

Anexo 8. Estructuras de datos y metodología estructurada básica

A. J. Pérez

[Estructuras de datos y metodología básica](#)

[Algoritmos y lenguajes](#)

[Lenguaje algorítmico y lenguaje de programación](#)

[Diagramas de Flujo](#)

[Organigramas](#)

[Ordinogramas](#)

[Pseudocódigo](#)

[Estructura general de un programa](#)

[Instrucciones](#)

[Instrucciones de declaración](#)

[Instrucciones de entrada](#)

[Instrucciones de salida](#)

[Instrucciones de asignación](#)

[Instrucciones de control](#)

[Instrucciones alternativas](#)

[Instrucciones repetitivas](#)

[Variables auxiliares](#)

[Contadores](#)

[Acumuladores](#)

[Interruptores, conmutadores, switches y banderas](#)

[Identificadores, constantes y variables](#)

[Expresiones y operadores](#)

[Operadores aritméticos](#)

[Operadores relacionales](#)

[Operadores lógicos](#)

[Asignaciones](#)

[Técnicas de programación](#)

[Programación estructurada](#)

[Teorema de la estructura](#)

[Herramientas de la programación estructurada](#)

[Programación modular y diseño descendente](#)

[Programa principal y subprogramas](#)

[Variables globales vs. locales](#)

[Paso de parámetros o argumentos](#)

[Recursividad](#)

[Funciones, procedimientos y subrutinas](#)

[Funciones](#)

[Llamada de una función](#)

[Ámbito de las variables](#)

[Procedimientos](#)

[Subrutinas](#)

[Estructuras de datos internas estáticas: arrays](#)

[Características](#)

[Operaciones con arrays](#)

[Arrays unidimensionales \(vectores\)](#)

[Arrays bidimensionales \(tablas\)](#)

[Arrays multidimensionales](#)

[Tratamiento secuencial de un array](#)

[Registros](#)

[Registros anidados](#)

[Registros con arrays](#)

[Ejercicios](#)

[Fuentes y bibliografía](#)

Estructuras de datos y metodología básica

Algoritmos y lenguajes

Un algoritmo es el conjunto de pasos, procedimientos o acciones ordenadas que permiten alcanzar un resultado o resolver un problema. La palabra “algoritmo” deriva del nombre latinizado del matemático árabe Mohamed Ibn Moussa Al Kow Rizimi, el cual escribió entre los años 800 y 825 su obra *Quitab Al Jabr Al Mugabala*, donde se recogía el sistema de numeración hindú y el concepto del cero. Fue Fibonacci, el que tradujo su obra al latín y la inicio con la obra *Algoritmi dicit*.

Lenguaje algorítmico y lenguaje de programación

Un programa, concepto desarrollado por Von Neumann en 1946, es un conjunto de instrucciones que ejecuta una computadora para alcanzar un resultado específico.

El lenguaje algorítmico permite realizar un análisis previo del problema y encontrar un método que permita resolverlo. El conjunto ordenado de las operaciones a realizar, se le denomina algoritmo.

Un lenguaje de programación está formado por un conjunto de reglas sintácticas y semánticas que hacen posible escribir un programa.

El lenguaje de programación es el que permite expresar un algoritmo para que sea interpretado por un ordenador. Según la cercanía del lenguaje a la máquina, se denomina lenguaje de alto nivel al que es más cercano a la comprensión humana y lenguaje de bajo nivel a aquellos que son utilizados directamente por la máquina.

Diagramas de Flujo

Los diagramas de flujo permiten representar el recorrido de la información desde su entrada como dato hasta su salida como resultado. Son el lenguaje para expresar algoritmos y programas.

Dependiendo de la fase de diseño en la que los utilizemos, hablaremos de:

- ORGANIGRAMAS: Diagramas de flujos del sistema. Representación gráfica de la circulación de datos e informaciones dentro de un sistema de información. Se utiliza en la fase de ANÁLISIS.
- ORDINOGRAMAS: Diagramas de flujos del programa. Representación gráfica de la secuencia de operaciones que se han de realizar en un programa. Se utiliza en la fase de PROGRAMACIÓN o DISEÑO.

Los diagramas de flujo buscan que el programador retenga en su mente el objetivo del programa con una mirada, por lo que deben cumplir las siguientes cualidades:

- **Sencillez:** Construcción fácil y sencilla.
- **Claridad:** Fácilmente comprensible para otras personas.
- **Normalización:** Utilizar las mismas normas de construcción.
- **Flexibilidad:** Susceptible de ser fácilmente modificado y ampliado.

Organigramas

Son representaciones gráficas del flujo de datos e informaciones que maneja un programa.

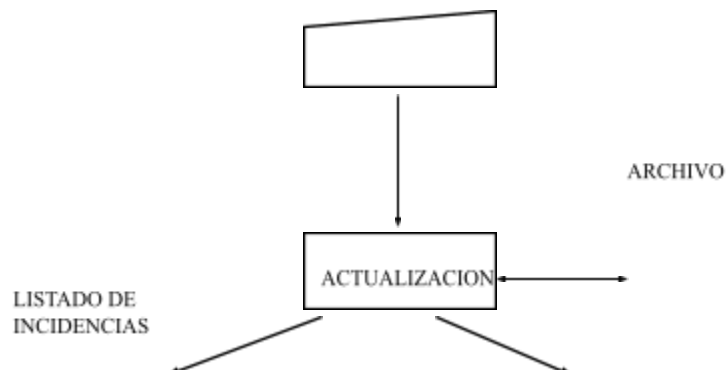
En un organigrama se deben mostrar los siguientes elementos:

- **Los soportes de los datos de entrada** (símbolos de soporte)
- **El nombre del programa** (rectángulo central)
- **Los soportes de los datos de salida** (símbolos de soporte)
- **El flujo de los datos** (líneas de flujo): la información circula entre los elementos anteriores, por eso es necesario establecer el camino a través de flechas que unan todos los elementos. Estas flechas indican el sentido de dichos datos.

Para representar un organigrama se deben seguir las siguientes **reglas**:

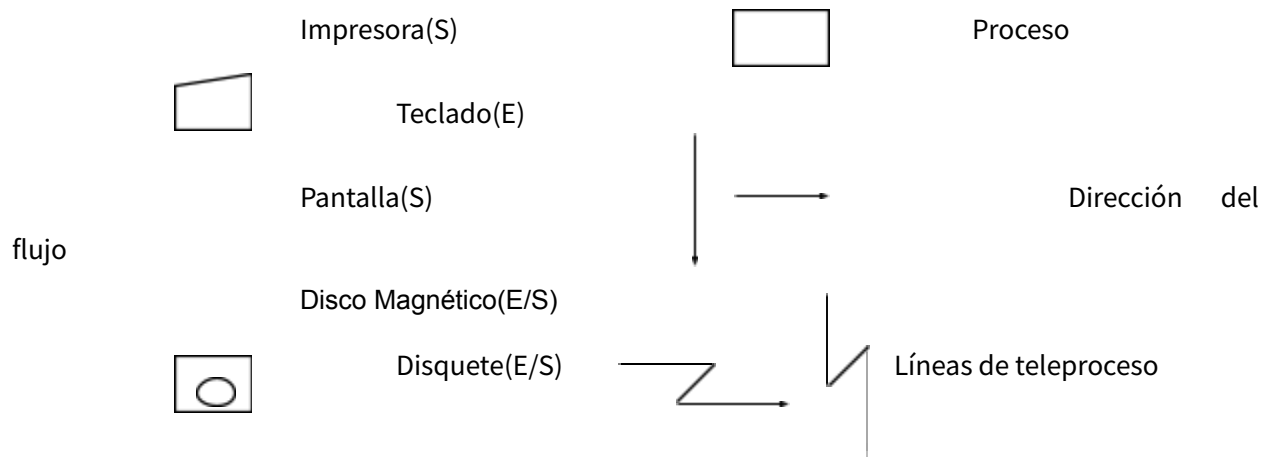
- 1) En el centro figurará el nombre del programa (rectángulo)
- 2) En la parte superior los periféricos de entrada
- 3) En la parte inferior los periféricos de salida
- 4) En las zonas de la derecha e izquierda, los soportes de los datos de entrada y salida.

Ejemplo: Organigrama de una aplicación de actualización del fichero de apuntes de contabilidad, con entrada de datos por teclado, consulta de datos por pantalla y listado de incidencias por impresora.



VISUALIZACIÓN
DE DATOS

Los símbolos utilizados en los organigramas se dividen en símbolos de **soporte**, símbolos de **proceso**, y **líneas de flujo**. Algunos ejemplos de estos símbolos son:



Ordinogramas

Representan gráficamente paso a paso todas las instrucciones del programa, es decir, la secuencia de operaciones que se realizan en un programa.








Un ordinograma debe representar con claridad:

- El inicio del programa.
- Las operaciones.
- La secuencia (el orden) en que se realizan las operaciones.
- El final del programa.

