorResist, CalcularResistencia, ValorCorriente(*)

(*) todo procedimiento es no local así mismo.

(1) son locales al principal son variables globales

(2) también son locales los procedimientos (acciones y funciones) definidos o especificados dentro de él.

```
// invocación o llamada a las acciones y
// funciones dentro del procedimiento principal
CalcularResistencia(factorResist);
corriente = ValorCorriente(voltios, resistencia);
… // otras instrucciones
FinAcción CalcVoltajes;
```

Alcance de Variables, Tipos de Datos y Resolución de alcance

El alcance o ámbito de una variable no restringe el tipo de dato que la variable puede usar; sean locales o globales, las variables pueden ser de tipo Carácter, String, Real, Entero, Lógico o de cualquier otro tipo de dato simple o estructurado.

El alcance de una variable está relacionado con el lugar o procedimiento donde ella está definida, ya que "dentro de este lugar" será reservado el almacenamiento para los valores de la variable. Por otro lado, el tipo de dato está relacionado con la declaración de la variable y con la palabra reservada asociada a cada tipo (Carácter, String, Real, Entero, Lógico, Arreglo, …)

Cuando una variable local tiene el mismo identificador que una variable global, digamos que ambas se llaman X, todas las referencias a este indicador dentro del procedimiento local se refieren a la variable local. Al volver a usar el mismo identificador X localmente, la variable global X se "tapa" y no es visible desde dentro del procedimiento que la redefine. La variable local X, tiene precedencia sobre el valor de la variable global X.

Sin embargo algunos lenguajes de programación, y también en pseudocódigo, suministran un **operador de resolución de alcance**, en muchos identificado como **::**<nombre_variable>

El operador :: colocado antes del nombre de la variable le indica al lenguaje que utilice el valor de la variable global, en lugar del valor de la variable local.

A continuación mostramos ejemplos de los valores tomados por variables globales y locales:

```
Acción Principal
Entero X1
Real Pi;
X1 = 10;  Pi = 3,1416;
A(X1);
…

Acción A (Entero X)
    Lógico Pi;
    Pi = (X mod 2) == 0;
    Escribir("El valor de Pi en A es: " + Pi);
FinAcción A;
Escribir("El valor de Pi en Principal es: " + Pi);
FinAcción Principal;
```

Salidas de este algoritmo:

1. El valor de Pi en A es: verdadero (hay redefinición de la variable Pi dentro de A)
2. El valor de Pi en Principal es: 3,1416

```
Acción Principal
Entero X1
Real Pi;
X1 = 10;  Pi = 3,1416;
A(X1);
…

Acción A (Entero X)
    Lógico Pi;
    Pi = (X mod 2) == 0;
    Escribir("El valor de Pi es: " + ::Pi);
FinAcción A;
FinAcción Principal;
```

Salidas de este algoritmo:

1. El valor de Pi es: 3,1416

El operador de resolución de alcance consulta el valor de la variable Pi en el procedimiento Principal y lo utiliza dentro de la instrucción Escribir.

Ejercicios Asignados

1. Dado el siguiente algoritmo indique: (Parcial Nro. 2, junio 2007, valor 6ptos)
   1.1. Variables globales, variables y procedimientos locales y no locales
   1.2. Resultados que escribe el algoritmo
   1.3. Parámetros actuales y formales, parámetros pasados por referencia
   1.4. ¿Qué hacen Ac1 y Ac2?

```
Acción Principal
    Entero a, b, c, d;   Real Res1, Res2;

Función Ac1 (Entero w, x, y, z) : Real
    Real a;
    a = w * z + x * y;    a = a / (x * z);
    Retornar (a);
FFunción Ac1;

Acción Ac2 (Entero w, x, y, z; ref Real w1)
    w1 = w * y / (x * z);
    x= 1;   y = z mod 1;
FAcción Ac2;

// Continuación …
Escribir("Suministre valores enteros para a, b, c y d");
Leer(a, b, c, d);
Res1 = Ac1(a, b, c, d);
Ac2(c, d, a, b, Res2);

Escribir ("Resultados del algoritmo, ");
Escribir ("para Ac1: " + Res1 + " para Ac2: " + Res2 );

fAcciónPrincipal;
```

2. Dado el siguiente algoritmo se quiere que muestre lo siguiente:
   2.1. Localidad / No localidad de las variables y procedimientos
   2.2. Que resultados escribe el algoritmo
   2.3. Parámetros actuales y formales

```
Acción ejercicio
ENTERO A, B, C;
    Función número1 (Ref Entero a, Entero b) : Entero
ENTERO C;
        c = b + 2 + a + 3;
        a = c;
        b = c;
        Escribir ("en número1 los valores de a, b y c
                  son: ");
        Escribir (a  + "-" + b + "-" + c );
        Retornar (a + c + b );
    Ffunción;

Acción cálculo (Entero b, Ref Entero a, Ref Entero c)
    a = a * 2 + b;
    c = a + 1;
    b = c + 1;
    escribir ("en cálculo los valores de a, b y c son: ");
    escribir (a  + "-" + b + "-" + c);
    c = número2 (c, b);
Facción;
```

```
Función número2 (Ref Entero b, Entero c) : Entero
    a = 1;    c = b + 2 + a + 3;
    a = b - c;      b = 5 + a;
    Escribir ("en número2 los valores de a, b y c
              son: ");
    Escribir (a  + "-" + b + "-" + c);
    Retornar ( a + c + b + número1(a, b) );
Ffunción;
```

```
Acción principal()
    Entero a;
    a = 3; b = 2; c = 1;
    cálculo (número1(c, a), b, a);
    Escribir ("en principal los valores de a, b y c
              son: ");
    Escribir (a + "-" + b + "-" + c);
    a = número1(b, a);
    Escribir ("nuevos valores: "a + "-" + b + "-" +c);
Facción;
Acción ejercicio;
```

Ejercicios para consolidar

1. Calcula la multiplicación de dos números enteros n1 y n2 a través de sumas sucesivas (usar ciclo **Para**)
2. Traduce la siguiente fórmula matemática a pseudocódigo y construye un algoritmo que calcule su resultado para valores n y x suministrados por el usuario. Recuerda realizar las validaciones necesarias sobre los valores de n y x. (usar **Para anidado**)

$$\prod_{i=n}^{1}(n/x)+\sum_{k=i}^{n}(k-nx)^2$$

3. Algoritmo que detecte y escriba la primera vocal leída, asuma que el usuario indica los datos de entrada carácter por carácter **(usar Repetir)**
4. Escribe un algoritmo que solicite las notas de un alumno, calcule su promedio final, escriba la nota promedio y si el estudiante aprobó no la materia. Considere en su solución: **(usar Mientras)**
    - La cantidad de notas a procesar será suministrada por el usuario.
    - Se debe verificar que cada nota suministrada esté en un rango válido (de 0 a 20 puntos).
    - Debe contar la cantidad de notas inválidas que suministró el usuario e informar esta cantidad.
    - Las notas inválidas no se toman en cuenta para el cálculo del promedio
5. Calcule n términos de la Serie de Fibonacci e indique cuantos y cuáles de esos términos son números primos. Considere en su solución:
    - Los números de Fibonacci son miembros de una serie en la cual cada número es la suma de los dos números anteriores, es decir:
      $F_i = F_{i-1} + F_{i-2}$ donde $F_1 = 1$ y $F_2 = 1$ (los dos primeros términos son iguales a 1)
      Por ejemplo: $F_3 = F_2 + F_1 = 1 + 1 = 2$ $F_4 = F_3 + F_2 = 2 + 1 = 3$ $F_5 = F_4 + F_3 = 3 + 2 = 5$
    - Un número es primo si solo es divisible por el mismo y por 1
6. Calcule y escriba los números que son divisores de un número X dado por el usuario **(crear 3 versiones: una con Para, otra con Mientras y la tercera con Repetir)**

Para Investigar y Meditar

1. ¿Qué razones o errores pueden llevar a que los ciclos no se detengan?
2. ¿Cuáles son las principales diferencias entre el Mientras y el Repetir?
3. ¿Cómo puedo realizar la estructura Para usando un Mientras o un Repetir?
4. ¿Cómo puedo realizar la estructura Repetir usando un Mientras y viceversa?
5. ¿En qué casos no debo usar un Para?

Fuentes consultadas:

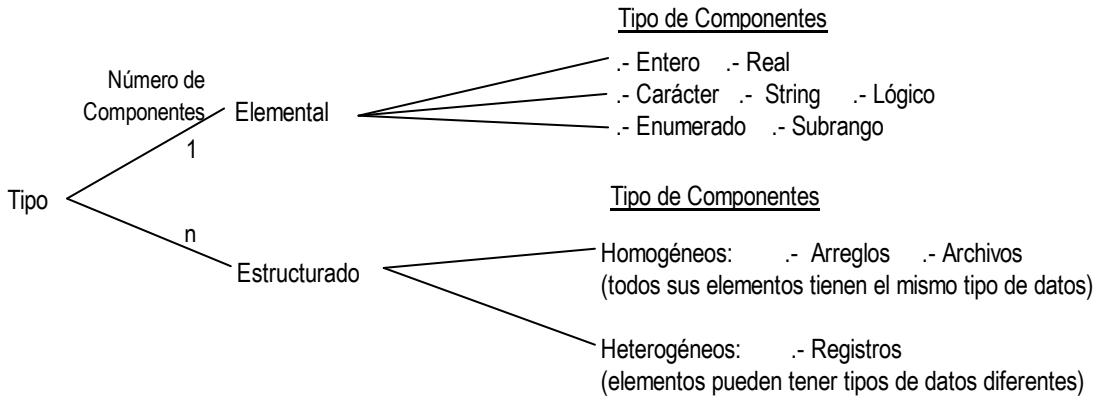
- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. **"Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos"**. Editorial McGraw-Hill, 2003. Capítulo 5, Subprogramas (Procedimientos y Funciones); pág 178 y siguientes.
- BRONSON, Gary J. **"C++ para Ingeniería y Ciencias".** Editorial Thomson, 2007. "da Edición. Capítulo 6, Modularidad con el uso de funciones, pág. 361 y siguientes.

TEMAS 7. TIPOS DE DATOS ESTRUCTURADOS ARREGLOS – REGISTROS – ARCHIVOS

7.1. Estructura de Datos (ED)

Es una herramienta mediante la cual es posible almacenar datos estructurados en la memoria del computador, permitiendo guardar datos conformados por varios elementos y manipularlos en forma sencilla. Estas estructuras de datos están formadas por más de un elemento, donde estos pueden ser todos del mismo tipo de dato (ED homogéneas como los arreglos y los archivos) o de tipos de datos diferentes (ED heterogénea, como los registros y los objetos).

Las estructuras de datos permiten el almacenamiento de información de manera organizada en la memoria principal del computador o en algún dispositivo externo (memoria auxiliar).



7.2. Arreglos

Estructuras de datos conformada por una sucesión de celdas, que permite almacenar en la memoria principal del computador un conjunto finito de elementos (hay un número máximo conocido) que tienen el mismo tipo de dato (son homogéneos).

Para hacer referencia a cualquiera de las celdas del arreglo es necesario el nombre del arreglo y el valor de uno de los elementos perteneciente al conjunto de **índices** asignado, lo que permite tener acceso aleatorio.

Características básicas de los Arreglos

- **Homogéneo**: los elementos del arreglo son todos del **mismo tipo de dato**
- **Ordenado**: cada elemento del arreglo puede ser identificado por el índice que le corresponde. El índice no es más que un entero que pertenece a un intervalo finito y determina la posición de cada elemento dentro del arreglo.

- **Acceso Secuencial o Directo**: El acceso a cada elemento del arreglo se realiza recorriendo los anteriores según el orden en que están almacenados o de manera directa (operación selectora), indicando el valor del índice del elemento requerido.
- **Sus elementos son tratados como variables simples:** Una vez seleccionado algún elemento del arreglo este puede ser utilizado en acciones de asignación, como parte de expresiones o como parámetros al llamar a acciones o funciones como si se tratara de una variable del tipo de dato simple declarado para todos los elementos del arreglo.
- **Uso de índice:** El índice es un valor de tipo entero (número entero o carácter con un código entero equivalente) que determina la posición de cada elemento en el arreglo. La cantidad de índices determina las dimensiones del arreglo. Así un arreglo unidimensional tiene un índice, un arreglo bidimensional tiene dos índices y uno n-dimensional tiene n índices.

