

ALGORITMOS Y PROGRAMAS

Resolución de problemas por computadoras

La principal razón para que las personas aprendan lenguajes de programación es utilizar un ordenador como una **herramienta para la resolución de problemas**. Tres fases pueden ser identificadas en el proceso de resolución :

- Fase de Identificación (qué nos plantean)
- Fase de resolución del problema
- Fase de implementación (realización) en un lenguaje de programación

Fase de resolución del problema

Esta fase incluye, a su vez, el análisis del problema así como el diseño y posterior verificación del algoritmo.

Análisis del problema

El primer paso para encontrar la solución a un problema es el análisis del mismo. Se debe examinar cuidadosamente el problema a fin de obtener una **idea clara sobre lo que se solicita** y determinar los datos necesarios para conseguirlo.

Diseño del algoritmo

Un algoritmo puede ser definido como la **secuencia ordenada** de pasos, sin ambigüedades, que conducen a la resolución de un problema dado y expresado en lenguaje natural, por ejemplo el castellano, Todo algoritmo debe ser:

- **Preciso:** Indicando el orden de realización de cada uno de los pasos.
- **Definido:** Si se sigue el algoritmo varias veces proporcionándole (consistente) los mismos datos, se deben obtener siempre los mismos resultados.
- **Finito:** Al seguir el algoritmo, este debe terminar en algún momento, es decir tener un número finito de pasos.

Para diseñar un algoritmo se debe comenzar por identificar las tareas más importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en esta primera descripción pueden requerir una revisión adicional antes de que podamos obtener un algoritmo claro, preciso y completo.

Este método de diseño de algoritmos en etapas, yendo de los conceptos generales a los de detalle, se conoce como método descendente (top-down).

En un algoritmo se deben de considerar tres partes:

- **Entrada:** Información dada al algoritmo.
- **Proceso:** Operaciones o cálculos necesarios para encontrar la solución del problema.
- **Salida:** Respuestas dadas por el algoritmo o resultados finales de los procesos realizados.

Como ejemplo supongamos que desea desarrollar un algoritmo que calcule la superficie de un rectángulo proporcionándole su base y altura. Lo primero que debemos hacer es plantearnos las siguientes preguntas:

Especificaciones de entrada

- ¿Que datos son de entrada?
- ¿Cuántos datos se introducirán?
- ¿Cuántos son datos de entrada válidos?

Especificaciones de salida

- ¿Cuáles son los datos de salida?
- ¿Cuántos datos de salida se producirán?
- ¿Qué formato y precisión tendrán los resultados?

El algoritmo que podemos utilizar es el siguiente:

- Paso 1. Entrada desde el teclado, de los datos de base y altura.
- Paso 2. Cálculo de la superficie, multiplicando la base por la altura.
- Paso 3. Salida por pantalla de base, altura y superficie calculada.

El lenguaje algorítmico debe ser independiente de cualquier lenguaje de programación particular, pero fácilmente traducible a cada uno de ellos. Alcanzar estos objetivos conducirá al empleo de **métodos** normalizados para la representación de algoritmos, tales como los diagrama de flujo o **pseudocódigo**.

Verificación de algoritmos

Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza las tareas para las que se ha diseñado y produce el resultado correcto y esperado. El modo más normal de comprobar un algoritmo es mediante su ejecución manual, usando datos significativos que abarquen todo el posible rango de valores y anotando en una hoja de papel las modificaciones que se den estos datos y en los del resto del algoritmo, en las diferentes fases hasta la obtención de los resultados. Este proceso se conoce como **prueba del algoritmo**.

Fase de implementación

Una vez que el algoritmo está diseñado, representado mediante pseudocódigo y verificado se debe pasar a la fase de codificación o traducción del algoritmo a un determinado lenguaje de programación, que deberá ser completada con la **ejecución y comprobación** del programa en el ordenador.

CODIFICACION DE ALGORITMOS EN PSEUDOCODIGO

[Nota: Aunque las normas de programación en pseudocódigo están cuasi-estandarizadas, en este curso las enfocaremos lo máximo posible a la nomenclatura que se emplea en lenguaje C para que en el momento de programar en esta lenguaje la adaptación sea lo mas rápida posible]

En el ejemplo del calculo del área de un rectángulo podemos observar que en la resolución de programas nos encontramos con datos que pueden ser números como por ejemplo la base o la altura y otros que pueden ser los mensajes que aparecen por pantalla (“ La superficie es”).

Quiere esto decir que previamente necesitamos conocer **qué TIPOS de datos** puede manejar un ordenador en un programa.

¿Porqué es importante este apartado?. Podríamos pensar en lo siguiente:

¿Qué tipo de numero es el 7? → Entero

¿Qué tipo de numero es el 3? → Entero

¿Qué tipo de numero es 7/3 ? → ???

Tipos de Datos básicos

- **entero:** Subconjunto finito de los números enteros, cuyo rango dependerá del lenguaje en el que posteriormente codifiquemos el algoritmo y del ordenador. El rango depende de cuantos bits utilice para codificar el numero,

normalmente **2 bytes**, Para números **positivos**, con 16 bits se pueden almacenar

$2^{16} = 65536$ números enteros diferentes: de 0 al 65535, y de

-32768 al 32767 para números con signo.

Por ejemplo 2, 14, -199,....

Operaciones asociadas al tipo entero:

Como norma general las operaciones asociadas a un tipo cualquiera serán aquellas cuyo **resultado** sea un **elemento del mismo tipo**, por tanto:

+, -, *, división → div (cociente), modulo → mod (resto),
sucesor, predecesor, es_par *

(*) es_par(n) → si n es par devuelve un 0 sino, un 1.