Los símbolos utilizados en los ordinogramas se dividen en símbolos de **operación**, símbolos de **decisión**, **líneas de flujo** y símbolos de **conexión**. Algunos ejemplos de estos símbolos son:

Símbolos

	SÍMBOLOS UTILIZADOS EN DIAGRAMAS DE FLUJO	
Símbolo	Explicación	Instrucción en pseudocódigo

	Se utiliza para marcar el y inicio y fin de un diagrama de flujo.	INICIA PROGRAMA TERMINA PROGRAMA
	Se utiliza para introducir datos de entrada (expresa lectura).	LEER(...)
	Se utiliza para indicar un proceso. En su interior se expresan asignaciones, operaciones aritméticas, etc.	<i>Identificador</i> ← <i>expresión</i>
	Representa una decisión. En el interior del símbolo se evalúa una condición y dependiendo de su resultado se sigue por una de las ramas o caminos alternativos.	SI (<i>condición</i>) ENTONCES Bloque de instrucciones FIN DE SI SI (<i>condición</i>) ENTONCES Bloque de instrucciones DE LO CONTRARIO Bloque de instrucciones FIN DE SI
	Representa una decisión múltiple y en su interior se almacena un selector, y dependiendo de su valor se sigue por una de las ramas o caminos alternativos.	EN CASO DE (selector) HACER VALOR 1: Bloque de instrucciones ... VALOR N: Bloque de instrucciones DE LO CONTRARIO Bloque de instrucciones FIN DE CASO
	Representa la impresión de un resultado (expresa escritura).	MOSTRAR (...)
	Se utiliza para representar un proceso predefinido.	LLAMAR ...
	Se utilizan para indicar la dirección de flujo del diagrama.	
	Se utiliza para conectar un diagrama de flujo dentro de la misma página.	
	Se utiliza para expresar conexión entre páginas diferentes.	

Pseudocódigo

Además de la utilización de representaciones gráficas, un programa se puede describir mediante un **lenguaje intermedio** entre el lenguaje natural y el lenguaje de programación. La utilización de una notación intermedia permite el diseño del programa sin depender de ningún lenguaje de programación.

Pseudocódigo: Es una notación mediante la cual podemos describir la solución de un problema en

forma de algoritmo, utilizando palabras y frases del lenguaje natural sujetas a unas determinadas reglas.

El pseudocódigo ha de considerarse como una herramienta para el diseño de programas más que una notación para la descripción de los mismos → debido a su **flexibilidad**, permite obtener la solución a un problema mediante aproximaciones sucesivas → **diseño descendente** (Programación modular).

La notación en pseudocódigo se caracteriza por:

- a) No puede ser ejecutado directamente por un ordenador, por lo que tampoco es considerado como un lenguaje de programación propiamente dicho
- b) Permite el diseño y desarrollo de algoritmos totalmente independientes del lenguaje de programación posteriormente utilizado en la fase de codificación del algoritmo, pues no está sujeto a las reglas sintácticas de ningún lenguaje excepto las del suyo propio
- c) Es sencillo de aprender y utilizar
- d) Facilita al programador el paso del algoritmo al correspondiente lenguaje de programación
- e) Permite una gran flexibilidad en el diseño del algoritmo a la hora de expresar acciones concretas
- f) Permite con cierta facilidad la realización de futuras correcciones o actualizaciones gracias a que no es un sistema de representación rígido
- g) La escritura o diseño de un algoritmo mediante el uso de esta herramienta, exige la “indentación” o “sangría” del texto en el margen izquierdo de las diferentes líneas
- h) Permite obtener la solución de un problema mediante aproximaciones sucesivas, es decir, lo que se conoce comúnmente como diseño descendente o **Top down** y que consiste en la descomposición sucesiva del problema en niveles o subproblemas más pequeños, lo que nos permite la simplificación del problema general

Todo pseudocódigo debe permitir la descripción de los siguientes elementos:

- Instrucciones de entrada/salida y de asignación (PRIMITIVAS o SIMPLES)
- Instrucciones de declaración
- Sentencias de control del flujo de ejecución
- Acciones compuestas (subprogramas) que hay que refinar posteriormente

La estructura general de un programa en pseudocódigo es la siguiente:

Programa NOMBRE DEL PROGRAMA

Entorno:

Descripción del conjunto de variables de un programa:
declaración de sus nombres y tipos

Algoritmo:

Secuencia de instrucciones que forman el programa

FinPrograma

Subprograma NOMBRE DEL SUBPROGRAMA

Entorno:

Algoritmo:

FinSubPrograma

Ejemplo.- Pseudocódigo de un programa que calcule el área de un rectángulo. Se debe introducir la base y la altura para realizar el cálculo:

Programa CALCULA_AREA

Entorno:

BASE, AREA, ALTURA son numéricas enteras (suponemos que son enteros)

Algoritmo:

 escribir "Introduzca la base y la altura"

 leer BASE, ALTURA

 AREA ← BASE*ALTURA

 escribir "El área del rectángulo es la siguiente ", AREA

FinAlgoritmo

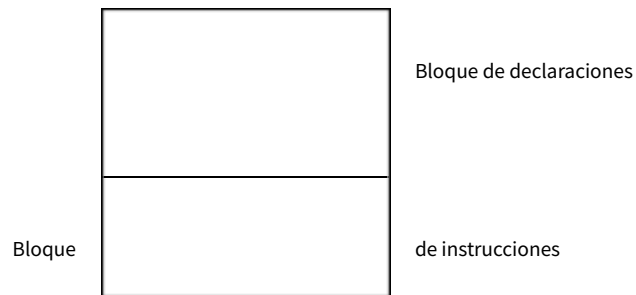
Estructura general de un programa

Un programa puede considerarse como una secuencia lógica de acciones (instrucciones) que manipulan un conjunto de variables (datos) para obtener unos resultados que serán la solución al problema que resuelve dicho programa-algoritmo.

Todo programa tiene dos partes o bloques. El primer bloque es el bloque de declaraciones. En éste se especifican todas las variables que utiliza el programa (constantes, variables, etc.) indicando sus

características. El segundo bloque es el de las instrucciones, formado por el conjunto de operaciones que se han de realizar para la obtención de los resultados deseados.

La ejecución de un programa consiste en la realización secuencial del conjunto de instrucciones de que se compone. Las instrucciones de un programa consisten en general en modificaciones sobre las variables del programa, desde un estado inicial hasta otro final. Al conjunto de variables de un programa se le llama también **entorno** del programa.



Las partes principales de un programa serían tres:

- 1) Entrada de datos (desde los dispositivos externos hasta memoria central).
- 2) Proceso (paso de un estado inicial a un estado final).
- 3) Obtención de resultados (desde memoria central hacia los dispositivos externos).

Instrucciones

Una instrucción se caracteriza por un **estado inicial**, representado por el valor de las variables antes de la ejecución de la instrucción, y otro **estado final** representado por el valor de las variables después de la ejecución de la misma.

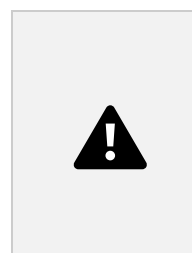
Instrucciones de declaración

Anuncian la utilización de variables en un programa indicando qué identificador, tipo y otras características corresponde a cada uno de ellos.

Instrucciones de entrada

Su misión consiste en tomar uno o varios datos desde un dispositivo de entrada y almacenarlos en las variables cuyos identificadores aparecen en la propia instrucción. Su sintaxis metodológica es:

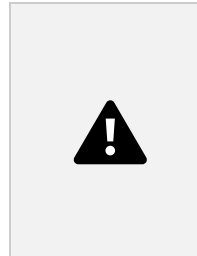
leer <lista de variables>



Instrucciones de salida

Es el conjunto de instrucciones que muestran el valor de algunas variables en los dispositivos de salida (monitor...). Sintaxis:

escribir <lista de variables>



Instrucciones de asignación

Permiten realizar cálculos evaluando una expresión y depositando su valor final en un objeto o realizar movimientos de datos de un objeto a otro. Su sintaxis es:


variable  expresión

←



Ejemplo.- EDAD  EDAD_MAX

Una expresión puede ser una variable o una combinación

EDAD  17 * EDAD

Esta instrucción se realiza en dos tiempos; primero se evalúa la expresión convirtiéndose en su valor final; segundo, el valor final se asigna al objeto borrándose el valor previo que éste pudiese tener.

El objeto y la expresión deben coincidir en tipo y se admite que el propio objeto que recibe el valor final de la expresión pueda intervenir en la misma pero entendiéndose que lo hace con su valor anterior.

Instrucciones de control

Son instrucciones que tienen como objetivo el controlar la ejecución de otras instrucciones o alterar el orden de ejecución normal de las mismas.

Instrucciones alternativas

Controlan la ejecución de uno o varios bloques de instrucciones dependiendo del cumplimiento o no de alguna condición o del valor final de una expresión.

Alternativa simple

Controla la ejecución de un conjunto de instrucciones por el cumplimiento o no de una condición.

Sintaxis:

si CONDICION **entonces**

Ins1

...

InsN

finSi

*Alternativa múltiple*

Controla la ejecución de varios conjuntos de instrucciones por el valor final de una expresión, de tal forma que cada conjunto de instrucciones está ligado a un posible valor de la expresión. Se ejecutará el conjunto que se encuentre relacionado con el valor que resulte de la evaluación de la expresión.

