Contenido

El Software y sus características.

El software en sus comienzos era la parte insignificante del hardware, lo que venía como añadidura, casi como regalo. Al poco tiempo adquirió una entidad propia. En la actualizad, el software es la tecnología individual más importante en el mundo. Nadie en la década de 1950 podría haber predicho que el software se convertiría en una tecnología indispensable en los negocios, la ciencia, la ingeniería; tampoco podría preverse que una compañía de software podría volverse más grande e influyente que la mayoría de las compañías de la era industrial; que una red construida con software, llamada Internet cubriría y cambiaría todo, desde la investigación bibliográfica hasta las compras de los consumidores y los hábitos de las personas. Nadie podría haber imaginado que estaría relacionado con sistemas de todo tipo: transporte, medicina, militares, industriales, entretenimiento, automatización de hogares.

El software puede definirse como “el alma y cerebro de la computadora, la corporización de las funciones de un sistema, el conocimiento capturado acerca de un área de aplicación, la colección de los programas, y los datos necesarios para convertir a una computadora en una máquina de propósito especial diseñada para una aplicación particular, y toda la información producida durante el desarrollo de un producto de software”. El software viabiliza el producto más importante de nuestro tiempo: la información.

Características del software:

1. El software es intangible, es decir, que se trata de un concepto abstracto.

2. Tiene alto contenido intelectual.

3. Su proceso de desarrollo es humano intensivo, es decir que la materia prima principal radica en la mente de quienes lo crean.

4. El software no exhibe una separación real entre investigación y producción.

5. El software puede ser potencialmente modificado, infinitamente.

6. El software no se desgasta

7. La mayoría del software, en su mayoría, aún se construye a medida.

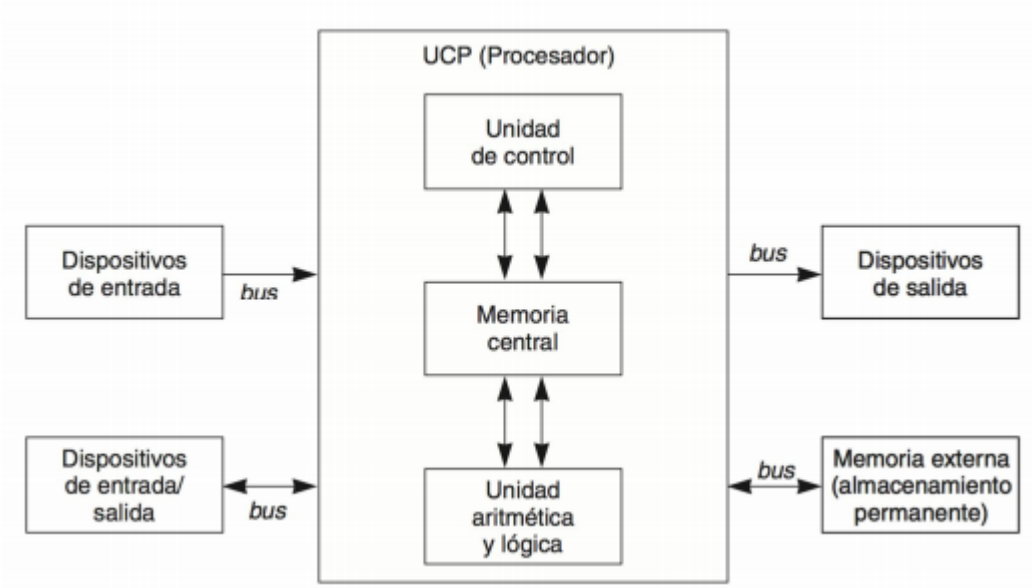
8. El software no se desarrolla en forma masiva, debido a que es único.

Estructura Interna de una Computadora

Una computadora moderna consta de uno o más procesadores, una memoria principal, discos, impresoras, un teclado, un ratón, una pantalla o monitor, interfaces de red y otros dispositivos de entrada/salida. En general es un sistema complejo. Si todos los programadores de aplicaciones tuvieran que comprender el funcionamiento de todas estas partes, no escribirían código alguno. Es más: el trabajo de administrar todos estos componentes y utilizarlos de manera óptima es una tarea muy desafiante. Por esta razón, las computadoras están equipadas con una capa de software llamada sistema operativo, cuyo trabajo es proporcionar a los programas de usuario un modelo de computadora mejor, más simple y pulcro, así como encargarse de la administración de todos los recursos antes mencionados.

La mayoría de las computadoras, grandes o pequeñas, están organizadas como se muestra en la siguiente figura. Constan fundamentalmente de tres componentes principales: Unidad Central de Proceso (UCP) o procesador, la memoria principal o central.

Si a los componentes anteriores se les añaden los dispositivos para comunicación con la computadora, aparece la estructura típica de un sistema de computadora: dispositivos de entrada, dispositivos de salida, memoria externa y el procesador/memoria central.



Los dispositivos de Entrada/Salida (E/S) (en inglés, Input/Output I/O) o periféricos permiten la comunicación entre la computadora y el usuario.

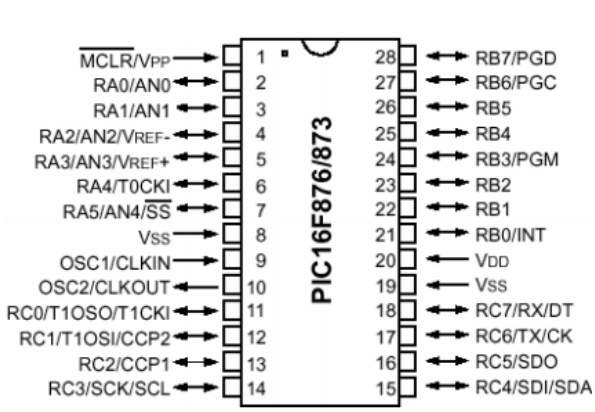
La memoria central de una computadora puede tener desde unos centenares de miles de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos de potencia de 2 para definir el tamaño de la memoria central: Kilobyte (KB o Kb) igual a 1.024 bytes (210) —prácticamente se consideran 1.000—; Megabyte (MB o Mb) igual a 1.024 × 1.024 bytes = = 1.048.576 (220) —prácticamente se consideran 1.000.000; Gigabyte (GB o Gb) igual a 1.024 MB (230), 1.073.741.824 = prácticamente se consideran 1.000 millones de MB.