- **real:** Subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión. Suelen ocupar 4, 6 ó 10 bytes. Se representan por medio de la **mantisa**, y un **exponente** ($1E-3 = 0'001$), utilizando **24 bits** para la mantisa (1 para el signo y 23 para el valor) y **8 bits** para el exponente (1 para el signo y 7 para el valor). El orden es de 10^{-39} hasta 10^{38} .
Por ejemplo 6.9, 33.00123, 3E-34.....
Las operaciones asociadas serían +, -, *, ÷, etc.
¿Y las operaciones de sucesor y predecesor?
- **Lógico:** Conjunto formado por los valores Cierto y Falso. '1' y '0'.
Operaciones: todas las lógicas y relacionales → AND, OR, >, <, == (igual), >=, <=, ≠ ...

- **Carácter:** Conjunto finito y ordenado de los caracteres que el ordenador

reconoce. se almacenan en un **byte**. Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).

Un carácter (*letra*) se guarda en un solo byte como un número entero, el correspondiente en el **código ASCII**, que se muestra en la Tabla para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores:

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Sabiendo la fila y la columna en la que está un determinado carácter puede componerse el número correspondiente. Por ejemplo, la letra A está en la fila 6 y la columna 5. Su número ASCII es por tanto el 65. El carácter % está en la fila 3 y la columna 7, por lo que su representación ASCII será el 37.

Los caracteres se representan entre comillas simples 'a', 'D'... y abarcan desde '0'..'9', 'a'..'z' y 'A'..'Z'. Un espacio en blanco (ð) también es un carácter. Por tratarse de un tipo de datos **enumerados** y con una

representación como la de un entero podemos aplicar, además de las operaciones de los tipos enteros, las siguientes:

$\text{ord}(\text{carácter}) \rightarrow \text{numero}$ $\text{car}(\text{numero}) \rightarrow \text{carácter}$

Así $\text{ord}('A') \rightarrow 65$ y $\text{car}(37) \rightarrow '\%$ ', $'A'+3='D'$!!!

- **Cadena:** Los datos de este tipo contendrán una serie finita de caracteres.

Podrán representarse de estas dos formas:

$'H' 'O' 'L' 'A' == \text{"HOLA"}$,

No es lo mismo "H" que 'H', ya que el primero es una

cadena y el segundo un carácter. Las operaciones

relacionadas con las cadenas son: Concatenar(C_1, C_2),

Longitud(C_1),

Extraer (C_1, N), **ejemplos:**

$C_1 = \text{"Hola"}$, $C_2 = \text{"Pepeeerl"}$

$C_3 = \text{Concatenar}(C_1, C_2) \rightarrow C_3 = \text{"HolaPepeeerl"}$

$X = \text{Longitud}(C_2) \rightarrow X = 8$

$C_3 = \text{Extraer}(C_1, 3) \rightarrow C_3 = \text{"Hol"}$

Entero, Real, Carácter, Cadena y Lógico son tipos predefinidos en la mayoría de los lenguajes de programación.

Además el usuario podrá definir sus propios tipos de datos, si estos facilitan de alguna manera la resolución del problema.

Tipos de datos definidos por el usuario

- **Enumerado o**

Escalar : Se hace una enumeración de los valores que va a tomar el tipo. Para ello se antepone la palabra **tipo:**

Ejemplo:

tipo

moneda=(peseta,libra,franco,marco)

dia=(L,M,X,J,V,S,D)

Cada elemento tiene un número de orden asociado, y comenzando desde el cero.

Estos tipos no se pueden leer por teclado.

- **Subrango:** Son un subconjunto finito de un tipo definido por el usuario y, en general, de cualquier tipo que tenga sucesor y predecesor, como los enteros y carácter.

Ejemplo:

tipo

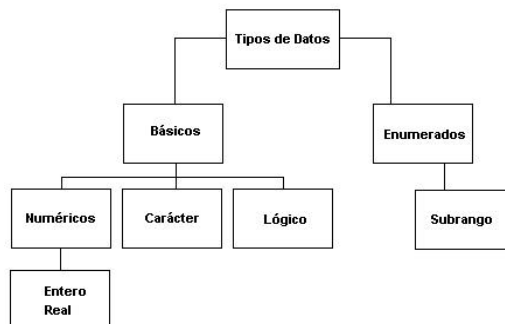
flojas=peseta..franco

laboral=L..V

minusculas='a'..'z'

aceptado=1000..1100

Muy esquemáticamente :



Dentro de un programa en pseudocódigo tendremos ocasiones en que los valores que toman los datos podrán ser de tres tipos:

- Un valor propiamente dicho p.ej. 45, 3.4, 'a', etc. ('constante')
- Una Constante p.ej. PI=3.141592
- Una Variable p.ej x,y,z,t.

Constantes

Son datos cuyo valor no cambia durante todo el desarrollo del programa. Las constantes podrán ser **literales** o con **nombres**. Las constantes literales serán un valor propiamente dicho y las que tienen nombre se identifican por su **nombre y el valor asignado**. Tendremos pues constantes:

- Numéricas enteras: 12,34,-44,22234, -9837,m=12,n=44592.....
- Numéricas reales: 12.55, -3E3, PI=3.14159.....
- Lógicas: Sólo existen dos constantes lógicas → 0,1
- Carácter: n='a', k='1',.....
- Cadena: saludo="hola", s="sí", n="no".....

Variables

Una variable es un **dato representado por una posición determinada de memoria** cuyo valor puede cambiar durante el desarrollo del algoritmo.