Sintaxis:

Opción EXPRESION **de**

V1 **hacer** Ins1, Ins2....

V2 **hacer** Ins1, Ins2...

....

VN **hacer** Ins1, Ins2...

otra hacer

finOpción



Ejemplo.- opcion EDAD de

4 hacer regalo <--- "JUGUETE"

15 hacer regalo <--- “CD”

otro hacer regalo <--- “LIBRO”

finOpción

Instrucciones repetitivas

Son aquellas que controlan la repetición de un conjunto de instrucciones denominado rango mediante la evaluación de una condición que se realiza cada nueva repetición o por medio de un contador asociado.

Instrucción MIENTRAS

El conjunto de instrucciones que configuran su rango se ejecutan mientras se cumpla la condición que será evaluada siempre antes de cada repetición, es decir, mientras la condición sea CIERTA.

mientras CONDICION **hacer**

Ins

finMientras

Ejemplo.- leer numero, potencia

resultado <---- 1

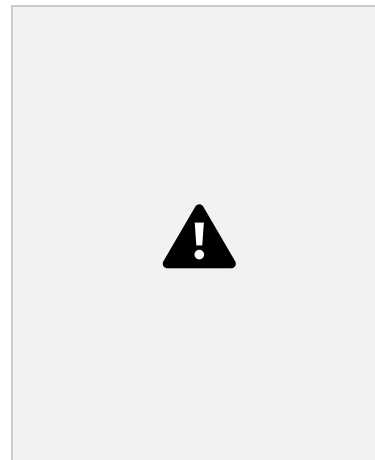
mientras (potencia >0) hacer

resultado <--- resultado * numero

potencia <---- potencia - 1

finMientras

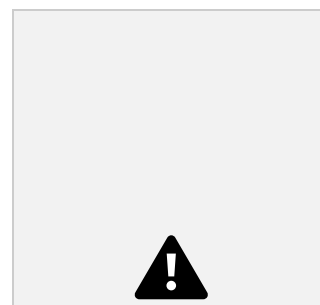
escribir resultado



Instrucción REPETIR

Controla la ejecución del conjunto de instrucciones que configuran su rango de tal forma que éstas se ejecutan hasta que se cumpla la condición que será evaluada siempre después de cada repetición, es decir, hasta que la condición sea CIERTA.

Sintaxis:



repetir

Ins

hasta CONDICION**Ejemplo.-** El ejemplo anterior pero con estructura Repetir sería

leer numero, potencia

resultado <---- 1

repetir

resultado <---- resultado * numero

potencia <----- potencia - 1

hasta (potencia = 0)

Instrucción PARA

Controla la ejecución del conjunto de instrucciones de su rango de tal forma que éstas se ejecutan un número determinado de veces que queda definido en lo que se denomina cabecera del bucle. En la cabecera se define una variable de control del bucle, su valor inicial, su valor final y su incremento o decremento.

Sintaxis:

para V de Vi a Vf con incremento paso hacer

Ins

FinPara

Ejemplo:

leer numero, potencia

resultado <---- 1







para H de 1 a potencia con incremento 1 hacer




resultado <----- resultado * numero

FinPara

Con esta clase de bucle se facilita el incremento del contador al estar especificada en la cabecera del bucle. Es ideal para repeticiones con incrementos fijos (positivos o negativos).



Tipo de Instrucción	PSEUDOCÓDIGO	ORDINOGRAMA	LENGUAJE C	Java
De entrada	leer <lista de variables>		Funciones de biblioteca (scanf, getch, getchar, ...)	Métodos de clases y objetos System.in.*
De salida	escribir <lista de variables>		Funciones de biblioteca (printf, putchar, ...)	Métodos de clases y objetos System.out.*
Asignación	variable = expresión		variable = expresión ;	variable = expresión ;
De control: Alternativa simple	si CONDICION entonces Ins1 ... InsN finSi		if (CONDICION) { Ins1 ... InsN }	if (CONDICION) { Ins1 ... InsN }
De control: Alternativa doble	si CONDICION entonces Ins1 sino Ins2 FinSi		if (CONDICION) { Ins1 } else { Ins2 }	if (CONDICION) { Ins1 } else { Ins2 }
De control: Alternativa múltiple	opción EXPRESION de V1 hacer Ins1, Ins2.... V2 hacer Ins1, Ins2... VN hacer Ins1, Ins2... otra hacer		switch (expresión) { case constante 1: instrucciones; break; case constante 2: instrucciones; break; ... default: instrucciones; }	switch (expresión) { case constante 1: instrucciones; break; case constante 2: instrucciones; break; ... default: instrucciones; }

	finOpción			
De control: Repetitiva Mientras	mientras CONDICION hacer instrucciones finMientras		while (CONDICION) { instrucciones; } 	while (CONDICION) { instrucciones; }
De control: Repetitiva Repetir Hasta	repetir instrucciones; hasta CONDICION		do { instrucciones; } while (CONDICION != 1);	do { instrucciones; } while (CONDICION != true);
			do { instrucciones; } while (CONDICION == 0); do	do { instrucciones; } while (CONDICION == false);
De control: Repetitiva Repetir Mientras	repetir instrucciones; mientras CONDICION		do { instrucciones; } while (CONDICION);	do { instrucciones; } while (CONDICION);
De control: Repetitiva Para	para V de Vi a Vf con j hacer instrucciones FinPara		for (inicialización; CONDICION; Incremento) { instrucciones; } 	for (inicialización; CONDICION; Incremento) { instrucciones; }

Variables auxiliares

Contadores

Son variables que se utilizan para contar cualquier evento que pueda ocurrir dentro de un programa. Se utilizan realizando sobre ellos dos operaciones básicas: **inicialización**, e **incremento**.

Ejemplo.- Programa que lee una lista de 50 notas de alumnos e indica cuantos están aprobados:

Programa: APROBADOS

Entorno:

NUMERO (contador) es numérico entero
 APROB (contador de aprobados) es numérico entero
 NOTA es numérico entero

Algoritmo:

APROB \leftarrow 0
 NUMERO \leftarrow 0
 Mientras NUMERO \leq 50 hacer
 NUMERO \leftarrow NUMERO + 1
 Leer NOTA
 Si NOTA \geq 5 entonces
 APROB \leftarrow APROB + 1
 FinSi

FinMientras
 Escribir APROB

FinAlgoritmo

Acumuladores

Se utilizan para realizar sumatorios o productos de distintas cantidades. Para el sumatorio se inicializan a 0 y para el producto a 1. Para utilizarlos se realizan sobre ellos dos operaciones básicas: **inicialización**, y **acumulación**.

Ejemplo.- Algoritmo que calcula y escribe la suma y el producto de los 10 primeros números naturales:

Programa: SUMANAT

Entorno:

SUMA (acumulador) es numérico entero
 PRODUCTO (acumulador) es numérico entero
 CONTA es numérico entero

Algoritmo:

SUMA \leftarrow 0
 PRODUCTO \leftarrow 1
 CONTA \leftarrow 1
 Mientras CONTA \leq 10 hacer
 SUMA \leftarrow SUMA + CONTA
 PRODUCTO \leftarrow PRODUCTO * CONTA
 CONTA \leftarrow CONTA + 1

FinMientras
 Escribir SUMA, PRODUCTO

FinAlgoritmo

Interruptores, conmutadores, switches y banderas

Son variables que pueden tomar dos valores exclusivamente. Normalmente 0 y 1, -1 y +1, cierto y falso, etc. Se utilizan para transmitir información de un punto a otro de un programa y para conmutar alternativamente entre dos caminos posibles. Se utilizan inicializándolos con un valor y en los puntos

en que corresponda se cambian al valor contrario, de tal forma que examinando su valor posteriormente podemos realizar la transmisión de información que deseábamos.

Ejemplo.- Algoritmo que lee una secuencia de notas (con valores que van de 0 a 10) que termina con el valor -1 y nos dice si hubo o no alguna nota con valor 10:

Programa: NOTAS

Entorno:

NOTA es numérico entero

NOTA10 booleano (switch)

Algoritmo:

NOTA10 ~~es~~ FALSO

Leer NOTA

Mientras NOTA <> -1 hacer

Si NOTA = 10 entonces

NOTA10 ~~es~~ CIERTO

FinSi

Leer NOTA

FinMientras

Si NOTA10 = CIERTO entonces

Escribir "Hubo 10"

Sino

Escribir "No hubo 10"

FinSi

FinAlgoritmo

Identificadores, constantes y variables

Los datos que procesa una computadora son almacenados en celdas de memoria para que su utilización posterior. Para poder hacer referencia al contenido de estas celdas es necesario asignarle un *nombre* para su fácil identificación. A ese nombre se llama **identificador** y se forma de la siguiente manera¹:

- El primer carácter que forma un identificador debe ser una letra (a..z, A..Z)
- Los caracteres restantes pueden ser una combinación tanto de letras como de números (0..9) o el símbolo especial: _

Los identificadores se pueden utilizar para nombrar a constantes, variables, funciones y procedimientos.