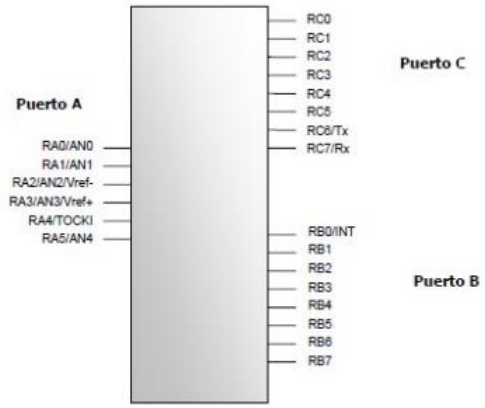
Byte	**Byte (B)**	*equivale a*	8 bits
Kilobyte	**Kbyte (KB)**	*equivale a*	1.024 bytes
Megabyte	**Mbyte (MB)**	*equivale a*	1.024 Kbytes
Gigabyte	**Gbyte (GB)**	*equivale a*	1.024 Mbytes
Terabyte	**Tbyte (TB)**	*equivale a*	1.024 Gbytes

1 **Tb** = 1.024 **Gb** = 1.024 × 1.024 **Mb** = 1.048.576 **Kb** = 1.073.741.824 **B**

Entradas/salidas digitales en el dsPIC16f873a

A partir de la configuración de pines vamos a ver los puertos de que dispone este microcontrolador.



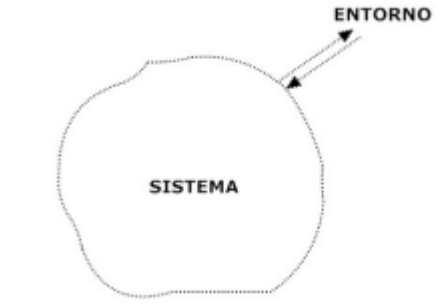


¿Qué es un Sistema?

Llamamos sistema a todo conjunto de elementos relacionados entre sí –puede ser por una finalidad en común-, que tienen un cierto orden u organización y que cumplen una función.

Los sistemas tienen composición (los elementos que lo forman), una estructura interna dada por el conjunto de relaciones entre sus componentes. Y también tienen un entorno o ambiente que es el conjunto de cosas que no pertenecen al sistema pero que actúan sobre él o sobre las que él actúa intercambiando materia, energía e información (MEI).

Los sistemas están inmersos en un entorno o ambiente, que es el conjunto de elementos que está fuera del sistema, es decir que no pertenecen al sistema pero que actúan sobre él o sobre las que el sistema actúa intercambiando materia, energía e información (MEI).



Diseño de Algoritmos

Un programador es antes que nada una persona que resuelve problemas; el programador procede a resolver un problema, a partir de la definición de un algoritmo y de la traducción de dicho algoritmo a un programa que ejecutará la computadora, pic, arduino, etc. En la oración anterior se nombran algunos conceptos que debemos profundizar:

Algoritmo: un algoritmo es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema. Así, como en la vida real, cuando tenemos que resolver un problema, o lograr un objetivo, por ejemplo: “Tengo que atarme los cordones”, para alcanzar la solución de ese problema, realizamos un conjunto de pasos, de manera ordenada y secuencial. Es decir, podríamos definir un algoritmo para atarnos los cordones de la siguiente forma:

1. Ponerme las zapatillas.

2. Agarrar los cordones con ambas manos.

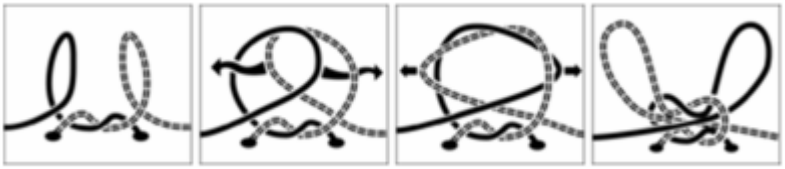
3. Hacer el primer nudo.

4. Hacer un bucle con cada uno de los cordones.

5. Cruzar los dos bucles y ajustar.

6. Corroborar que al caminar los cordones no se sueltan y la zapatilla se encuenta correctamente atada.

Algoritmo gráfico para atarse los cordones.



Programa: luego de haber definido el algoritmo necesario, se debe traducir dicho algoritmo en un conjunto de instrucciones, entendibles por la computadora, que le indican a la misma lo que debe hacer; este conjunto de instrucciones conforma lo que se denomina, un programa.

Para escribir un programa se utilizan lenguajes de programación, que son lenguajes que pueden ser entendidos y procesados por la computadora. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Algoritmos: Es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados, tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

• Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.

• Un algoritmo debe estar específicamente definido. Es decir, si se ejecuta un mismo algoritmo dos veces, con los mismos datos de entada, se debe obtener el mismo resultado cada vez.

• Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos. Debe tener un inicio y un final.

• Un algoritmo debe ser correcto: el resultado del algoritmo debe ser el resultado esperado.

• Un algoritmo es independiente tanto del lenguaje de programación en el que se expresa como de la computadora que lo ejecuta.

Como vimos anteriormente, el programador debe constantemente resolver problemas de manera algorítmica, lo que significa plantear el problema de forma tal que queden indicados los pasos necesarios para obtener los resultados pedidos, a partir de los datos conocidos. Lo anterior implica que un algoritmo básicamente consta de tres elementos: Datos de Entrada, Procesos y la Información de Salida.

