Facultad de Ingeniería

Universidad Nacional de Jujuy

FUNDAMENTOS DE PROGRAMACIÓN

Conceptos de Teoría
2023

AGRADECIMIENTOS A LA CÁTEDRA DE PROGRAMACIÓN ESTRUCTURADA DE LA CARRERA APU - FI

CONCEPTO BÁSICOS

¿Qué es un sistema?

Un sistema es un conjunto de componentes interrelacionados que trabajan juntos para alcanzar un objetivo predefinido. Por ejemplo, el sistema circulatorio humano (Figura 1).

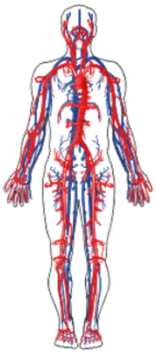


Figura 1. Sistema circulatorio humano.

¿Qué es un sistema de procesamiento de información?

Un *sistema de procesamiento* de información (Figura 2) es un sistema que transforma datos brutos en información organizada, significativa y útil. Un sistema de procesamiento de información consta de 3 componentes: entrada (datos), procesador (métodos de transformación de información) y salida (información procesada).

Figura 2. Sistema de procesamiento de información.

Para llevar a cabo el procesamiento de la información se requiere de un conjunto de instrucciones que especifiquen la secuencia de operaciones a realizar, en orden, para resolver un problema específico o clase de problemas. A este conjunto de instrucciones se denomina *algoritmo* (Figura 3).

Figura 3. Un sistema de información utiliza un algoritmo para realizar la transformación de la entrada.

Cuando el procesador de información es una computadora, el algoritmo debe expresarse como programa. Un programa se escribe en algún lenguaje de programación y la tarea de expresar un algoritmo como programa se denomina *programación*. Los pasos de un algoritmo se traducen a instrucciones de programa, que indican las operaciones a realizar por la computadora.

¿Qué es una computadora?

Una computadora es un dispositivo electrónico utilizado para procesar información y obtener resultados. Los componentes físicos de una computadora, que permiten introducir datos y obtener información se denominan *hardware*. El conjunto de programas y datos que controlan el funcionamiento de una computadora se denomina *software*.

El hardware consta básicamente de los siguientes componentes:

- unidad central de proceso (UCP)
- memoria central o principal
- dispositivos de almacenamiento secundario
- periféricos o dispositivos de entrada, salida y entrada/salida

Unidad Central de Proceso

La UCP (CPU en inglés, Central Processing Unit) dirige y controla el proceso de información realizado por la computadora. La UCP procesa o manipula información almacenada en memoria; puede recuperar información desde memoria (instrucciones o datos de programa). También puede almacenar los resultados de procesos en memoria. Es decir, que la UCP puede leer y escribir datos en memoria.

La UCP consta de 2 componentes principales:

Unidad de Control (UC), cuya función es coordinar las actividades de la computadora y determinar qué operaciones deben realizarse y en qué orden; además de controlar y sincronizar todo el proceso de la computadora.

Unidad Aritmético-Lógica, cuya función es realizar las operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones.

Memoria Central

La memoria central o principal de una computadora se utiliza para almacenar información. En general, la información almacenada en memoria puede ser de 2 tipos: instrucciones de programa o datos con los que operan las instrucciones. Para que un programa se ejecute debe cargarse (load) en memoria principal, asimismo los datos utilizados en el procesamiento también deben ser cargados.

La memoria se organiza en unidades de almacenamiento individual (celdas) denominadas registros o palabras. Un registro o palabra es un conjunto, generalmente, de 8 bits al que se llama byte (octeto). Un bit es la unidad mínima de información que puede almacenarse y representa un valor cero o uno. Cada registro o palabra de memoria tiene asociado 2 conceptos: dirección y contenido. La dirección de una palabra indica su posición en la memoria, y permite especificar que byte será leído o escrito. El contenido de una palabra de memoria hace referencia al valor almacenado en esa posición.

Memoria Secundaria

Para ejecutar un programa es necesario que éste se cargue en la memoria principal. La memoria tiene capacidad limitada (tamaño) y pierde su contenido cuando la computadora se apaga o sufre la interrupción del suministro de energía eléctrica. Es por ello que los dispositivos de almacenamiento masivo se hacen necesarios para guardar tanto datos como programas de modo permanente. Los discos rígidos, cintas magnéticas (tape backup), soportes ópticos y dispositivos extraíbles en general, constituyen las memorias auxiliares, externas o secundarias. En comparación con éstas, la memoria principal es más rápida (en la lectura y escritura de datos) y costosa, pero también volátil (se borra al no tener energía).

Dispositivos de entrada, salida y entrada/salida

Estos dispositivos permiten la comunicación entre el usuario y la computadora. Los dispositivos o periféricos de entrada, sirven para introducir datos para el procesamiento. Los datos se leen desde la entrada y se almacenan en memoria principal. Los dispositivos de entrada convierten la información de entrada para que pueda ser almacenada en la memoria principal. Ejemplos de dispositivos de entrada son: teclado, mouse, lápiz óptico, lectores de códigos de barras, etc.

Los dispositivos de salida permiten presentar los resultados del procesamiento de datos. Ejemplos de dispositivos de salida son: monitor, impresoras, plotters (trazadores gráficos), parlantes, etc.

Los dispositivos de entrada/salida cumplen tanto la función de ingreso de datos como la de presentación de resultados. Por ejemplo, las impresoras multifunción permiten escanear imágenes (entrada) y obtener resultados impresos (salida).

La figura 4 presenta la estructura física general de una computadora.

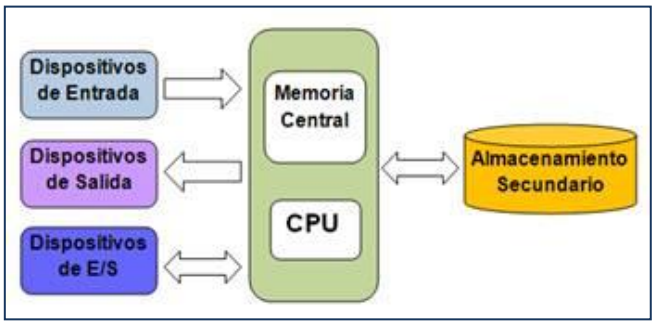


Figura 4. Estructura física de una computadora.

El componente lógico de una computadora, el software, permite especificar las operaciones que se realizarán sobre el hardware. En particular el software de sistemas se ocupa de proporcionar servicio a otros programas y administrar los recursos (memoria, los discos, los dispositivos, etc.) y tiempo de una computadora. Ejemplo de software de sistemas son los sistemas operativos, editores de texto, compiladores/intérpretes y programas de utilidad.

Resolución de Problemas con Computadora

Como ya se mencionó, un algoritmo especifica el conjunto de pasos que debe seguirse para resolver un problema. En el contexto de la Informática un algoritmo se traduce a un programa mediante algún lenguaje de programación, este proceso se denomina *programación*. Por lo tanto, el objetivo de la programación es proporcionar soluciones basadas en computadora que resuelvan problemas considerados engorrosos o difíciles para una persona. La metodología necesaria para resolver problemas mediante computadoras se llama *metodología de la programación*. El punto de partida de esta metodología es el algoritmo. Todo algoritmo debe cumplir con las siguientes características:

- un algoritmo debe ser preciso, es decir, debe indicar el orden de realización de cada paso,
- un algoritmo debe ser definido, es decir, si un algoritmo se sigue 2 veces (considerando el mismo conjunto de datos de entrada) los resultados que se obtengan deben ser los mismos.
- un algoritmo debe ser finito, es decir, si se sigue un algoritmo éste debe finalizar en algún momento (debe tener un número finito de pasos)

Los pasos para obtener una solución basada en computadora son:

1.- Análisis del problema
2.- Diseño del algoritmo
3.- Codificación
4.- Compilación y ejecución
5.- Verificación y depuración
6.- Documentación y mantenimiento

A continuación se describen los pasos enunciados.

Análisis del Problema

En este paso se debe definir claramente el problema a resolver, la información que se utilizará para calcular la solución (datos de entrada), el resultado o solución deseada (datos de salida) y el objetivo del algoritmo que alcanzará la solución.

Diseño del Algoritmo

En la fase de diseño se determina cómo el algoritmo llevará a cabo el procesamiento de la información de entrada para obtener el resultado deseado. Entonces el diseñador descompone el problema en subproblemas y éstos a su vez en subproblemas más simples, aumentando en cada nivel de descomposición el detalle de descripción del algoritmo. El enfoque de dividir el problema en subproblemas se denomina *diseño descendente* (top-down design) y la descripción más detallada de los pasos del algoritmo en cada nivel de descomposición se denomina *refinamiento sucesivo*.

Las ventajas más importantes del diseño descendente son:

- el problema se comprende más fácilmente al dividirse en partes más simples denominadas módulos (conjunto de acciones que tienen un punto de entrada y un punto de salida)
- las modificaciones en los módulos son más fáciles
- la comprobación del problema se puede realizar fácilmente

El diseño abarca:

- diseño descendente
- refinamiento por pasos
- herramientas de programación para la representación de algoritmos

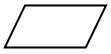
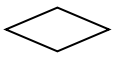
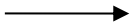
Las herramientas de programación permiten representar un algoritmo a través de técnicas que independizan al diseñador de la sintaxis de un lenguaje de programación específico. Los métodos más usuales para representar algoritmos son:

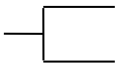
- diagramas de flujo,
- diagramas N-S,
- pseudocódigo,
- lenguaje español (una descripción narrativa) o
- fórmulas

Diagramas de Flujo

Un *diagrama de flujo* (flowchart) es una técnica de representación de algoritmos que utiliza símbolos (cajas) estándar y que tiene los pasos del algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se deben ejecutar. La tabla 1 presenta los símbolos más utilizados en los diagramas de flujo.

Tabla 1. Símbolos del Diagrama de Flujo.

SÍMBOLO	FUNCIÓN
	Terminal, representa el “inicio” y el “fin” de un programa.
	Entrada/salida, representa cualquier tipo de introducción de datos en la memoria desde los periféricos, “entrada”, o registro de la información procesada en un periférico, “salida.”
	Proceso, representa cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.
	Decisión/Decisión Múltiple, indica operaciones lógicas o de comparación entre datos, y en función del resultado determina cuál de los caminos alternativos del programa seguir.
	Conector, permite enlazar 2 partes de un ordinograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo, señala el sentido de la ejecución de las operaciones.
	Línea conectora, permite conectar 2 símbolos.

SÍMBOLO	FUNCIÓN
	Conector, permite conectar 2 puntos del organigrama situado en páginas diferentes.
	Llamada a subrutina o un proceso predeterminado. Una subrutina es un módulo independiente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal.
	Pantalla, se utiliza en ocasiones en lugar del símbolo E/S.
	Impresora, se utiliza en ocasiones en lugar del símbolo E/S.
	Teclado, se utiliza en ocasiones en lugar del símbolo E/S.
	Comentarios, permite añadir comentarios a los símbolos del diagrama de flujo, se pueden dibujar a cualquier lado del símbolo.

Diagramas Nassi-Schneiderman (N-S)

El diagrama N-S de Nassi-Schneiderman (también conocido como diagrama de Chapin) es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Pseudocódigo

El pseudocódigo es un lenguaje de especificación de algoritmos. El uso de tal lenguaje hace el paso de codificación final relativamente fácil. La ventaja del pseudocódigo es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje de programación específico. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a los lenguajes estructurados.