Una **constante** es un valor que, una vez fijado no cambia durante la ejecución de un programa.

El valor de una **variable**, a diferencia de las constantes, puede cambiar a lo largo de la ejecución de un programa.

Expresiones y operadores

Los **operadores** son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales y lógicos. A partir de estos se pueden expresar

¹ En un lenguaje de programación estas reglas varían dependiendo del lenguaje.

expresiones de distinta complejidad.

Operadores aritméticos

Con los siguientes operadores se pueden representar expresiones aritméticas entre operandos: números, constantes o variables. El resultado de una operación aritmética será un número.

Operador	Símbolo	Ejemplo	Resultado
Potencia	\wedge	$4 \wedge 3$	64
Multipliación	*	$2.5 * 2$	5
División	/	$17 / 2$	8.5
Suma	+	$6.5 + 10$	16.5
Resta	-	$7 - (-3)$	10
Módulo	MOD	$5 \text{ MOD } 2$	1
División Entera	DIV	$15 \text{ DIV } 2$	7

Operadores relacionales

Los operadores relacionales son aquellos que permiten comparar dos operandos. Los operandos pueden ser números, caracteres o cadena de caracteres, constantes o variables. El resultado de una expresión con operadores relacionales es VERDADERO o FALSO.

Operador	Símbolo	Ejemplo	Resultado
Mayor que	>	$-8 > 1$	FALSO
Menor que	<	$5 < -10$	FALSO
Mayor o igual que	>=	$4 >= 4$	VERDADERO
Menor o igual que	<=	$15 <= 0$	VERDADERO
Igual que	=	$-3 = 3$	FALSO
		'hola' = 'bola'	FALSO
Diferente a	<>	$5 <> 50$	VERDADERO
		'peso' <> 'pesos'	VERDADERO

Operadores lógicos

Los operadores lógicos son operadores que permiten formular condiciones complejas a partir de condiciones simples.

- O El operador lógico **O** da como resultado el valor lógico FALSO, si ambos operandos son FALSO. Si uno de los operandos es VERDADERO, el resultado será VERDADERO.
- Y El operador lógico **Y** da como resultado el valor lógico VERDADERO, si ambos operandos son distintos de FALSO. Si uno de ellos es FALSO el resultado será FALSO.
- NO El operador lógico **NO** da como resultado el valor lógico FALSO si el operando tiene el valor de

VERDADERO y FALSO en caso contrario.

O	Resultado		Y	Resultado		NO	Resultado
F O F	F		F Y F	F		NO F	V
F O V	V		F Y V	F		NO V	F
V O F	V		V Y F	F			
V O V	V		V Y V	V			

Una **expresión** es una combinación de operadores y operandos que dan lugar a un único valor. Al evaluar expresiones se debe respetar la jerarquía de cada operador. A continuación se presenta una tabla que muestra la jerarquía de todos operadores anteriores.

Operador	Jerarquía
()	(mayor)
^	
*, /, MOD, DIV	
+, -	
>, <, >=, <=, =, <>	
NO	
Y	
O	(menor)

Para resolver una expresión se deben respetar las siguientes reglas:

1. Si una expresión contiene subexpresiones entre paréntesis, éstas se evalúan primero; respetando la jerarquía de los operadores en esta subexpresión. Si las subexpresiones se encuentran anidadas por paréntesis primero se evaluarán las subexpresiones mas internas.
2. Los operadores se aplican teniendo en cuenta la jerarquía de izquierda a derecha.

EJEMPLOS

1) Cual es el resultado de la siguiente expresión: $7 * 8 * (160 \text{ MOD } 3 ^ 3) \text{ DIV } 5 * 13 - 28$

$$7 * 8 * (160 \text{ MOD } 3^3) \text{ DIV } 5 * 13 - 28$$

$$7 * 8 * (160 \text{ MOD } 27) \text{ DIV } 5 * 13 - 28$$

$$7 * 8 * 25 \text{ DIV } 5 * 13 - 28$$

$$56 * 25 \text{ DIV } 5 * 13 - 28$$

$$1400 \text{ DIV } 5 * 13 - 28$$

$$280 * 13 - 28$$

$$3640 - 28$$

$$3612$$

2) Cual es el resultado de la siguiente expresión: $(X * 5 + B^3 / 4) \leq (X^3 \text{ DIV } B)$

Si $X=6$ y $B=7.8$

$$(X * 5 + B^3 / 4) \leq (X^3 \text{ DIV } B)$$

$$(X * 5 + 474.552 / 4) \leq (X^3 \text{ DIV } B)$$

$$(X * 5 + 474.552 / 4) \leq (X^3 \text{ DIV } B)$$

$$(30 + 474.552 / 4) \leq (X^3 \text{ DIV } B)$$

$$(30 + 118.638) \leq (X^3 \text{ DIV } B)$$

$$148.638 \leq (X^3 \text{ DIV } B)$$

$$148.638 \leq (216 \text{ DIV } B)$$

$$148.638 \leq 27.6923$$

FALSO

3) Cual es el resultado de la siguiente expresión: $\text{NO}(15 \geq 7^2) \text{ O } (43 - 8^2 \text{ DIV } 4 \leq 3 * 2)$

$$\text{NO}(15 \geq 7^2) \text{ O } (43 - 8^2 \text{ DIV } 4 \leq 3 * 2)$$

NO (**15 >= 14**) O ($43 - 8 \wedge 2 \text{ DIV } 4 <> 3 * 2$)

NO VERDADERO O ($43 - \mathbf{8 \wedge 2 \text{ DIV } 4} <> 3 * 2$)

NO VERDADERO O ($43 - \mathbf{64 \text{ DIV } 4} <> 3 * 2$)

NO VERDADERO O ($43 - 16 <> \mathbf{3 * 2}$)

NO VERDADERO O ($\mathbf{43} - \mathbf{16} <> 6$)

NO VERDADERO O ($\mathbf{27} <> \mathbf{6}$)

NO VERDADERO O VERDADERO

FALSO O VERDADERO

VERDADERO

Asignaciones

Una expresión de asignación se usa para dar un valor a un variable utilizando el operador de asignación =. El formato de una asignación es el siguiente:

Variable \leftarrow expresión o valor

Donde *expresión* puede ser aritmética o lógica, o una constante o una variable

Ejemplo: Suponga que tiene la siguiente secuencia de asignaciones, donde I, ACUM, J son variables de tipo entero, CAR de tipo carácter, REAL de tipo real y FLAG de tipo lógico.

- 1) $I \leftarrow 0$
- 2) $ACUM \leftarrow 0$
- 3) $I \leftarrow I + 1$
- 4) $J \leftarrow 2 \wedge 3 \text{ DIV } 3$
- 5) $CAR \leftarrow 'a'$
- 6) $ACUM \leftarrow J * 2$
- 7) $REAL \leftarrow ACUM / 3$
- 8) $FLAG \leftarrow (8 > 5) Y (I = J)$

El estado de las celdas de memoria siguiendo un orden secuencial de las asignaciones anteriores cambia de la siguiente manera:

	I	J	ACUM	REAL	CARACTER	BANDERA
0	?	?	?	?	?	?

1	0	?	?	?	?	?
2	0	?	0	?	?	?
3	1	?	0	?	?	?
4	1	2	0	?	?	?
5	1	2	0	?	a	?
6	1	2	4	?	?	?
7	1	2	4	1.33333	a	?
8	1	2	4	1.33333	a	FALSO

Técnicas de programación

Programación estructurada

Las técnicas de desarrollo y diseño de programas que se utilizan en la programación no estructurada tienen inconvenientes, sobre todo a la hora de verificar y modificar un programa. Para evitar estos inconvenientes surgieron técnicas de programación que pretenden facilitar la comprensión de los programas y permiten, de forma rápida, el mantenimiento de los mismos.

La programación estructurada fue desarrollada en sus principios por Edsger W. Dijkstra y se basa en el denominado **Teorema de la Estructura**, desarrollado por Böhm y Jacopini.

En la programación convencional no estructurada se suele hacer un uso indiscriminado y sin control de las instrucciones de salto condicional e incondicional, lo cual produce complejidad en la lectura y en las modificaciones de un programa. **La programación estructurada se ha definido como la técnica de programación sin saltos condicionales e incondicionales.**

Todo programa estructurado puede ser leído de principio a fin sin interrupciones en la secuencia normal de lectura. De esta forma se obtiene una mayor claridad en el programa, y un mantenimiento y documentación más rápidos.

Los programadores realizan cada tarea en **bloques** o **módulos**. Puesto que no siempre se dispone de los mismos programadores, es muy importante que un programa realizado por una persona sea fácil de modificar y mantener por otra. La programación estructurada ofrece muchas ventajas para lograr estos objetivos.

Un programa estructurado es:

- Fácil de leer y comprender.
- Fácil de codificar en una amplia gama de lenguajes y en diferentes sistemas.
- Fácil de mantener.
- Eficiente, aprovechando al máximo los recursos del sistema.

- Modular.