Cuando dicho algoritmo se transforma en un programa de computadora:

• Las entradas se darán por medio de un dispositivo de entrada (como los vistos en el bloque anterior), como pueden ser el teclado, disco duro, teléfono, etc. Este proceso se lo conoce como entrada de datos, operación de lectura o acción de leer.

• Las salidas de datos se presentan en dispositivos periféricos de salida, que pueden ser pantalla, impresora, discos, etc. Este proceso se lo conoce como salida de datos, operación de escritura o acción de escribir.

Dado un problema, para plantear un algoritmo que permita resolverlo, es conveniente entender correctamente la situación problemática y su contexto, tratando de deducir del mismo los elementos ya indicados (entradas, procesos y salida). En este sentido entonces, para crear un algoritmo:

1. Comenzar identificando los resultados esperados, porque así quedan claros los objetivos a cumplir.

2. Luego, individualizar los datos con que se cuenta y determinar si con estos datos es suficiente para llegar a los resultados esperados. Es decir, definir los datos de entrada con los que se va a trabajar para lograr el resultado.

3. Finalmente si los datos son completos y los objetivos claros, se intentan plantear los procesos necesarios para pasar de los datos de entrada a los datos de salida.

Para comprender esto, veamos un ejemplo:

Problema:

Obtención del área de un rectángulo:

Altura: 5 cm

Base: 10 cm

1. Resultado esperado: área del rectángulo. Salida: área

Fórmula del área: base x altura.

2. Los datos con los que se dispone, es decir las entradas de datos son:

Dato de Entrada 1: altura: 5 cm

Dato de Entrada 2: base: 10 cm

3. El proceso para obtener el área del rectángulo es:

área = base * altura = área = 50

Herramientas para la representación gráfica de los algoritmos

Como se especificó anteriormente, un algoritmo es independiente del lenguaje de programación que se utilice. Es por esto, que existen distintas técnicas de representación de un algoritmo que permiten está diferenciación con el lenguaje de programación elegido. De esta forma el algoritmo puede ser representado en cualquier lenguaje. Existen diversas herramientas para representar gráficamente un algoritmo. En este material presentaremos dos:

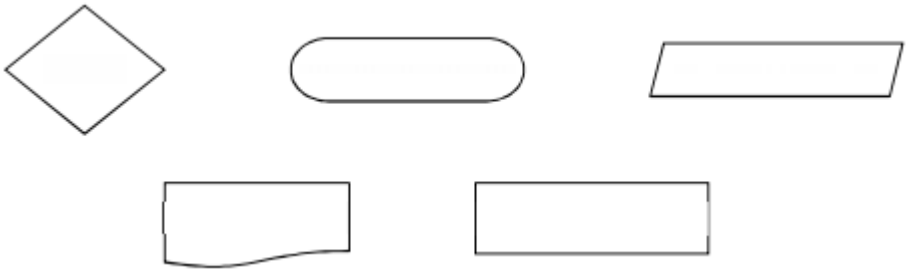
1. Diagrama de flujo.

2. Lenguaje de especificación de algoritmos: pseudocódigo.

Diagramas de Flujo

Un diagrama de flujo hace uso de símbolos estándar que unidos por flechas, indican la secuencia en que se deben ejecutar.

Estos símbolos son, por ejemplo:



Pseudocódigo

Conocido como lenguaje de especificación de algoritmos, el pseudocódigo tiene una estructura muy similar al lenguaje natural y sirve para poder expresar algoritmos y programas de forma independiente del lenguaje de programación. Además, es muy utilizado para comunicar y representar ideas que puedan ser entendidas por programadores que conozcan distintos lenguajes. El pseudocódigo luego se traduce a un lenguaje de programación específico ya que la computadora no puede ejecutar el pseudocódigo. Su uso tiene ventajas porque permite al programador una mejor concentración de la lógica y estructuras de control y no preocuparse de las reglas de un lenguaje de programación específico.

Un ejemplo básico de pseudocódigo, considerando el ejemplo utilizado anteriormente, es el siguiente:

```
INICIO FUNCION CALCULAR_AREA
    DEFINIR BASE: 5
    DEFINIR ALTURA: 10
    DEFINIR AREA: BASE * ALTURA
    DEVOLVER AREA
FIN
```

Lenguajes de Programación

Los lenguajes de programación son lenguajes que pueden ser entendidos y procesados por la computadora.

Tipos de Lenguajes de Programación

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

- **Lenguajes máquina**
- **Lenguaje de bajo nivel (ensamblador)**
- **Lenguajes de alto nivel**

La elección del lenguaje de programación a utilizar depende mucho del objetivo del software. Por ejemplo, para desarrollar aplicaciones que deben responder en tiempo real como por ejemplo, el control de la velocidad crucero en un sistema de navegación de un auto, debemos tener mayor control del hardware disponible, por lo que privilegiaremos lenguajes de más bajo nivel que nos permitan hacer un uso más eficiente de los recursos. En cambio, para aplicaciones de escritorio como sistemas de gestión de productos, calendarios, correo electrónico, entre otras privilegiaremos la elección de lenguajes de más alto nivel que nos permitan ser más eficientes en cuanto a la codificación ya que, en términos generales, es necesario escribir menos líneas de código en los lenguajes de alto nivel, que para su equivalente en bajo nivel.

El lenguaje máquina

Los lenguajes máquina son aquellos que están escritos en lenguajes cuyas instrucciones son cadenas binarias (cadenas o series de caracteres -dígitos- O y1) que especifican una operación, y las posiciones (dirección) de memoria implicadas en la operación se denominan instrucciones de máquina o código máquina. El código máquina es el conocido código binario.

En los primeros tiempos del desarrollo de los ordenadores era necesario programarlos directamente de esta forma, sin embargo, eran máquinas extraordinariamente limitadas, con muy pocas instrucciones por lo que aún era posible; en la actualidad esto es completamente irrealizable por lo que es necesario utilizar lenguajes más fácilmente comprensibles para los humanos que deben ser traducidos a código máquina para su ejecución.