El pseudocódigo utiliza un conjunto de palabras (palabras reservadas) para representar las estructuras de control que implementan las acciones que el algoritmo debe realizar. La escritura de pseudocódigo exige normalmente la indentación de diferentes líneas. Además se pueden incluir comentarios en el pseudocódigo que permitan documentarlo, éstos se indican precediéndolos por el símbolo "//".

Codificación

Codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo en un programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural expresarlas en el lenguaje de programación correspondiente.

Es conveniente aclarar que los lenguajes de programación se clasifican básicamente en:

- lenguajes máquina,
- lenguajes de bajo nivel (ensamblador) y
- lenguajes de alto nivel

Los lenguajes máquina son aquellos que están escritos en lenguajes directamente inteligibles por la computadora, ya que sus instrucciones son cadenas binarias (secuencias de 0's y 1's) que especifican una operación y las posiciones de

memoria implicadas en la operación. Las instrucciones en lenguaje máquina dependen del hardware de la computadora, y por tanto, serán diferentes de una computadora a otra. Esto lleva a que el programador requiera de un profundo conocimiento de la estructura interna de la computadora para la que escribe el programa.

Los lenguajes de bajo nivel son más fáciles de utilizar que los lenguajes máquina, pero, al igual que ellos, dependen del hardware de la computadora. El lenguaje de bajo nivel por excelencia es el ensamblador. Las instrucciones en ensamblador son conocidas como nemotécnicos (por ejemplo, ADD es una instrucción de suma). Estos nemotécnicos se utilizan en lugar de los códigos máquina de las instrucciones correspondientes. Un programa en lenguaje ensamblador no es directamente ejecutable por la computadora, sino que requiere una traducción a lenguaje máquina.

Los lenguajes de alto nivel son los más utilizados por los programadores y están diseñados para que las personas escriban y entiendan los programas fácilmente. Además los programas escritos en lenguajes de alto nivel son independientes del hardware de la computadora, por lo que se dice que son portables o transportables (pueden ser ejecutados en diferentes computadoras con poco o ninguna modificación). Los programas en lenguajes de alto nivel deben ser traducidos por programas traductores (compiladores/intérpretes) para poder ser ejecutados.

Compilación y Ejecución

Una vez que el algoritmo se ha traducido en un programa fuente (generalmente en un lenguaje de alto nivel), es preciso introducirlo en memoria mediante el teclado y almacenarlo en disco. Esta operación se realiza con un programa editor, posteriormente el programa fuente se convierte en un archivo de programa que se graba en disco.

El programa fuente debe ser traducido a lenguaje máquina. Este proceso se realiza con el compilador y el sistema operativo que se encarga de la compilación. Si tras la compilación se presentan errores (errores de compilación) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Esto se repite hasta que no se producen errores, obteniéndose el programa objeto que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de montaje o enlace (link), que carga el programa objeto con las librerías del programa compilador. El montaje genera un programa ejecutable. La Figura 5 ilustra el proceso de compilación y ejecución de un programa.

Cuando el programa ejecutable se ha creado, se puede ejecutar. Suponiendo que no se producen errores durante la ejecución se obtendrán los resultados del programa.

Verificación y Depuración

La verificación de un programa es el proceso de ejecución del programa con una amplia variedad de datos, llamados datos de test o prueba, que determinarán si el programa tiene errores (“bugs”). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

1.- *Errores de compilación*. Se producen normalmente por el uso incorrecto de las reglas del lenguaje de programación y suelen ser errores de sintaxis. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.

2.- *Errores de ejecución*. Estos errores se producen por instrucciones que la computadora puede comprender pero

no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.

3.- *Errores lógicos*. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente, compilar y ejecutar una vez más.

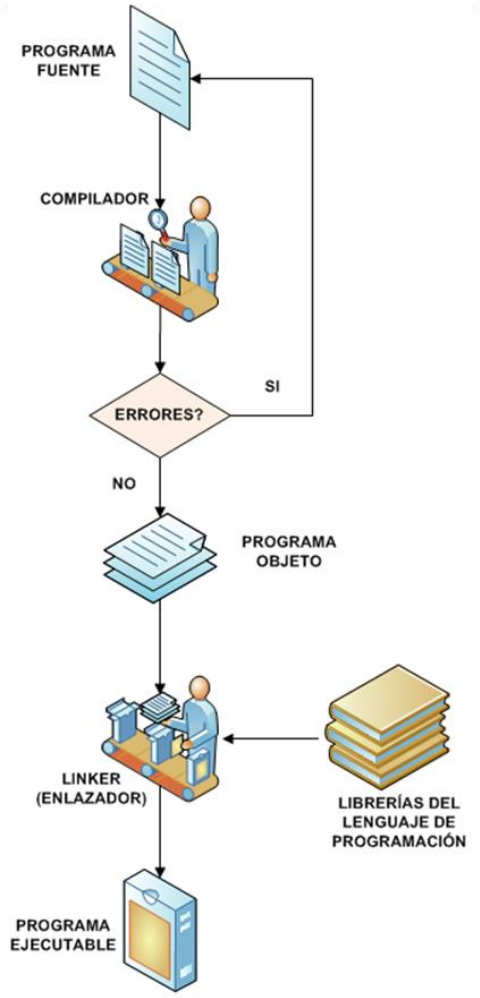


Figura 5. Proceso de compilación y ejecución de un programa.

Documentación y Mantenimiento

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de un problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser interna y externa. La *documentación interna* es la contenida en las líneas de comentarios que aparecen en el programa fuente. La *documentación externa* incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar un programa. Tales cambios se denominan *mantenimiento* del programa. Después de cada cambio la documentación debe ser actualizada para

facilitar cambios posteriores. En la práctica, se suele numerar las sucesivas versiones de los programas 1.0, 1.1, 2.0, 2.1, etc. (si los cambios introducidos son importantes, se varía el primer dígito [1.0, 2.0, …], en caso de pequeños cambios sólo se varía el segundo dígito [2.0, 2.1, …]).

Partes de un Programa

Los componentes básicos de un programa son instrucciones y datos. Las instrucciones o acciones representan operaciones que ejecutará una computadora al *interpretar* un programa. Los lenguajes de programación brindan un conjunto de instrucciones que permiten escribir programas (soluciones algorítmicas) para resolver problemas específicos. Los datos son los valores que deben ser procesados por la computadora (que ejecuta el programa) para obtener resultados o información. Los datos pueden ser constantes (si no cambian) o variables (si sufren modificaciones durante la ejecución del programa).

Documentación de Algoritmos

Un programa bien documentado es más fácil de leer y mantener. Para ello la *documentación* es fundamental. La mayoría de los lenguajes de programación proporcionan algún mecanismo de documentación, por ejemplo, a través de comentarios en el programa (documentación interna). Es recomendable utilizar un comentario general para el objetivo del programa en cuestión, que refleje la especificación del problema a resolver de la forma más completa posible. Además deben describirse todos los objetos de datos que se utilicen en el programa así como fecha de creación, autor, etc. Cuando se realizan modificaciones a un programa es necesario que se actualicen los comentarios para reflejar los cambios llevados a cabo sobre el código.

Corrección de Algoritmos

Un algoritmo es correcto cuando cumple con su especificación, esto significa que cumple con los requerimientos propuestos. Para determinar cuáles son esos requerimientos se debe tener una especificación completa, precisa y libre de ambigüedades (imprecisiones) del problema a resolver antes de escribir el mismo.

Una de las formas de determinar si un programa es correcto consiste en realizar pruebas con un juego de datos reales que permitan establecer si cumple con su especificación.

Eficiencia de Algoritmos

Un problema puede tener varias soluciones algorítmicas. Sin embargo, el uso de los recursos de la computadora (tiempo, memoria) que ejecuta cada una de las soluciones puede ser muy diferente.

La eficiencia se define como una medida de la calidad de los algoritmos que está asociada a la utilización óptima de los recursos de la computadora que ejecuta el algoritmo.

Mantenimiento de Programas

El mantenimiento de un programa involucra básicamente dos tareas: corregir errores y modificar código. Los errores pueden ocurrir antes que el programa esté terminado o bien pueden ser descubiertos más tarde (cuando ya está en uso). En estas situaciones se debe identificar los errores, determinar las posibles causas y corregir el código correspondiente. Las modificaciones del código, en general, están asociadas a la necesidad de agregar funcionalidades a los programas o mejorar el rendimiento actual de éstos.

Según la naturaleza de los cambios que deban realizarse en un programa, el mantenimiento puede ser:

- Correctivo: una vez entregado, un programa aún puede contener errores que el usuario descubre al utilizarlo. El mantenimiento correctivo tiene por objetivo localizar y eliminar estos errores.
- Preventivo: este tipo de mantenimiento consiste en la modificación del software para mejorar sus propiedades sin alterar su funcionalidad (las tareas que realiza). Por ejemplo, se pueden incluir sentencias que comprueben la validez de los datos de entrada, reestructurar los programas para mejorar su legibilidad, o incluir nuevos comentarios que faciliten su comprensión.
- Adaptativo: este tipo de mantenimiento consiste en la modificación de un programa debido a cambios en el entorno (hardware o software) en el cual se ejecuta. Este mantenimiento permite al software adaptarse a las nuevas condiciones en las que debe trabajar.
- Perfectivo: el mantenimiento perfectivo se define como el conjunto de actividades para mejorar o añadir nuevas funcionalidades requeridas por el usuario.

La experiencia demuestra que un programa bien diseñado reduce el costo de mantenimiento del software.

Reusabilidad de Código

En el desarrollo de soluciones basadas en computadora es común que se escriban componentes de programa de propósito general. Esto hace posible que cada vez que se crea un nuevo programa se usen los componentes diseñados anteriormente, reduciéndose así los costos, tiempo y esfuerzo invertidos en la creación de software.

DATOS SIMPLES Y OPERACIONES BÁSICAS

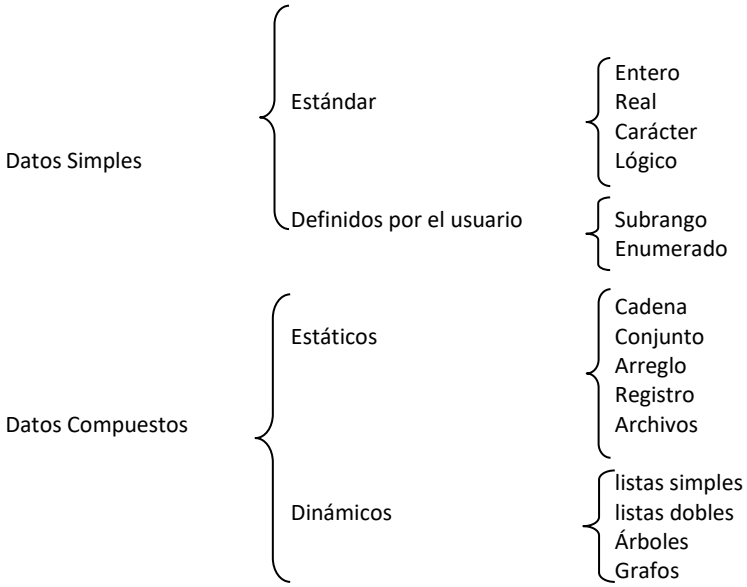
Un programa es un conjunto de instrucciones y datos que permiten solucionar problemas de forma eficiente y rápida, agilizando el trabajo de las personas. Los datos e información que procesa un programa, así como las operaciones que realiza tienen asociados una serie de conceptos que aquí se presentan.