Se identifica por su **nombre y por su tipo**, que podrá ser cualquiera, y es el que determina el conjunto de valores que podrá tomar la variable. En los programas la declaración de cada una de las variables origina que se reserve un determinado espacio en memoria etiquetado con el correspondiente identificador. El nombre asignado a la variable se denomina **identificador** y tiene las siguientes reglas:

1. Un **identificador** se forma con una secuencia de **letras** (minúsculas de la **a** a la **z**; mayúsculas de la **A** a la **Z**; y **dígitos** del **0** al **9**).
2. El carácter **subrayado** se considera como una letra más.
3. Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (*,;,:-+, etc.).
4. **El primer carácter de un identificador debe ser siempre una letra** o un (**_**), es decir, no puede ser un dígito.
5. Se hace distinción entre letras mayúsculas y minúsculas. Así, **Masa** es considerado como un identificador distinto de **masa** y de **MASA**.
6. ANSI C permite definir identificadores de hasta 31 caracteres de longitud.

Ejemplos de identificadores válidos son los siguientes:

tiempo, distancia1, caso_A, PI, velocidad_de_la_luz,x,mm,media

Por el contrario, los siguientes nombres no son válidos:

1_valor, tiempo-total, dolares\$, %final →¿Por qué?

En general es muy aconsejable **elegir los nombres** de las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas.

Para **declarar** una variables utilizaremos el siguiente convenio:

tipo variable1, <variable2>, ... ;

Ejemplo: entero velocidad; real media, sumatotal; carácter x , rpta;

Operadores, Expresiones y Sentencias

Operadores

Un **operador** es un carácter o grupo de caracteres que actúa sobre una, dos o más variables para realizar una determinada **operación** con un determinado **resultado**. Ejemplos típicos de operadores son la suma (+), la diferencia (-), el producto (*), etc. Los operadores pueden ser **unarios**, **binarios** y **ternarios**, según actúen sobre uno, dos o tres operandos, respectivamente.

OPERADORES ARITMÉTICOS

Los **operadores aritméticos** son los más sencillos de entender y de utilizar. Todos ellos son operadores binarios. Se utilizan los cinco operadores siguientes:

Suma:	+
Resta:	-
Multiplicación:	*
División:	/
Resto:	% (resto de la división entera . Este operador se aplica solamente a constantes, variables o expresiones de tipo entero).*

(*) $23\%4$ es 3, puesto que el resto de dividir 23 por 4 es 3.
Si $a\%b$ es cero, a es múltiplo de b .

Una **expresión** es un conjunto de variables y constantes relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es el siguiente: $3*x - x*x/2$

Las expresiones pueden contener **paréntesis** (...) que agrupan a algunos de sus términos. (en ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones)

OPERADORES DE ASIGNACIÓN

Los **operadores de asignación** atribuyen a una variable –es decir, depositan en su zona de memoria correspondiente– el resultado de una expresión o el valor.

variable (en realidad, una variable es un caso particular de una expresión).

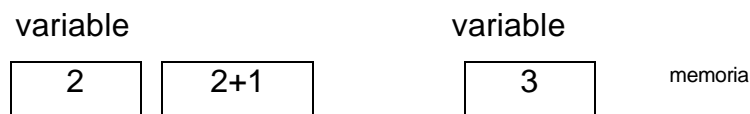
El operador de asignación más utilizado es el **operador de igualdad (=)**, que no debe ser confundido con la igualdad lógica (==). Su forma general es:

nombre_variable = expresión;

Primero se evalúa **expresión** y el resultado se pone en **nombre_variable**, sustituyendo cualquier otro valor que hubiera en esa posición de memoria anteriormente. Una posible utilización de este operador es la siguiente:

variable = variable + 1;

Desde el punto de vista matemático este ejemplo no tiene sentido pues Equivale a $0 = 1!$, pero sí lo tiene considerando que en realidad **el operador de asignación (=) representa una sustitución**, pues se toma el valor de la **variable** contenido en la memoria(2),



le suma 1 y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador **variable**, sustituyendo al valor que había anteriormente. Esta operación se denomina **acumulación**

OPERADORES RELACIONALES

Una característica imprescindible de cualquier lenguaje de programación es la de **considerar alternativas**, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los **operadores relacionales** permiten estudiar si se cumplen o no esas condiciones.

En un programa si una condición se cumple, el resultado es **cierto**; en caso contrario, el resultado es **falso**. Un 0 representa la condición de **falso**, y cualquier número distinto de 0 equivale a la condición **cierto**. Los **operadores relacionales**

son los siguientes:

- Igual que: ==
- Menor que: <
- Mayor que: >
- Menor o igual que: <=
- Mayor o igual que: >=
- Distinto que: !=

Todos los **operadores relacionales** son operadores **binarios** (tienen dos operandos),

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1) // resultado=0 porque la condición no se cumple
(3<=3) // resultado=1 porque la condición se cumple
(3<3) // resultado=0 porque la condición no se cumple
(1!=1) // resultado=0 porque la condición no se cumple
```

OPERADORES LÓGICOS

Los operadores lógicos son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen las condiciones necesarias. Como **operadores lógicos** tenemos: el operador Y (&&), el operador O (||) y el operador NO (!). En inglés son los operadores AND, OR y NOT. Su forma general es la siguiente:

```
expresion1 && expresion2, expresion1 || expresión, !expresión
```

Cuando las **operaciones lógicas son bit a bit**, típicas para hacer tests de uno o varios bits (o máscaras) disponemos de los siguientes:

- ~ inversión de cada bit (Complemento a 1)
- & And lógica bit a bit
- | Or lógico bit a bit
- ^ Xor
- >> Desplazamiento a la derecha
- << Desplazamiento a la Izquierda

Los operadores && y || se pueden combinar entre sí paréntesis. Por ejemplo:

```
(2==1) || (-1==-1) // el resultado es 1
(2==2) && (3==-1) // el resultado es 0
((2==2) && (3==3)) || (4==0) // el resultado es 1
((6==6) || (8==0)) && ((5==5) && (3==2)) // el resultado es 0
((250)&&(330)) 250>0,330>0 // el resultado es 1
```

FUNCIONES

En los lenguajes de programación existen ciertas funciones predefinidas o internas que aceptan unos argumentos y producen un valor denominado resultado. Como funciones numéricas, normalmente se usarán :

<i>Función</i>	<i>Descripción</i>
<i>abs(x)</i>	Valor Absoluto
<i>cos(x),sin(x)</i>	Coseno, Seno
<i>cuadrado(x)</i>	x^2
<i>ent(x)</i>	Parte entera
<i>exp(x)</i>	e^x
<i>ln(x)</i>	Ln(x)
<i>log(x)</i>	Log ₁₀ (x)
<i>raiz(x)</i>	Raiz Cuadrada)
<i>redondeo (x)</i>	Redondear numero

Las funciones se utilizarán escribiendo su nombre, seguido de los argumentos adecuados encerrados entre paréntesis, en una expresión.

Expresiones

Ya han aparecido algunos ejemplos de expresiones en las secciones anteriores. Una expresión es una combinación de variables y/o constantes, y operadores. Por ejemplo, 1+5 es una expresión formada por dos operandos (1 y 5) y un operador (el +); esta expresión es equivalente al valor 6, lo cual quiere decir que esta expresión, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos.

Por ejemplo, la solución de la ecuación de segundo grado: se escribe, en la siguiente forma:

$$x = \frac{-b \pm \sqrt{b^2 \cdot 4ac}}{2a} \quad x = (-b + \text{raiz}((b*b) - (4*a*c)))/(2*a);$$

donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

$$x = (-b + \text{sqrt}((b * b) - (4 * a * c)))/(2 * a)$$

Sentencias

Las expresiones son ítems elementales de una entidades que son las sentencias. Si nos fijamos en la expresión anterior, la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una sentencia.

Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

SENTENCIA O ACCION SIMPLE

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;) de finalización. Un caso típico son las declaraciones de variables o las sentencias aritméticas. Por ejemplo, declaradas ya las variables:

```
espacio = espacio_inicial + velocidad * tiempo;
```

SENTENCIAS o ACCION COMPUESTA

Muchas veces es necesario poner realizar un **grupo de acciones** determinado un lugar del programa , de acuerdo a una condición. Esto se realiza por medio de sentencias compuestas. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves {...}. También se conocen con el nombre de bloque. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas. Un ejemplo de sentencia compuesta es el siguiente:

```
{   velocidad=espacio/tiempo ;  
    velocidad=velocidad+media_velocidad ;  
    masa = 9.5 ;  
    m=masa*velocidad_media ;  
}
```

Instrucciones para Entrada y Salida de datos

Salida

La instrucción es **Escribir()** o **Escribe()**. Presentan por el dispositivo de salida estándar (pantalla), las expresiones, valores, variables o cadenas contenidas en la función, por ejemplo:

```
Escribe ("Introducir el valor de la base");
```

```
Escribe ("El resultado es");
```

```
Escribe (x);
```

Una sintaxis especial se requiere si deseamos presentar conjuntamente cadenas y variables:

- Escribimos la cadena
- Separamos la variable con una coma , .
- Si después va otra cadena entonces ponemos otra coma y escribimos la cadena.

Ejemplo:

Escribe ("la base ",b,"por la altura ",a,"es la superficie ",s);

Entrada

La instrucción es **Leer()**. Toman los valores de forma interactiva desde el dispositivo de entrada estándar (teclado), introduciéndolos en las variables que forman parte de la instrucción. Los tipos de datos introducidos deben coincidir con los de las variables que los recogen. Hay dos formas de usarlas:

```
....  
....  
x=leer() ;  
  
leer(x) ;
```

Cuando queremos recoger mas de un dato podemos utilizar esta forma:

Leer (x,y) → Se respetará el orden de recogida con cual si se introduce desde el teclado 12,34, la variable x=12 y la y=34.

El primer programa

Normas: (Adaptadas un poco al lenguaje C)

- Todos los programas comienzan por la palabra **Inicio** seguido de un grupo de sentencias.
- Por tanto debemos recordar que un grupo de sentencias va entre llaves {...}. Una sentencia simple aislada no necesita llaves.
- Cada sentencia termina por un **;** ; (**Separador de sentencias**)
- Como norma se declaran todas las variables al inicio del programa.
- Un grupo de sentencias relacionado guardan la misma vertical.
- Si necesitamos un nuevo grupo de sentencias se tabula a la derecha.
- Lápiz, papel cuadriculado y letra clara, grande y limpia!!!.
- En comentario comienza por **// si es de 1 línea** y por **/**/** si abarca a un párrafo.

Supongamos que queremos hacer un programa que nos calcule el área de un triángulo. Primero debemos pensar que $A=(b*h)/2$. Por lo tanto necesito pedir dos datos y presentar un resultado. Una posible solución usando una secuencia de sentencias y expresiones sería:

Inicio