Ejemplo de una instrucción:

1110 0010 0010 0001 0000 0000 0010 0000

El lenguaje de bajo nivel

Los lenguajes de bajo nivel son más fáciles de utilizar que los lenguajes máquina, pero, al igual, que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el ensamblador o assembler. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como mnemotécnicas (mnemonics). Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés, ADD, SUB, DIV, etc.; en español, SUM, para sumar, RES, para restar, DIV, para dividir etc.

Lenguajes de alto nivel

Los lenguajes de alto nivel son los más utilizados por los programadores. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y

ensambladores. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la máquina; esto es, las instrucciones del programa de la computadora no dependen del diseño del hardware o de una computadora en particular. En consecuencia, los programas escritos en lenguaje de alto nivel son portables o transportables, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras; al contrario que los programas en lenguaje máquina o ensamblador, que sólo se pueden ejecutar en un determinado tipo de computadora. Esto es posible porque los lenguajes de alto nivel son traducidos a lenguaje máquina por un tipo de programa especial denominado "compilador". Un compilador toma como entrada un algoritmo escrito en un lenguaje de alto nivel y lo convierte a instrucciones inteligibles por el ordenador; los compiladores deben estar adaptados a cada tipo de ordenador pues deben generar código máquina específico para el mismo.

Ejemplos de Lenguajes de Alto Nivel:

C, C++, Java, Python, VisualBasic, C#, JavaScript

¿Qué es un Programa?

Es un algoritmo escrito en algún lenguaje de programación de computadoras.

Pasos para la construcción de un programa

DEFINICIÓN DEL PROBLEMA

En este paso se determina la información inicial para la elaboración del programa. Es donde se determina qué es lo que debe resolverse con el computador, el cual requiere una definición clara y precisa.

Es importante que se conozca lo que se desea que realice la computadora; mientras la definición del problema no se conozca del todo, no tiene mucho caso continuar con la siguiente etapa.

ANÁLISIS DEL PROBLEMA

Una vez que se ha comprendido lo que se desea de la computadora, es necesario definir:

- Los datos de entrada.
- Los datos de salida
- Los métodos y fórmulas que se necesitan para procesar los datos.

Una recomendación muy práctica es la de colocarse en el lugar de la computadora y analizar qué es lo que se necesita que se ordene y en qué secuencia para producir los resultados esperados.

DISEÑO DEL ALGORITMO

Se puede utilizar algunas de las herramientas de representación de algoritmos mencionadas anteriormente. Este proceso consiste en definir la secuencia de pasos que se deben llevar a cabo para conseguir la salida identificada en el paso anterior.

CODIFICACIÓN

La codificación es la operación de escribir la solución del problema (de acuerdo a la lógica del diagrama de flujo o pseudocódigo), en una serie de instrucciones detalladas, en un código reconocible por la computadora. La serie de instrucciones detalladas se conoce como código fuente, el cual se escribe en un lenguaje de programación o lenguaje de alto nivel.

PRUEBA Y DEPURACIÓN

Se denomina prueba de escritorio a la comprobación que se hace de un algoritmo para saber si está bien realizado. Esta prueba consiste en tomar datos específicos como entrada y seguir la secuencia indicada en el algoritmo hasta obtener un resultado, el análisis de estos resultados indicará si el algoritmo está correcto o si por el contrario hay necesidad de corregirlo o hacerle ajustes.

Elementos de un Programa

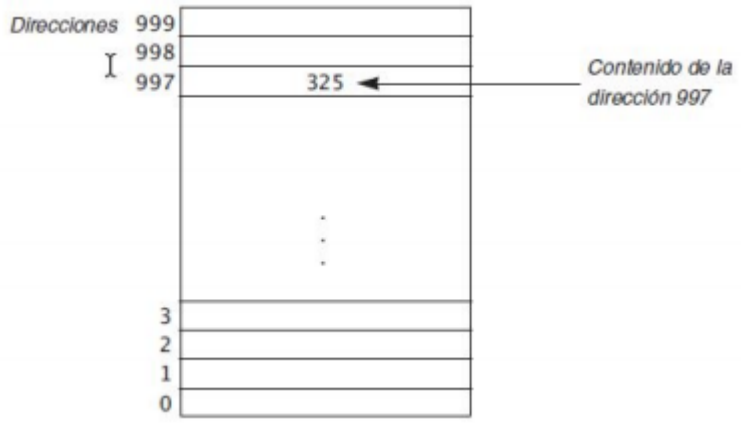
Variables y Constantes

A la hora de elaborar un programa es necesario usar datos; en el caso del ejemplo del cálculo del área del rectángulo, para poder obtener el área del mismo, necesitamos almacenar en la memoria de la computadora el valor de la base y de la altura, para luego poder multiplicar sus valores.

Recordemos que no es lo mismo grabar los datos en memoria que grabarlos en el disco duro. Cuando decimos grabar en memoria nos estaremos refiriendo a grabar esos datos en la RAM. Ahora bien, para grabar esos datos en la RAM podemos hacerlo utilizando dos elementos, llamados: variables y constantes. Los dos elementos funcionan como fuentes de almacenamiento de datos, la gran diferencia entre los dos es que en el caso de las constantes su valor dado no varía en el transcurso de todo el programa.

Podría decirse que tanto las variables como las constantes, son direcciones de memoria con un valor, ya sea un número, una letra, o valor nulo (cuando no tiene valor alguno, se denomina valor nulo). Estos elementos permiten almacenar temporalmente datos en la computadora para luego poder realizar cálculos y operaciones con los mismos. Al almacenarlos en memoria, podemos nombrarlos en cualquier parte de nuestro programa y obtener el valor del dato almacenado, se dice que la variable nos devuelve el valor almacenado.

A continuación, se muestra un esquema, que representa la memoria RAM, como un conjunto de filas, donde, en este caso, cada fila representa un byte y tiene una dirección asociada.