Declaración de los arreglos

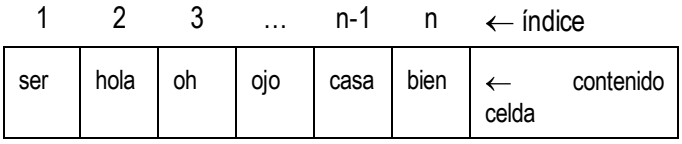
Para declarar un arreglo se debe indicar:

- El nombre del arreglo
- El tipo base: tipo de dato de todos los componentes del arreglo
- El tipo índice: intervalo de valores que podrá tomar el índice del arreglo, indicando el límite inferior (Li) y el límite superior (Ls) del rango

Para declarar arreglos tendremos dos opciones: **declaración por Variable y declaración por Tipo**. Más adelante se indicará la sintaxis de cada declaración

7.2.1 Arreglos Unidimensionales (o Vectores)

Tipos de Datos Estructurados (TDE) donde todos sus elementos pertenecen al mismo tipo y existe una correspondencia uno a uno de cada elemento con un subconjunto de los enteros (índices).



Declaración de Arreglos Unidimensionales

A. Declaración por Variable: se declara la variable tipo arreglo como si se tratara de una variable de tipo de dato simple.

La sintaxis a utilizar será:

Arreglo <identificador> de <tipo de dato> [Li .. Ls];

Donde: **Li** es el límite inferior del arreglo, **Ls** es el límite superior del arreglo

Ejemplos:

```
Arreglo A de Entero [1..5];          // arreglo de 5 enteros
Arreglo B de Lógico [-5..5];         // arreglo de 11 lógicos
Arreglo NOMBRE de String [0..99];    // arreglo de 100 palabras
Arreglo NOTAS de Entero [1..n];      // arreglo de n notas
```

B. Declaración por Tipo: se realiza en dos pasos, (1) Se declara el arreglo como un nuevo tipo de dato y (2) Se le asigna ese tipo a una o más variables.

La ventaja de esta clase de declaración reside en que no es necesario repetir todos los datos del arreglo cada vez que se necesite uno, sino que se utilizan tantas variables como arreglos se necesiten y además simplifica la escritura del algoritmo.

Para declarar un tipo, sólo hace falta anteceder a la especificación la palabra clave **Tipo**

La sintaxis a utilizar será:

```
// paso 1, declaración del tipo de dato arreglo <identificador>
Tipo  Arreglo <identificador> de <tipo de dato> [Li .. Ls];

// paso 2, declaración de las variables de tipo <identificador>
<identificador>  Arreg1, Arreg2;
```

Ejemplo:

```
Tipo  Arreglo Asistentes de String [1..3000];  // declaración del tipo de dato arreglo llamado Asistentes
Participantes Fiesta, Convención;     // declara dos variables (Fiesta y Convención) del tipo arreglo Asistentes
```

Operaciones básicas en Arreglos Unidimensionales

A. Operación constructora B. Operación selectora C. Recorrido Secuencial

A. Operación Constructora: Permite asociarle al nombre de un arreglo un valor estructurado de las mismas dimensiones del arreglo, y con componentes del mismo tipo. Esto sólo se puede hacer en la declaración.

Ejemplos:
```
Arreglo A de Entero[1..5] = { 10, 15, 20, 25, 30 };
Arreglo A de String[1..2] = { "Lunes", "Martes" };
```

B. Operación Selectora: permite hacer referencia a un elemento individual del arreglo, mediante el índice unívoco del elemento. El índice es evaluado, por lo que puede ser una constante, variable o expresión.

Ejemplo: Selección en Arreglos

```
Acción SeleccionArreglos
// muestra distintas operaciones de selección y asignación de valores a un arreglo
    Entero N, i;
    N = 3;
    Arreglo A de Entero[1..5];     Arreglo B de Caracter[1..N];
    i = 3;
    A[1] = 100;              // se asigna el valor 100 a la posición 1 del arreglo
    A[i] = 30;          // se asigna el valor 30 a la posición 3 del arreglo
    A[i*2-1] = 1;            // se asigna el valor 1 a la posición 5 del arreglo (i*2-1
    Para i=1 hasta N en 1 hacer
        B[i] = 'x';  // inicializa el arreglo B almacenando en sus 3 posiciones el carácter 'x'
    FPara;
    A[4] = 3 + A[1] + A[3];
    A[2] = Factorial(A[1]);  // la función Factorial calcula el factorial del valor almacenado en la
                             // posición 1 del arreglo y ese resultado se almacena en la celda 2
Facción;
```

C. Recorrido Secuencial: Esta operación se realiza cuando se utiliza una estructura de control iterativa para tratar todos y cada uno de los elementos del arreglo de acuerdo al orden en que se almacenan. El tratamiento es el mismo para todo el arreglo, y se aplica tanto cuando se desea leer el arreglo, buscar un elemento en un arreglo, listar todo el contenido del mismo, y muchos otros.

El recorrido puede ser de manera ascendente (el más utilizado), partiendo del límite inferior hasta el superior e incrementando en uno; o de manera descendente, partiendo del límite superior hasta el inferior y decrementando en uno.

La estructura de control **Para** es la más conveniente para realizar de manera el recorrido, indicando solo el limite inferior y superior, pues el incremento en uno del índice es automático. Sin embargo, si existiera alguna condición adicional o si se necesita usar una condición lógica para finalizar el recorrido, no basta con llegar al límite superior del índice, hay que cambiar de la estructura iterativa Para, y utilizar un **Mientras** o un **Repetir.**

Ejemplo: Copia de un arreglo a otro

```
Acción PrincipalCopia
    // solicita y valida el tamaño de un arreglo, lo crea y llama luego a acciones que manipulan el arreglo
            Entero N;
            Repetir
               Escribir("Suministre el tamaño del arreglo");   Leer(N);
            Hasta N>0; //se valida que el tamaño sea positivo y que el arreglo tenga al menos 1 celda

            // ⊙j⊙, luego de conocer el tamaño suministrado por el usuario definimos el tipo de dato arreglo
               Tipo Arreglo Números de Entero [1..N];
        Números A, B;     // se declaran dos arreglos (A y B) con las mismas características que Números
            Leer_Arreglo (A, N);              // inicializa el arreglo con valores dados por el usuario
        Copiar (A, B, N);         //copia los valores de arreglo A al arreglo B
        Escribir_Arreglo (B, N); // muestra los valores almacenados en el arreglo B

    // acción que llena un arreglo pasado como parámetro de limite superior N
    Acción Leer_Arreglo (Ref Números C; Entero N)
        Entero i;                 // variable índice para recorrer el arreglo
        Para i=1 hasta N en 1 hacer   // lectura de los valores que se almacenan en el arreglo
          Escribir ("suministre el valor para el elemento" + i + "del arreglo");   Leer(C[i]);
        FPara;
    fAcción Leer_Arreglo;

    // acción que copia dos arreglos de un mismo tipo
    Acción Copiar (Números A, Ref Números B; Entero N)
        Entero i;                 // variable índice para recorrer el arreglo
        Para i=1 hasta N hacer
          B[i] = A[i];                  // se copia cada elemento de A a B
        Fpara;
    fAcción Copiar;
```

```
    // acción que imprime un arreglo pasado como parámetro de limite superior N
    Acción Escribir_Arreglo (Números C, Entero N)
        Entero i;                    // variable índice para recorrer el arreglo
        Escribir("A continuación se escriben los valores almacenados en el arreglo");
        Para i=1 hasta N en 1 hacer   // se escriben los valores del arreglo, separándolos con --
          Escribir(C[i] + " -- ");
        FPara;
    fAcción Escribir_Arreglo;
fAcción PrincipalCopia;
```

7.2.2 Arreglos Bidimensionales (Matrices o Tablas)

Un arreglo bidimensional (tabla o matriz) es un arreglo que tiene dos índices. Para localizar o almacenar un valor en el arreglo se deben especificar **dos subíndices** (dos posiciones), uno para la fila y otro para la columna. Los diferentes tipos de índices no necesitan ser subrangos del mismo tipo. Los elementos se referencian con el formato:

<identificador>[3,4] donde: identificador, es el nombre del arreglo o de la variable

[3,4] indica el elemento de la fila 3 y columna 4

Declaración de Arreglos Bidimensionales

Al igual que en los arreglos unidimensionales, los arreglos bidimensionales **también se pueden crear con declaraciones de variable o de tipo**, el cambio, es que se debe indicar dos intervalos de índices: uno para las filas y otro para las columnas. Estos intervalos o subrangos no tienen que ser de la misma dimensión.

A. Declaración por Variable

La sintaxis a utilizar será:

Arreglo <identificador> de <tipo de dato> [Li_{fila} .. Ls_{fila} , Li_{col} .. Ls_{col}];

Donde: **Li** es el límite inferior del arreglo, **Ls** es el límite superior del arreglo, usados en el rango de filas y de columnas

Ejemplo: Arreglo Montos de Real [1...10 , 1…25];

B. Declaración por Tipo

La sintaxis a utilizar será:

// paso 1, declaramos el tipo arreglo

Tipo Arreglo <identificador> de <tipo de dato> [Li_{fila} .. Ls_{fila} , Li_{col} .. Ls_{col}];

// paso 2, declaración de la variable Tabla de tipo <identificador>

<identificador> Tabla1;

Ejemplo:

```
Tipo  Arreglo Matriz de Entero [1..10], [-5…5];  // declaración del tipo de dato arreglo bidimensional llamado Tabla
Matriz Tabla, T;                                 // declara dos variables (Tabla y T) del tipo arreglo bidimensional Matriz
```

Operaciones básicas en Arreglos Bidimensionales

Las mismas que para arreglos unidimensionales, **pero utilizando dos subíndices** para identificar a cada elemento.

A. Operación constructora B. Operación selectora C. Recorrido Secuencial

Cualquier proceso que queramos realizar sobre un arreglo bidimensional involucrará a los dos subíndices. Se usan dos ciclos anidados para recorrer todo el arreglo.

Ejemplo: Lectura de una tabla o matriz por filas y por columnas

Sean las variables

```
Entero Fila, Columna;
```

Recorrido por filas

```
Para Fila = 1 hasta 10 en 1 hacer
   Para Columna = 1 hasta 25 en 1 hacer
         Escribir(A[Fila, Columna]);
   FPara;
FPara;
```

Recorrido por columnas

```
Para Columna = 1 hasta 25 en 1 hacer
    Para Fila = 1 hasta 10 en 1 hacer
        Escribir(A[Fila, Columna]);
    FPara;
FPara;
```

7.3. Algoritmos de Búsqueda

7.3.1 Algoritmos de Búsqueda en Arreglos

La búsqueda de un elemento, es una operación muy frecuente en cuanto a programación se trata, ya que permite verificar la existencia de un elemento dentro de una estructura de datos. En el caso de los arreglos unidimensionales, una operación de búsqueda puede implicar recorrer, de manera secuencial, desde 1 hasta N elementos.

El resultado de un algoritmo de búsqueda puede ser:

- Un lógico indicando que existe (verdad) o no (falso) el elemento buscado.
- Un número entero que indique la posición o índice donde está el elemento en el arreglo. Si no existe devuelve -1, 0 o cualquier valor fuera del intervalo definido para el arreglo.
- Un número entero, que indique la cantidad de veces que se encuentra el elemento buscado (0 o más repeticiones), devuelve cero (0) si el elemento no existe.

A continuación se estudiarán los algoritmos de **búsqueda lineal** y **búsqueda binaria**. ⊙j⊙ **ESTUDIAR** ⊙j⊙

7.3.2 Búsqueda Lineal

Consiste en comparar de manera secuencial, cada elemento del arreglo con el elemento clave (o buscado). Es eficiente cuando el arreglo es pequeño y **no necesariamente está ordenado**.