Tipos de datos, tipos de datos abstractos y estructuras de datos

El término *tipo de datos* hace referencia a un conjunto de valores, mientras que el término *tipo de datos abstracto* (TDA) comprende tanto el conjunto de valores como las operaciones que pueden aplicárseles. Por ejemplo, sobre valores de tipo entero se pueden realizar tales como suma, resta, producto, cociente, etc.

Una *estructura de datos* se refiere a la implementación física de un *tipo de datos abstracto*, caracterizándolos por su organización y operaciones.

Los tipos de datos más utilizados en los diferentes lenguajes de programación son:



- Datos Simples: Los tipos de datos simples caracterizan a las formas de presentación más sencillas de objetos de datos e información de un programa. Los lenguajes de programación permiten representar y manipular datos enteros, reales, caracteres y lógicos. Además es posible que el usuario defina tipos especiales, como subrango o enumerado, de acuerdo a los requerimientos que presente un problema en particular.
- Datos Compuestos: Los tipos de datos estructurados están construidos a partir de los tipos de datos simples y pueden ser estáticos o dinámicos. Las estructuras de datos estáticas son aquellas cuyo tamaño en memoria debe ser definido antes que el programa se ejecute y que no pueden modificar dicho tamaño durante su ejecución.

  Las estructuras de datos dinámicas pueden modificar el tamaño de memoria que ocupan mientras se ejecuta el programa, solicitando o liberando memoria según se requiera. Mediante el uso de un tipo de datos especial, denominado puntero, se pueden construir estructuras dinámicas soportadas por la mayoría de los lenguajes.

Es importante destacar que los tipos de datos simples tienen como característica común que cada objeto de datos representa a un elemento; mientras que los tipos compuestos tienen como característica común que un identificador (nombre de un objeto de datos) puede representar múltiples datos individuales, pudiendo referenciarse cada uno en forma

independiente.

A continuación, se describen los tipos de datos simples, definidos por el usuario y algunos tipos estructurados.

Datos Simples

Los diferentes objetos de información con los que trabaja un programa de computadora se conocen como datos. Todos los datos tienen un tipo asociado a ellos. El tipo de un dato determina el conjunto de valores que éste puede asumir. Los tipos de datos simples utilizados en casi todos los lenguajes de programación son:

Datos Numéricos

El tipo numérico es el conjunto de valores numéricos. Éstos pueden representarse en dos formas:

- Enteros: este tipo es un subconjunto de los números enteros, es decir, se trata de números sin parte decimal y que pueden ser positivos o negativos. Por ejemplo: -123, 0, 48, etc.
- Reales: este tipo es un subconjunto de los números reales, es decir, se trata de números con parte entera y parte decimal, que pueden ser positivos o negativos. Por ejemplo: -234.33, 0.0, 78.21, etc.

Datos Lógicos

El tipo lógico es un tipo ordinal, es decir, que tiene un número fijo de posibles valores y orden definido para éstos.

- Lógico: el tipo lógico o booleano puede tomar sólo 2 valores: Verdadero (V) o Falso (F). En general, este tipo se utiliza para representar la ocurrencia o no de un suceso o condición. Se considera que el valor Falso es menor que el valor Verdadero.

Datos Carácter

El tipo carácter representa una letra (‘a’, ‘A’), un dígito (‘0’,’9’) o símbolo especial (‘@’, ‘&’, ‘#’). Los datos de tipo carácter son ordinales.

- Caracteres: este tipo es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo carácter contiene un solo símbolo, dígito o letra.

Datos Tipo Cadena de Caracteres

Una cadena de caracteres es un conjunto de caracteres (incluido el espacio en blanco) reconocidos por la computadora, que se almacenan en posiciones de memorias contiguas. La longitud de una cadena es el número de caracteres que ésta contiene. La cadena que no contiene ningún carácter se denomina vacía o nula.

La representación de las cadenas suele utilizar comillas simples o dobles. Por ejemplo, la cadena *‘hola mundo’* está delimitada por comillas simples.

Una subcadena es un conjunto de caracteres extraído de una cadena de mayor longitud. Por ejemplo, si se considera la cadena *‘Escuela de Minas’*, la cadena *‘Escuela’* es subcadena de la primera.

Constantes y Variables

Una *constante* es un objeto de datos cuyo valor no cambia durante la ejecución de un programa. Una *constante* recibe su valor al momento de la compilación del programa y este valor no será modificado durante la ejecución.

Una *variable* es un objeto de datos de programa cuyo valor puede cambiar durante la ejecución de un programa. Este cambio se producirá mediante sentencias ejecutables. Una variable es una posición de memoria con nombre. El nombre de la posición se llama nombre de variable, y el valor almacenado en la posición se llama valor de la variable.

Tanto constantes como variables se definen con un tipo específico (numérico, lógico, carácter).

Operadores

Los operadores permiten realizar acciones sobre los datos de acuerdo al tipo de éstos. A continuación se presentan los operadores más utilizados en los lenguajes de programación.

Tipo	Símbolo	Nombre	Función
Paréntesis	( )		Anida expresiones
Aritméticos	** ó ^ *, / +, - div, mod	Potencia Producto, división Suma, diferencia División entera, resto	Conectan objetos o campos numéricos
Alfanuméricos	+	Concatenación	Conectan campos alfanuméricos
Relacionales	= < <= > >= <>	Igual a Menor que Menor o igual que Mayor que Mayor o igual que Distinto a	Conectan objetos, campos o expresiones de cualquier tipo. Su evaluación da como resultado "Verdadero" o "Falso".
Lógicos	NOT AND OR	Negación Conjunción Disyunción	Conectan expresiones de tipo lógico. Su evaluación da como resultado "Verdadero" o "Falso".

Los operadores lógicos NOT, AND y OR se evalúan según tablas de verdad. Considerando las variables booleanas *a* y *b*, las siguientes tablas indican los valores que se obtienen al aplicarles los operadores indicados.

Operador NO (NOT)

A	*NO a*
Verdadero	Falso
Falso	Verdadero

Operador Y (AND)

a	*b*	*a Y b*
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

Operador O (OR)

A	*b*	*a O b*
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Cuando se combinan operadores, es necesario considerar el orden en que se resuelven para obtener un resultado correcto. La siguiente tabla indica la precedencia de operadores.

Operador	Prioridad
NO, ^	Más alta (se evalúa primero)
*, /, Y,div, mod	
+, -, O	
<, <=, =, <>, >=, >	Más baja (se evalúa al final)
Si se utilizan paréntesis, las expresiones encerradas se evalúan primero.	

Expresiones

Las expresiones son combinaciones de constantes, variables, símbolos de operación y nombres de funciones especiales. Una expresión consta de *operandos* y *operadores*. De acuerdo al tipo de datos que manipulan las expresiones, éstas se clasifican en *aritméticas*, *alfanuméricas* y *lógicas*.

Las expresiones aritméticas son análogas a fórmulas matemáticas, en donde se utilizan operadores aritméticos y variables y constantes numéricas (reales o enteras).

Las expresiones alfanuméricas son aquellas en las que se utilizan operadores alfanuméricos y se producen resultados de tipo alfanumérico.

Las expresiones lógicas utilizan tanto operadores lógicos como relacionales para representar alguna condición. Al evaluar esta condición se obtiene un valor VERDADERO o FALSO.

Expresión	Tipo	Observación
3 + 9	Expresión numérica (entera)	
2.5 * 6.8 + 2	Expresión numérica (real)	
A + B * 2	Expresión numérica	A y B deben ser variables o de tipo numérico.

Falso Y Verdadero	Expresión lógica	
23 >= 30	Expresión lógica	
(A < B) Y (C = Falso)	Expresión lógica	A y B deben ser variables del mismo tipo y C debe ser variable de tipo lógico.

Escritura de Expresiones Algorítmicas

Las expresiones que tienen 2 o más operandos requieren de reglas matemáticas que permitan determinar el orden de las operaciones, estas reglas se denominan de prioridad o precedencia y son:

- las operaciones encerradas entre paréntesis se resuelven primero. Si existen paréntesis anidados se resuelven primero los más interiores.
- las operaciones aritméticas dentro de una expresión siguen el orden de precedencia indicado en la tabla Operador/Prioridad presentada anteriormente.

En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de prioridad es de izquierda a derecha.

Considerando las reglas de prioridad y los operadores aritméticos definidos, las expresiones matemáticas regulares pueden traducirse a expresiones aritméticas algorítmicas equivalentes que utilizan los lenguajes de programación. Por ejemplo, dadas las siguientes expresiones matemáticas sus equivalentes expresiones aritméticas algorítmicas son:

Expresión Matemática	Expresión Aritmética Algorítmica
$a \times x^2 + b \times x + c$	a * x ^ 2 + b * x + c
$\frac{-b + \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$	(-b + (b ^ 2 – 4 * a * c) ^ (1 / 2))/(2 * a)
$\frac{a+2}{b} - \frac{c-4}{d \times 2}$	(a + 2) / b - ((c - 4) / (d * 2))

Asignación

La operación de *asignación* es el modo de darle valores a una variable. La operación de asignación se representa con el símbolo u operador ←. La operación de asignación se conoce como instrucción o sentencia de asignación cuando se refiere a un lenguaje de programación.

El formato general de la operación asignación es:

Nombre_Variable ← Expresión

La asignación es destructiva, esto significa que cuando se le asigna un valor a una variable se pierde el valor anterior. Por ejemplo, si la variable **suma** tiene valor 5, y luego se le asigna 12, éste será el valor final de la variable.

Las acciones de asignación se clasifican según el tipo de expresiones en: aritméticas, lógicas y de caracteres.

Asignación	Descripción	Ejemplos
Aritméticas	Las expresiones en las operaciones de asignación son aritméticas.	cantidad ← 100 total ← suma * 2 + 30
Lógicas	La expresión que se evalúa en la operación de asignación es lógica.	mayor ← 23 > 4 estado ← verdadero y (16 = 2 * valor)
Caracteres	La expresión que se evalúa es de tipo carácter o cadena de caracteres.	letra ← ‘a’ palabra ← ‘programación’

Entrada y Salida de Información

Los cálculos que realizan las computadoras requieren, para ser útiles, de la entrada de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, la salida. Las operaciones de entrada permiten leer determinados valores y asignarlos a variables específicas. Esta entrada se conoce como operación de *Lectura*.

Los datos se introducen al procesador mediante dispositivos de entrada (teclado, unidades de disco, etc.). La salida puede aparecer en un dispositivo de salida (pantalla, impresora, etc.). La operación de salida se denomina *Escritura*.

Las operaciones de *Lectura* y *Escritura* presentan el siguiente formato general:

LEER lista de variables de entrada

ESCRIBIR lista de expresiones de salida

Por ejemplo,

- la operación LEER *a*, *b*, *c* representa la lectura de 3 valores que se asignan a las variables *a*, *b* y *c*.
- la operación ESCRIBIR *'Hola Mundo'* visualiza en el dispositivo de salida elegido el mensaje "Hola Mundo"

Funciones Internas

Las operaciones que realizan los programas exigen en numerosas ocasiones, además de las operaciones básicas, un número determinado de operaciones especiales que se denominan funciones internas, incorporadas o estándar. Los lenguajes de programación incorporan estas funciones para facilitar el trabajo del programador. La siguiente tabla presenta algunas de las funciones internas incluidas en los lenguajes de programación.