Variables

Son elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato, que puede variar en el desarrollo del programa. Una variable es un grupo de bytes asociado a un nombre o identificador, y a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a esa variable.

En una variable se puede almacenar distintos tipos de datos. De acuerdo al tipo de dato, definido para cada lenguaje de programación, será la cantidad de bytes que ocupa dicha variable en la memoria.

BYTE	0-255	Variable sin signo de 8 bits
INT	+/- 32768	Variable de 16 bits con signo
UINT	0-65535	Variable de 16 bits sin signo
LONG	+/- 21474836648	Variable de 32 bits con signo
ULONG	0-4294967295	Variable de 32 bits sin signo
Cadena	0-255	Matriz sin signo de 8 bits, tamaño predeterminado es 20.
Coma flotante	-Inf a +Inf	Variable con signo de 32 bits

Lo más importante de la definición de las variables y la elección del tipo de datos asociados es el significado de la variable, o su semántica: ya que en base al tipo de datos seleccionado serán las operaciones que podamos realizar con esa variable, por ejemplo: si tenemos la variable edad deberíamos seleccionar un tipo de datos como integer (número entero) ya que las operaciones relacionadas serán de comparación, sumas o restas y no es necesario tener una profundidad de decimales.

Constantes

Elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato pero que no varía durante la ejecución del programa. Se podría pensar en un ejemplo de necesitar utilizar en el programa el número pi, como el mismo no varía, se puede definir una constante pi y asignarle el valor 3.14.

Operadores

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Los operadores son símbolos especiales que sirven para ejecutar una determinada operación, devolviendo el resultado de la misma.

Para comprender lo que es un operador, debemos primero introducir el concepto de Expresión. Una expresión es, normalmente, una ecuación matemática, tal como 3 + 5. En esta expresión, el símbolo más (+) es el operador de suma, y los números 3 y 5 se llaman operandos. En síntesis, una expresión es una secuencia de operaciones y operandos que especifica un cálculo.

Existen diferentes tipos de operadores:

Operador de asignación

Es el operador más simple que existe, se utiliza para asignar un valor a una variable o a una constante.

El signo que representa la asignación es el = y este operador indica que el valor a la derecha del = será asignado a lo que está a la izquierda del mismo.

Ejemplo en pseudocódigo:

Entero edad = 20

Decimal precio = 25.45

Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (+), resta (-), multiplicación (*), división (/) y resto de la división entera (%), por ejemplo 50%8=2 porque necesitamos obtener un número entero que queda luego de determinar la cantidad de veces que 8 entra en 50.

Ejemplo:

Expresión	Operador	Operandos	Resultado arrojado
5 * 7	*	5 , 7	35
6 + 3	+	6 , 3	9
20 - 4	-	20 , 4	16
50 % 8	%	50 , 8	2
45/5	/	45,5	9

Tabla Resumen Operadores Aritméticos:

Operador	Significado
+	Suma
-	Resta
*	Producto
/	División
%	Resto de la división entera

Los operadores unitarios

Los operadores unitarios requieren sólo un operando; que llevan a cabo diversas operaciones, tales como incrementar/decrementar un valor de a uno, negar una expresión, o invertir el valor de un booleano.

Operador	Descripción	Ejemplo	Resultado
++	operador de incremento; incrementa un valor de a 1	int suma=20; suma++;	suma=21
--	operador de decremento; Reduce un valor de a 1	int resta=20; resta--;	resta=19
!	operador de complemento lógico; invierte el valor de un valor booleano	boolean a=true; boolean b= !a;	b=false

Operadores Condicionales

Son aquellos operadores que sirven para comparar valores. Siempre devuelven valores booleanos:

TRUE O FALSE. Pueden ser Relacionales o Lógicos.

Operadores relacionales

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor.

Los operadores relacionales determinan si un operando es mayor que, menor que, igual a, o no igual a otro operando. La mayoría de estos operadores probablemente le resultará familiar. Tenga en cuenta que debe utilizar " == ", no " = ", al probar si dos valores primitivos son iguales.

Operador	Significado
==	Igual a
!=	No igual a
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que

Expresión	Operador	Resultado
a > b	>	true: si a es mayor que b false: si a es menor que b
a >= b	>=	true:si a es mayor o igual que b false: si a es menor que b
a < b	<	true: si a es menor que b false: si a es mayor que b
a <= b	<=	true: si a es menor o igual que b false: si a es mayor que b.
a == b	==	true: si a y b son iguales. false: si a y b son diferentes.
a != b	!=	true: si a y b son diferentes false: si a y b son iguales.

Operadores lógicos

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. Se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales.

Expresión	Nombre Operador	Operador	Resultado
a && b	AND	&&	true: si a y b son verdaderos. false: si a es falso, o si b es falso, o si a y b son falsos
a || b	OR	||	true: si a es verdadero, o si b es verdadero, o si a y b son verdaderos. false: si a y b son falsos.

Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser true y el primero es false ya se sabe que la condición de que ambos sean true no se va a cumplir).

Rutinas

Las rutinas son uno de los recursos más valiosos cuando se trabaja en programación ya que permiten que los programas sean más simples, debido a que el programa principal se compone de diferentes rutinas donde cada una de ellas realiza una tarea determinada.

Una rutina se define como un bloque, formado por un conjunto de instrucciones que realizan una tarea específica y a la cual se la puede llamar desde cualquier parte del programa principal. Además, una rutina puede opcionalmente tener un valor de retorno y parámetros. El valor de retorno puede entenderse como el resultado de las instrucciones llevadas a cabo por la rutina, por ejemplo si para una rutina llamada sumar(a, b) podríamos esperar que su valor de retorno sea la suma de los números a y b. En el caso anterior, a y b son los datos de entrada de la rutina necesarios para realizar los cálculos correspondientes. A estos datos de entrada los denominamos parámetros y a las rutinas que reciben parámetros las denominamos funciones, procedimientos o métodos, dependiendo del lenguaje de programación.