Ejemplo: Búsqueda de un elemento dentro de un arreglo con el algoritmo de Búsqueda Lineal

Enunciado: Utilice el método de Búsqueda Lineal para recorrer un arreglo A de 100 elementos enteros e indicar si se encuentra en el arreglo un elemento ELEM suministrado por el usuario.

```
Acción Principal_BúsquedaLineal
// realiza la búsqueda de un elemento en un arreglo de 100 valores enteros
      // Declaración del tipo de dato y la constante para el límite superior
        Entero N;  N = 100;          // tamaño del arreglo
        Tipo Arreglo Vector de Entero [1..N];      // declaración del tipo arreglo Vector
        Vector A;                    // declaración de la variable arreglo A, donde se hará la búsqueda
        Entero ELEM;                       // elemento a buscar en el arreglo
        Lógico ENC;                        // indica si se encontró el elemento

      Llenar_Arreglo(A, N);                // acción que inicializa o llena el arreglo A
      Escribir ("Indique el número que quiere buscar dentro del arreglo");
      Leer (ELEM);                         // leemos el valor que el usuario quiere buscar

      // Ei = {Se tiene un arreglo A de N elementos, N > 1; N y ELEM Є Z (son valores de tipo entero) }
      ENC = Buscar_Lineal(A, ELEM, N);
        Selección
        ENC == verdad:  Escribir ("El valor: " + ELEM + " existe en el arreglo");
          ENC == falso:  Escribir ("El valor: " + ELEM + " no está en el arreglo");
        FSelección

        // Ef = {Se realizó la búsqueda del elemento ELEM en el arreglo A; ENC indica si se encontró el
        elemento}
  Función Buscar_Lineal (Vector A, Entero E, N) : Lógico
  //realiza la búsqueda del valor E en un arreglo de N enteros
      Entero I;                  // índice del arreglo
        Lógico L;                 // variable de retorno
        L = falso;   I = 1;       // inicialización de las variables

        Mientras   I <= N  y  L == falso  Hacer
              Si  A[I] == E  entonces  L = verdadero;  FSi;
              I  =  I + 1;
        FMientras;
      Retornar (L);              // se devuelve el valor lógico de la función

    FFunción Buscar_Lineal;
FAcción Principal_BúsquedaLineal;
```

EJERCICIO ASIGNADO: Indicar por Búsqueda Secuencial cuántas veces se encuentra un elemento en un arreglo

Enunciado: Adaptar el algoritmo de Búsqueda Secuencial para indicar cuántas veces se encuentra una nota X dada por el usuario, dentro de un arreglo unidimensional llamado Notas con 48 valores de tipo real.

7.3.3 Búsqueda Binaria

Algoritmo de búsqueda que se emplea de manera eficiente cuando se tiene un arreglo muy grande, y **está ordenado**

Pasos:

a. Verificar el valor clave (elemento a buscar) contra el elemento que está justo en el centro del arreglo.
b. Si es mayor, el algoritmo buscará solo en la mitad superior del arreglo, si es menor buscará en la mitad inferior, si es igual retorna la posición encontrada.
c. Si busca en alguna de las mitades del arreglo, el algoritmo aplicará nuevamente los pasos a y b, y así sucesivamente hasta que la clave sea igual al elemento que está justo en la mitad del subarreglo, ó, el subarreglo se convierta en un arreglo de 1 elemento el cual es distinto a la clave o elemento buscado.

Ejemplo: Búsqueda de un elemento dentro de un arreglo con el algoritmo de Búsqueda Binaria

Enunciado: Utilice el método de Búsqueda Binaria para recorrer un arreglo B que almacena 100 elementos de tipo carácter e indicar la primera posición en que se encuentra un elemento Car suministrado por el usuario. De no encontrarse en el arreglo se retorna cero (0).

```
Acción Principal_BúsquedaBinaria
// realiza la búsqueda de un elemento en un arreglo con 100 valores enteros

      // Declaración del tipo de dato y la constante para el límite superior
        Entero N;  N = 100;          // tamaño del arreglo
        Tipo Arreglo Símbolos de Caracter [1..N];       // declaración del tipo arreglo Símbolos
        Símbolos B;                  // declaración de la variable arreglo B, donde se hará la
búsqueda
        Carácter Car;                // Elemento a buscar en el arreglo
        Entero POS;                  // Posición del elemento en el arreglo, es cero si no existe
      Llenar_Arreglo(B, N);          // acción que inicializa o llena el arreglo B

        Escribir ("Indique el carácter que quiere buscar dentro del arreglo");
        Leer (Car); // leemos el valor que el usuario quiere buscar

        // Ei = {Se tiene un arreglo B de N elementos, N > 1; N y POS ∈ Z (son valores de tipo entero) }
      POS = Buscar_Binario(B, Car, N);
        Selección
          POS ≠ 0: Escribir ("El carácter: " + Car + " está en la posición: "+ POS);
            POS == 0: Escribir ("El carácter: " + Car + " no está en el arreglo");
        FSelección
        // Ef = {Se realizó la búsqueda del elemento clave Car en el arreglo B; POS indica si se encontró el
        elemento y su posición en el arreglo, posición = 0 en caso de que no se encuentre el elemento }

Función Buscar_Binario (Símbolos B, Carácter C, Entero N) : Entero
// realiza la búsqueda binaria del valor C en un arreglo ya ordenado, llamado B con N caracteres
```

```
    //variables locales a la función
    Entero I;                  // índice del arreglo
      Lógico ENC;                    // indica si se encontró el elemento
      Entero INFERIOR, MITAD, SUPERIOR;   //Delimitan las dos mitades para hacer la búsqueda
      Entero P;          // Posición donde se encuentra el elemento buscado

      // inicialización de variables
      ENC = falso;      INFERIOR = 1;           SUPERIOR = N;    P = 0;

      Mientras  (INFERIOR <= SUPERIOR) y no(ENC) hacer
           MITAD  =  (INFERIOR + SUPERIOR) div 2;

           Selección
              B[MITAD] == Car:  ENC = verdad;   P = MITAD;  // el elemento fue encontrado
              B[MITAD]  >  Car:  SUPERIOR = MITAD - 1;         // seleccionamos la mitad inferior del
              arreglo
              B[MITAD]  <  Car:  INFERIOR = MITAD + 1;       // seleccionamos la mitad superior del
              arreglo
           FSelección;
      FMientras;
    Retornar(P);              // se devuelve el valor entero de la función
```

FFunción Buscar_Binario;
FAcción Principal_BúsquedaBinaria;

EJERCICIO ASIGNADO: Adaptar el algoritmo de Búsqueda Binaria para buscar en un arreglo de registros.

Enunciado: Dado un arreglo que almacena información de N productos (código, descripción y precio), adapte el algoritmo de Búsqueda Binaria para indicar el precio de un producto con un código C dado. Asuma que no hay productos repetidos en el arreglo.

7.4. Algoritmos de Ordenamiento en Arreglos

El ordenamiento es una de las tareas que se efectúa con más frecuencia cuando se desarrollan programas de tratamiento de información estructurada. Sin embargo esta tarea puede llegar a consumir mucho tiempo de ejecución. Por esta razón, muchos investigadores de la computación han desarrollado cientos de algoritmos de ordenamiento que buscan hacer más eficiente el proceso.

A continuación se estudiarán los algoritmos de ordenamiento **Selección directa (*ripple sort*)** e **Intercambio (*burbuja o bubble sort*)**

⊙j⊙ IMPORTANTE ESTUDIAR ⊙j⊙

7.3.1 Ordenamiento por Selección Directa (*ripple sort*):

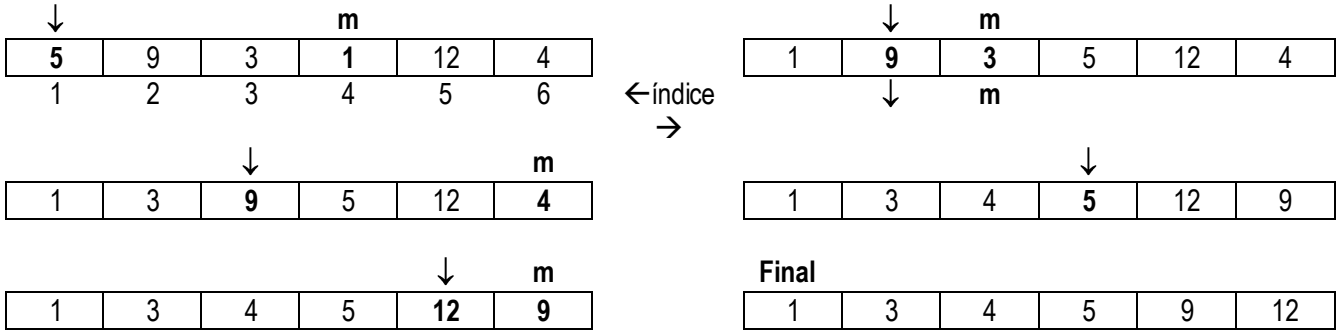
Consiste en el ordenamiento ascendente de un arreglo unidimensional (vector) de N elementos, basándose en buscar el elemento más pequeño del arreglo y colocarlo de primero, luego, buscar el segundo más pequeño y colocarlo en la segunda posición y así sucesivamente.

Los pasos que sigue este algoritmo son:

a. Seleccionar el primer elemento del arreglo.
b. Buscar el menor entre todos los elementos restantes (2, 3, 4, … ,N). Si el elemento encontrado es menor que el primero, se intercambian.
c. Seleccionar el segundo elemento.
d. Compararlo con los N-2 restantes (3, 4, … ,N) e intercambiar cuando se encuentre uno menor.
e. Repetir hasta procesar todos los elementos.

Ejemplo Gráfico:

- La flecha (↓) indica el elemento seleccionado para comparar con el resto de los valores en el arreglo. Observe como los elementos ubicados antes de ella están ordenados ascendentemente.
- La letra (**m**) indica cual es la posición del elemento menor con el que se va a intercambiar el seleccionado.



Ejemplo: Ordenar un arreglo utilizando Selección Directa

Enunciado: Luego de llenar un arreglo A con 100 valores enteros suministrados por el usuario, utilice el algoritmo de Ordenamiento por Selección Directa para ordenar este arreglo ascendentemente.

```
Acción Principal_OrdenamientoSel
// declara un arreglo, lo llena con sus valores iniciales y llama a una acción que lo ordena
    Entero N = 100;  N= 100;                  // límite superior del arreglo
    Tipo Arreglo Vector de Entero [1..N];          // declaración del arreglo a ordenar
    Vector A;
    Llenar_Arreglo(A);          // acción que inicializa o llena el arreglo A
    Ord_Selección(A, N);        // acción que ordena el arreglo
    EscribirElementos(A, N) // acción que muestra los elementos del arreglo ya ordenado
FAcción Principal_OrdenamientoSel;
```

```
Acción Ord_Selección (Ref Vector A; Entero N)
// Acción que realiza el ordenamiento ascendente de un arreglo unidimensional por el método de selección
    // Variables locales a la acción
    Entero I, J;                    // Índices para recorrer el arreglo
    Entero MENOR;                   // Contiene el valor del menor elemento del arreglo
    Entero POSMENOR;                // Contiene la posición del menor elemento

    // Ei = { Se tiene un arreglo de N elementos el cual se quiere ordenar, N > 1}

    Para I = 1 hasta N-1 en 1 hacer
        POSMENOR =  I;
      MENOR =  A[I];
      Para J = I +1 hasta N hacer          // búsqueda del elemento menor
          Si (A[J] < MENOR) entonces
              MENOR = A[J];
              POSMENOR =  J;
          FSi
      FPara
        // Intercambio del elemento mínimo
      A[POSMENOR] =  A[I];
      A[I] = MENOR;
    FPara;
    // Ef = { El arreglo A, de N elementos, fue ordenado ascendentemente }

FAcción Ord_Selección;
```

EJERCICIO ASIGNADO: Ordenar descendentemente un arreglo.

Enunciado: Dado un arreglo unidimensional A de N elementos enteros, modifique el algoritmo de Selección Directa para que ordene el arreglo descendentemente.

7.3.2 Ordenamiento por Intercambio (burbuja o *bubble sort*)

Consiste en el ordenamiento ascendente de un arreglo (vector) de N elementos. Consiste en insertar un elemento del vector en una parte ya ordenada y comenzar de nuevo con los elementos restantes. Se basa en comparaciones y desplazamientos sucesivos.