Teorema de la estructura

Conceptos necesarios:

- **Programa propio:** es aquel programa que cumple las siguientes condiciones:
 - Posee un solo inicio y un solo fin.
 - Todo elemento del programa es accesible, es decir, existe al menos un camino desde el inicio al fin que pasa a través de él.
 - No posee bucles infinitos.
- **Equivalencia de programas:** Dos programas distintos son equivalentes si proporcionan, ante cualquier situación de datos, el mismo resultado.

TEOREMA DE LA ESTRUCTURA: Todo programa propio tiene, al menos, un programa equivalente que sólo utiliza las estructuras básicas de la programación:

- 1) La secuencia
- 2) La selección o alternativa
- 3) La repetición

El teorema quiere decir que **diseñando programas con sentencias primitivas (lectura, escritura y asignación) y estructuras básicas**, no sólo podremos hacer cualquier trabajo sino que además **conseguiremos mejorar la creación, lectura, comprensión y mantenimiento de los programas**.

Herramientas de la programación estructurada

La programación estructurada utiliza:

- a) Diseño descendente (TOP-DOWN).
- b) Recursos abstractos.
- c) Estructuras básicas.

Diseño descendente. Los programas se diseñan de lo general a lo particular por medio de sucesivos refinamientos o descomposiciones que nos van acercando a las instrucciones finales del programa.

Utilización de recursos abstractos. En cada descomposición se supone que todas las partes

resultantes están resueltas, dejando los detalles para el siguiente refinamiento y considerando que todas ellas pueden llegar a estar definidas en instrucciones y estructuras disponibles en los lenguajes de programación. **Ejemplo:** un viaje por etapas.

Estructuras básicas. El teorema de la estructura dice que toda acción se puede realizar utilizando tres estructuras básicas de control: la estructura secuencial, alternativa y repetitiva.

Ejemplos de métodos que utilizan la programación estructurada son: método de WARNIER, método de JACKSON, método de BERTINI, ..., que en general son métodos para la representación de programas.

Programación modular y diseño descendente

Ante programas complejos y/o de gran tamaño, lo más adecuado es descomponer el problema, ya desde su fase de análisis, en partes cuya resolución sea más asequible. La programación de cada una de estas partes se realiza independientemente de las otras, incluso por diferentes personas.

El **diseño descendente** o diseño **top-down** consiste en una serie de descomposiciones sucesivas del problema inicial que describen el refinamiento progresivo de las de instrucciones que van a formar parte del programa.

La utilización de esta técnica tiene los siguientes objetivos:

- **Simplificación del problema** y de los subproblemas resultantes de cada descomposición.
- Las **diferentes partes del problema pueden ser programadas de modo independiente** e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloques o **módulos** lo que hace más **sencilla** su lectura y mantenimiento.

Programa principal y subprogramas

Un programa diseñado siguiendo esta técnica, quedará constituido por dos partes:

- ❑ **Programa principal:** Describe la solución completa del problema de manera general; consta principalmente de llamadas a subprogramas que se encargan de alguna parte específica del proceso -Principio de delegación-. Las llamadas son indicaciones al procesador de que debe continuar la ejecución del programa en el subprograma invocado regresando al punto de partida una vez lo haya concluido.

El programa principal puede contener además instrucciones primitivas y sentencias de control pero prestando atención a que contenga pocas líneas de manera que se vean claramente los diferentes pasos del proceso que se ha de seguir para la obtención de los resultados deseados.

- ❑ **Subprogramas:** Su estructura coincide básicamente con la de un programa. Su función es

resolver de modo independiente una parte del problema. Es importante que realice una función concreta y única en el contexto del problema. -Principio de responsabilidad única-

Los subprogramas pueden ser básicamente de tres tipos :

- ❖ Subrutinas.
- ❖ Procedimientos.
- ❖ Funciones.

Ejemplo:

Programa

Entorno

...

Algoritmo

llamada A

llamada B

FinPrograma

Subprograma A

entorno

...

Algoritmo

...

...

FinSubprograma

Subprograma B

entorno

...

algoritmo

...

...

FinSubprograma

Variables globales vs. locales

Los diferentes objetos (variables) que se manejan en un programa se clasifican según su ámbito, es decir, según la porción de programa o subprogramas en que son conocidos y por tanto pueden ser utilizados.

Son variables globales los que tienen como ámbito al programa principal y a todos sus subprogramas.

Son variables locales a un subprograma aquellos cuyo ámbito está restringido a él mismo.

Cuando se declaran variables que en el programa principal son globales y después otras con los mismos nombres como locales, siempre se tomará el valor y tipo de la local dentro de ese subprograma en cuestión. NO CONVIENE hacer esto para evitar malentendidos. Lo que sí suele hacerse es declarar varias variables iguales en nombre para distintos subprogramas.

Paso de parámetros o argumentos

El proceso de emisión y recepción de datos y resultados se realiza mediante los parámetros.

Cada vez que se realiza una llamada a un subprograma, los datos de entrada le son pasados por medio de los parámetros de entrada y cuando termina la ejecución del subprograma los resultados regresan o son devueltos mediante los parámetros de salida.

Recursividad

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración (bucles) para resolver determinado tipo de problemas. Consiste en permitir que un subprograma se llame a sí mismo para resolver una versión reducida del problema original.

Frente a determinados problemas se puede optar por una solución iterativa (no recursiva) o una solución recursiva. Existen situaciones en las que el uso de la recursividad permite soluciones (programas) mucho más simples y elegantes. *No conviene abusar de la recursividad, pues podría dar lugar a resultados impredecibles y de difícil comprensión.*

La recursividad es especialmente apropiada cuando el problema a resolver (factorial) o la estructura de datos a procesar (árboles) tienen una clara definición recursiva.

Ejemplo.- Si se desea calcular el factorial de un número n , entero positivo, se hará a partir de su definición:

$$0! = 1$$

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1, \text{ si } n > 0$$

Esto daría lugar a una solución iterativa:

Subprograma CALCULAR_FACTORIAL

Recibe N numérica entera (mayor o igual que 0)

Trasforma F numérica entera

Algoritmo:

F \leftarrow 1

Mientras N > 0 hacer

 F \leftarrow F * N

 N \leftarrow N - 1

FinMientras
FinSubprograma

Pero existe otra definición “recursiva” de la función factorial:

$$0! = 1$$

$$n! = n \cdot (n - 1)!, \text{ si } n > 0$$

Esta definición da lugar a una solución recursiva:

Subprograma CALCULAR_FACTORIAL

Recibe N numérica entera (mayor o igual que 0)
Trasforma F numérica entera

Entorno:

Algoritmo:

Si N = 0 entonces

 F ← 1

Sino

 F ← N * CALCULAR_FACTORIAL (N-1)

FinSi

FinSubprograma

Se dice que un programa es recursivo si entre sus instrucciones tiene una llamada a sí mismo.

Si utilizamos funciones recursivas, lo que nunca debemos olvidar es incluir alguna sentencia de control (tipo if) que permita que exista una situación en que se ejecute la función sin volver a llamarse a sí misma. Si no hacemos esto, la función se llamará a sí misma infinitamente hasta que el contenido de la pila desborde la capacidad de la memoria del sistema, provocando un error y finalizando la ejecución.

Ejemplo: Programa modular o estructurado, donde cada problema se resuelve por separado en módulos o procedimientos (funciones).

Programa NOTAS

Entorno

notas es tabla (100) tipo numérico entero

numalu, max es numérico entero

media es numérico real

variables locales al programa principal

Algoritmo

INTRODATOS(notas ; numalu, notas)

CALCMEDIA(numalu, notas ; media)

CALCMAX(numalu, notas ; max)

MUESTRARES(media, max ;)

finPrograma

Subprograma INTRODATOS

recibe

 notas

devuelve

 numalu

```

Entorno
    Ind, numalu tipo numérico entero
Algoritmo
    leer numalu
    para Ind de 1 hasta numalu hacer
        leer notas (Ind)
    finPara
finSubprograma

Subprograma CALCMEDIA
recibe
    numalu, notas
devuelve
    media
Entorno
    Ind, media tipo numérico entero
    Observa que lo que aparece en devuelve,
    parámetro de salida, debe declararse en el entorno
    del programa principal y en el subprograma.
Algoritmo
    media  $\approx$  0
    para Ind de 1 hasta numalu hacer
        media  $\approx$  media + notas (Ind)
    finPara
    media  $\approx$  media / numalu
finSubprograma

Subprograma CALCMAX
recibe
    numalu, notas
devuelve
    max
Entorno
    Ind, max
    Lo mismo ocurre con MAX (parámetro de salida)
Algoritmo
    max  $\approx$  0
    para Ind de 1 hasta numalu hacer
        si (notas (Ind) > max) entonces
            max  $\approx$  notas (Ind)
        finSi
    finPara
finSubprograma

Subprograma MUESTRARES
recibe
    media, max
devuelve
    ---
    No devuelve nada
Entorno
    ---
    No hay entorno. Variables locales

Algoritmo
    escribir "La media es: ", media
    escribir "El máximo es ", max
FinSubprograma

```

Funciones, procedimientos y subrutinas

Funciones

La idea de función es la de una "caja negra" en la que nosotros introducimos datos, dentro de esa caja pasa "algo", y entonces, de la caja, sale un resultado, un producto.