Ejemplos de rutinas:

SumarPrecioProductos(precioProducto1, precioProducto2)

Rutina que realiza la suma de los precios de los productos comprados por un cliente y devuelve el monto total conseguido.

AplicarDescuento(montoTotal)

Rutina que a partir de un monto total aplica un descuento de 10% y devuelve el monto total con el descuento aplicado.

Desarrollo de programas

Estructuras de programación

Un programa puede ser escrito utilizando tres tipos de estructuras de control:

a) secuenciales

b) selectivas o de decisión

c) repetitivas

Las Estructuras de Control determinan el orden en que deben ejecutarse las instrucciones de un algoritmo: si serán recorridas una luego de la otra, si habrá que tomar decisiones sobre si ejecutar o no alguna acción o si habrá repeticiones.

Estructura secuencial

Es la estructura en donde una acción (instrucción) sigue a otra de manera secuencial. Las tareas se dan de tal forma que la salida de una es la entrada de la que sigue y así en lo sucesivo hasta cumplir con todo el proceso. Esta estructura de control es la más simple, permite que las instrucciones que la constituyen se ejecuten una tras otra en el orden en que se listan. Por ejemplo, considérese el siguiente fragmento de un algoritmo:

En este fragmento se indica que se ejecute la operación 1 y a continuación la operación 2.

Estructura alternativa

Estas estructuras de control son de gran utilidad para cuando el algoritmo a desarrollar requiera una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existe un número de posibles alternativas que resultan de la evaluación de una determinada condición.

Este tipo de estructuras son utilizadas para tomar decisiones lógicas, es por esto que también se denominan estructuras de decisión o selectivas.

En estas estructuras, se realiza una evaluación de una condición y de acuerdo al resultado, el algoritmo realiza una determinada acción. Las condiciones son especificadas utilizando expresiones lógicas.

Las estructuras selectivas/alternativas pueden ser:

- Simples
- Dobles
- Múltiples

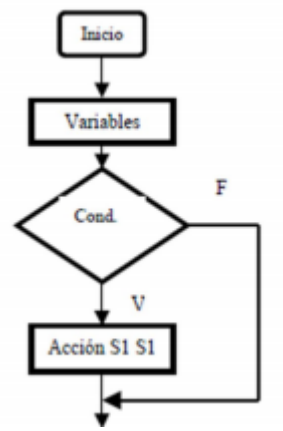
Alternativa simple (si-entonces/if-then)

La estructura alternativa simple si-entonces (en inglés if-then) lleva a cabo una acción al cumplirse una determinada condición. La selección si-entonces evalúa la condición y:

- Si la condición es verdadera, ejecuta la acción

S1

- Si la condición es falsa, no ejecuta nada.



En español:	En Inglés:
Si <condición> Entonces <acción S1> Fin_si	If <condición> Then <acción S1> End_if

Ejemplo:

INICIO

ENTERO edad = 18

SI (edad > 18)

ENTONCES:

puede manejar un auto

FIN_SI

FIN

Alternativa Doble (si-entonces-sino/if-then-else)

Existen limitaciones en la estructura anterior, y se necesitará normalmente una estructura que permita elegir dos opciones o alternativas posibles, de acuerdo al cumplimiento o no de una determinada condición:

- Si la condición es verdadera, se ejecuta la acción S1
- Si la condición es falsa, se ejecuta la acción S2

En español:

Si <condición>

entonces <acción S1>

sino <acción S2>

Fin_Si

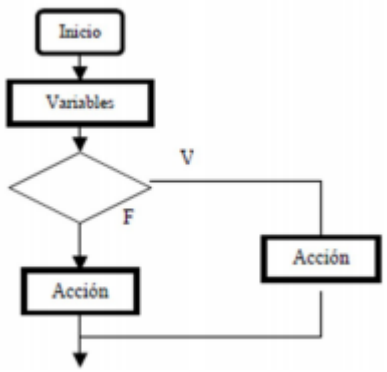
En inglés:

If <condición>

then<acción>

else<acción S2>

End_if



Ejemplo:

INICIO

BOOLEANO afueraLlueve = verdadero

SI (afueraLlueve es verdadero)

ENTONCES:

me quedo viendo películas

SINO:

salgo al parque a tomar mates

FIN_SI

FIN

Alternativa de Decisión múltiple (según_sea, caso de/case)

Se utiliza cuando existen más de dos alternativas para elegir. Esto podría solucionarse por medio de estructuras alternativas simples o dobles, anidadas o en cascada. Sin embargo, se pueden plantear serios problemas de escritura del algoritmo, de comprensión y de legibilidad, si el número de alternativas es grande.

En esta estructura, se evalúa una condición o expresión que puede tomar n valores. Según el valor que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor.

PSEUDOCÓDIGO:

Según sea <expresión>

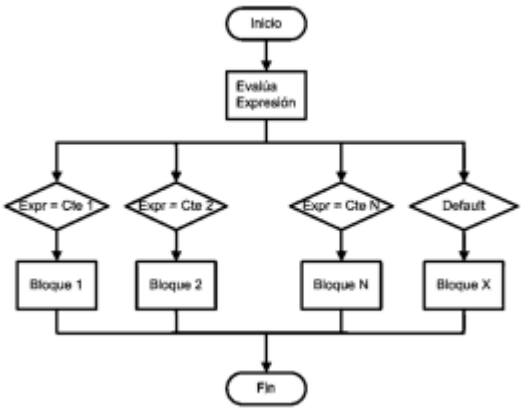
<Valor1>: <acción1>

<valor2>: <acción2>

.....