Función	Descripción	Tipo de argumento	Resultado
abs(x)	valor absoluto de x	entero o real	igual que el argumento
arctan(x)	arco tangente de x	entero o real	Real
cos(x)	coseno de x	entero o real	Real
exp(x)	exponencial de x	entero o real	Real
ln(x)	logaritmo neperiano de x	entero o real	Real
log10(x)	logaritmo decimal de x	entero o real	Real
redondeo(x)	redondeo de x	Real	Entero
sen(x)	seno de x	entero o real	Real
cuadrado(x)	cuadrado de x	entero o real	igual que el argumento
raíz2(x)	raíz cuadrada de x	entero o real	Real

Operaciones con Cadenas

El tratamiento de cadenas contempla básicamente las siguientes operaciones:

- Cálculo de longitud
- Comparación
- Concatenación
- Extracción de subcadenas

Cálculo de Longitud de una Cadena

La longitud de una cadena es el número de caracteres de la cadena. Por ejemplo, **'Hola Mundo!!!'** es una cadena de que tiene 13 caracteres (incluido el espacio en blanco).

La operación para determinar la longitud de una cadena se representará por la función longitud, cuyo formato es:

Longitud(cadena de caracteres**)**

Esta función, cuyo argumento es una cadena, retorna un valor entero que indica el número de caracteres de la cadena.

Comparación de Cadenas

La comparación de cadenas (igualdad y desigualdad) se basa en el orden numérico del código o juego de caracteres (por ejemplo, código ASCII) que admite una computadora.

Dos cadenas *a* y *b* de longitudes *m* y *n* son iguales si:

- El número de caracteres de *a* es igual al número de caracteres de *b* (*m*=*n*)
- Cada carácter de *a* es igual a su correspondiente de *b*; si $a=a_1a_2a_3\ldots a_m$ y $b=b_1b_2b_3\ldots b_n$, se debe verificar que $a_i=b_i$.

Por ejemplo, las siguientes expresiones comprueban la igualdad de cadenas:

'hola mundo' = 'hola mundo'
'informática'='computadora'
'Escuela de Minas' = 'escuela de minas'

La primera expresión es VERDADERA, la segunda es evidentemente FALSA, y la tercera expresión a primera vista puede parecer VERDADERA, sin embargo, los caracteres 'E' y 'e' son diferentes (el código binario que identifica a cada uno es distinto) por cuanto la expresión es FALSA.

Para comprobar la desigualdad de cadenas, en general, se utilizan los operadores relacionales (>, <, >=, <=, <>) y se ajustan a una comparación sucesiva de caracteres correspondientes en ambas cadenas hasta conseguir dos caracteres diferentes.

Por ejemplo, las siguientes expresiones comprueban la desigualdad de cadenas:

'zapato'<'avestruz'
'CARAMELO'<'golosina'
'amarillo'>='Amarillo'

La primera expresión es FALSA, al comparar el primer carácter de ambas cadenas (en realidad, sus correspondientes códigos binarios) se comprueba que 'z' no es menor que 'a' (considerando los caracteres ASCII). La segunda expresión es FALSA, porque las mayúsculas tienen códigos binarios menores a los de las minúsculas (código ASCII). Finalmente la tercera expresión es VERDADERA.

Concatenación

La concatenación es la operación de reunir varias cadenas de caracteres en una sola, pero conservando el orden de los caracteres de cada una de ellas.

El símbolo que representa la concatenación varía de un lenguaje a otro. Los más utilizados son: +, // y &.

Por ejemplo, la siguiente operación de concatenación combina las cadenas almacenadas en las variables A y B en la variable C.

A← 'facultad'
B← 'de minas'
C←A + B

El contenido de la variable C tras realizar la concatenación es 'facultadde minas'.

Subcadenas

La operación de extraer una parte específica de una cadena se representa por la función *subcadena*. Esta función presenta el siguiente formato:

subcadena(cadena**,** p_inicial**,** p_final**)**

Como puede observarse la función utiliza 3 parámetros, siendo opcional el último. El primer parámetro especifica la cadena original, el segundo indica la posición del primer carácter de la subcadena y el tercero, si se especifica, indica la posición del carácter final de la subcadena.

Por ejemplo:

palabra←subcadena('domingo',3,5)

La variable palabra (tipo cadena de caracteres) almacena cadena 'min'.

frase← 'hola mundo'
subfrase←subcadena(frase,6)

La variable subfrase (tipo cadena de caracteres) almacena la cadena ‘mundo’. Obsérvese que si el último parámetro de la función subcadena no se especifica, los caracteres se extraen desde la posición inicial hasta el último carácter de la cadena original.

CONCEPTOS PROGRAMACIÓN ESTRUCTURADA

El diseño de algoritmos constituye una de las fases más importantes en la resolución de problemas basados en computadora. En esta fase se indican las acciones que se llevarán a cabo para dar solución a un problema específico. El paradigma de la programación estructurada utiliza 3 estructuras de control (secuenciales, condicionales y repetitivas) para representar los pasos de resolución de un algoritmo. A continuación se aborda este paradigma.

Programación Estructurada

La *Programación Estructurada* se refiere al conjunto de técnicas empleadas para aumentar la productividad de los programas, reduciendo el tiempo necesario para escribir, verificar, depurar y mantener los programas. El paradigma de la Programación Estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas, y por consiguiente, reducen los errores.

Este paradigma hace que los programas sean más fáciles de escribir, verificar, leer y mantener. Los programas deben estar dotados de estructura.

La Programación Estructurada es el conjunto de técnicas que incorporan:

- Diseño descendente
- Recursos Abstractos
- Estructuras Básicas

Diseño Descendente

El *diseño descendente* (*TOP-DOWN*) es el proceso mediante el cual un problema se descompone en una serie de niveles o pasos sucesivos de refinamiento (*stepwise*). La metodología descendente consiste en efectuar una relación entre las sucesivas etapas de estructuración, de modo que se relacionen unas con otras mediante entradas y salidas de información. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de modo que se puede considerar cada estructura desde dos puntos de vista: ¿lo que hace? (Figura 2) y ¿cómo lo hace? (Figura 3)

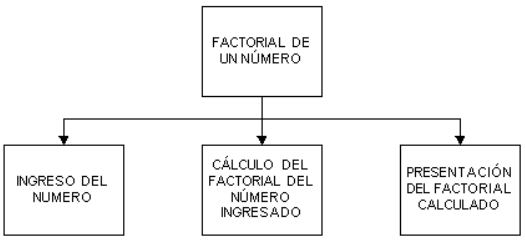


Figura 2. Diseño Top Down para el cálculo del factorial de un número.

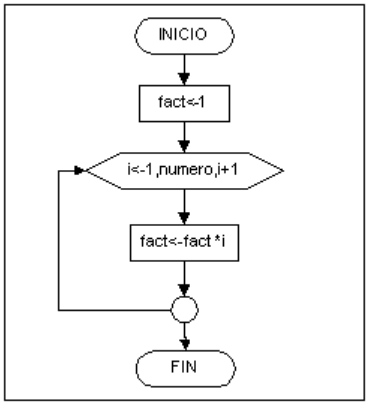


Figura 3. Algoritmo (diagrama de flujo) para el cálculo del factorial de un número.

El diseño descendente comienza con un problema general y se van diseñando soluciones específicas para cada uno de los subproblemas en los que ha sido divido el programa general.

Recursos Abstractos

La Programación Estructurada se auxilia de *recursos abstractos* en lugar de los recursos concretos de que se dispone (un determinado lenguaje de programación).

Descomponer un programa en términos de recursos abstractos, consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples capaces de ser ejecutadas por una computadora y que constituirán sus instrucciones.

Estructuras Básicas: Teorema de la Programación Estructurada

El *Teorema de la Programación Estructurada* establece que un programa propio puede ser escrito utilizando solamente las siguientes estructuras de control:

- Secuenciales
- Selectivas
- Repetitivas

Un programa se define como propio si cumple con las siguientes características:

- tiene exactamente una entrada y una salida para control del programa,
- existen caminos que se pueden seguir desde la entrada hasta la salida que conducen por cada parte del programa, es decir, no existen lazos infinitos ni instrucciones que no se ejecutan.

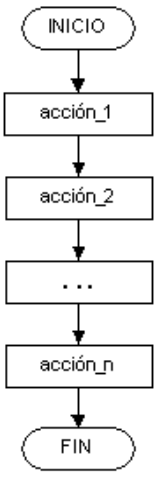
Secuenciales

La estructura de control más simple está representada por una sucesión de operaciones, en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones. La representación en pseudocódigo y diagrama de flujo de una estructura secuencial se presenta a continuación:

Pseudocódigo

```
INICIO
    acción_1
    acción_2
    . . .
    acción_n
FIN
```

Diagrama de Flujo



Observe que *INICIO* y *FIN* indican el punto de entrada y punto de salida, respectivamente, del algoritmo.

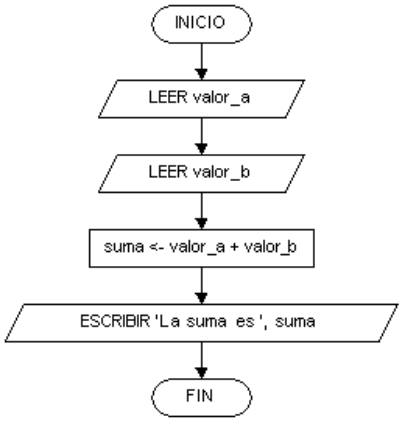
El siguiente ejemplo muestra un algoritmo (pseudocódigo y diagrama de flujo) que resuelve la suma de dos valores (*valor_a* y *valor_b*) ingresados por el usuario a través de instrucciones secuenciales (lectura, asignación, escritura).

En el ejemplo puede observarse el uso de las instrucciones LEER, ESCRIBIR y ASIGNAR (←); así como los símbolos de diagrama de flujo correspondientes a cada instrucción.

Pseudocódigo

```
INICIO
    LEER valor_a
    LEER valor_b
    suma ← valor_a + valor_b
    ESCRIBIR 'La suma es:', suma
FIN
```

Diagrama de Flujo



Selectivas, Condicionales o de Decisión

En general, los problemas que se resuelven utilizando computadoras requieren más que la ejecución de instrucciones una a continuación de la otra. Es por ello, que es prácticamente imposible que las instrucciones sean secuenciales puras. Es necesario tomar decisiones en función de los datos del problema.

Las estructuras selectivas se utilizan para tomar decisiones lógicas. Estas estructuras evalúan una condición y en función del resultado de ésta se realiza una acción u otra. Una *condición* es una expresión lógica que al evaluarse devuelve un valor lógico VERDADERO o FALSO para tomar la decisión.