Qué pasa dentro de esa "caja negra" depende; si somos nosotros quienes hemos de programarla, lo sabremos, pero si no, no tenemos por qué. Para poder usar la función sólo necesitaremos saber qué datos de entrada admite, y de qué tipo será el resultado. Hay que remarcar un detalle importante: las funciones devuelven un **ÚNICO VALOR**.

Por ejemplo; dentro de un programa, podemos querer calcular la media aritmética de una serie de datos. En principio, nosotros lo escribimos cuando tenemos que hacer los cálculos. Pero ahora, resulta que más adelante tenemos que volver a calcular la media aritmética de otros datos. Y más adelante, otra vez. ¿Vamos a escribir el código tantas veces? ¿No sería más lógico definirnos una función que se encargara de esa parte, y llamarla cuando la necesitemos?

La definición de una función es la siguiente:

funcion NOMBRE (arg1,...,argN) : TIPO

// Declaracion de variables

Inicia funcion

accion1

...

accionN

Resultado ← Valor

Termina funcion

TIPO es el tipo de dato que devolverá la función al terminar de hacer su trabajo. **NOMBRE** es el nombre que le vamos a dar a la función; por ejemplo *Media_Aritmetica*. **arg1 ... argN** es la lista de parámetros que vamos a pasar a la función. La sección "variables" es una sección donde se declararán las variables a usar por la función.

accion1 ... accionN son todas las instrucciones que debe hacer la función. Al final de éstas, la función devuelve un Resultado, que es el que hemos especificado como "Valor". ¿Y a dónde va a parar ese "Valor"? Bueno, es que para poder usar una función, tenemos que invocarla, llamarla de alguna manera. Si las funciones son cajas que devuelven valores, tendremos que disponer algún sitio para almacenar ese valor que nos devuelva la función.

Llamada de una función

Para poder llamar a una función, tendremos que tener definida en nuestra declaración de variables

una variable del mismo tipo que devuelva la función. Entonces, lo que hacemos es asignar a esa variable lo que nos devuelva la función, haciendo lo siguiente:

Variable \leftarrow Nombre_Funcion(arg1, ..., argN)

Esta línea hace lo siguiente: llama a la función Nombre_Funcion, pasándole los parámetros arg1, ..., argN; entonces, se ejecuta el código de la función, hasta que llega al final, momento en que devuelve un valor, y este valor devuelto es asignado a la variable Variable.

Ejemplo

Supongamos que se desea hacer un programa que calcule, en varios puntos, la suma de los N primeros números naturales, pero este N varía conforme el programa lo necesita. Queremos hacer una función que nos simplifique el trabajo. ¿Cómo lo hacemos? Bueno, lo primero que hay que plantearse siempre es qué parámetros necesita la función para trabajar, qué tipo de valor va a devolver y, por último, cómo va a hacer lo que tenga que hacer la función.

En nuestro caso, la función sólo necesita saber quién es N, que será de tipo entero; como la suma de naturales es natural, el resultado a devolver también tendrá que ser una variable de tipo entero. Falta ver cómo implementamos esa función. Por ejemplo, lo podemos hacer así:

funcion Suma_N_Naturales(N:ENTERO) : ENTERO

Suma,i: ENTERO

Inicia funcion

Suma \leftarrow 0

Para i desde 1 hasta N hacer

Suma \leftarrow Suma+i

Fin de Para

Resultado \leftarrow Suma

Termina funcion

y ahora, vamos a usarla. En nuestro programa podemos poner:

N, Suma: ENTERO

Inicia programa

Para N desde 1 hasta 200 hacer

Suma \leftarrow Suma_N_Naturales(N);

Mostrar("La suma de los" ,N, "primeros naturales es" ,Suma)

Fin de PARA Termina programa

Con esto, hacemos 200 veces, incrementando en 1 cada vez N, la asignación a la variable Suma del resultado obtenido por la función Suma_N_Naturales, y mostrando por pantalla el resultado. Cada vez que se llegue a la línea de la asignación, se llamará a la función Suma_N_Naturales, se ejecutará el código de esa función, y al devolver el resultado, el programa principal recupera el control de la ejecución.

Sin embargo, dentro de la función tenemos declarada una variable que se llama Suma, y en el programa principal hay otra variable que se llama Suma... ¿cómo sabemos cuál es la buena? ¿no se mezclan ni nada parecido los valores?

Ámbito de las variables

Como ya hemos visto, un programa no tiene por qué estar formado por un único módulo (vamos a llamarle así) principal, si no que puede estar formado por muchas funciones, y por muchos procedimientos (de los que hablaremos más adelante). Cada función, o cada procedimiento, puede tener, dentro de su sección de declaración de variables, sus propias variables, aunque se llamen igual que las de la función de más arriba, puesto que, al declarar una función o un procedimiento, las variables que usan son LOCALES a ellos, es decir, sólo ellos saben que existen y, por tanto, pueden usarlas.

Como contraposición a las variables locales, tenemos las GLOBALES; éstas variables son reconocidas por cualquier función o procedimiento que exista en nuestro programa, cualquiera puede modificar su valor en cualquier momento.

Ahora, ¿y si hay una variable global que tiene el mismo nombre que una variable local en una función que estoy usando? En ese caso, se usa la variable que es local a la función.

En nuestro ejemplo no se da este conflicto, al no haber una sección de variables globales (eso implica que no las hay), ya que cada variable Suma pertenece a una función distinta.

Algunas notas respecto al tema de funciones:

Es una buena costumbre escribir, justo antes de la definición de la función, un comentario sobre qué hace la función, para qué nos van a servir los parámetros que vamos a pasarle, y qué es lo que devuelve.

Hay que distinguir entre lo que se llama parámetros FORMALES y parámetros ACTUALES.

Cuando definimos una función, en su CABECERA (la línea donde pone su nombre, los argumentos que recibe y el tipo de valor que devuelve) aparecen nombrados los argumentos. El nombre que ponemos en ese momento es lo que se llama parámetros formales. Pero, cuando la llamamos, por ejemplo,

Suma_N_Naturales(27), le estamos pasando el parámetro concreto 27: a estos parámetros se les llama parámetros actuales.

Procedimientos

Se llama así a un subprograma que ejecuta unas ciertas acciones sin que valor alguno de retorno esté asociado a su nombre. En otras palabras: NO devuelven valores (en cierto sentido).

Los procedimientos son normalmente llamados desde el algoritmo principal mediante su nombre y una lista de parámetros actuales (como las funciones) a través de una instrucción específica:

LLAMAR

Se diferencian de las funciones en que los parámetros de llamada pueden ser modificados si así se especifica dentro del procedimiento; en ese sentido se puede interpretar como que devuelven valores.

La forma de declararlos es la siguiente:

PROCEDIMIENTO Nombre (Lista de parámetros formales)

// Declaracion de variables

Inicia procedimiento

acción1

.....

acciónN

Termina Procedimiento

y la forma de usarlos:

LLAMAR Nombre(Lista de parámetros actuales)

Vamos a ver un ejemplo de todo esto: queremos tener una forma de calcular la suma, la resta, el producto y el cociente de dos números cualesquiera. Obviamente, vamos a necesitar 6 variables; 2 de ellas serán los factores, y las otras 4, el resultado de las correspondientes operaciones. Podríamos pensar en 4 funciones que devolvieran cada una de ellas un número (entero, real, ...), pero podemos hacer esto de forma más compacta con un procedimiento. Veamos cómo lo declararíamos:

PROCEDIMIENTO Cuentas(a, b, sum, dif, mul, div: ENTERO)

sum ← a+b

dif ← a-b

mul ← a*b

div ← a/b

Fin Procedimiento

y luego lo podríamos llamar así:

LLAMAR Cuentas(5, 3, SUMA, RESTA, MULT, DIV)

con lo que a las variables SUMA, RESTA, MULT, DIV les serían asignados sus correspondientes valores; estas variables se supone que ya están declaradas previamente

Subrutinas

Una subrutina es un conjunto de sentencias que realizan una tarea determinada. No reciben argumentos o datos de entrada y no retorna un resultado al programa principal. En lenguajes estructurados suelen ser un caso particular de función o procedimiento.

Estructuras de datos internas estáticas: arrays

Ejemplo de introducción: En el ejercicio 1 (algoritmo que lee 100 números y cuenta cuántos de ellos son positivos) de la relación 2, ¿qué pasaría si quisiéramos imprimir los 100 números en orden inverso? ¿deberíamos emplear 100 variables numéricas? ¿o si queremos leer 50 nombres, ordenarlos y mostrarlos por pantalla? necesitamos tipos de datos más complejos: estructuras de datos.