[<otro>: <acciones>]

fin según



Ejemplo en pseudocódigo:

INICIO

ENTERO posicion DeLlegada = 3

SEGUN SEA posicionDeLlegada

1: entregar medalla de oro

2: entregar medalla de plata

3: entregar medalla de bronce

otro: entregar mención especial

FIN

```
INICIO
ENTERO posicionDeLlegada = 3
SI (posicionDeLlegada = 1)
  ENTONCES:
    entregar medalla de oro
  SINO:
    SI (posicionDeLlegada = 2)
    ENTONCES:
      entregar medalla de plata
    SINO:
      SI (posicionDeLlegada = 3)
      ENTONCES:
        entregar medalla de bronce
      SINO:
        entregar mención especial
      FIN_SI
    FIN_SI
  FIN_SI
FIN
```

Podemos ver que usar condiciones anidadas podemos resolver el mismo problema, pero la estructura resultante es mucho más compleja y difícil de modificar.

Estructura repetitiva o iterativa

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan BUCLES. Y cada repetición del bucle se llama iteración.

Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

Un bucle se denomina también lazo o loop. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalización del bucle no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores principiantes.

Hay distintos tipos de bucles:

- **Mientras, en inglés: While**
- **Hacer Mientras, en inglés: Do While.**
- **Para, en inglés: For**

Estructura mientras (while, en inglés)

Esta estructura repetitiva “mientras”, es en la que el cuerpo del bucle se repite siempre que se cumpla una determinada condición.

Diagrama de Flujo

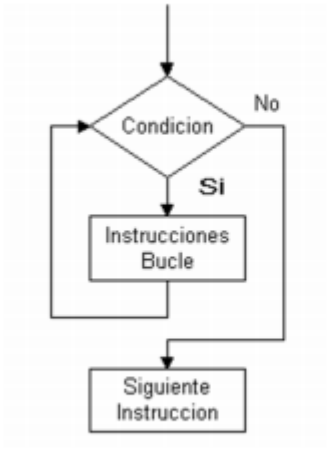


Figura 19: Diagrama de Flujo de la estructura "mientras"

Pseudocódigo:

```
Mientras <cond> hacer
  <acciones>
fin mientras
```

Ejemplo:

INICIO

BOOLEANO tanqueLleno = falso

MIENTRAS (tanqueLleno == falso)

HACER:

llenar tanque

FIN_MIENTRAS

// el tanque ya está lleno :)

FIN

Estructura hacer-mientras (do while, en inglés)

Esta estructura es muy similar a la anterior, sólo que a diferencia del while el contenido del bucle se ejecuta siempre al menos una vez, ya que la evaluación de la condición se encuentra al final. De esta forma garantizamos que las acciones dentro de este bucle sean llevadas a cabo, aunque sea una vez independientemente del valor de la condición.

Diagrama de Flujo

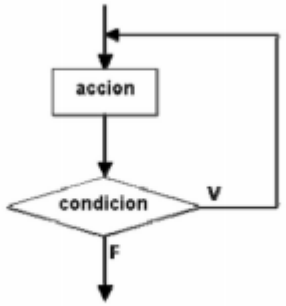


Figura 20: Diagrama de flujo de "hacer-mientras"

Pseudocódigo:

```
Hacer
  <acciones>
Mientras <cond>
Fin mientras
```

Ejemplo:

INICIO

BOOLEANO llegadaColectivo=false;

HACER: esperar en la parada

MIENTRAS (llegadaColectivo == false)

FIN_MIENTRAS

FIN

Estructura para (for, en inglés)

La estructura for es un poco más compleja que las anteriores y nos permite ejecutar un conjunto de acciones para cada elemento de una lista, o para cada paso de un conjunto de elementos. Su implementación depende del lenguaje de programación, pero en términos generales podemos identificar tres componentes: la inicialización, la condición de corte y el incremento.

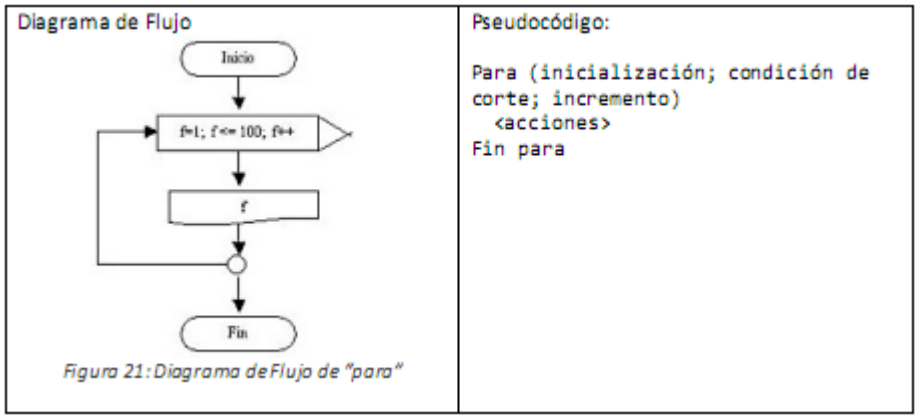


Figura 21: Diagrama de Flujo de "para"

Ejemplo:

INICIO

PARA (ENTERO RUEDA = 1; RUEDA <= 4; RUEDA++)

inflar_rueda (RUEDA)

FIN_PARA

FIN

La ejecución del pseudocódigo anterior dará como resultado las siguientes llamadas a la función inflar():

1. inflar_rueda (1)
2. inflar_rueda (2)
3. inflar_rueda (3)
4. inflar_rueda (4)

Luego de esto podríamos suponer que hemos inflado las 4 cubiertas del auto y estamos listos para seguir viaje.

Recursividad

La recursividad es un elemento muy importante en la solución de algunos problemas de computación.

Por definición, un algoritmo recursivo es aquel que utiliza una parte de sí mismo como solución al problema. La otra parte generalmente es la solución trivial, es decir, aquella cuya solución será siempre conocida, es muy fácil de calcular, o es parte de la definición del problema a resolver. Dicha solución sirve como referencia y además permite que el algoritmo tenga una cantidad finita de pasos.