Las estructuras selectivas pueden ser:

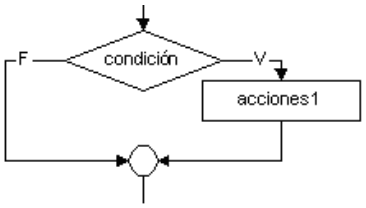
- Simples
- Dobles
- Múltiples

Las estructuras *selectivas simples* (SI-ENTONCES-FIN_SI) permiten realizar un conjunto de acciones si la condición que se evalúa es VERDADERA, caso contrario, dichas acciones no se realizan. La representación en pseudocódigo y diagrama de flujo de una estructura selectiva simple se presenta a continuación:

Pseudocódigo

```
SI condición ENTONCES
    acciones
FIN_SI
```

Diagrama de Flujo

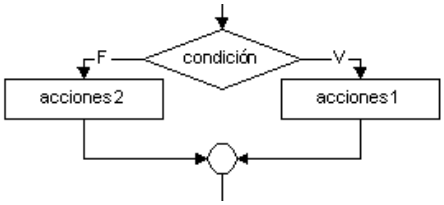


Las estructuras *selectivas dobles* (SI-ENTONCES-SINO-FIN_SI) presentan 2 caminos alternativos de acción, los que se eligen de acuerdo al valor de una determinada condición (VERDADERA o FALSA). La representación en pseudocódigo y diagrama de flujo de una estructura selectiva doble se presenta a continuación:

Pseudocódigo

```
SI condición ENTONCES
    acciones_1
SINO
    acciones_2
FIN_SI
```

Diagrama de Flujo



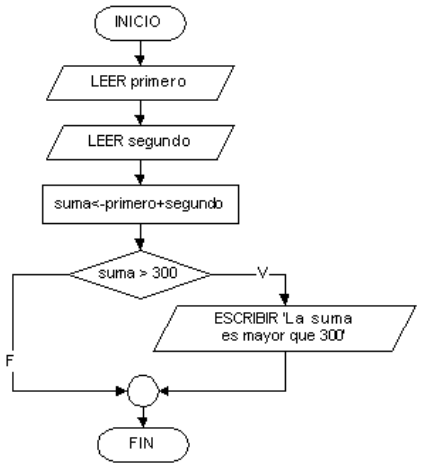
A continuación, se presentan algoritmos que ejemplifican el uso de las instrucciones selectivas en la resolución de problemas simples.

Ejemplo 1: Diseñe un algoritmo que determine si la suma de 2 valores ingresados por el usuario es mayor a 300.

Pseudocódigo

```
INICIO
    LEER primero
    LEER segundo
    suma←primero + segundo
    SI suma > 300 ENTONCES
        ESCRIBIR 'La suma es mayor a 300'
    FIN_SI
FIN
```

Diagrama de Flujo

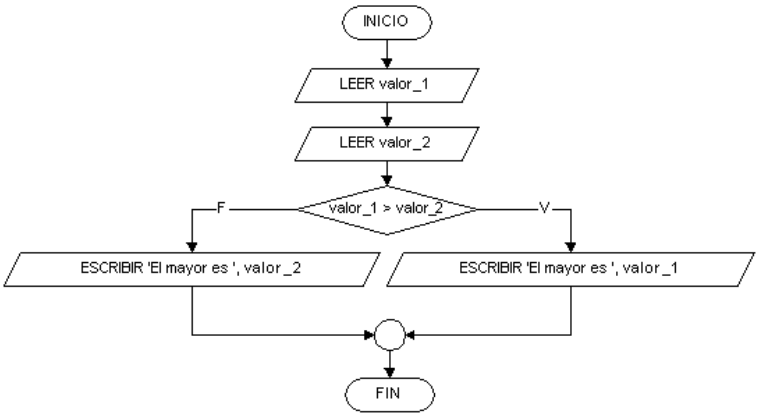


Ejemplo 2: Diseñe un algoritmo que determine cuál es el mayor de dos valores ingresados por el usuario.

Pseudocódigo

```
INICIO
    LEER valor_1
    LEER valor_2
    SI valor_1 > valor_2 ENTONCES
        ESCRIBIR 'El mayor valor es:', valor_1
    SINO
        ESCRIBIR 'El mayor valor es:', valor_2
    FIN_SI
FIN
```

Diagrama de Flujo



Repetitivas e Iterativas

Las computadoras están especialmente diseñadas para todas aquellas aplicaciones en las que una operación o conjunto de ellas deben repetirse muchas veces.

Las estructuras que permiten repetir una secuencia de instrucciones se denominan bucles, y se llama iteración al hecho de repetir la ejecución de una secuencia de acciones.

Las sentencias que permiten especificar la repetición de un conjunto de instrucciones son:

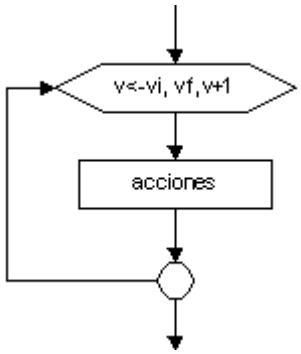
- Para (PARA-FIN_PARA)
- Mientras (MIENTRAS-FIN_MIENTRAS)
- Repetir (REPETIR-HASTA_QUE)

La estructura *iterativa PARA* permite repetir un conjunto de acciones un número conocido de veces. Esta estructura comienza con un *valor inicial* (vi) de la variable de control o índice y las acciones especificadas en el cuerpo del bucle se ejecutan hasta que el índice alcanza el *valor final* (vf). Por defecto, el incremento o decremento del índice se realiza en una unidad, aunque puede especificarse otro valor (PASO n). La variable de control debe ser tipo ordinal, y por lo general se la designa i, j, k.

La representación en pseudocódigo y diagrama de flujo de una estructura iterativa PARA se presenta a continuación:

Pseudocódigo **Diagrama de Flujo**

```
PARA v DESDE vi HASTA vf HACER CON PASO n
      acciones
FIN_PARA
```



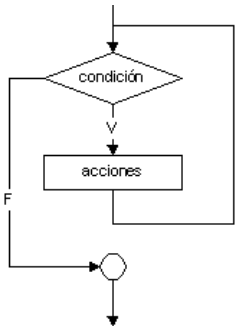
La estructura *iterativa MIENTRAS* permite repetir un conjunto de acciones en tanto la condición evaluada sea VERDADERA, no conociéndose de antemano el número de repeticiones. Esta estructura se clasifica en pre-condicional, ya que evalúa la condición y, si es VERDADERA, entonces ejecuta el bloque de acciones; por cuanto el bloque se puede ejecutar 0, 1 o más veces. En general, esta estructura se utiliza para cálculos aritméticos.

La representación en pseudocódigo y diagrama de flujo de una estructura iterativa MIENTRAS se presenta a continuación:

Pseudocódigo

```
MIENTRAS condición HACER
      acciones
FIN_MIENTRAS
```

Diagrama de Flujo



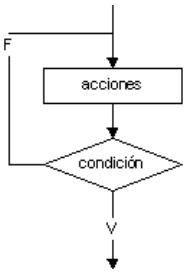
La estructura *iterativa REPETIR* permite repetir un conjunto de acciones en tanto la condición evaluada sea FALSA, no se conoce de antemano el número de repeticiones. Esta estructura se clasifica en pos-condicional, ya que primero ejecuta el bloque de acciones y luego se evalúa la condición; si ésta es FALSA, el bloque de acciones se ejecuta nuevamente. A diferencia de las estructuras pre-condicionales, el bloque de acciones se puede ejecutar 1 o más veces. En general, esta estructura se utiliza para el ingreso de datos.

La representación en pseudocódigo y diagrama de flujo de una estructura iterativa REPETIR se presenta a continuación:

Pseudocódigo

```
REPETIR
      acciones
HASTA_QUE condición
```

Diagrama de Flujo



<u>Diferencias entre las estructuras MIENTRAS y REPETIR</u>

- La estructura *mientras* finaliza cuando la condición es FALSA, en tanto que *repetir* termina cuando la condición es VERDADERA.
- En la estructura *repetir* el cuerpo del bucle se ejecuta al menos una vez, por el contrario, *mientras* es más general y permite la posibilidad de no ejecutar el bucle.

Bucles Infinitos

- Algunos bucles no exigen fin, sin embargo otros no encuentran fin por errores en su diseño. Un bucle que nunca termina se denomina bucle infinito o sin fin. Los bucles sin fin no intencionados son perjudiciales para la programación y siempre deben evitarse.

Finalización de Bucles

- Si un algoritmo o programa ejecuta un bucle cuyas iteraciones dependen de una condición, este bucle puede estar controlado por un valor centinela, una bandera o un contador.
- Un *bucle controlado por centinela* es aquel en el que un valor que se ingresa en el bucle permite finalizar las iteraciones. Este valor especial se denomina valor centinela.
- Un *bucle controlado por bandera* es aquel en el que una variable lógica permite determinar la ocurrencia de un evento y de este modo finalizar o no las iteraciones del bucle. Esta variable lógica se denomina bandera.
- Un *bucle controlado por contador* es aquel en el que una variable se utiliza para contabilizar la cantidad de veces que ocurre algún evento y de este modo finalizar o no las iteraciones del bucle. Esta variable se denomina contador.

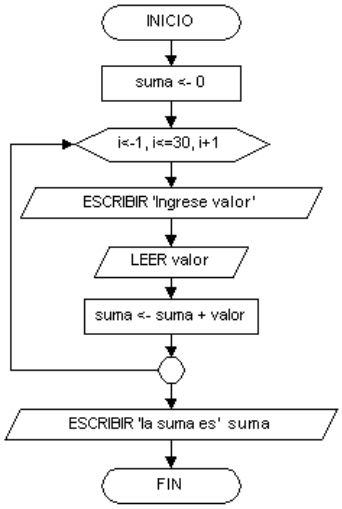
Ejemplos: A continuación, se presentan 3 algoritmos que ejemplifican el uso de las instrucciones repetitivas en la resolución de problemas simples.

Ejemplo 1: Diseñe un algoritmo que sume 30 valores ingresados por el usuario.

Pseudocódigo

```
INICIO
    suma <- 0
    PARA i DESDE 1 HASTA 30 HACER
        ESCRIBIR 'ingrese valor:'
        LEER valor
        suma <- suma + valor
    FIN_PARA
    ESCRIBIR 'la suma es:' suma
FIN
```

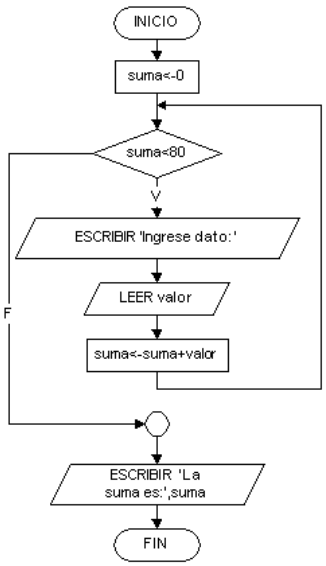
Diagrama de Flujo



Ejemplo 2: Diseñe un algoritmo que calcule la suma de valores ingresados por el usuario en tanto la suma sea menor a 80.

Pseudocódigo

```
INICIO
    suma←0
    MIENTRAS suma < 80 HACER
        ESCRIBIR 'Ingrese dato:'
        LEER valor
        suma←suma+valor
    FIN_MIENTRAS
    ESCRIBIR 'La suma es:', suma
                FIN
```

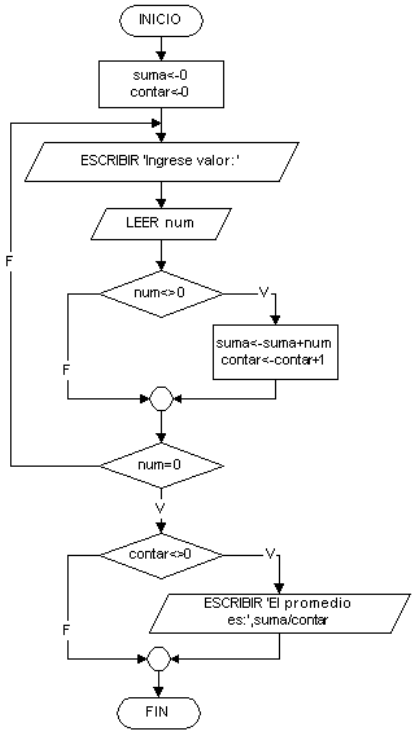


Ejemplo 3: Diseñe un algoritmo que calcule el promedio de valores introducidos por el usuario. El ingreso finaliza cuando el usuario introduce un valor cero.