```
{
    entero a,b,h;           // Declaro las variables
    Escribe ("Introducir el valor de la Base y la Altura :"); // mensaje en
                                                                    pantalla
    Leer(b);                // Leemos la base
    Leer(h);                // Leemos la altura
    a=(b*h)/2;             // calculamos ...
    Escribe ("la superficie es : ",a); // Presentamos el resultado
}
```

Problemas de pseudocódigo: Declaración de Variables y Programas secuenciales

1) Decir si son correctos o no los siguientes identificadores de variables:

- | | | |
|----------------|--------------|---------------|
| - contador | - temporal1 | - num_bytes |
| - 2temporal | - N_numeros | - Media_Aritm |
| - Valor\$curso | - N-terminos | - MeDiaGeom |

2) Escribir en pseudocódigo las siguientes expresiones matemáticas:

$$\frac{(x+5)}{y-2} \quad \frac{\text{Sen}(x) + \text{Cos}(y)}{\sqrt{2-y}} \quad (x-3)^2 + 2(z-5) + 3y^3 - 7$$

3) Dada la siguiente declaración de variables, comprobar el siguiente pseudocódigo indicando los posibles reales y los de 'alta probabilidad de error':

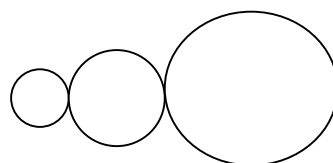
x: entero;
y: carácter;
z,t: real;
m, n, p: entero;
c, s: cadena

```

x=7;           p=z/10.0;       t=m+1;
m=x+1;        y='c'+1;
y=x;          s='Hola'
n=x*x;        c= Concat("Hola", 'Pepe');
z=x-m;        m=Long("Sabado");
    
```

4) Realizar en pseudocódigo un programa que calcule de forma individual la velocidad de 5 cuerpos, introduciendo por teclado el espacio y el tiempo, hallando posteriormente la media de cada una de las tres magnitudes.

5) Realizar en pseudocódigo un programa que calcule la longitud y el área total de tres circunferencias sabiendo que la 1ª de ellas tiene radio R que será introducido por teclado, la 2ª tiene radio 2R y la 3ª tiene radio 3R



Programación en Pseudocódigo: Estructuras de programas

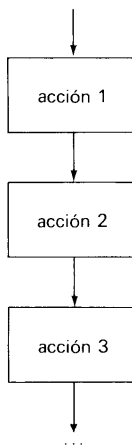
Cualquier programa puede ser realizado con una combinación de las siguientes estructuras de sentencias que definimos a continuación:

- **Secuencial**
- **Selectiva**
- **Repetitiva**

Estructura secuencial

En esta estructura cada acción se ejecuta en el orden preestablecido por como son enumeradas a lo largo del programa. Se ejecutan de forma secuencial (una detrás de otra) y no puede verse alterado el orden de ejecución.

Gráficamente:



En primer lugar se ejecutaría la acción1, luego la acción2, y así sucesivamente. Por ejemplo:

```

x=5;           // asignamos a x el valor 5
y=7;           // lo mismo con y
z=x+y;         // asignamos a z el valor x+y
x=x+3          // incrementamos x en 3
.....
.....
    
```

Estructuras de control selectivas

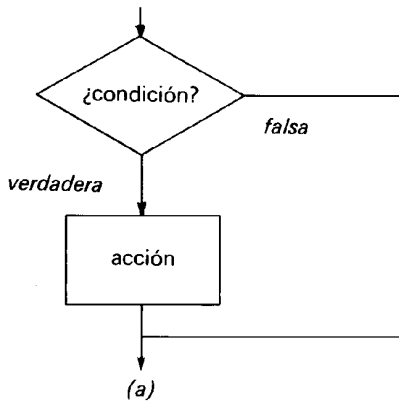
En ciertos programas la evolución natural del mismo durante su ejecución, puede necesitar unas variaciones de acuerdo con el cumplimiento o no de alguna-as condiciones. Mediante las estructuras selectivas podemos tomar decisiones, en las cuales se **evalúa una condición** y en función del resultado **se ejecutará** o no una acción o acciones.

Las distintas estructuras de las que disponemos son:

Selección Incompleta (simple)

La estructura alternativa simple **si-entonces** ejecuta una determinada acción cuando se cumple una determinada condición. La selección **sí-entonces** evalúa la condición y -si la condición es verdadera, entonces ejecuta la acción (o acciones) y si la condición es falsa, entonces no hacer nada y el programa continua con la siguiente instrucción.

Las representación gráfica de la estructura condicional simple se muestran en la siguiente figura:



Pseudocódigo en español

si (condición) **entonces** < acción1 >

Pseudocódigo en español (varias acciones)

```

si (condición) entonces {<acción1 >
                           <acción2>
                           .....
                           .....
                           }
  
```

Por ejemplo: Aquí contemplamos sólo una acción en el caso en que se cumpla la condición: (solo hallamos la raíz)

```

....
leer (x); // Obtenemos x desde el teclado
si (x>=0) entonces y= raíz(x); // si x es positiva hallamos la raíz
x=x*x; // hacemos x2
Escribe (x); // Escribimos el valor de x
....
  
```

En este vemos como se representa **un grupo de acciones**:

```

....
leer (x);
si (x>=0) entonces { y= raíz(x);
                        Escribe (x,y);
                        }
x=x*x*x;
Escribe (raíz (x));
....
    