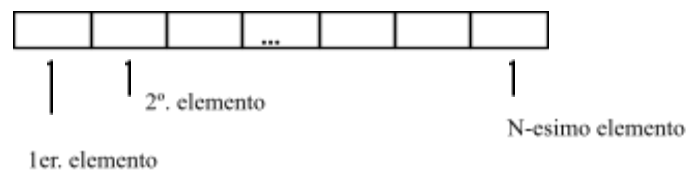
En un gran número de problemas es necesario manejar un conjunto de datos, más o menos grande, que están relacionados entre sí, de tal forma que constituyen una unidad para su tratamiento. Por ejemplo, si se quiere manipular una lista de 100 nombres de personas, es conveniente tratar este conjunto de datos de forma unitaria en lugar de utilizar 100 variables, una para cada dato simple.

Un **array** es una estructura de datos interna que consiste en un número finito de elementos, todos del mismo tipo y bajo un nombre común para todos ellos.

Como estructura de datos es un conjunto de datos homogéneos organizados secuencialmente y que se tratan como una sola unidad.

Como estructura de datos interna, se almacena en memoria central y puede resolverse de dos maneras:

- **Estática:** Tiene un número fijo de elementos que queda determinado desde la declaración de la estructura en el comienzo del programa.
- **Dinámica:** Tienen un número de elementos que varía a lo largo de la ejecución del programa.



Características

- **Nombre:** Nombre de una tabla es un identificador común para todos sus elementos, distinguiéndose cada uno de ellos por una lista de índices, que complementan a dicho nombre para referenciarlos.
- **Componentes:** Son los elementos de una tabla.
- **Dimensión:** Es el número de índices que se utilizan con una tabla.
- **Longitud o tamaño:** Es el número de elementos que contiene.
- **Tipo:** El tipo de un array es el tipo de sus componentes (numérico, alfanumérico, etc.).

La posición de cada componente dentro de la array está determinada por uno o varios **índices**, de forma que a cada componente se puede acceder (para lectura o escritura) de forma directa indicando el nombre de la tabla y sus índices. **Ejemplo:** De forma parecida a como se especifica cada posición de un tablero de ajedrez. Pero en este caso todos los índices son números enteros, que en unos casos empiezan desde 1 y en otros (como en C), empiezan desde 0.

Ejemplo.- vector Alumnos

Alumnos

ARIADNA	LUIS	PENELOPE	AITANA	ROBERTO
1	2	3	4	5

En pseudocódigo se utilizará el siguiente formato:
ALUMNOS es array[5] de tipo alfanumérico.

En C: `char *alumnos[5];`

En Java: `String [] alumnos = {"ARIADNA", "LUIS", "PENELOPE", "AITANA", "ROBERTO"};`

Para hacer referencia a un elemento de un array se utiliza:

- Nombre del array
- El índice del elemento

NOMBRE DEL VECTOR: ALUMNOS

COMPONENTES: ALUMNOS[1], ALUMNOS[2], ALUMNOS[3], ...

ÍNDICE: Los números del 1 al 5.

DIMENSIÓN: 1

LONGITUD: 5

TIPO: Alfanumérica.

En el ejemplo tendríamos una tabla de cinco elementos alfanuméricos.

Para referenciar un elemento de la tabla, podemos utilizar:

- Un valor numérico entero (3):

ALUMNOS[3]

- Una variable numérica entera (IND):

IND = 3

ALUMNOS[IND]

- Una expresión numérica entera (IND + 2)

IND + 1

ALUMNOS[IND + 2]

Ejemplo: Suponga que se tiene el siguiente array: **arrnum : ARRAY[1..10] de enteros**

Elementos:	5	23	1	8	2	13	7	11	6	5
Numero de casilla:	1	2	3	4	5	6	7	8	9	10
	arrnum[1]				arrnum[5]					

arrnum[5] contiene el valor de **2**, ya que el índice o número de casilla es el **5**.

arrnum[1] contiene el valor de **5**, ya que el índice o número de casilla es el **1**.

arrnum[7] contiene el valor de **7**, ya que el índice o número de casilla es el **7**.

Operaciones con arrays

- ❖ Lectura/Escritura
- ❖ Asignación
- ❖ Actualización
 - Inserción

- Eliminación
- Modificación
- ❖ Ordenación
- ❖ Búsqueda

A continuación se listan los programas en pseudocódigo para las tres operaciones básicas (lectura, asignación y escritura) para arrays unidimensionales:

Lectura de un array	Asignación de un array	Escritura de un array
.	.	.
.	.	.
.	.	.
PARA i DESDE 1 HASTA n	PARA i DESDE 1 HASTA n	PARA i DESDE 1 HASTA n
LEER(array1[i])	array1[i] ← 0	MOSTRAR(array1[i])
FIN DE PARA	FIN DE PARA	FIN DE PARA
.	.	.
.	.	.
.	.	.

Ejemplo1: Construya un algoritmo que dados como datos una array unidimensional de 10 enteros y un número entero, determine cuantas veces se encuentra éste número dentro del array.

PROGRAMA cuenta_numeros_en_array

arrray: ARRAY[1..10] de entero

i, cont, num: entero

INICIA PROGRAMA

cont ← 0

PARA i DESDE 1 HASTA 10

 LEER(arrray[i])

FIN DE PARA

LEER(num)

PARA i DESDE 1 HASTA 10

 SI arrray[i] ← num ENTONCES

 cont ← cont + 1

FIN DE PARA

MOSTRAR("El numero: ", num, "aparece ", cont, " veces en el array")

TERMINA PROGRAMA

Ejemplo2: Se tienen registradas en un array unidimensional, las calificaciones obtenidas de un examen para un grupo de alumnos. Cada calificación es un número entero comprendido entre 0 y 10. Escribe un algoritmo que calcule e imprima la frecuencia de cada uno de los posibles valores.

```
PROGRAMA frecuencia_de_calificaciones
  arrcal : ARRAY[1..50] de enteros
  arrfrec: ARRAY[0..10] de enteros
  i: entero
```

```
INICIA PROGRAMA
LEER( n alumnos )
PARA i DESDE 1 HASTA n alumnos
  LEER( arrcal[ i ] )
FIN DE PARA
```

```
PARA i DESDE 0 HASTA 10
  arrfrec[ i ] ← 0
FIN DE PARA
```

```
PARA i DESDE 1 HASTA n alumnos
  arrfrec[ arrcal[ i ] ] ← arrfrec[ arrcal[ i ] ] + 1
FIN DE PARA
```

```
PARA i DESDE 0 HASTA 10
  MOSTRAR( "La calificación de ", i, " fue obtenida por ", arrfrec[ i ] , " alumnos")
FIN DE PARA
TERMINA PROGRAMA
```

Ejemplo 3: Escribe un programa que dado como entrada un array unidimensional que contiene números enteros, determine cuántos de ellos son positivos, negativos o nulos.

Ejemplo 4: Escribe un programa que dado como entrada un array unidimensional que contiene el número de asaltos de todo un año en la Ciudad de México, determine cual fue la cantidad mayor y menor de asaltos.

Arrays unidimensionales (vectores)

Son tablas de una única dimensión, es decir, un único índice. La tabla ALUMNOS del ejemplo anterior sería de este tipo.

Si quisiéramos intercambiar los contenidos de las componentes primera y cuarta, lo podríamos hacer utilizando una variable alfanumérica auxiliar AUX:

```
AUX ⇌ ALUMNOS[1]
```

```
ALUMNOS[1] ⇌ ALUMNOS[4]
```

```
ALUMNOS[4] ⇌ AUX
```

Los elementos de un vector se almacenan en la memoria interna del ordenador de forma consecutiva.

Ejemplo: Un programa que lee las calificaciones de un alumno en 10 asignaturas, las almacena en una tabla(vector) y calcula e imprime su media.

Programa Nota_media

Entorno

NOTAS es tabla[10] de tipo numérico real.

MEDIA es tipo numérico real, para acumular las notas y calcular la media

IND es tipo numérico entero, contador asociado a los bucles e índice de la tabla

Algoritmo

para IND de 1 a 10 con incremento 1 hacer

 escribir "Teclee la nota nº ", IND

 leer NOTAS[IND]

finPara

MEDIA \approx 0

para IND de 1 a 10 con incremento 1 hacer

 MEDIA \approx MEDIA + NOTAS[IND]

finPara

MEDIA \approx MEDIA/10

escribir "La nota media es: ", MEDIA

FinPrograma

Arrays bidimensionales (tablas)

Estos arrays tienen dos índices, uno por dimensión, por lo cual cada componente de ella se direcciona mediante el nombre de la tabla seguido de los dos índices separados por coma y entre corchetes. Los índices siempre han de ser enteros, tanto en tipos de tablas unidimensionales o multidimensionales.

A	1	2	3		N
1					
2					
3					
				* * *	
M					

Tabla A de M filas y N columnas

M x N componentes

Componentes: A [fila , columna]

La fila y la columna será un valor, variable o expresión numérica entera.

Ejemplo: Tabla de cuatro filas y cinco columnas que contiene el número de alumnos matriculados en cada grupo de un centro docente por asignatura. Las filas corresponden a los grupos y las columnas a las asignaturas.

MATRICULADOS	1	2	3	4	5
1	35	30	25	29	33
2	32	21	33	18	7
3	37	15	28	34	19
4	18	31	9	21	22

MATRICULADOS es tabla[4,5] de tipo numérico entero

En C: `int matriculados[4][5];`

NOMBRE DE LA TABLA: MATRICULADOS

COMPONENTES: MATRICULADOS[i,j]

INDICES: Del 1 al 4 y del 1 al 5. (En C sería de 0 a 3 y de 0 a 4).