Pseudocódigo

```
INICIO
    suma←0
    contar<-0
    REPETIR
        ESCRIBIR 'Ingrese valor:'
        LEER num
        SI num <> 0 ENTONCES
            suma←suma+num
            contar←contar+1
        FIN_SI
    HASTA_QUE num=0
    SI contar<> 0 ENTONCES
        ESCRIBIR 'El promedio es:' suma/contar
    FIN_SI
FIN
```

Diagrama de Flujo



Anidamiento de Estructuras de Control

Para diseñar algoritmos que solucionen diversos tipos problemas mediante programas de computadora, la Programación Estructurada permite al diseñador combinar las estructuras de control básicas de una manera flexible. En muchos casos es necesario colocar una estructura de control dentro de otra, a esta forma de combinar las estructuras se denomina anidamiento. El anidamiento debe cumplir con las siguientes reglas de construcción:

- la estructura interna debe quedar completamente incluida dentro de la externa, y
- no puede existir solapamiento.

Los siguientes ejemplos muestran anidamientos válidos de sentencias selectivas.

a)

```
SI condición_1 ENTONCES
    SI condición_2 ENTONCES
        acciones
    FIN_SI
FIN_SI
```

b)

```
SI condición_1 ENTONCES
    SI condición_2 ENTONCES
        acciones
    FIN_SI
ELSE
    acciones
FIN_SI
```

c)

```
SI condición_1 ENTONCES
    SI condición_2 ENTONCES
        acciones
    ELSE
        acciones
    FIN_SI
ELSE
    SI condición_3 ENTONCES
        acciones
    FIN_SI
FIN_SI
```

d)

```
SI condición_1 ENTONCES
    SI condición_2 ENTONCES
        acciones
    ELSE
        acciones
    FIN_SI
ELSE
    SI condición_3 ENTONCES
        acciones
    ELSE
        acciones
    FIN_SI
FIN_SI
```

Obsérvese que para una mejor comprensión y lectura del algoritmo se utilizaron sangrías que permiten diferenciar las estructuras anidadas. Esta es una práctica recomendable al momento de escribir un programa.

Los siguientes ejemplos muestran anidamientos válidos de sentencias repetitivas.

a)
```
MIENTRAS condición_1 HACER
        MIENTRAS condición_2 HACER
                acciones
        FIN_MIENTRAS
FIN_MIENTRAS
```

b)
```
REPETIR
        REPETIR
                acciones
        HASTA_QUE condición_2
HASTA_QUE condición_1
```

c)
```
PARA i DESDE vi HASTA vf HACER
        PARA k DESDE vi_2 HASTA vf_2 HACER
                acciones
        FIN_PARA
FIN_PARA
```

d)
```
MIENTRAS condición_1 HACER
        PARA i DESDE vi HASTA vf HACER
                acciones
        FIN_PARA
FIN_MIENTRAS
```

e)
```
PARA i DESDE vi HASTA vf HACER
        REPETIR
                acciones
        HASTA_QUE condición_1
FIN_PARA
```

f)
```
REPETIR
        MIENTRAS condición_2 HACER
                acciones
        FIN_MIENTRAS
HASTA_QUE condición_1
```

En los bucles anidados, el bucle interno ejecuta todas sus repeticiones para cada una de las iteraciones del bucle externo.

Los siguientes ejemplos muestran anidamientos inválidos de sentencias selectivas y repetitivas.

a)
```
MIENTRAS condición_1 HACER
        SI condición_2 ENTONCES
                acciones
        FIN_MIENTRAS
FIN_SI
```

b)
```
PARA i DESDE vi HASTA vf HACER
        MIENTRAS condición_1 HACER
                acciones
        FIN_PARA
FIN_MIENTRAS
```

c)
```
REPETIR
        MIENTRAS condición_2 HACER
            SI condición_3 ENTONCES
                acciones
        FIN_MIENTRAS
            ELSE
                SI condición_4 ENTONCES
                    acciones
                FIN_SI
            FIN_SI
HASTA_QUE condición_1
```

d)
```
SI condición_1 ENTONCES
        PARA i DESDE vi HASTA vf HACER
        acciones
        FIN_PARA
REPETIR
ELSE
        acciones
FINS_SI
HASTA_QUE condición_2
```

En cada ejemplo se destacan en negritas las estructuras que están incorrectamente anidadas.

Estructura Básica de Programa

Cada vez que se escribe un programa se debe seguir en detalle ciertas reglas de estructura. La estructura básica de un programa es:

- Nombre del programa: es el nombre que se utilizará para designar al programa, este nombre debe estar relacionado con las acciones que realizará el programa. En pseudocódigo, antes del nombre del programa se escribe la palabra *programa*; en los lenguajes de programación se utiliza la palabra *program*. Por ejemplo: programa calcular_superficies, programa sumar_numeros, etc.
- Declaración de variables y constantes: en esta sección se definirán con nombre y tipo, las constantes y variables que se utilizarán dentro del programa.

- Inicio del programa: siempre se debe indicar el punto en el que comienzan las acciones correspondientes al programa. En pseudocódigo se denota con la palabra *inicio* y en los lenguajes de programación por lo general con la palabra *begin*.
- Cuerpo del programa: en esta parte están contenidos todos los procesos y acciones que realizará el programa.
- Fin del programa: así como se indica el inicio de las acciones del programa, también se debe dejar marcado el final de estas acciones, en pseudocódigo esto hace con la palabra *fin* y en lenguajes de programación con la palabra *end*.

Elementos Básicos de Programa

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, y reglas que permiten combinar tales elementos. Estas reglas se denominan sintaxis del lenguaje. Sólo las instrucciones sintácticamente correctas podrán ser interpretadas por la computadora.

Los elementos básicos constitutivos de un programa o algoritmo son:

- Palabras reservadas: son aquellas que tienen un significado especial y permiten indicar partes de un programa o instrucciones. Por ejemplo, las palabras INICIO y FIN delimitan el alcance del ámbito de un programa.
- Identificadores: los identificadores son los nombres con los que se distingue a las variables. Por ejemplo, 2 variables de tipo entero pueden identificarse como *suma* y *producto*.
- Caracteres especiales: los caracteres especiales son aquellos símbolos que tienen una significación propia para el lenguaje.
- Constantes: las constantes son objetos de datos que no varían su valor durante la ejecución de un programa.
- Variables: las variables son objetos de datos cuyo valor se modifica durante la ejecución de un programa.
- Expresiones: las expresiones combinan operandos (variables, constantes, funciones) y operadores (+, -, *, /).
- Instrucciones: las instrucciones son las operaciones que puede realizar la computadora.

Además existen otros elementos que forman parte de los programas, éstos son: bucles, contadores, acumuladores, interruptores y estructuras de control.

Un *bucle* o *lazo* es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un número determinado de veces o mientras se cumpla una determinada condición. Una condición puede ser VERDADERA o FALSA y se comprueba en cada iteración del bucle (total de instrucciones que se repiten en el bucle). Un bucle consta de tres partes: decisión, cuerpo del bucle, salida del bucle.

Los *contadores*, en general, se utilizan para controlar las iteraciones de un bucle. Un *contador* es una variable cuyo valor se incrementa o decrementa en una cantidad constante en cada iteración. El contador puede ser positivo o negativo.

Un *acumulador* es una variable cuya misión es almacenar cantidades variables resultantes de sumas o productos sucesivos. Realiza la misma función que un contador, con la diferencia que el incremento o decremento es variable en lugar de constante.

Un *interruptor* o *bandera* es, en general, una variable lógica que puede tomar valor VERDADERO o FALSO a lo largo de la ejecución de un programa y que permite comunicar información de una parte a otra del mismo.

Las *estructuras de control* secuenciales, selectivas y repetitivas ya fueron explicadas en detalle.

Comprobación de Algoritmos: Prueba de Escritorio

La *Prueba de Escritorio* es una herramienta útil para comprobar que un algoritmo realiza la tarea para la que fue diseñado. Esta prueba consiste en “ejecutar a mano” el algoritmo propuesto como solución del problema planteado. Para esto deben utilizarse datos representativos y anotarse los valores que toman las variables en cada paso. Es recomendable dar diferentes datos de entrada y considerar todos los posibles casos, aún los de excepción o no esperados, para asegurar que el programa codificado a partir del algoritmo no produzca errores en tiempo de ejecución al recibir tales entradas.

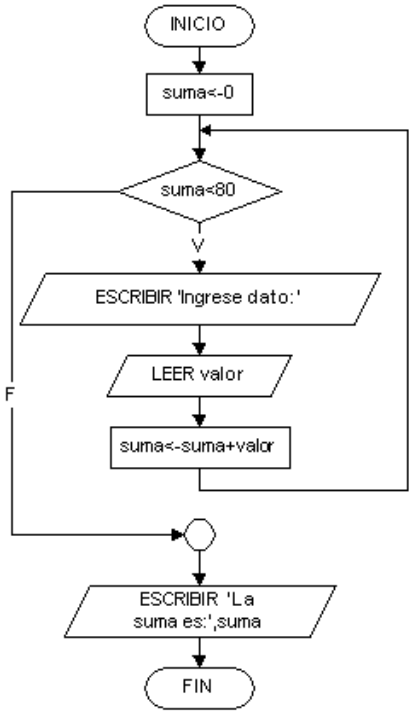
De este modo la prueba de escritorio permite comprobar, en tiempo de diseño, si el algoritmo es correcto o si es necesario realizar ajustes.

A continuación se muestra un ejemplo de prueba de escritorio del siguiente algoritmo:

Ejemplo: Diseñe un algoritmo que calcule la suma de valores ingresados por el usuario en tanto la suma sea menor a 80.

	Variables y Condiciones		
Pasos	suma	suma<80	valor
01	0	-	-
02		VERDADERO	-
03			30
04	30		
05		VERDADERO	
06			14
07	44		
08		VERDADERO	
09			25
10	69		
11		VERDADERO	
12			21
13	90		
14		FALSO	
15	LA SUMA ES: 90		

```
INICIO
    suma←0
    MIENTRAS suma < 80 HACER
        ESCRIBIR 'Ingrese dato:'
        LEER valor
        suma←suma+valor
    FIN_MIENTRAS
    ESCRIBIR 'La suma es:', suma
FIN
```



En la tabla anterior puede observarse cómo las *variables* (suma, valor) y la *condición* (suma<80) se van modificando al ejecutar, paso a paso, las instrucciones del algoritmo.

Nótese, además, que el número de repeticiones del bucle (que realiza la suma) depende de los datos ingresados por el usuario, finalizando éste en el momento en que la variable suma alcanza o supera el valor 80.

CODIFICACIÓN EN LENGUAJE C/C++

Equivalencias entre Pseudocódigo y Lenguaje C/C++.