```

Estructura de selección completa

Se emplea cuando queremos matizar qué acción (a partir de ahora en el lugar donde se cite acción se sobreentiende que también puede ser acciones) se realizará cuando **sí** se cumple la condición y cual se hará cuando **no** se cumpla. Su pseudocódigo es el siguiente:

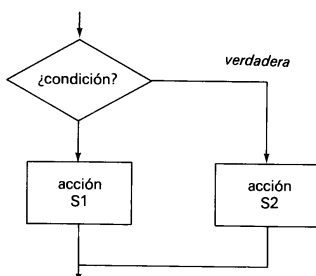
Pseudocódigo en español

```

si (condición) entonces <acción1 >
sino <acción2>
    
```

(Ver la tabulación a →)

Gráficamente:



Pseudocódigo en español

(con acción compuesta)

```

si (condición) entonces
{ accion1;
  accion2;
  ...
}
sino { acción3;
      .....
      }
    
```

Ejemplo:

```

leer (x);
si (x>=0) entonces { Escribe (x);
                        y= raíz(x);
                        }
sino Escribe ("No podemos con x<0)
    
```

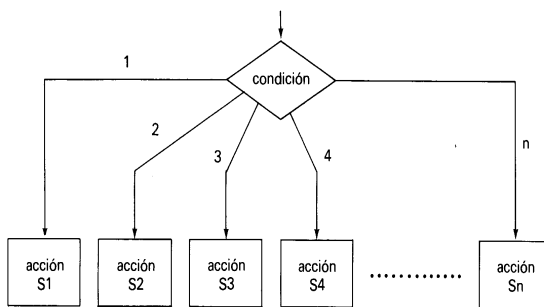
Selección múltiple (según (valor))

A veces es necesario que existan más de dos elecciones posibles Este problema se podría resolver por **estructuras selectivas simples o completas** que **estuvieran anidadas o en cascada**; sin embargo por este método si el número de alternativas es grande puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple **evaluará** una expresión que podrá **tomar n valores distintos** (siempre un valor enumerado). Según que elija uno de estos valores en la condición, se realizará **una de las n acciones**.

La representación gráfica de la estructura de decisión múltiple es:

Diagrama de flujo



En pseudocódigo:

según (valor)

V1: accion1

V2: accion2

V3: accion3

V4,V5: accion4

... ..

Vn: accionn

Ejemplo:

....

Escribe (“Elegir la opción del menú: “);

Leer (opción)

según (opción)

```
1:  { Leer(x);
      Raíz(x);
    }
```

```
2:  { Leer (x);
      x=1/x;
    }
```

ESTRUCTURAS DE CONTROL REPETITIVAS

Algunas veces nos podremos encontrar ciertas tareas dentro de un programa que deben repetirse un **numero determinado o indeterminado** de veces. Este es un tipo muy importante de estructura, ya que, por un lado nos permite ahorrar muchas líneas de programa y en otros casos no sería posible resolverlo.

Las estructuras que repiten una secuencia de instrucciones un número **determinado** de veces se denominan **bucles** y se denomina **iteración** al hecho de repetir la ejecución de una secuencia de acciones.

Por ejemplo: supongamos que se desea sumar una lista de 25 números escritos desde teclado -por ejemplo, calificaciones de los alumnos de una clase. Hasta ahora lo que haríamos es leer los números en una variable y añadir sus valores a una variable SUMA que contenga las sucesivas sumas parciales. La variable SUMA se hace igual a cero ya continuación se incrementa en el valor del número cada vez que uno de ellos se lea. En pseudocódigo:

```

Inicio
{   entero numero,suma;
    suma=0;
    Leer (numero);
    suma=suma+numero;
    Leer (numero);
    suma=suma+numero;
    ....
    ....
}

```

y así sucesivamente para cada número de la lista (los 25 alumnos!!!) . En otras palabras, el algoritmo repite muchas veces las acciones.

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuántos números se han de sumar. Para ello necesitaremos conocer

algún medio para finalizar el bucle. En el ejemplo anterior se repetiría **desde** 1 **hasta** 25 la acción de pedir la nota.

Otra variante del bucle la podemos pensar con el siguiente ejercicio; Un profesor quiere meter las notas pero debe tener la posibilidad de parar de meterlas cuando el quiera, por ejemplo cuando escriba el **-1**. Con lo que conocemos de programación hasta ahora ¿Cómo se plantearía el programa?, ¿Cuántas líneas tipo Leer(x) habría que poner? (¿Comooorl?);

Necesitaríamos ‘algo’ que permitiera **repetir** el leer datos **hasta** que escribiera el **-1**, o bien que **mientras** lo que leemos sea distinto de **-1**, leamos notas. Pues esas son las **estructuras de repetición**.

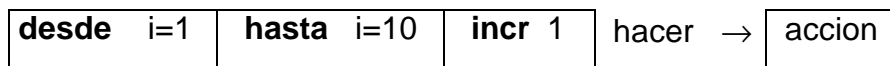
Estructura desde...hasta

Una variable contendrá los valores el numero de veces que queramos realizar una secuencia de código, por lo tanto debemos **conocer obligatoriamente** el numero de veces (iteraciones) que se realizaran.

En pseudocódigo:

i=índice

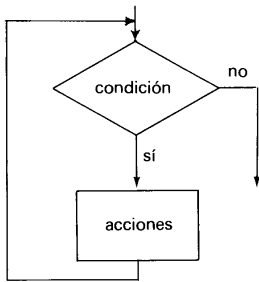
desde i=valor_inicial **hasta** i=valor_final **incremento** valor <acción>;
si es 1 se omite



Estructura mientras

Es aquella en la que la acción o grupo de acciones se repetirá **mientras** se cumpla la condición y en el momento en que no se cumpla se saldrá del bucle:

Gráficamente:



En pseudocódigo:

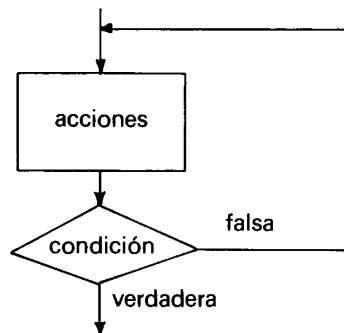
```

mientras (condición)
    { acciones;
      ....
      ....
    }
    
```

Estructura repetir

Es aquella en la que la acción o grupo de acciones se repetirá **hasta** que se cumpla la condición y en el momento en que se cumpla se saldrá del bucle:

Gráficamente:



En pseudocódigo:

```

repetir
    < acciones >
hasta (condición)
    
```

Con una estructura **repetir**, el cuerpo del bucle se ejecuta siempre al menos una vez. Cuando una instrucción **repetir** se ejecuta, la primera cosa que sucede es la ejecución del bucle ya continuación se evalúa la expresión booleana resultante de la condición. Si se evalúa como falsa, el cuerpo del bucle se repite y la expresión booleana se evalúa una vez. Después de cada iteración del cuerpo del bucle, la expresión booleana se evalúa; si es *verdadera*, el bucle termina y el programa sigue en la siguiente instrucción a **hasta**.

*Diferencias de las estructuras **mientras** y **repetir***

- La estructura **mientras** termina cuando la condición es falsa, mientras que **repetir** termina cuando la condición es verdadera.
 - En la estructura **repetir** el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, **mientras** es más general y permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura **mientras** se debe estar seguro de que se cumplirá la condición de entrada.

Otros tipos de datos

Vectores, matrices y cadenas de caracteres

Las **matrices o vectores** surgen en informática por la necesidad de manejar un **número determinado** de datos **del mismo tipo**. La idea esta tomada del concepto matemático de matriz.

Por tanto una matriz es **una colección de datos del mismo tipo**, y que además están alojados consecutivamente en la memoria. A los elementos de la matriz se puede acceder de forma individual mediante un **índice**, y cada elemento de la matriz ocupa en memoria lo mismo que si estuviera almacenado en una variable.

Las matrices son estructuras **estáticas** de datos, lo que significa que al declarar una matriz se **reserva la memoria** que necesita, independientemente del numero de valores que haya en la matriz.

Cuando la matriz es de **una dimensión** se denomina **vector**, y a partir de 2 dimensiones se llama **matriz**.

La forma de declarar una matriz es:

1 dimensión:

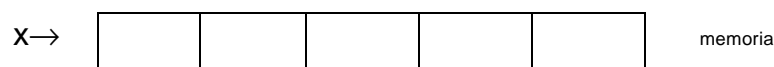
tipo nombre [nº _elementos]

2 o mas:

tipo nombre [nº _elementos] [nº _elementos]

Así por ejemplo un vector de 5 elementos de tipo entero se declara y representa:

entero x [5]



Una matriz de 2 x 5 sería:

entero x [2] [5] // filas y columnas

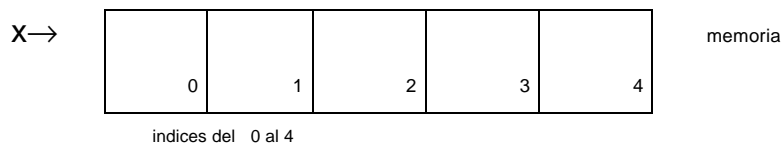
Los elementos se numeran y referencian con un índice que va desde **0 hasta (numero_elementos-1)**. El tamaño de un vector puede definirse con cualquier expresión constante entera. Para definir tamaños son particularmente útiles las constantes simbólicas.

Para referenciar los elementos de un vector se utiliza la siguiente notación:

nombre_matriz [nº_elemento]

Así, si declaramos el vector entero x [5], el primer elemento será el x[0], el segundo el x[1] y el ultimo el x[4]. Si declaramos entero m [2][5] el primer elemento es el m[0][0], el segundo el m[0][1] y el ultimo m[1][4].

Gráficamente:



0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4

No se puede operar con todo un vector o toda una matriz como una única entidad, sino que hay que tratar sus elementos individualmente por medio de bucles o los elementos propiamente dichos.

La forma de introducir o leer el contenido de los elementos en la matriz es similar a la asignación o lectura del valor de una variable, pero esta variable es una posición de la matriz: Ejemplos

```
x[4] = 0.8;
x[3] = 30. * x[0];
x[0] = 3. * x[1] - x[1]/x[5];
x[3] = (x[0] + x[2])/x[3];
```

INICIALIZACIÓN DE VECTORES Y MATRICES

La inicialización de un vector se puede hacer de varias maneras:

– Declarando el vector como tal e inicializándolo luego mediante lectura o asignación por medio de un bucle **desde... hasta**:

```
n=12
real vect[n];
....
desde i=0 hasta i=n-1 incr 1 Leer (vect[i] );
....
```

– Inicializándolo en la misma declaración, en la forma:

```
real v[6]      = {1., 2., 3., 3., 2., 1.};
real d[ ]     = {1.2, 3.4, 5.1}; // equivale a real d[3]  ®   está implícito
                                                         por el numero de
                                                         valores

entero f[100] = {0}; // todo se inicializa a 0
entero h[10]  = {1, 2, 3}; // restantes elementos a 0
entero mat[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

Una cadena de caracteres no es sino un vector de tipo carácter, con alguna particularidad que conviene resaltar. Las cadenas suelen contener texto (nombres, frases, etc.), y éste se almacena en la parte inicial de la cadena (a partir de la posición cero del vector). Para separarla parte que contiene texto de la parte no utilizada, se utiliza un carácter fin de texto que es el carácter nulo ('\0') según el código ASCII. Este carácter se introduce automáticamente al leer o inicializar las cadenas de caracteres, como en el siguiente ejemplo:

```
carácter ciudad[20] = "Albacete";
```

donde a los 8 caracteres del nombre de esta ciudad se añade un decimocuarto: el '\0'. El resto del espacio reservado –hasta la posición ciudad[19]– no se utiliza. De modo análogo, una cadena tal como "marco" ocupa 6 bytes (para las 5 letras y 1 para el '\0').