Pasos a seguir:

a. Tomar un elemento J del arreglo (inicialmente J = 1)

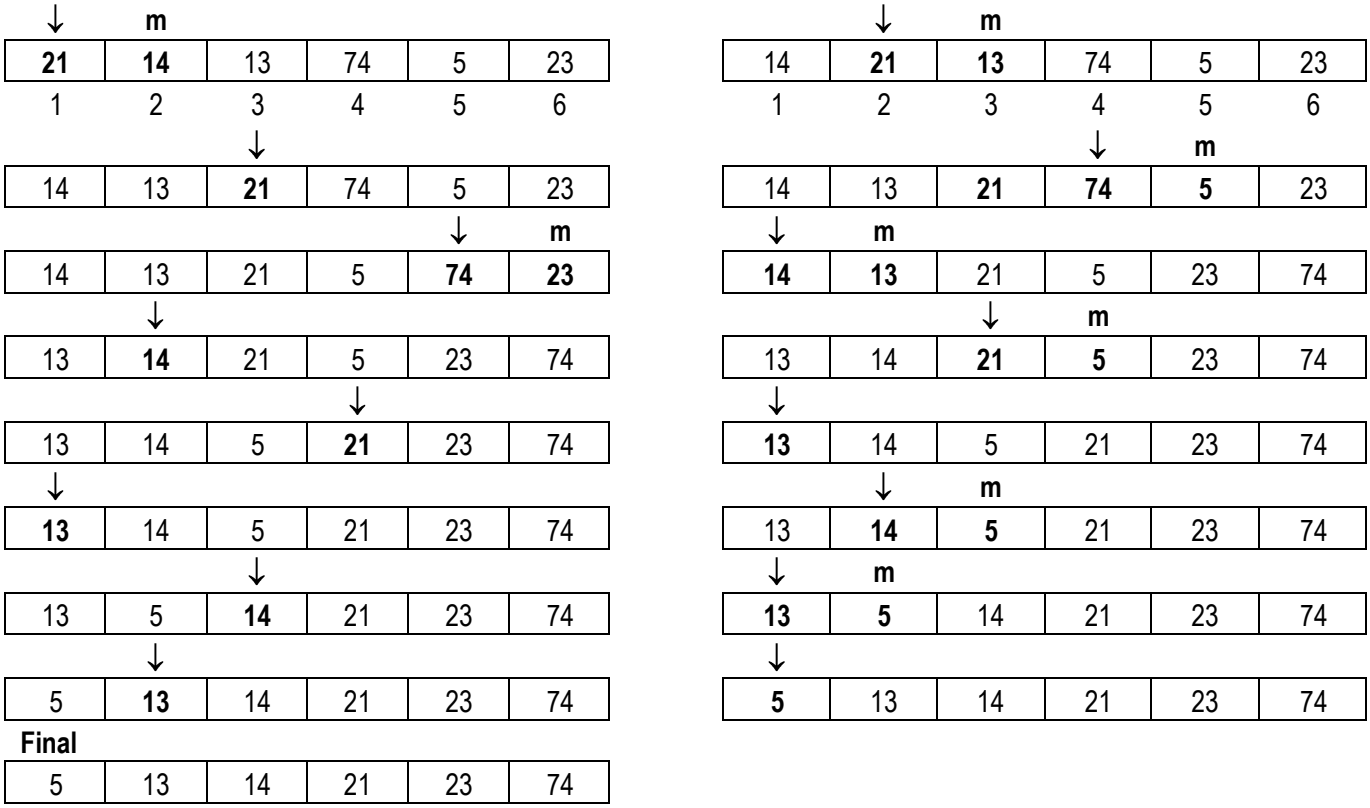
b. Compararlo con su elemento adyacente superior (el que se encuentra a su derecha).

c. Si el elemento seleccionado es mayor que su adyacente, intercambiar.

d. Repetir el proceso hasta que no hayan más intercambios (se controla con un ciclo que va desde I = 1 hasta N – 1).

Ejemplo Gráfico:

- La flecha (↓) indica el elemento seleccionado para comparar con el resto de los valores en el arreglo.
- La letra (**m**) indica cual es la posición del elemento con el que se va a intercambiar el seleccionado. En este caso se intercambia si el adyacente es menor que el seleccionado.



Ejemplo: Ordenar un arreglo utilizando Intercambio

Enunciado: Luego de llenar un arreglo A con 100 valores enteros suministrados por el usuario, ordene sus valores utilizando el método de Ordenamiento por Intercambio.

```
Acción Principal_OrdenamientoBurb
// declara un arreglo, lo llena con sus valores iniciales y llama a una acción que lo ordena
      Entero N = 100;  N= 100;                  // límite superior del arreglo
      Tipo Arreglo Vector de Entero [1..N];          // declaración del arreglo a ordenar
      Vector A;
      Llenar_Arreglo(A);            // acción que inicializa o llena el arreglo A
      Ord_Burbuja(A, N);            // acción que ordena el arreglo
      EscribirElementos(A, N) // acción que muestra los elementos del arreglo ya ordenado
FAcción Principal_OrdenamientoBurb;

Acción Ord_Burbuja (Ref Vector A; Entero N)
// Acción que realiza el ordenamiento ascendente de un arreglo unidimensional por el método de la
burbuja

Entero I, J;              // Índices para recorrer el arreglo según el método Burbuja
Entero AUX;               // Variable auxiliar para hacer los intercambios
// Ei = { Se tiene un arreglo de N elementos el cual se quiere ordenar, N = n; n  > 1}

Para I = 1 hasta N–1 hacer     // indica las veces que se recorre el arreglo para hacer intercambios
      Para J = 1 hasta N–I hacer   // ciclo para buscar si el siguiente (J+1) es menor que el actual (J)
            Si A[J] > A[J+1] entonces
            // intercambio de los valores de los dos elementos ayudándonos con la variable AUX
                  AUX = A[J];     A[J] = A[J+1];     A[J+1] = AUX;
            Fsi;
      FPara;
FPara;
// Ef = { El arreglo A de N elementos fue ordenado ascendentemente}

FAcción Ord_Burbuja;
```

EJERCICIO ASIGNADO: Ordenar las filas de una matriz utilizando el método de Intercambio.

Enunciado: Adapte el algoritmo anterior para ordenar los valores enteros de una matriz de 5 filas por 3 columnas utilizando el método de Intercambio o Burbuja.

7.3.3 Versión Mejorada: Uso de una bandera lógica y de la acción Intercambio

Utiliza una variable lógica para controlar cuando no es necesario realizar más intercambios porque el arreglo ya está ordenado, lo cual permite finalizar el algoritmo realizando menos iteraciones. El algoritmo se detiene cuando se detecta por primera vez que ya no se realizan intercambios entre los valores del arreglo porque ya está ordenado. Esta versión utiliza una llamada a la acción Intercambiar, pasándole los valores correspondientes a los elementos a intercambiar en el ordenamiento lo cual da mayor modularidad al algoritmo, al separar el recorrido del intercambio de los valores.

A continuación se suministra una versión de la acción Ord_Burbuja_Mejorada aplicado sobre un arreglo llamado A con N elementos de tipo entero.

```
Acción Ord_Burbuja_Mejorado (Ref Vector A; Entero N)
// Acción que realiza el ordenamiento ascendente de un arreglo unidimensional por el método de la
// burbuja de manera más eficiente

// Variables locales a la acción
Entero I, J; // Índices para recorrer el arreglo
Lógico LOG;// Indica si no existen más intercambios

// Ei = { Se tiene un arreglo de N elementos el cual
        se quiere ordenar, N = n; n  > 1}

I =  1;
Repetir
      LOG = falso;       //indica si hay intercambios
      Para J = 1 hasta N–1 en 1 hacer
            Si A[J] > A[J+1] entonces
                  Intercambiar (A[J], A[J+1]);
                  LOG = verdad;
            Fsi
      FPara
      I = I + 1;
Hasta (I == N – 1) o (no(LOG));
// donde no(LOG) es equivalente a LOG = falso

// Ef = { El arreglo A de N elementos fue ordenado
        ascendentemente}

FAcción Ord_Burbuja_Mejorado;
```

```
Acción Intercambio (Ref Entero X, Y)
   //acción que realiza el intercambio
   // de los valores de dos números enteros
   // utilizando la variable AUX para no
   // perder uno de los dos valores

   Entero AUX;          //variable auxiliar
   AUX = X;
   X = Y;
   Y = AUX;
FAcción Intercambio;
```

Ejercicios Asignados de Arreglos (Vectores y Matrices):

1. Inicializar un arreglo con los montos en BsF de las ventas mensuales para 2008 de 5 productos. Calcular e informar luego el promedio de ventas por mes y por producto.

2. Dada una matriz de M x N elementos enteros, se quiere:
   2.1. Calcular la suma de los cuadrados de sus elementos.
   2.2. Encontrar la posición (fila, columna) del primer elemento igual a un valor V dado por el usuario.
   2.3. Obtener la posición de todo elemento de la matriz que sea múltiplo de un entero H dado.
   2.4. Dados I y J, intercambiar la fila I con la fila J.
   2.5. Dado un arreglo A con P elementos, indicar la posición de los elementos de A que aparecen en la matriz (sólo la primera aparición), debe indicar también la posición en la matriz.
3. Modificar la función **Buscar_Lineal** para que cuente y devuelva la cantidad de veces que se encuentra el elemento **elem** (cero si no está). Escribir los resultados en la acción.
4. Modificar la función **Buscar_Lineal** para que trabaje con una matriz de 3 x 2, busque y muestre todos los valores menores a un entero **elem** suministrado por el usuario. La función deberá retornar un string (o cadena) con todos los elementos encontrados separados por `–` o una cadena vacía en caso de que no se encuentre ningún elemento menor a **elem** en el arreglo.
5. Dada una matriz cuadrada de M filas (M suministrado por el usuario) con valores reales, elabora un algoritmo que calcule la suma de los valores de su diagonal principal y la suma de los valores de su diagonal secundaria, para indicar luego cual de los dos resultados es el menor.
6. Dado un arreglo con 1000 números enteros con valores entre -100 y 100, crea un algoritmo que indique cuál es el número (o los números) que más se repite(n). Utilice un su solución un arreglo auxiliar de contadores. Por ejemplo, si el arreglo contiene: -2, 4, 1, -2, 3, 1, la salida sería: -2 1.
7. Dados dos arreglos unidimensionales A y B, con elementos enteros, los cuales presentan un ordenamiento en forma ascendente, se quiere crear un arreglo C con el mismo orden a partir de la mezcla de los dos anteriores, adicionalmente se pide que C no contenga elementos repetidos.

7.5. Registros

Estructura de datos formada por una colección finita de elementos llamados campos, no necesariamente homogéneos (del mismo tipo) y que permiten almacenar una serie de datos relacionados entre sí bajo un nombre y una estructura común.

7.5.1 Características básicas de los Registros

- Permiten almacenar un grupo de elementos bajo un nombre y un estructura común
- Los elementos (campos) de un registro no tienen que ser homogéneos, de hecho, generalmente son de diferentes tipos
- No están disponibles en todos los lenguajes de programación, razón por la cual muchas veces es necesario simularlo o definirlo.
- Cada campo del registro se comporta como una variable simple, de manera que puede ser usado en una expresión de asignación, como parte de otra expresión, en operaciones o como parámetro al invocar una acción o función.

7.5.2 Declaración de registros

A. Declaración por Variable: se declara la variable de tipo registro identificándola a través de su nombre, se indica la estructura del registro suministrando la definición de sus campos mediante sus tipos de dato y sus nombres

La sintaxis a utilizar para declarar un registro será:

```
Registro  <identificador> =                  // se indica el nombre del registro
        <Tipo de dato>1 <Identificador>1     //  tipo de dato y nombre del campo 1
       <Tipo de dato>2 <Identificador>2      // tipo de dato y nombre del campo 2
            …
        <Tipo de dato>N <Identificador>N     // tipo de dato y nombre del campo N
Fregistro;
```

B. Declaración por Tipo: Al igual que con los arreglos, para declarar un tipo de registro definido por el usuario, se antecede a la especificación la palabra clave **Tipo** y luego se definen las variables del tipo. El **uso de la declaración por tipo facilita** la declaración de variables con una estructura común, así como el pase de parámetros.

7.5.3 Operaciones básicas en Registros

A. Operación constructora B. Operación selectora

A. Operación Constructora: Permite asociarle al nombre de un registro un dato estructurado, el cual se corresponde componente a componente con la declaración del registro. Esta operación **permite inicializar** los campos del registro.

Ejemplo:

```
// Declaración del registro
Tipo Registro Fecha =
      Entero día, mes, año;
FRegistro;
```

```
// Construcción de una variable registro llamada Persona
Persona ← {10234223, "Carlos Morales", {10, 10, 1986} }
```

Valor campo CI — Valor campo Nombre — Valor campo FNacimiento

```
Registro  Persona =
          Entero CI;
          String Nombre;
          Fecha  FNacimiento;
FRegistro;
```

B. Operación Selectora: Permite Vareferenciar o seleccionar un campo particular del registro. Su sintaxis es:

<identificador_registro>.<identificador_campo> Ejemplos: Persona.CI Persona.FNacimiento

Nombre de la variable tipo Registro

Nombre del campo que se está consultando dentro del registro

Al igual que los arreglos, los registros o sus campos pueden ser pasados como parámetro.