ASIGNACIÓN	
`suma ← suma + 10`	`suma=suma + 10;`
LECTURA	
`leer valor`	`cin >> valor;`
ESCRITURA	
`escribir "hola mundo!!!"`	`cout << "hola mundo!!!";`
CONDICIONALES O SELECTIVAS SIMPLES	

`si condición entonces`
`  acciones`
`fin_si`
	`if (condición)`
	`  acciones;`

CONDICIONALES O SELECTIVAS DOBLES	

`si condición entonces`
`  acciones1`
`sino`
`  acciones2`
`fin_si`
	`if (condición)`
	`  acciones1;`
	`else`
	`  acciones2;`

CONDICIONALES O SELECTIVAS MÚLTIPLES	

`según opción hacer`
`  op1: acciones_1`
`  op2: acciones_2`
`  …`
`  opn: acciones_n`
`de otro modo`
`  acciones`
`fin_segun`
	`switch (opción)`
	`{`
	`  case op1: acciones_1; break;`
	`  case op2: acciones_2; break;`
	`  …`
	`  case opn: acciones_n; break;`
	`  default: acciones;`
	`}`

REPETIR	

`Repetir`
`  acciones`
`hasta_que condición`
	`do`
	`{`
	`  acciones;`
	`} while (condición);`

MIENTRAS	

`mientras condición hacer`	`while (condición)`
`acciones`	`{`
`fin_mientras`	`acciones; }`
PARA	
`para v desde vi hasta vf con paso n hacer`	`for (v=vi; v<=vf; v++)`
`acciones`	`{`
`fin_para`	`acciones; }`

Programas en C. Estructura General

Un programa codificado en lenguaje *C* se organiza, básicamente, en 3 secciones

- Declaraciones
  - Librerías del lenguaje: la directiva *#include* permite indicar al compilador qué librerías debe incluir en el programa objeto para generar el programa ejecutable correspondiente. Las librerías a utilizar se indican entre paréntesis angulares, por ejemplo, *<stdio.h>* (librería de las funciones estándar de entrada/salida). De esta manera, el programador puede utilizar las funciones internas del lenguaje.
  - Módulos del programador: se especifica el tipo y argumentos de los módulos escritos por el programador.
  - Variables globales: se especifica el tipo de dato y nombre de las variables globales del programa.
  - Constantes: se especifica el tipo, nombre y valor de las constantes del programa.
- Programa principal (función *main*): la función *main()* contiene declaraciones de variables e instrucciones necesarias para controlar la secuencia de operaciones que ejecuta el programa a fin de resolver el problema para el que fue pensado.
- Módulos del programador: se especifica el código correspondiente a cada uno de los módulos creados por el programador. Un módulo contiene la declaración de variables e instrucciones que resuelven un subproblema específico.

A continuación, se presenta un modelo de la estructura general para un programa codificado en lenguaje *C*.

```
/* Comentario inicial: nombre del programa del programador, fecha, etc */

/* Archivos de cabecera (prototipos de funciones de librería) */
#include <archivo_cabecera.h>
#include <archivo_cabecera.h>

// Prototipos de funciones y procedimientos escritos por el programador
tipo_dato modulo1 (argumentos);
tipo_dato modulo2 (argumentos);

/* Variables y constantes globales */
tipo_dato variable_global;

const tipo_dato constante_global=valor;
#define PI 3.14

/* Algoritmo principal */
tipo_dato main(argumentos)
```

```
{
   /* Variables locales del algoritmo principal */
   tipo_dato nombre_variable1, nombre_variable2;
   tipo_dato nombre_variable3, nombre_variable4;
   ...
   /* Instrucciones del algoritmo principal */
   ...
   modulo1(argumentos);
   ...
   modulo2(argumentos);
   ...
   return valor;
}

/* Código completo de las funciones escritas por el programador */
tipo_dato modulo1 (argumentos)
{
   /* Variables locales e instrucciones del módulo */
}
tipo_dato modulo2 (argumentos)
{
   /* Variables locales e instrucciones del módulo */
}
```

Tipos de datos básicos

En un programa en *C*, los datos que se utilicen pueden definirse según los tipos presentados en la siguiente tabla.

Nombre	Descripción	Tamaño	Rango
char	Caracter (código ASCII)	8 bits	Con signo: -128 … 127 Sin signo: 0 … 255
short int (short)	Número entero corto	16 bits	Con signo: -32768 … 32767 Sin signo: 0 … 65535
int	Número entero	32 bits	Con signo: -2147483648 … 2147483647 Sin signo: 0 … 4294967295
long int (long)	Número entero largo	64 bits	Con signo: -9223372036854775808, 9223372036854775807 Sin signo: 0 … 18446744073709551615
float	Número real	32 bits	3,4*10-38 … 3,4*10+38 (6 decimales)
double	Número real en doble precisión	64 bits	1,7*10-308 … 1,7*10+308 (15 decimales)
long double	Número real largo de doble precisión	80 bits	3,4*10-4932 … 1,1*10+4932
bool	Valor booleano	1 bit	true (VERDADERO) o false (FALSO)

La selección del tipo de dato adecuado para los elementos del programa se realiza teniendo en cuenta el rango de valores a representar y las operaciones que se deben aplicar.

Funciones Básicas de Entrada/Salida de datos.

Las funciones de entrada/salida estándar de *C* están definidas en la biblioteca *stdio*. Cuando estas funciones se utilizan en un programa fuente es preciso incluir el archivo *stdio.h* mediante la directiva de precompilación *#include <stdio.h>.* Entre las funciones que contiene esta librería se encuentran *printf*, *scanf* y *gets*.

La función *printf* es la salida genérica por consola utilizada por *C*, mientras que la función *scanf* es la entrada estándar asociada al teclado. Tanto la función *printf* como la función *scanf* permiten especificar el formato en el que se van a escribir o leer los datos, esto se conoce como entrada/salida formateada.

La función de entrada *gets* permite almacenar una cadena de caracteres ingresada por teclado. Es decir, lee datos de la entrada estándar y los almacena en la variable de tipo cadena que utiliza como argumento. La sintaxis de esta función es la siguiente:

```
gets(nombre_variable);
```

En *C++* además de las funciones *printf* y *scanf*, que siguen estando vigentes, se pueden utilizar las "funciones" *cin* y *cout*. Para utilizarlas es necesario incluir la librería *iostream* especificando la directiva *#include <iostream>, cin* y *cout* responden a la siguiente sintaxis:

```
cin >> nombre_variable;
```

```
cout << "cadena de caracteres" << nombre_variable << endl;
```

Utilizando *cin* y *cout* no es necesario especificar el tipo de dato que se imprimirá o leerá, asociándolo con un formato determinado (como ocurre con *printf* y *scanf*), sino que es el propio programa el que decide el tipo de dato en tiempo de ejecución. De este modo, *cin* y *cout* admiten tanto los tipos predefinidos como aquellos tipos de datos definidos por el usuario. Al ser C++ una ampliación del lenguaje C, es necesario agregar nuevas palabras reservadas. Estas palabras reservadas están en un espacio de nombres o *"namespace"*. Para usar las palabras reservadas *cout* y *cin*, que están el *namespace std* (standard), se debe incorporar la instrucción:

```
using namespace std;
```

al incorporar en la cabecera del programa esta directiva estamos indicando al compilador que use el espacio de nombres *std*, y que busque e interprete todos los elementos definidos en el archivo.

Documentación Interna. Comentarios

Los comentarios en un programa fuente *C* pueden especificarse para líneas individuales o párrafos completos. Un comentario de línea se indica con el símbolo // y un comentario de párrafo se especifica con los símbolos /* (inicio del comentario) */ (fin del comentario).

Operadores en C

La siguiente tabla presenta los operadores básicos utilizados en *C*.

Tipo	Operadores
Aritmético	Potencia: ***pow(x,y)*** (librería *math.h*); *x, y* valores numéricos Producto: ***** Cociente: **/** Módulo o resto: **%** Sumar: **+** Diferencia: **-**
Alfanuméricos	Operaciones con cadenas (librería *string.h*)

Tipo	Operadores
	strcat(s,t); concatena *t* al final de *s.* *strcmp(s,t);* compara *s* y *t*, retornando negativo, cero, o positivo para: *s<t*, *s==t*, *s > t.* *strcpy(s,t);* copia *t* en *s.* *strlen(s);* retorna la longitud de *s*. donde *s* y *t* son variables de tipo cadena.
Lógicos	Negación (NO, NOT): **!** Conjunción (Y, AND): **&&** Disyunción (O, OR): **||**
Relacionales	Igual: **==** Distinto: **!=** Mayor: **>** Mayor o igual: **>=** Menor: **<** Menor o igual: **<=**
Asignación	**=**

SOFTWARE DE PROGRAMACIÓN EN C++:

Dev-C++ es un entorno de desarrollo integrado para programar en lenguaje C/C++. Usa MinGW, una versión de GCC, como compilador. Dev-C++ puede además ser usado en combinación con Cygwin y cualquier otro compilador basado en GCC. El Entorno está desarrollado en el lenguaje Delphi de Borland. Fuentes y sitio de descarga https://www.bloodshed.net/index.html.

CodeBlocks es un entorno de desarrollo integrado libre y multiplataforma para el desarrollo de programas en lenguaje C++. Está basado en la plataforma de interfaces gráficas WxWidgets, por lo que puede usarse libremente en diversos sistemas operativos, y está bajo GNU. Es una herramienta para desarrollar programas en C++ que ofrece una interfaz sencilla a los usuarios. Fuentes y sitio de descarga http://www.codeblocks.org/downloads/binaries.

ZinjaI es un IDE (entorno de desarrollo integrado) libre y gratuito para programar en C/C++. Pensado originalmente para ser utilizado por estudiantes de programación durante el aprendizaje, presenta una interfaz incial muy sencilla, pero sin dejar de incluir funcionalidades avanzadas que permiten el desarrollo de proyectos tan complejos como el propio ZinjaI. Fuentes y sitio de descarga: http://zinjai.sourceforge.net/

Opciones online:

Onlinegdb se accede ingresando el siguiente link en su navegador de preferencia (el navegador no debe tener activado el traductor): https://www.onlinegdb.com/online_c++_compiler

OmegaUp Es un proyecto web enfocado a elevar el nivel de competitividad de desarrolladores de software en América Latina mediante la resolución de problemas de algoritmos, con un enfoque competitivo. Un gestor de concursos basado en cómputo en la Nube (Cloud-Computing) y la plataforma oficial para la Olimpiada Mexicana de Informática.

https://omegaup.com/

PROGRAMACIÓN MODULAR

Estructura General de Programa

Los programas, en general, presentan una estructura común en la que pueden distinguirse las siguientes partes:

- Cabecera del programa: permite indicar el nombre de programa, a través de una palabra reservada. Por ejemplo, PROGRAMA *Calcula_Promedio*
- Declaración de Constantes: permite especificar los valores constantes que se utilizarán en el programa. Por ejemplo, MAXIMO=40
- Declaración de Tipos: permite especificar los tipos de datos definidos por el usuario que serán utilizados en el programa.
- Declaración de Variables: permite especificar las variables que serán utilizadas en el programa, indicando para cada una el tipo requerido. Por ejemplo, contador: entero (la variable contador se declara de tipo entero)
- Declaración de Procedimientos y Funciones: permite especificar las subrutinas que contendrá el programa y que facilitarán la escritura del programa. Por ejemplo, puede escribirse un subprograma que realice el cálculo del factorial de un número ingresado por el usuario.
- Programa Principal: permite especificar las secuencias de pasos generales del algoritmo, e invocar a los subprogramas necesarios cuando se requiera.