DIMENSIÓN: 2

TAMAÑO: $4 \times 5 = 20$

TIPO: Numérica entera

Ejemplo:- Programa que carga la tabla del ejemplo anterior y a continuación calcula e imprime el total de alumnos matriculados por asignatura. Suponemos que los cuatro grupos son del mismo curso y que tienen las mismas asignaturas.

Programa MATRICULACION

Entorno

MATRICULADOS es una tabla[4,5] numérica entera

SUMA, F, C son numéricas enteras

ASIGNATURAS es tabla[5] alfanumérica

Algoritmo

ASIGNATURA[1] \Leftarrow "R.A.L."

ASIGNATURA[2] \Leftarrow "S.O.M.M."

ASIGNATURA[3] \Leftarrow "F.O.L."

ASIGNATURA[4] \Leftarrow "R.L."

ASIGNATURA[5] \Leftarrow "F.P."

para F de 1 a 4 con incremento 1 hacer

para C de 1 a 5 con incremento 1 hacer

escribir "Grupo ", F, "Asignatura ", ASIGNATURA[C]

```

                leer MATRICULA[F,C]
            finPara
        finPara
    escribir "Alumnos matriculados"
    escribir " ASIGNATURA ", "NUM. ALUMNOS "
    para C e 1 a 5 con incremento 1 hacer
        SUMA  $\Leftarrow$  0
        para F de 1 a 4 con incremento 1 hacer
            SUMA  $\Leftarrow$  SUMA + MATRICULADOS[F,C]
        finPara
        escribir ASIGATURA[C], SUMA
    finPara
FinPrograma

```

Arrays multidimensionales

Son tablas de tres o más dimensiones. Por ejemplo, la de tres dimensiones se representa con un cubo.

Ejemplo: Poliedro de tres dimensiones que consta de 24 componentes agrupados de la siguiente forma:

- Primera dimensión, que indica el número de un artículo de 1 a 4 (tenemos 4 artículos diferentes).
- Segunda dimensión, que indica la talla, de 1 a 3 (tenemos tres tallas).
- Tercera dimensión, que indica el tipo de venta, de 1 a 2 (1 para venta al por mayor y 2 para venta al detalle).

Cada componente (elemento) almacena el precio para los artículos según la talla y el tipo de venta.

NOMBRE DEL ARRAY: PRECIOS

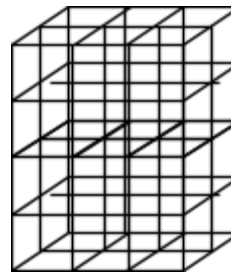
COMPONENTES: PRECIOS[i,j,k]

INDICES: De 1 a 4, de 1 a 3 y de 1 a 2.

DIMENSION: 3

TAMAÑO: $4 \times 3 \times 2 = 24$

TIPO: Numérica real.



El elemento PRECIOS[4,3,1] almacena el precio del artículo número 4, de talla 3, que se vende al por mayor.

Ejemplo:- Programa que carga el array del ejemplo anterior y posibilita sucesivas consultas de precios introduciendo como datos de entrada el número de artículo, su talla y el tipo de venta. Para terminar las consultas se introducirá un 0 en el número de artículo.

Programa CONSULTAS

Entorno:

PRECIOS es tabla [4,3,2] numérica real
ARTICULO, TALLA, VENTA son numéricas enteras

Algoritmo:

```

para ARTICULO de 1 a 4 hacer
    para TALLA de 1 a 3 hacer
        para VENTA de 1 a 2 hacer
            escribir ARTICULO, TALLA, VENTA
            leer PRECIOS[ARTICULO, TALLA, VENTA]
        finPara
    finPara
finPara
escribir "ARTICULO 1-4 (para terminar 0)"
leer ARTICULO
mientras ARTICULO <> 0 hacer
    escribir "TALLA 1-3"
    leer TALLA
    escribir "VENTA 1-2"
    leer VENTA
    escribir "PRECIOS=",PRECIOS[ARTICULO,TALLA,VENTA],"Ptas."
    escribir "ARTICULO 1-4 (para terminar 0)"
    leer ARTICULO
finMientras

```

FinPrograma

Tratamiento secuencial de un array

Consiste en la realización de algún cálculo o comparación con todos y cada uno de los elementos que forman la tabla siguiendo habitualmente el orden ascendente del o de los índices. Requiere tantos bucles PARA anidados como dimensiones tenga la tabla.

```

para IND de 1 a 10 ... // de 1 a 5 ...

    para J de 1 a 5 ... // de 1 a 10 ...

        leer (J,IND) // leer (IND,J)

    ...

```

Registros

Un registro es un dato estructurado, donde cada uno de sus componentes se denomina campo. Estos campos de un registro pueden ser todos de diferentes tipos (inclusive arrays o registros). Cada campo se identifica por un nombre único, es decir, el identificador de campo.

El formato para la declaración en un programa de pseudocódigo será el siguiente:

```
Identificador_reg : REGISTRO
    id_campo1: tipo1
    id_campo1: tipo1
    ...
    id_campon: tipon
FIN DE REGISTRO
```

Ejemplos:

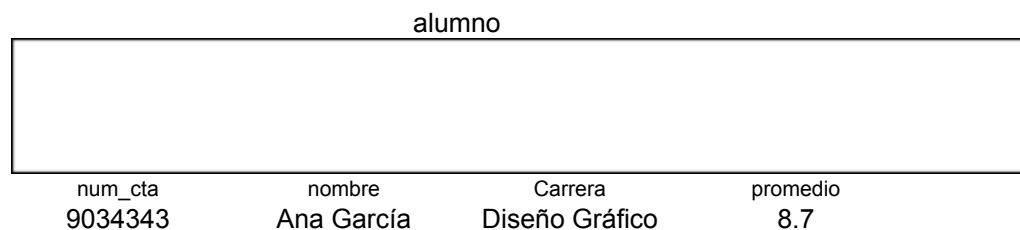
domicilio : REGISTRO	alumno : REGISTRO	empleado : REGISTRO
calle: cadena de caracteres	num_cta: entero	num_nomina: entero
colonia: cadena de caracteres	nombre: cadena de	nombre: cadena de
codpostal: entero	caracteres	caracteres
FIN DE REGISTRO	carrera: cadena de caracteres	depto: cadena de caracteres
	promedio: real	sueldo: real
	FIN DE REGISTRO	FIN DE REGISTRO

Como se podrá observar un registro posee un mayor poder de manejo de datos, ya que permite agrupar distintos tipos de datos a diferencia de los arrays, en los cuales solo se maneja un solo tipo de dato.

Para acceder a los campos de un registro se sigue el siguiente formato:

variable_registro.identificador_campo

Ejemplo: Tomando el ejemplo del registro que se definió anteriormente, para acceder al dato de la carrera que cursa el alumno Ana García la sintaxis sería la siguiente: **alumno.carrera**



alumno.carrera

Como se menciono con anterioridad, un campo de un registro puede ser de cualquier tipo, por lo tanto

puede ser un array o un registro. A su vez cada componente de un array puede ser un registro. Las combinaciones que se pueden dar entre arrays y registros es la siguiente:

- Arrays de registros
- Registros anidados
- Registros con arrays

Arrays de registros

En esta combinación, cada elemento del array es un registro. **Ejemplo:**

Una empresa registra para cada uno de sus empleados los siguientes datos: número de nómina, nombre, departamento, puesto y sueldo. Suponiendo que la empresa tiene 200 empleados, entonces se necesitará un array de 200 elementos de tipo empleado:

```
Empleado: REGISTRO
    num_nomina: entero
    nombre: cadena de caracteres
    depto: cadena de caracteres
    puesto: cadena de caracteres
    sueldo: real
FIN DE REGISTRO

arreglo: ARRAY [1..200] de empleado
```

Registros anidados

En este caso, al menos un campo del registro es de tipo registro. **Ejemplo:**

Suponga que la empresa en el ejemplo anterior desea agregar la dirección del empleado al registro, pero a su vez el dato dirección es también de tipo registro:

```
domicilio: REGISTRO
    calle: cadena de caracteres
    colonia: cadena de caracteres
    codpostal: entero
    ciudad: cadena de caracteres
FIN DE REGISTRO

Empleado: REGISTRO
    num_nomina: entero
    nombre: cadena de caracteres
    dirección: domicilio
    depto: cadena de caracteres
    puesto: cadena de caracteres
    sueldo: real
FIN DE REGISTRO
```

Registros con arrays

Los registros con arrays tienen por lo menos un campo que es un array. **Ejemplo:**

El Centro Meteorológico de México registra para cada uno de los estados de la República Mexicana los siguientes datos: Nombre del estado y lluvias mensuales del ultimo año.

Lluvias: REGISTRO

 estado: cadena de caracteres

 lluvias_mensuales: ARRAY[1..12] de enteros

FIN DE REGISTRO

Ejercicios

Fuentes y bibliografía