7.5.4 Ejemplo del uso de Registros (pasos a considerar) ⊙j⊙ IMPORTANTE ⊙j⊙

1. Declarar la estructura de los registros a utilizar (declarar los Registros):

```
Tipo Registro Asignación =
      Real Transporte;
      Real Comida;
      Real Vacaciones;
FRegistro;
```

```
Tipo Registro Deducción =
      Real Seguro;
      Real Vivienda;
      Real Impuesto;
FRegistro;
```

```
Tipo Registro Empleado =
      Entero Cédula;
      String Nombre;
      Lógico Activo;
      Carácter Sexo;
      Real Sueldo_Hora;
      Entero Horas_Trab;
      Asignación AS;
      Deducción DE;
FRegistro;
```

2. Declarar las variable que se necesitan de cada tipo de registro, en este caso, se declara la variable E de tipo Empleado

```
Empleado E;
```

3. Inicializar cada campo del (los) registro(s) a utilizar

3.1 Usando la Operación Constructora

E = {10234223, "Juan Mata", Verdad, 'M', 6000, 160, {15000, 50000, 0.0}, {2500.75, 3687.25, 1547.15} }

3.2 Usando la operación selectora para cada campo del registro

```
Acción Inicializar (Ref Empleado E)
        E.Cédula = 10234223;                    E.AS.Vacaciones = 0,0;
        E.Nombre = "Juan Mata";                 E.DE.Seguro = 2500,75;
        E.Activo = Verdad;                      E.DE.Vivienda = 3687,25;
        E.Sexo = 'M';                           E.DE.Impuesto = 1547,15;
        E.Sueldo_Hora = 6000;               Facción Inicializar
        E.Horas_Trab = 160;
        E.AS.Transporte = 15000;
        E.AS.Comida = 50000;
                        sigue ↗
```

4. Crear los algoritmos de las acciones, funciones o métodos necesarios, en este ejemplo, se indica el algoritmo de una función para calcular el sueldo de un empleado activo

```
Función Sueldo (Empleado E) : Real
// Calcula el sueldo de un empleado activo
    Real A;          // sumatoria de asignaciones del empleado
    Real D;          // sumatoria de deducciones del empleado
    Real S;          // sueldo total del empleado (variable de retorno)

    Si E.Activo entonces
        A = E.AS.Transporte + E.AS.Comida + E.AS.Vacaciones;
        D = E.DE.Seguro + E.DE.Vivienda + E.DE.Impuesto;
        S = ((E.Sueldo_Hora * E.Horas_Trab) + A) – D:

    Sino
        S = 0,0;
    fSi
    Retornar(S);
Ffunción Sueldo;
```

7.6. Archivos

Un archivo puede definirse como una secuencia de elementos del mismo tipo, que residen generalmente en memoria auxiliar. Los archivos son utilizados cuando el volumen de datos a procesar es grande y no cabe en la memoria central del computador. Adicional a la memoria principal, los archivos pueden ser almacenados en dispositivos de memoria auxiliar como discos, disquetes, cintas, cartuchos zip, cd, pen drive o memory flash, entre otros.

7.6.1 Características básicas de los Archivos

- Los archivos son utilizados cuando se desea que los datos puedan recuperarse aún después de haber apagado la máquina y también cuando se manejan grandes volúmenes de información.
- Residen en memoria auxiliar, la cual generalmente, es más económica, grande, pero lenta que la memoria principal del computador.
- Los elementos que almacena un archivo pueden ser elementales (texto, números) o estructurados (arreglos, registros). Para propósitos del curso, sólo se puede leer un archivo elemento por elemento.
- La cantidad de componentes que constituyen el archivo, es en teoría, ilimitada (no se conoce a priori). Si tiene cero elementos se dice que es un archivo vacío.
- Cuando la estructura de un archivo no parece contener elementos de un solo tipo, si decimos que todos los elementos del archivo es un gran registro, no rompe la definición anterior (es un caso particular de un archivo con un solo elemento de tipo registro).
- Los archivos más comunes son los archivos secuenciales. Los elementos de un archivo secuencial están dispuestos uno tras otro, y para acceder un elemento en particular, hay que pasar por todos los anteriores. Por esos se les llama de acceso secuencial, a diferencia de los arreglos que son de acceso aleatorio. Sin embargo, los elementos del archivo también puedes ser extraídos de manera directa, pero este objetivo escapa del alcance del curso.

7.6.2 Declaración de Archivos

Consiste en indicar el nombre de la variable que será utilizada en el algoritmo para hacer Referencia al archivo.

A. Declaración por Variable: se declara la variable tipo archivo como si se tratara de una variable de tipo de dato simple.

```
Archivo <Identificador>A;    # se indica el nombre del archivo
```

Ejemplos: `Archivo A;` `Archivo Ar, X;`

B. Declaración por Tipo: al igual que los arreglos y los registros, se antecede la palabra reservada **Tipo** a la declaración y luego se declaran las variables del tipo.

```
Tipo Archivo <Identificador_Tipo>;

<Identificador_Tipo> <Identificador>A;
```

Ejemplos: `Tipo Archivo A;` `A Ar, X;`

7.6.3 Operaciones básicas en Archivos Secuenciales

A. Abrir el archivo B. Cerrar el archivo C. Verificar Fin de archivo (EOF) D. Leer del archivo E. Escribir en el archivo

Estas operaciones se consideran operaciones primitivas y reservadas del pseudo-código.

A. Abrir el archivo: Ubica y prepara el archivo para poder trabajar con él.

```
Acción AbrirArchivo (Ref Archivo A; String <ruta_del_archivo>, <Argumentos>)
```

Donde:

<**ruta_del_archivo**> indica la ruta en memoria o al menos el nombre del archivo con el cual se va a trabajar, por ejemplo, C/Mis documentos/datosprueba.txt o DatosParticipantes.doc

<**Argumentos**> es uno o más de las siguientes palabras reservadas:

- Escritura: indica que el archivo se abre de solo escritura
- Lectura: indica que el archivo de abre de solo lectura
- Añadir: indica que el archivo se abre de escritura pero todo lo que se escriba se añade al final del archivo
- Texto: indica que el archivo a abrir es un archivo de texto
- Binario: indica que el archivo a abrir es un archivo binario (un archivo diferente a un archivo de texto).

Los argumentos pueden ser combinados con el operador lógico y.

Por ejemplo: AbrirArchivo(A, 'prueba.txt', Lectura y Texto)

En el parámetro argumentos, normalmente se indica como mínimo, uno de los tipos de archivo (lectura, escritura, añadir) y uno de los tipos de datos para sus elementos (texto, binario). Los argumentos también son llamados *flags*.

NOTA: El archivo se abre una sola vez en el algoritmo, al principio del mismo y debe ser cerrado al finalizar el mismo

B. Cerrar el archivo: Cuando se desea dejar de trabajar con un archivo, debe cerrarse para que esté disponible para otras aplicaciones. Sólo los archivos que están abiertos pueden cerrarse. Si un archivo no se cierra se puede perder información en el mismo.

```
Acción CerrarArchivo(Ref Archivo A)
```

C. Fin de archivo (FDA, EOF): indica si no hay más elementos para leer en el archivo. Sólo se usa para archivos de lectura. Retorna verdadero si se alcanzó el fin del archivo y falso en caso contrario.

```
Función FDA(Ref Archivo A) : lógico
```

D. Leer del archivo: Lee un elemento. Aunque una lectura no modifica el archivo físicamente en disco, la variable Archivo si es modificada (por ejemplo, cambia la posición actual del cursor del archivo), por lo tanto debe pasarse por Referencia.

```
Acción LeerArchivo(Ref Archivo A, Ref <elemento_tipo> x)
```

E. Escribir en el archivo: escribe un elemento en el archivo.

```
Acción EscribirArchivo(Ref Archivo A, <elemento_tipo> x)
```

7.6.4 Recorrido secuencial para tratar un archivo

Supongamos que utilizamos dos archivos de texto: **A** es de lectura y será el archivo a recorrer a través de operaciones de lectura, **B** es de escritura y se utilizará para mostrar los resultados.

La idea fundamental es recorrer el archivo de entrada o lectura una sola vez a través del uso de una estructura de iteración que finalice cuando la **operación FDA** retorne verdadero. Los ciclos **Mientras** o **Repetir** son los más adecuadas para este recorrido, el **Para** no resulta tan conveniente, a menos que conozcamos cuantos registros tiene el archivo o el problema este limitado a leer una cantidad determinada de ellos. El archivo de salida, si se pide que se genere, se creará en el ciclo de recorrido ya que la idea es realizar la lectura del archivo origen y la creación del archivo resultado al mismo tiempo, es decir, en un solo ciclo.

La acción de impresión del archivo de salida requiere que el archivo abierto como escritura sea cerrado y reabierto como de lectura, de manera que sea recorrido y en cada paso se escriba el contenido de cada uno de sus registros.

A continuación se suministrará un esquema para este recorrido, que si bien no es 100% general o no es aplicable para todos los problemas de recorrido de archivos, puede servir como base para la resolución de los ejercicios clásicos donde se usa un archivo de entrada, se genera un archivo de salida y se realizan cálculos adicionales. En el ejemplo se contarán la cantidad de caracteres existentes en el archivo de entrada.

⊙j⊙ IMPORTANTE ⊙j⊙

Los PASOS GENERALES PARA RECORRER UN ARCHIVO se pueden resumir en los siguientes:

1. **Declarar** los archivos a utilizar y las variables que almacenarán elementos del archivo
2. **Abrir el archivo de entrada** (que solo se podrá leer) como tipo lectura.
3. **Validar** si la función **FDA** devuelve verdadero. Si esto ocurre mostrar un mensaje indicativo e ir al paso **f** para finalizar el ciclo. Si es falso ir al paso **d** para continuar el recorrido.
4. **Abrir el o los archivos de salida** a utilizar e inicializar las variables necesarias en el procesamiento dentro del ciclo.
5. **Recorrer** con un ciclo **Repetir…Hasta FDA** o **Mientras no(FDA) hacer…FMientras**.
6. Dentro de este ciclo la primera operación es la de lectura de cada registro. Si se realizan escrituras en el archivo de salida, estas van en este mismo ciclo.
7. Al finalizar el ciclo, **Cerrar** el archivo de salida y mostrar resultados.
8. **Cerrar** el archivo de entrada

Acción Procesar_Archivos

```
// pseudocódigo de las acciones llamadas dentro de la Acción Principal
Acción Imprimir_Archivo (Archivo T; String Nombre)
  // acción que muestra el contenido de todo el archivo

    Carácter P;              // se usa para obtener cada carácter
    AbrirArchivo(T, Nombre, Escritura y Texto)
       Mientras no( FDA(T) ) hacer
          LeerArchivo(T, P);
               Escribir(P);          // mostrar contenido del registro actual
    FMientras;
    CerrarArchivo(T);
fAcción Imprimir_Archivo;
```

```
// instrucciones de la acción Procesar_Archivos (acción principal)

Archivo A, B;     // Variables para los archivos a utilizar en el algoritmo
Carácter C;       // Variable utilizada para las operaciones de lectura y escritura
Entero CONT;      // Contador de caracteres existentes en el archivo A

AbrirArchivo(A, "entrada.txt", Lectura y Texto);
                                                                    Selección
   FDA(A):        Escribir ("El archivo de entrada fue suministrado vacío");
   No( FDA(A) ): AbrirArchivo (B, "listado.txt", Escritura y Texto);
                 CONT = 0;
                 Repetir
                     LeerArchivo(A,C);   // obtener el carácter actual a tratar
                     CONT = CONT + 1; // contar cantidad de caracteres
                     <otros_procesos>  // otras operaciones para procesar el registro actual
                     ...
                 EscribirArchivo(B,C);          // escribir en el archivo de salida
                 Hasta FDA(A)
                 CerrarArchivo(B);

            Escribir("Cantidad de caracteres encontrados: " + CONT);
          Imprimir_Archivo(B, "listado.txt");     // acción para imprimir el archivo
   fSelección
      CerrarArchivo(A);                  // acción común para los dos casos

Facción Procesar_Archivos;
```

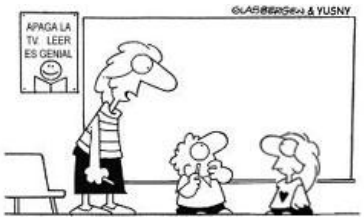
Ejercicios Asignados de Arreglos, Registros y Archivos:

1. Dado un archivo de texto, contar la cantidad de vocales que contiene.
2. Dado un archivo de estudiantes, imprimir aquellos cuya cédula de identidad sea mayor que 10 millones.
3. Dado el nombre de dos archivos binarios, generar un tercero que sea la concatenación de los anteriores.
4. Dado dos archivos de estudiantes ordenados ascendentemente por el campo CI, generar un tercer archivo que sea la mezcla ordenada de los anteriores.

```
Tipo  Registro Estudiante =
      Entero CI;
      String Nombre;
      Entero ND;
FRegistro;
```

5. Almacenar en un arreglo de registros el nombre, cédula, ingreso mensual durante un año y fecha de ingreso a la empresa de 10 empleados. A partir de esta información calcular y almacenar los años de antigüedad de cada empleado y el monto total que cobró en el año.
6. Dado un archivo con caracteres y símbolos, cuyos valores iniciales son suministrados por el usuario, indicar cuántos de esos caracteres son vocales y cuántos son '*'. Indicar luego la frecuencia de cada uno sobre el total de caracteres del archivo.