El siguiente ejemplo ilustra la estructura general de un programa.

```
PROGRAMA Calculo_Area_Circular
CONSTANTES
        PI=3.141592654
VARIABLES
        radio: real
        area:real
INICIO
    ESCRIBIR 'Ingrese el radio del círculo:'
    LEER radio
    area←PI * radio^2
    ESCRIBIR 'El área del círculo es:', area
FIN
```

En la cabecera del programa se especificó el nombre de programa, en este caso, *Calculo_Area_Circular*. En la sección de constantes, se definió la constante *PI* igual a 3.141592654. En la sección de variables, se definieron las variables de tipo real *radio* y *area*. En el cuerpo del programa principal, delimitado por las palabras reservadas *INICIO* y *FIN*, se especificaron las sentencias que implementan el algoritmo. Nótese que no se utilizaron procedimientos o funciones porque es preciso definir formalmente estos conceptos antes de ejemplificar su aplicación.

Descomposición de Problemas

En general los problemas que se presentan en el mundo real son complejos y extensos, por lo que su resolución no es directa. Para encarar tales problemas la programación cuenta con herramientas de abstracción y descomposición de problemas. La abstracción permite representar los objetos relevantes de un problema, y la descomposición, basada en el paradigma "Divide y vencerás", permite dividir problemas complejos en subproblemas y éstos a su vez en subproblemas más pequeños, y así sucesivamente. Cada uno de los subproblemas finales será entonces más simple de resolver.

Los programas que dan solución a problemas complejos están diseñados de tal forma que sus componentes (partes independientes, llamadas módulos) realizan actividades o tareas específicas (que resuelven subproblemas). Cada uno de estos componentes se programa utilizando las estructuras de control (secuenciales, selectivas y repetitivas) de la programación estructurada. La integración de los módulos, a través de métodos ascendentes o descendentes, permite obtener un programa modular que resuelve el problema planteado.

Programación Modular

La programación modular es un método de diseño flexible y potente que mejora la productividad de un programa. En programación modular, un programa se divide en subprogramas (módulos) que realizan tareas específicas y que se codifican independientemente (figuras 4 y 5). Cada módulo se analiza, codifica y pone a punto por separado.

En cada programa existe un módulo especial, llamado principal, que controla la ejecución de las tareas del programa. El principal transfiere temporalmente el control a los módulos (y éstos a su vez a submódulos, si es necesario) para que realicen una determinada tarea y luego retornen el control al módulo que los invocó.

En general los módulos o subrutinas se clasifican en procedimientos y funciones.

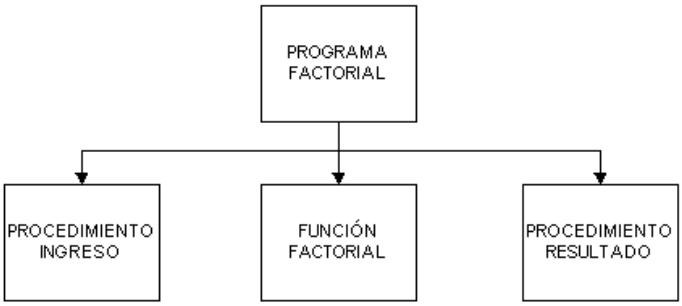


Figura 4. Estructura modular de un programa para el cálculo del factorial de un número.

```
function factorial (num:integer):integer;
var
      fact,i:integer;
begin
      fact:=1;
      for i:=1 to num do
            fact:=fact*i;
      factorial:=fact;
end;
```

Figura 5. Módulo (función) para el cálculo del factorial de un número.

Módulos: Funciones.

La resolución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños (subproblemas). La resolución de estos subproblemas se realiza mediante subalgoritmos. Los subalgoritmos pueden ser de 2 tipos: procedimientos y funciones. Los subalgoritmos o subprogramas se diseñan para realizar una tarea específica, y pueden ser llamados desde distintas partes del programa. Cuando un subprograma es invocado, quien realiza el llamado (el programa principal u otro subprograma) transfiere el control de las acciones temporalmente a esta subrutina para que ejecute las acciones que contiene. Finalizado el procedimiento o la función, devuelve un resultado y el control a quién realizó la invocación.

Funciones

Matemáticamente una función es una operación que toma uno o más valores llamados argumentos y produce un valor denominado resultado (valor de la función para los argumentos dados).

En programación, una función es un módulo o subprograma que toma una lista de valores llamados argumentos o parámetros y devuelve un único valor. Como las funciones retornan un valor, éstas deben definirse del tipo de dato apropiado (entero, real, carácter, lógico), por lo que deben incluirse en expresiones (el nombre de la función está asociado a un valor) o visualizarse.

Todos los lenguajes de programación tienen funciones incorporadas o intrínsecas (por ejemplo, seno, coseno) y funciones definidas por el usuario. Las funciones incorporadas al sistema se denominan funciones internas o intrínsecas y las funciones definidas o creadas por el usuario, funciones externas.

Las funciones son diseñadas para realizar tareas específicas: tomar una lista de valores llamados argumentos o parámetros, procesarlos y devolver un único valor.

Cuando se invoca una función debe utilizarse su nombre en una expresión, indicando los argumentos actuales o reales encerrados entre paréntesis.

Cada vez que se llama a una función desde el algoritmo principal se establece automáticamente una correspondencia entre los parámetros formales (cabecera de la declaración de función) y los parámetros actuales (valores que se envían a la función cuando se realiza la invocación). Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se supone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a una función implica los siguientes pasos:

1. A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.
2. Se ejecuta el cuerpo de acciones de la función.
3. Se asigna el resultado al nombre de la función y se retorna al punto de llamada.

Ámbito de la Variables

En el desarrollo de un sistema complejo con gran número de módulos, realizados por diferentes programadores, debe tenerse en cuenta que cada uno de ellos escribirá el código correspondiente a un módulo. Estos implica que cada cual define sus propios datos, con identificadores apropiados y como consecuencia de ello se podrían observar algunos inconvenientes tales como:

- Demasiados identificadores.
- Conflictos entre los nombres de los identificadores de cada programador.
- Integridad de los datos lo que implica que se puedan utilizan datos que tengan igual identificador pero que realicen funciones diferentes.
- Side Effects no previstos.

Si se considera que los módulos también pueden anidarse (módulos dentro de otros módulos), deben aplicarse 2 reglas muy importantes que gobiernan el alcance de los identificadores:

- El alcance de un identificador es el bloque de programa donde se lo declara.
- Si un identificador declarado en un bloque es nuevamente declarado en un bloque interno al primero, el segundo bloque es excluido del alcance de la primera sección.

Variables Globales y Locales

Las variables utilizadas en los programas principales y subprogramas se clasifican en 2 tipos:

- Variables Locales
- Variables Globales

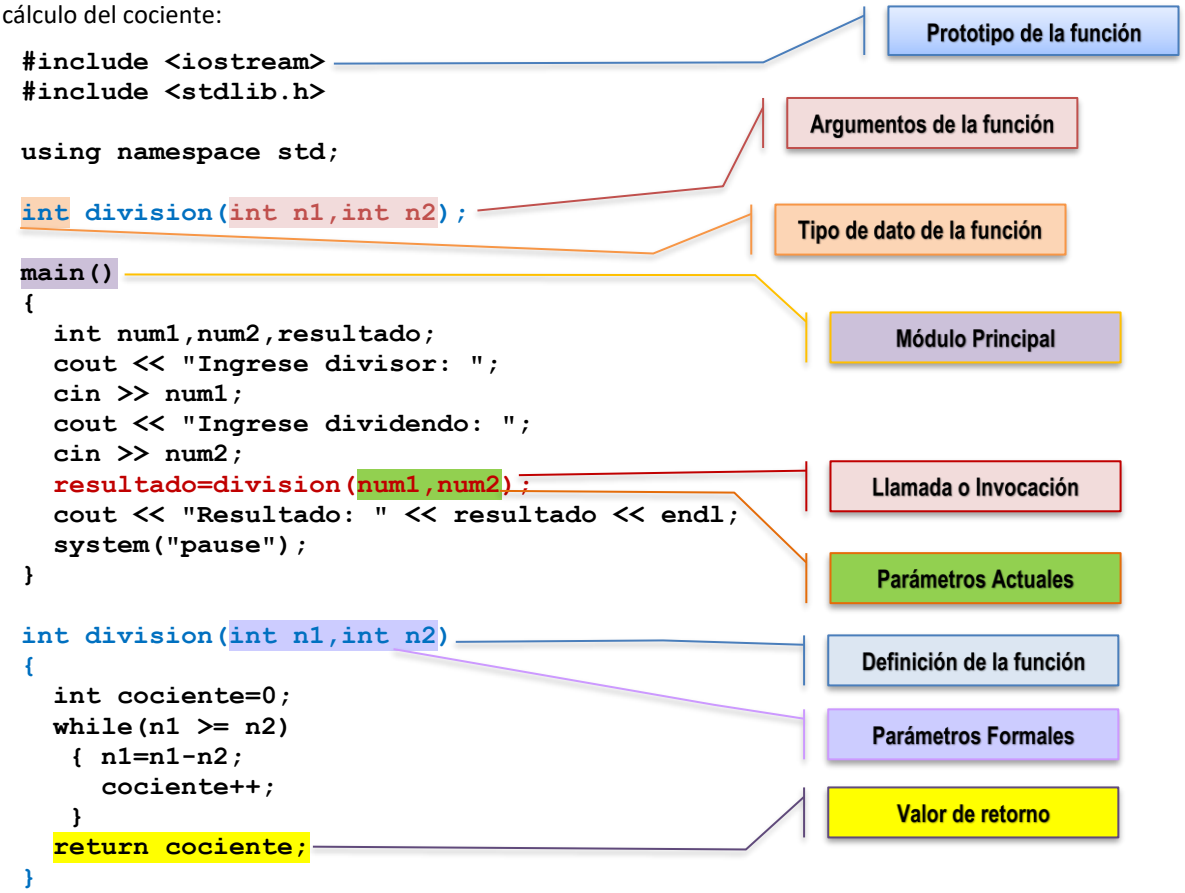
Una variable local es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese programa y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. El significado de una variable se confina al subprograma en el que está declarada. Una variable local no tiene ningún significado en otros subprogramas, es decir, el valor asignado a una variable local en un subprograma no es accesible por los otros procedimientos o funciones del programa.

Un variable global es aquella que está declarada en el programa o algoritmo principal, del que dependen todos los subprogramas. Las variables globales son conocidas por todos los procedimientos y funciones que conforman un programa, y pueden ser modificadas por éstos. Esta situación permite utilizar variables sin necesidad de usarlas como parámetros, sin embargo, las modificaciones no intencionales a una variable global pueden ocasionar errores en el programa.

Ocultamiento y Protección de Datos

El concepto *Data Hidding* se utiliza para significar que todo dato que es relevante para un módulo debe ocultarse a otros módulos. De esta manera se evita que en el programa principal se declaren datos que sólo son relevantes para algún módulo en particular y, además, se protege la integridad de los datos.

A continuación, se presenta el código de un programa modular que cuenta con un módulo tipo función para el cálculo del cociente:



Como puede observarse al escribir un programa modular se añaden algunos elementos a la estructura básica de programa:

- Prototipo del módulo (función o procedimiento): define qué módulos que serán utilizados en el programa especificando su tipo, nombre y parámetros.
- Llamada o invocación del módulo (función o procedimiento): indica qué módulo será utilizado en ese momento y cuáles serán los valores con los que operará.
- Definición del módulo (función o procedimiento): indica qué datos serán utilizados por el módulo (parámetros formales), el conjunto de operaciones a ejecutar, y el/los resultados a generar. En el caso de las funciones se incluye la instrucción *return* para devolver el resultado del cálculo.

También puede observarse que la estructura del programa principal (*main*) se simplifica al reducir a una sola línea el cálculo del cociente. La "complejidad" de este cálculo queda confinada al módulo que implementa la operación.