La implementación de estos algoritmos se realiza generalmente en conjunto con una estructura de datos, la pila, en la cual se van almacenando los resultados parciales de cada recursión.

Un ejemplo es el cálculo de factorial de un número, se puede definir el factorial de un número entero positivo x como sigue:

```
x! = x * (x-1) * (x-2)... * 3 * 2 * 1
```

donde "!" indica la operación unaria de factorial Por ejemplo para calcular el factorial de 5tenemos:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Definimos, además:

```
1! = 1 y 0! = 1
```

Sin embargo, podemos observar que la definición del factorial de un número x, puede expresarse, a su vez, a través del factorial de otro número:

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

Es decir, para conocer el factorial de x basta con conocer el factorial de x-1 y multiplicarlo por x. Para conocer el factorial de x-1 basta con conocer el factorial de x-2, y multiplicarlo por x-1. Este proceso se realiza recursivamente, hasta llegar a la solución trivial, donde necesitamos el factorial de 1, el cual es

1.

Lo importante a notar en la igualdad anterior es que expresa un proceso recursivo, donde definimos una operación en términos de sí misma.

El pseudocódigo queda así:

```
Factorial (x)
  SI (x == 1 O x == 0)
     ENTONCES:
              DEVOLVER 1
```

SINO:

DEVOLVER x * Factorial (x-1)

FIN_SI

Ventajas

● Algunos problemas son esencialmente recursivos, por lo cual su implementación se facilita mediante un algoritmo de naturaleza recursiva, sin tener que cambiarlo a un método iterativo, por ejemplo.

● En algunas ocasiones el código de un algoritmo recursivo es muy pequeño.

Desventajas

● Puede llegar a utilizar grandes cantidades de memoria en un instante, pues implementa una pila cuyo tamaño crece linealmente con el número de recursiones necesarias en el algoritmo. Si los datos en cada paso son muy grandes, podemos requerir grandes cantidades de memoria lo que a veces puede agotar la memoria de la computadora donde está corriendo el programa.

Estructuras de Datos: Pilas, Colas y Listas

En esta sección veremos estructuras de datos que nos permiten coleccionar elementos. Para poder agregar u obtener elementos de estas estructuras de datos tenemos dos operaciones básicas:

- COLOCAR
- OBTENER.

Dependiendo de cómo se realicen las operaciones sobre los elementos de la colección es que definimos pilas, colas y listas.

Listas

Una lista (en inglés array) es una secuencia de datos. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3, etc. En una lista los datos deben ser del mismo tipo. Es importante destacar que la mayoría de las estructuras de datos en los lenguajes de programación son zero-based, es decir que el primer elemento siempre tendrá asignado el número de orden 0, lo que significa que la cantidad de elementos total será igual al número del último elemento más 1. El tipo de elementos almacenados en la lista puede ser cualquier tipo de dato. Normalmente la lista se utiliza para almacenar tipos de datos, tales como cadenas de texto, números enteros o decimales.

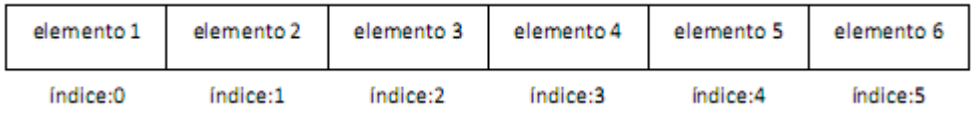
Una lista puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de asientos que tiene un colectivo de larga distancia. Cada ítem de una lista se denomina elemento. Una lista tiene definido una longitud, que indica la cantidad de elementos que contiene la misma. Por ejemplo, si tenemos una lista que contiene los meses del año, entonces dicha lista tiene en total 12 elementos, donde el primer elemento “Enero” tiene el número de orden 0 y “Diciembre” el número 11.

Los elementos de una lista se enumeran consecutivamente 0, 1, 2, 3, 4, 5, etc. Estos números se denominan valores índices o subíndice de la lista.

Los índices o subíndices de una lista son números que sirven para identificar unívocamente la posición de cada elemento dentro de la lista. Entonces, si uno quiere acceder a un elemento determinado de la lista, conociendo su posición, es decir su índice, se puede obtener el elemento deseado fácilmente.

Podemos representar una lista de la siguiente forma:

Si consideramos una lista de longitud 6:

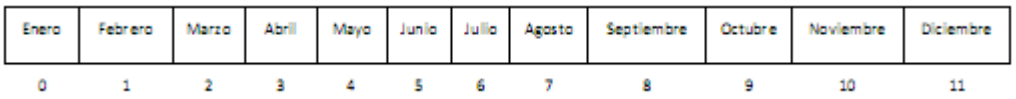


Ejemplo, volviendo al ejemplo planteado en el párrafo superior, se muestra a continuación una lista que contiene todos los meses del año:

Nombre de la Lista: mesesDelAño

Longitud de la Lista: 12

Tipo de Datos: String



Si quisiéramos acceder a un elemento determinado de la lista, simplemente debemos conocer cuál es la posición del elemento deseado.

Ejemplo:

INICIO

LISTA mesesDelAño

OBTENER(mesesDelAño, 11) // esto nos devuelve “diciembre”

OBTENER(mesesDelAño, 0) // esto nos devuelve “enero”

OBTENER(mesesDelAño, 7) // esto nos devuelve “agosto”

OBTENER(mesesDelAño, 12) // error: no existe el elemento 12

FIN

Si quisiéramos asignar un valor a una posición determinada de la lista, necesitamos conocer por un lado la posición que queremos asignar, y el elemento que vamos a asignar a dicha posición:

INICIO

LISTA mesesDelAño

COLOCAR(mesesDelAño, 10, “noviembre”) // “noviembre”, en la posición 10

COLOCAR(mesesDelAño, 0, “enero”) // esto asigna “enero”, en la posición 0

COLOCAR(mesesDelAño, 3, “abril”) // esto asigna “abril”, en la posición 3

FIN