Ejercicios Asignados de Arreglos, Registros y Archivos:



7. Dado un arreglo de registros con información de N compradores, se debe crear un archivo de salida llamado "Result" en donde se identifique a todos los clientes que hicieron compras por un valor en BsF mayor al promedio general del arreglo. Del archivo Result deben mostrase luego los clientes que hicieron las compras de mayor monto en BsF.
8. Dado un archivo con información de estudiantes y la nota definitiva obtenida en una materia, recorrerlo para indicar cuántos estudiantes cursaron la materia, cuántos aprobaron, cuántos reprobaron y quiénes se retiraron.
9. Dado dos archivos con valores de tipo real que representan los precios de los productos vendidos en una tienda de Caracas (archivo C) y en una tienda de Valencia (archivo V), crear un tercer archivo (archivo Salida) que almacene la mezcla de los archivos anteriores. Asuma que los archivos de entrada no tienen valores repetidos.
10. Dado un archivo de entrada con información de viviendas en venta, crear un archivo de salida con la información de todas las viviendas que tienen un precio menor o igual a un valor dado por el usuario. La información del archivo de salida debe indicar el código y metraje de la vivienda, además de permitir llamar a la persona que lo está vendiendo.
11. Una Empresa tiene la información referente a los precios en cada año de cinco productos, durante un periodo de 10 años. A partir de dicha información, se quiere que Usted:

    11.1. Cree un archivo secuencial, para almacenar la información dada.

    11.2. A partir de la información almacenada en el archivo creado, calcule:
    - Para cada año el producto de menor precio.
    - Para cada producto, el año en que tuvo menor precio en el periodo dado.
    - El menor precio existente en general, indicando el producto y año correspondiente.

TEMA 8.
PROGRAMACIÓN ORIENTADA A OBJETOS

8.1. Conceptos Básicos

1. Programación Orientada a Objeto (POO)

La programación Orientada a Objetos es una metodología que basa la estructura de los programas en torno a los objetos.

Los lenguajes de POO ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (Clase) que describe a un conjunto de objetos que poseen las mismas propiedades.

2. Objeto

Es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos. Un objeto es una entidad caracterizada por sus atributos propios y cuyo comportamiento está determinado por las acciones o funciones que pueden modificarlo, así como también las acciones que requiere de otros objetos. Un objeto tiene identidad e inteligencia y constituye una unidad que oculta tanto datos como la descripción de su manipulación. Puede ser definido como una encapsulación y una abstracción: una encapsulación de atributos y servicios, y una abstracción del mundo real.

Para el contexto del Enfoque Orientado a Objetos (EOO) un objeto es una entidad que encapsula datos (atributos) y acciones o funciones que los manejan (métodos). También para el EOO un objeto se define como una instancia o particularización de una clase. Los objetos de interés durante el desarrollo de software no sólo son tomados de la vida real (objetos visibles o tangibles), también pueden ser abstractos. En general son entidades que juegan un rol bien definido en el dominio del problema. Un libro, una persona, un carro, un polígono, son apenas algunos ejemplos de objeto.

Cada objeto puede ser considerado como un proveedor de servicios utilizados por otros objetos que son sus clientes. Cada objeto puede ser a al vez proveedor y cliente. De allí que un programa pueda ser visto como un conjunto de relaciones entre proveedores clientes. Los servicios ofrecidos por los objetos son de dos tipos:

1.- Los datos, que llamamos **atributos**.

2.- Las acciones o funciones, que llamamos **métodos**.

Características Generales

- ***Un objeto se identifica por un nombre o un identificador único que lo diferencia de los demás.*** Ejemplo: el objeto Cuenta de Ahorros número 12345 es diferente al objeto Cuenta de Ahorros número 25789. En este caso el identificador que los hace únicos es el número de la cuenta.
- ***Un objeto posee estados.*** El estado de un objeto está determinado por los valores que poseen sus atributos en un momento dado.
- ***Un objeto tiene un conjunto de métodos.*** El comportamiento general de los objetos dentro de un sistema se describe o representa mediante sus operaciones o métodos. Los métodos se utilizarán para obtener o cambiar el estado de los objetos, así como para proporcionar un medio de comunicación entre objetos.

- ***Un objeto tiene un conjunto de atributos.*** Los atributos de un objeto contienen valores que determinan el estado del objeto durante su tiempo de vida. Se implementan con variables, constantes y estructuras de datos (similares a los campos de un registro).
- ***Los objetos soportan encapsulamiento****.* La estructura interna de un objeto normalmente está oculta a los usuarios del mismo. Los datos del objeto están disponibles solo para ser manipulados por los propios métodos del objeto. El único mecanismo que lo conecta con el mundo exterior es el paso de mensajes.
- ***Un objeto tiene un tiempo de vida dentro del programa o sistema que lo crea y utiliza.*** Para ser utilizado en un algoritmo el objeto debe ser creado con una instrucción particular (*New* ó Nuevo) y al finalizar su utilización es destruido con el uso de otra instrucción o de manera automática.

3. Clase

La clase es la unidad de modularidad en el EOO. La tendencia natural del individuo es la de clasificar los objetos según sus características comunes (clase). Por ejemplo, las personas que asisten a la universidad se pueden clasificar (haciendo abstracción) en estudiante, docente, empleado e investigador.

La clase puede definirse como la agrupación o colección de objetos que comparten una estructura común y un comportamiento común.

Es una plantilla que contiene la descripción general de una colección de objetos. Consta de atributos y métodos que resumen las características y el comportamiento comunes de un conjunto de objetos.

Todo objeto (también llamado instancia de una clase), pertenece a alguna clase. Mientras un objeto es una entidad concreta que existe en el tiempo y en el espacio, una clase representa solo una abstracción.

Todos aquellos objetos que pertenecen a la misma clase son descritos o comparten el mismo conjunto de atributos y métodos. Todos los objetos de una clase tienen el mismo formato y comportamiento, son diferentes únicamente en los valores que contienen sus atributos. Todos ellos responden a los mismos mensajes.

Su sintaxis algorítmica es:

Clase <Nombre de la Clase>
…
FClase <Nombre de la Clase>;

Características Generales

- *Una clase es un nivel de abstracción alto.* La clase permite describir un conjunto de características comunes para los objetos que representa. Ejemplo: La clase Avión se puede utilizar para definir los atributos (tipo de avión, distancia, altura, velocidad de crucero, capacidad, país de origen, etc.) y los métodos (calcular posición en el vuelo, calcular velocidad de vuelo, estimar tiempo de llegada, despegar, aterrizar, volar, etc.) de los objetos particulares Avión que representa.
- *Un objeto es una instancia de una clase.* Cada objeto concreto dentro de un sistema es miembro de una clase específica y tiene el conjunto de atributos y métodos especificados en la misma
- *Las clases se relacionan entre sí mediante una jerarquía.* Entre las clases se establecen diferentes tipos de relaciones de herencia, en las cuales la clase hija (subclase) hereda los atributos y métodos de la clase padre (superclase), además de incorporar sus propios atributos y métodos.
  Ejemplos, Superclase: Clase Avión
  Subclases de Avión: Clase Avión Comercial, Avión de Combate, Avión de Transporte
- Los nombres o identificadores de las clases deben colocarse en singular (clase Animal, clase Carro, clase Alumno).

4. Relación entre Clase y Objeto

Algorítmicamente, las **clases son descripciones netamente estáticas o plantillas** que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones.

Por el contrario, **los objetos son instancias particulares** de una clase. Las clases son una especie de molde de fábrica, en base al cual son construidos los objetos. Durante la ejecución de un programa sólo existen los objetos, no las clases.

La **declaración** de una variable de una clase **NO crea** el objeto.

La asociación siguiente: <Nombre_Clase> <Nombre_Variable>; (por ejemplo, Rectángulo R), no genera o no crea automáticamente un objeto Rectángulo. Sólo indica que R será una referencia o una variable de objeto de la clase Rectángulo.

La **creación de un objeto**, debe ser indicada explícitamente por el programador, de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un **método Constructor** (ver punto Métodos).

5. Atributo

Son los datos o variables que caracterizan al objeto y cuyos valores en un momento dado indican su estado.

Un atributo es una característica de un objeto. Mediante los atributos se define información oculta dentro de un objeto, la cual es manipulada solamente por los métodos definidos sobre dicho objeto. Un atributo consta de un nombre y un valor. Cada atributo está asociado a un tipo de dato, que puede ser simple (entero, real, lógico, carácter, *string*) o estructurado (arreglo, registro, archivo, lista, etc.)

Su sintaxis algorítmica es: **<Modo de Acceso> <Tipo de dato> <Nombre del Atributo>;**

Los **modos de acceso** son:

- ***Público:*** Atributos (o Métodos) que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella. Este modo de acceso **también se puede** representar con el símbolo **+**
- ***Privado:*** Atributos (o Métodos) que sólo son accesibles dentro de la implementación de la clase. **También** se puede representar con el símbolo **–**
- ***Protegido:*** Atributos (o Métodos) que son accesibles para la propia clase y sus clases hijas (subclases). **También** se puede representar con el símbolo **#**

6. Método

Son las operaciones (acciones o funciones) que se aplican sobre los objetos y que permiten crearlos, cambiar su estado o consultar el valor de sus atributos. Los métodos constituyen la secuencia de acciones que implementan las operaciones sobre los objetos. La implementación de los métodos no es visible fuera de objeto.

*La **sintaxis algorítmica*** de los métodos expresados como funciones y acciones es:

Para **funciones** se pueden usar cualquiera de estas dos sintaxis:

<Modo de Acceso> Función <Nombre> [(Lista Parámetros)]: <Descripción del Tipo de datos>

Para **acciones**:

<Modo de Acceso> Acción <Nombre> [(Lista Parámetros)] donde los parámetros son opcionales

Ejemplo: Un rectángulo es un objeto caracterizado por los atributos Largo y Ancho, y por varios métodos, entre otros Calcular su área y Calcular su perímetro.

Características Generales

- *Cada método tiene un nombre, cero o más parámetros (por valor o por referencia) que recibe o devuelve y un algoritmo con el desarrollo del mismo.*
- *En particular se destaca el método constructor, que no es más que el método que se ejecuta cuando el objeto es creado.* Este constructor suele tener el mismo nombre de la clase/ objeto, pero aunque es una práctica común, el método constructor no necesariamente tiene que llamarse igual a la clase (al menos, no en pseudos-código). Es un método que recibe cero o más parámetros y lo usual es que inicialicen los valores de los atributos del objeto.
- En lenguajes como Java y C++ se puede definir más de un método constructor, que normalmente se diferencian entre sí por la cantidad de parámetros que reciben.
- *Los métodos se ejecutan o activan cuando el objeto recibe un mensaje*, enviado por un objeto o clase externo al que lo contiene, o por el mismo objeto de manera local.

Creación de Objetos y Métodos Constructores:

Cada objeto o instancia de una clase debe ser creada explícitamente a través de un método u operación especial denominado **Constructor**. Los atributos de un objeto toman valores iniciales dados por el constructor. Por convención el método constructor tiene el mismo nombre de la clase y no se le asocia un modo de acceso (**es público**).

Algunos lenguajes proveen un método constructor por defecto para cada clase y/o permiten la definición de más de un método constructor.

Método de Destructores de objetos:

Los objetos que ya no son utilizados en un programa, ocupan inútilmente espacio de memoria, que es conveniente recuperar en un momento dado. Según el lenguaje de programación utilizado esta tarea es dada al programador o es tratada automáticamente por el procesador o soporte de ejecución del lenguaje.

En la **notación algorítmica** NO tomaremos en cuenta ese problema de administración de memoria, por lo tanto no definiremos formas para destruir objetos. En cambio al utilizar lenguajes de programación si debemos conocer los métodos destructores suministrados por el lenguaje y utilizarlos a fin de eliminar objetos una vez no sean útiles.

7. Mensaje

Es la petición de un objeto a otro para solicitar la ejecución de alguno de sus métodos o para obtener el valor de un atributo público.

Estructuralmente, un mensaje consta de 3 partes:

- *Identidad del receptor:* Nombre del objeto que contiene el método a ejecutar.
- *Nombre del método a ejecutar: Solo* los métodos declarados públicos.
- *Lista de Parámetros* que recibe el método (cero o mas parámetros)

Su sintaxis algorítmica es: **<Variable_Objeto>.<Nombre_Método> ([<Lista de Parámetros>]);**

Cuando el objeto receptor recibe el mensaje, comienza la ejecución del algoritmo contenido dentro del método invocado, recibiendo y/o devolviendo los valores de los parámetros correspondientes, si los tiene ya que son opcionales: (**[ ]**)

Ejemplo 1, Definición de la Clase Rectángulo

IMPLEMENTACIÓN

```
Clase Rectángulo;
    // Atributos
      Privado:
              Real Largo, Ancho;
   // Métodos
      // método constructor
        Acción Rectángulo(Real lar, anc);
            Largo = lar;
            Ancho = anc;
      FAcción;

        Público Función Área: Real
            // Retorna el área o superficie ocupada por el rectángulo
            retornar(Largo * Ancho);
      FFunción Área;

        Público Función Perímetro: Real
              // Retorna el perímetro del rectángulo
            retornar(2 * (Largo + Ancho));
      FFunción Perímetro;

FClase Rectángulo;


// Uso de la clase rectángulo
Acción Principal
        Rectángulo R;   // se declara una variable llamada R, de tipo objeto Rectángulo
        Real L, A;        // se declaran las variables reales L y A para largo y ancho del rectángulo
        Escribir("Suministre a continuación los valores para el largo y el ancho");
        Leer(L);  Leer(A);
        R.Rectángulo(L, A);
        Escribir("Resultados de los cálculos:");
        Escribir("Área: "+ R.Área + "  -  Perímetro " + R.Perímetro);
FAcción Principal;
```

Rectángulo
Privado Real Largo Privado Real Ancho
Constructor Acción Rectángulo(lar, anc) Público Función Área: Real Público Función Perímetro: Real

8.2. Representación en Notación Algorítmica de una Clase

Las clases son el elemento principal dentro del enfoque orientado a objetos. En este lenguaje las declaraciones forman parte de su propio grupo, el grupo [Clases]. Cada una de las declaraciones de clase debe tener el siguiente formato:

```
Clase <Identificador> [Hereda de: <Clases>]
   Atributos
         Público:
               [Constantes];   [Variables];   o  [Estructuras];
            Privado:
               [Constantes];   [Variables];   o  [Estructuras];
            Protegido:
               [Constantes];   [Variables];   o  [Estructuras];

      Operaciones
         Público:
               [Métodos]
            Privado:
               [Métodos]
         Protegido:
               [Métodos]
FClase <Identificador>
```

En el caso de la especificación de los Atributos o de los Métodos los modos de acceso Público (+), Privado (-) o Protegido (#) pueden omitirse, solamente en el caso en el que los grupos [Constantes], [Estructuras] y [Variables] son vacíos, es decir, no se estén declarando Atributos ó Métodos.

NOTA: Otra forma de especificar los modos de acceso es colocándolo antes del nombre DE CADA UNO de los atributos o métodos

8.3. Diagramas de Clase

La representación gráfica de una o varias clases se hará mediante los denominados Diagramas de Clase. Para los diagramas de clase se utilizará la notación que provee el Lenguaje de Modelación Unificado (UML, ver www.omg.org), a saber:

- Las clases se denotan como rectángulos divididos en tres partes. **La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.**
- Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y – respectivamente, al lado derecho del atributo. (+ público, # protegido, - privado).

En la figura se muestra el método "color" que no tiene ningún parámetro y retorna un valor entero y el método "modificar_tamaño" que tiene un real como parámetro y no retorna nada (es una acción).

Notación: **Ejemplos**

Nombre Clase	**Ventana**	**Rectángulo**
Atributos	+ Real área; # Lógico visible;	Privado Real Lado
Métodos	+ color: Entero + modificar_tamaño(Real porcentaje)	Acción Cuadrado(Entero lad) Público Función Área: Real Público Función Perímetro: Real

8.4. Relaciones entre Clases

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de rescribirlos.

La posibilidad de establecer jerarquías entre las clases es una característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, ello debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que rescribirlo cada vez que se necesite.

Los cuatro tipos de relaciones entre clases estudiados en este curso serán:

- Herencia (Generalización / Especialización o Es-un)
- Agregación (Todo / Parte o Forma-parte-de)
- Composición (Es parte elemental de)
- Asociación (entre otras, la relación Usa-a)

1. Relación de Herencia (Generalización / Especialización, Es un)

Es un tipo de jerarquía de clases, en la que cada subclase contiene los atributos y métodos de una (herencia simple) o más

superclases (herencia múltiple).

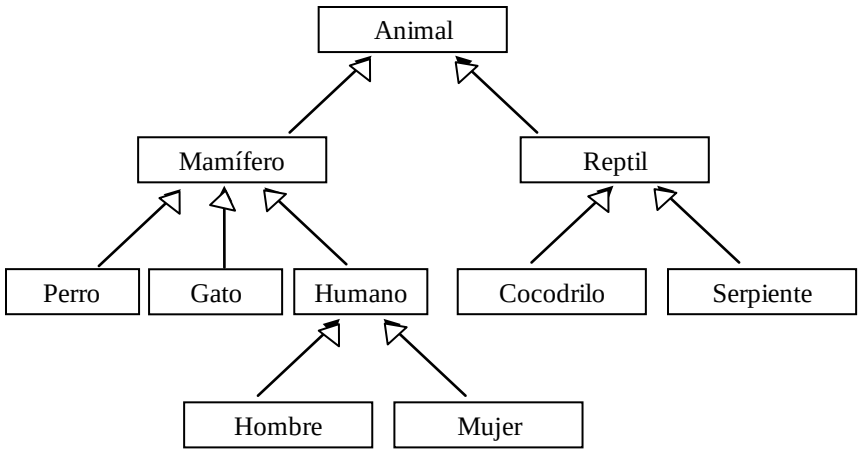
Mediante la herencia las instancias de una clase hija (o subclase) pueden acceder tanto a los atributos como a los métodos públicos y protegidos de la clase padre (o superclase). Cada subclase o clase hija en la jerarquía es siempre una extensión (esto es, conjunto estrictamente más grande) de la(s) superclase(s) o clase(s) padre(s) y además incorporar atributos y métodos propios, que a su vez serán heredados por sus hijas.

Representación:

- *En la notación algorítmica:* Se coloca el nombre de la clase padre después de la frase Hereda de del encabezado de la clase y se usan sus atributos y métodos públicos o protegidos. Ejemplo: Clase Punto3D Hereda de Punto2D
- *En el diagrama de clases:* La herencia se representa mediante una relación de generalización/especificación, que se denota de la siguiente forma:

clase **hija** ⟶ clase **madre** **subclase** ⟶ **superclase**

Ejemplo: **El siguiente diagrama de clases muestra la relación de Herencia entre la Clase Animal y sus hijas**



2. Relación de Agregación (Todo / Parte, Forma parte de)

Es una relación que representa a los objetos compuestos por otros objetos. Indica Objetos que a su vez están formados por otros. El objeto en el nivel superior de la jerarquía es el todo y los que están en los niveles inferiores son sus partes o componentes.

Representación en el Diagrama de Clase

La relación forma parte de, no es más que una asociación, que se denota:

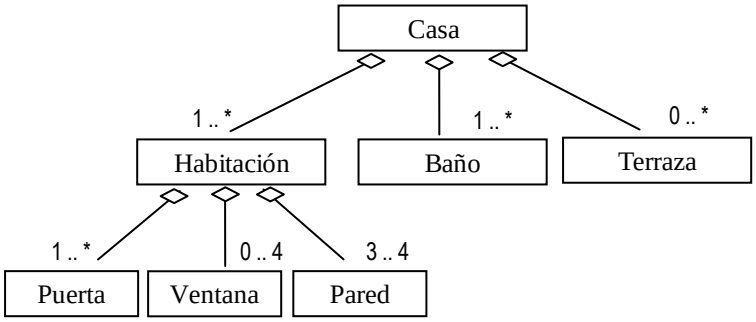
Nombre del **objeto de nivel superior** ◇—— multiplicidad —— Nombre del **objeto que lo compone**

carro ◇—— 4 .. * ——ruedas

Si motor forma parte de carro, la flecha apunta a la clase motor, y el diamante va pegado a carro.

La **multiplicidad** es el rango de cardinalidad permitido que puede asumir la asociación, se denota LI..LS. Se puede usar * en el limite superior para representar una cantidad ilimitada (ejemplo: 3..*).

Ejemplo: **Objeto Casa descrito en términos de sus componentes**



3. Relación de Composición

Un componente es parte esencial de un elemento. La relación es más fuerte que el caso de agregación, al punto que si el componente es eliminado o desaparece, la clase mayor deja de existir.

Representación en el Diagrama de Clases

La relación de *composición*, se denota de la siguiente forma:

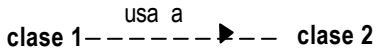
4. Relación de Asociación («uso», usa, cualquier otra relación)

Es una asociación que se establece cuando dos clases tienen una dependencia de utilización, es decir, una clase utiliza atributos y/o métodos de otra para funcionar. Estas dos clases no necesariamente están en jerarquía, es decir, no necesariamente una es clase padre de la otra, a diferencia de las otras relaciones de clases.

El ejemplo mas común de esta relación es de objetos que son utilizados por los humanos para alguna función, como Lápiz (se usa para escribir), tenedor (se usa para comer), silla (se usa para sentarse), etc. Otro ejemplo son los objetos Cajero y Cuenta. El Cajero "usa a" la cuenta para hacer las transacciones de consulta y retiro y verificar la información del usuario.

Representación en el Diagrama de Clases

La relación de *uso*, se denota con una dependencia estereotipada:



Ejemplos:

8.5. Fundamentos del Enfoque Orientado a Objeto

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son: la Abstracción, el Encapsulamiento, la Modularidad y la Herencia. Otros elementos a destacar (aunque no fundamentales) en el EOO son: Polimorfismo, Enlace dinámico (o *binding*), Concurrencia y Persistencia.

Fundamento 1: Abstracción

Es el principio de ignorar aquellos aspectos de un fenómeno observado que no son relevantes, con el objetivo de concentrarse en aquellos que si lo son. Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Dentro del EOO son muy usados mecanismos de abstracción: la Generalización, la Agregación y la clasificación.

- La **generalización** es el mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas. En consecuencia, a través de la generalización, la superclase almacena datos generales de las subclases, y las subclases almacenan sólo datos particulares. La **especialización** es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.
- La **agregación** es el mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la **descomposición**.
- La **clasificación** consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La **ejemplificación** es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular.

Fundamento 2: Encapsulamiento (Ocultamiento de Información)

Es la propiedad del EOO que permite ocultar al mundo exterior la representación interna del objeto. Esto quiere decir que el objeto puede ser utilizado, pero los datos esenciales del mismo no son conocidos fuera de él.

La idea central del encapsulamiento es esconder los detalles y mostrar lo relevante. Permite el ocultamiento de la información separando el aspecto correspondiente a la especificación de la implementación; de esta forma, distingue el "qué hacer" del "cómo hacer". **La especificación es visible al usuario, mientras que la implementación se le oculta.**

El encapsulamiento en un sistema orientado a objeto se representa en cada clase u objeto, definiendo sus atributos y métodos con los siguientes **modos de acceso**:

- *Público (+)* Atributos o Métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.
- *Privado (-)* Atributos o Métodos que solo son accesibles dentro de la implementación de la clase.
- *Protegido (#):* Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases).

Los atributos y los métodos que son públicos constituyen la interfaz de la clase, es decir, lo que el mundo exterior conoce de la misma. Normalmente lo usual es que se oculten los atributos de la clase y solo sean visibles los métodos, incluyendo entonces algunos de consulta para ver los valores de los atributos. El método constructor (Nuevo, *New*) siempre es Público.

Fundamento 3: Modularidad

Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.

Fundamento 4: Herencia

Es el proceso mediante el cual un objeto de una clase adquiere propiedades definidas en otra clase que lo preceda en una jerarquía de clasificaciones**.** Permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), evitando repetición de código y permitiendo la reusabilidad.

Las **clases** heredan los datos y métodos de la **superclase**. Un método heredado puede ser sustituido por uno propio si ambos tienen el mismo nombre.

La herencia puede ser simple (cada clase tiene sólo una superclase) o múltiple (cada clase puede tener asociada varias superclases). La clase Docente y la clase Estudiante heredan las propiedades de la clase Persona (superclase, herencia simple). La clase Preparador (**subclase**) hereda propiedades de la clase Docente y de la clase Estudiante (herencia múltiple).

Fundamento 5: Polimorfismo

Es una propiedad del EOO que permite que un método tenga múltiples implementaciones, que se seleccionan en base al tipo objeto indicado al solicitar la ejecución del método.

El **polimorfismo operacional o Sobrecarga operacional** permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión. En este tipo de polimorfismo, los métodos son interpretados en el contexto del objeto particular, ya que los métodos con nombres comunes son implementados de diferente manera dependiendo de cada clase. Por ejemplo, el área de un cuadrado, rectángulo y círculo, son calculados de manera distinta; sin embargo, en sus clases respectivas puede existir la implementación del área bajo el nombre común Área. En la práctica y dependiendo del objeto que llame al método, se usará el código correspondiente.

Ejemplos, Superclase: Clase Animal Subclases: Clases Mamífero, Ave, Pez

Se puede definir un método Comer en cada subclase, cuya implementación cambia de acuerdo a la clase invocada, sin embargo el nombre del método es el mismo.

Mamifero.Comer ≠ Ave.Comer ≠ Pez.Comer

Otro ejemplo de polimorfismo es el operador +. Este operador tiene dos funciones diferentes de acuerdo al tipo de dato de los operandos a los que se aplica. Si los dos elementos son numéricos, el operador + significa suma algebraica de los mismos, en cambio si por lo menos uno de los operandos es un String o Carácter, el operador es la concatenación de cadenas de caracteres.

Otro ejemplo de sobrecarga es cuando tenemos un método definido originalmente en la clase madre, que ha sido adaptado o modificado en la clase hija. Por ejemplo, un método Comer para la clase Animal y otro Comer que ha sido adaptado para la clase Ave, quien está heredando de la clase Animal.

Cuando se desea indicar que se está invocando (o llamando) a un método sobrecargado y que pertenece a otra clase (por ejemplo, a la clase padre) lo indicamos con la siguiente sintaxis:

```
Clase_Madre::nombre_método;
```

Para el ejemplo, la llamada en la clase hija Ave del método sobrecargado Comer de la clase madre Animal sería:

```
Animal::Comer
```

8.6. Ejemplos de Programación Orientada a Objetos

Ejemplo 1, Definición de la Clase Rectángulo


```
Clase Rectángulo;
      // Atributos
      Privado:
            Real Largo, Ancho;

      // Métodos
      Constructor Rectángulo(Real lar, anc);
            Largo = lar;
            Ancho = anc;
      FConstructor;

      Público Función Área: Real
            // Retorna el área del rectángulo
            retornar(Largo * Ancho);
      FFunción Área;

        Público Función Perímetro: Real
              // Retorna el perímetro del rectángulo
            retornar(2 * (Largo + Ancho));
      FFunción Perímetro;
FClase Rectángulo;


Clase Principal
Acción Usa_Rectángulo
       Rectángulo R;   // se declara una variable de tipo objeto Rectángulo, a la cual llamaremos R
       Real L, A; // se declaran las variables reales L y A para leer el largo y ancho dado por el usuario
       Escribir("Suministre a continuación los valores para el largo y el ancho");
       Leer(L, A);
       R.Rectángulo(L, A);
       Escribir("Resultados de los cálculos:");
       Escribir("Área: "+ R.Área + "  -  Perímetro " + R.Perímetro);
FAcción Usa_Rectángulo;
FClase Principal;
```

Ejemplo 2, Clase que implementa un punto en el plano real

```
Clase Punto2D

  //Atributos
      Protegido  Real X, Y;    // representan las coordenadas X e Y de un punto

  // Métodos u operaciones
      Público:
          //constructores
          Acción Punto2D
              X = 0;    Y = 0;
          FAcción;

          Acción Punto2D(Real CX, CY)
              X = CX;    Y = CY;
          FAcción;

          // otros métodos de la clase
          Función CoordenadaX: Real
          // devuelve el valor de la coordenada X del punto que hace la llamada
              retornar(X);
          Ffunción;

          Función CoordenadaY: Real
          // devuelve el valor de la coordenada Y del punto que hace la llamada
              retornar(Y);
          Ffunción;

              Acción CambiarX(Real NX)
          // modifica el valor de la coordenada X del punto que hace la llamada
              X = NX;
          Ffunción;

          Acción CambiarY(Real NY)
          // modifica el valor de la coordenada Y del punto que hace la llamada
              Y = NY;
          Ffunción;

        Función Distancia(Real CX, CY): Real
          // devuelve la distancia entre el punto actual y otro con coordenadas CX y CY
              Real D;              // variable donde se almacenará la distancia
              D = (RestaC(X, CX) + RestaC(Y, CY)) ^ (1/2);
              retornar(D);
          Ffunción;

          Función Distancia(Ref Punto2D P): Real;
          //devuelve la distancia entre el punto actual y otro punto P
              retornar(Distancia(P.X, P.Y));
          Ffunción Distancia;
```

```
        Protegido  Función RestaC (Real X,Y): Real;
            // devuelve la resta entre X y Y elevada al cuadrado
                retornar((X – Y) ^ 2));
            Ffunción Resta_C;
FClase Punto2D;


Clase Prueba_Punto

  Acción Principal
    Real CX, CY;        // valores para las coordenadas del punto a crear
    Real D;       // contiene la distancia entre los puntos
    Escribir ("Introduzca las coordenadas X e Y del punto a crear: ");
    Leer(CX, CY);
    Punto P = nuevo Punto2D(CX , CY);
    Punto P1 = nuevo Punto2D;    // crea un punto en el origen, es decir, en el punto (0.0 , 0.0)
    D = P.Distancia(P1.CoordenadaX , P1.CoordenadaY;
      Escribir ("Distancia entre el punto P y el origen: " + D);
      P1.CambiarX (15.0); P1.CambiarY(18.75);  // se cambia a al punto (15.0 , 18.75)
      Escribir ("Nuevo punto P1: ("  +  P1.CoordenadaX + " , "+   P1.CoordenadaY + ") ");
      D = P.Distancia(P1);
    Escribir ("La nueva Distancia entre P y P1 es: "+ D);

  fAcción Principal;

FClase Prueba_Punto;
```

Ejemplo 3, Clase Punto3D que se deriva de la clase Punto2D

```
Clase Punto3D Hereda de Punto2D;
  // Atributos
        Protegido Real Z; // coordenada Z del punto
  // Métodos u operaciones
        Público:
        // métodos constructores Punto3D
            Acción Punto3D
            //  Crea un punto con coordenadas (0,0,0) reutilizando el constructor de un punto de 2D
                Punto2D;
                Z = 0;
            FAcción;

            Acción Punto3D(Real CX, CY, CZ)
            //  Crea un punto de tres coordenadas reutilizando el constructor de un punto de 2D
                Punto2D(CX, CY);
                Z = CZ;
            FAcción;
```

```
        // otros métodos de la clase
        Función CoordenadaZ: Real
        //  devuelve el valor de la coordenada Z del punto que hace la llamada
            retornar(Z);
        Ffunción;

        Acción CambiarZ(Real NZ)
        // modifica el valor de la coordenada Z del punto que hace la llamada
            Z = NZ;
        Ffunción;

        Función Distancia3D(Real CX, CY, CZ): Real
        // devuelve la distancia entre el punto actual y otro con coordenadas CX, CY y CZ
            Real D;             //contiene la distancia
            D = RestaC(X, CX) + RestaC(Y, CY) + RestaC(Z, CZ);
           D = D ^ (1/2);
            retornar(D);
        Ffunción;

        Función Distancia3D(Ref Punto3D P): Real
        // devuelve la distancia entre el punto actual y otro punto P
            retornar(Distancia3D(P.X, P.Y, P.Z));
        Ffunción;
FClase Punto3D;
```

Plantea tus reflexiones y respuestas a:

1. ¿Qué atributos y métodos hereda la clase Punto3D de la clase Punto2D?
2. ¿Qué atributos y métodos hereda la clase Punto2D de la clase Punto3D?
3. ¿Quiénes pueden usar los atributos y métodos de Punto2D?
4. Plantea un algoritmo principal que utilice las clases Punto2D y Punto3D

Ejemplo 4, Clase Cuenta Bancaria

```
Clase CuentaBancaria

    // atributos
  Privado Entero Saldo;
    Privado String NroCuenta;
    Privado String Titular;

    // métodos
  Acción CuentaBancaria(Entero montoInicial; String num, nombre)
            // asigna a los atributo de la clase sus valores iniciales
            Saldo = montoInicial;
            NroCuenta = num;
            Titular = nombre;
    Facción;

    Público Acción depositar(Entero cantidad)
            // incrementa el saldo de la cuenta
            Saldo = Saldo + cantidad;
    Facción;

    Público Acción retirar(Entero cantidad)
            // disminuye el saldo de la cuenta
            Saldo = Saldo - cantidad;
    Facción;

    Público Función      obtenerSaldo: Entero
    // permite conocer el monto disponible o saldo de la cuenta
    Retornar(saldo);
    FFunción;

FinClase CuentaBancaria
```

Reflexiona y plantea propuestas para:

1. Crear un algoritmo principal que utilice la clase CuentaBancaria, creando varios objetos cuenta y actualizando sus saldos. Recuerda agregar en tu algoritmo principal las solicitudes de datos al usuario que consideres necesarias y las validaciones de estos datos de entrada
2. ¿Desde tu algoritmo principal podrías actualizar directamente el valor del atributo saldo? ¿Cómo lo puedes hacer?
3. Utiliza los métodos de la clase CuentaBancaria para verificar, en el algoritmo principal, que no se retire un monto que el saldo no puede cubrir.

Ejemplo 5, Clase Raíces Ecuación de 2do Grado

Desarrollar un algoritmo bajo el enfoque orientado a objetos, que permita calcular las raíces (reales e imaginarias) de una ecuación de segundo grado ax^2+bx+c, dados los coeficientes a, b y c, con a>0.

Análisis:

El objeto principal a ser tratado en este problema es la ecuación y se implementará a través de la clase **Ecuación2doGrado**.

Se considerará que cada ecuación tiene asociadas como atributos sus raíces, las cuales deben ser calculadas por el método constructor a través de una acción llamada **Resolver**. El método constructor tendrá tres parámetros, que son los coeficientes de la ecuación (a, b y c).

La clase **Ecuación2doGrado** tendrá un método público para escribir las raíces llamado **Imprimir**.

Ecuación2doGrado
Privado Real RaízReal1
Privado Real RaízImaginaria1
Privado Real RaízReal2
Privado Real RaízImaginaria2
Constructor Ecuación2doGrado(Real a, b, c) // Crea un Objeto Ecuacion2doGrado
Privado Acción Resolver(Real a, b, c) // Calcula las raíces de la ecuación
Público Acción Imprimir // Muestra las raíces de la ecuación

```
Clase Ecuación2doGrado;
      Privado Real r1, r2, i1, i2;

        Constructor Ecuación2doGrado(Real a, b, c);
        Resolver(a,b,c);
      FConstructor;

        Privado Acción Resolver(Real a,b,c);
      // Calcula las raíces de la ecuación ax^2+bx+c
        Real delta;
        delta = b * b - (4*a*c);
        Si delta ≥ 0 entonces      // Cálculo de raíces reales
            Si b > 0 entonces r1 = - ( b + delta^(1/2) ) /  (2*a);
                    sino r1 = - (delta^(1/2) - b) /  (2*a);
            Fsi;
            r2 = c  /  (r1*a);  i1 = 0;   i2 = 0;
        sino // Cálculo de raíces complejas
            r1 = -b  /  (2*a);       r2 = -b  /  (2*a);
            i1 = (-delta)^(1/2)  /  (2*a);
            i2 = -i1;
        Fsi;
      FAcción Resolver;


        Público Acción Imprimir;
      # Muestra las raíces de la ecuación
        Escribir(r1, i1);
        Escribir(r2, i2);
      FAcción Imprimir;
FClase Ecuación2doGrado;
```

```
// Usamos la clase Ecuacion2doGrado
Acción CalcularEcuaciones
   Real a,b,c;
   Ecuación2doGrado E, F;
   Escribir("suministre coeficientes ecuación 1");
   Leer(a, b, c);
   E.Ecuación2doGrado(a, b, c);
   E.Imprimir;
   Escribir("suministre coeficientes ecuación 2");
   Leer(a, b, c);
   F.Ecuación2doGrado(a, b, c);
   F.Imprimir;
FAcción;
```

BIBLIOGRAFÍA RECOMENDADA

- BOOCH, Grady. **"Object-oriented Analysis and Design with Applications".** 2da. edición, Benjamín/Cummings Publishing, 1993.
- CARMONA, Rhadamés. "**El Enfoque Orientado a Objetos"**. Guía para estudiantes de Algoritmos y Programación. UCV. 2004.
- COTO, Ernesto. **Notación Algorítmica** para estudiantes de Algoritmos y Programación. UCV.
- DEITEL & DEITEL. **"Cómo Programar en Java"**. Pearson Prentice Hall, 2004
- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. **"Texto: Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos"**. McGraw-Hill, 2003
- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. **"Libro de problemas: Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos"**. McGraw-Hill, 2003
- WU, Thomas C. **"Introducción a la Programación Orientada a Objetos en Java"**. McGraw-Hill. 2001