Es de destacar el estudio de los vectores de 2 dimensiones :las matrices. Se declaran de forma análoga, con corchetes independientes para cada subíndice.

La forma general de la declaración es:

```
tipo nombre[numero_filas][numero_columnas];
```

donde tanto las filas como las columnas se numeran también a partir de 0.

Las matrices (m×n), en la memoria se almacenan por filas, en posiciones consecutivas de memoria. Si una matriz tiene N filas (numeradas de 0 a N-1) y M columnas (numeradas de 0 a la M-1), el elemento (i, j) ocupa el lugar:

$$\text{posición_elemento}(0, 0) + i * M + j$$

primer elemento

A esta fórmula se le llama fórmula de direccionamiento de la matriz.

Estructuras o Registros

Uno de los inconvenientes que tiene una matriz es que por definición todos los datos son del mismo tipo, es decir, no podemos mezclar un real con un carácter. Una **estructura** es una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador.

Su forma general es mediante la palabra **registro o estructura** :

```

estructura nombre_estr {
    tipo nombre1;
    tipo nombre2;
    .....
    .....
};

```

De momento solo hemos declarado un **tipo de datos**. Para declarar una **variable de tipo estructura o registro** se emplea la siguiente notación:

```

estructura nombre_estr <nombre_variable>

```

Por ejemplo, supóngase que se desea diseñar una **estructura** que guarde los **datos** correspondientes a un alumno, esta estructura, a la que se llamará **alumno**, deberá guardar el **nombre**, la **dirección**, el **número** de matrícula, el teléfono, y las notas en las 10 asignaturas.

Cada uno de estos datos se denomina **campo o miembro** de la estructura. El modelo de esta estructura puede crearse del siguiente modo:

```

estructura alumno {
    carácter nombre[31];           // cadena de caracteres
    carácter direccion[21];
    entero no_matricula;
    entero telefono;
    real notas[10];
};

```

El pseudocódigo anterior crea el **tipo de dato alumno**, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Recordar que es un separador de sentencias. Para declarar una **variable de tipo alumno** se debe utilizar la palabra **estructura** y el nombre que le hemos dado (alumno) :

```
estructura alumno alumno1, alumno2;
```

También podrían haberse definido alumno1 y alumno2 al mismo tiempo que se definía la estructura de tipo alumno. Para ello bastaría haber hecho:

```
estructura alumno {  
    carácter nombre[31];           // cadena de caracteres  
    carácter dirección[21];  
    entero no_matricula;  
    entero telefono;  
    real notas[10];  
} alumno1,alumno2 ;
```

Para acceder a los miembros de una estructura se utiliza el operador punto (.), precedido por el nombre de la variable y seguido del nombre del campo :

variable . campo

Por ejemplo, para **dar valor al telefono del alumno alumno1** el valor 903456, se escribirá:

```
alumno1.telefono = 903456;
```

y para guardar la dirección de este mismo alumno, se escribirá:

```
alumno1.direccion = "C/ Rios Rosas 1,2-A";
```

De esta forma podemos tanto leer el contenido de un campo del registro, como escribirlo:

```
x= alumno1.telefono
calculo=9.7;
alumno1 .notas[3]=calificacion;
```

De todas formas no es efectivo ni muy útil el uso de 1 solo registro. Lo normal es emplear varios, y la forma más inmediata es mediante el empleo de una matriz, con lo cual habrá que declarar **una matriz de registros**, lo cual no debe preocuparnos pues el nombre de la estructura es considerado como un **TIPO** :

Por ejemplo:

```
estructura alumno alumno1 , clase[300];
```

En este caso, alumno1 es una estructura de tipo alumno y consta de 1 sólo registro, y **clase[300]** es una **matriz de estructuras** con espacio para almacenar los datos de 300 alumnos.

Para acceder al contenido de las diferentes estructuras utilizaremos la misma notación; sabemos que una matriz es una colección de variables del mismo tipo y que se pueden utilizar de forma independiente mediante su índice, luego el número de matrícula del alumno 264 podrá ser accedido como **clase[264].no_matricula.**

Para terminar una estructura puede, a su vez, contener a otra estructura, por ejemplo:

```
estructura fecha {
    entero dia,mes,año;
};

estructura ficha {
    carácter nombre[20];
    fecha nacimiento;
}
```

La declaración sería:

estructura ficha alumno;

Las operaciones serían:

alumno.nombre="Pepe Garcia";

alumno.nacimiento.dia=12;

mes1= alumno.nacimiento.mes;

Subprogramas : Programar con funciones

La resolución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños (subproblemas).

La solución de estos subproblemas se realiza con **subprogramas**. Su uso permite al programador desarrollar programas más complejos utilizando un método de diseño **descendente**. Se denomina descendente ya que se inicia en la parte superior con un problema general y luego abajo el diseño específico de las soluciones en los subproblemas. Normalmente las partes en que se divide un programa deben poder desarrollarse independientemente entre sí, planteando los requerimientos de variables, el cuerpo del programa principal y luego el código de los subprogramas.

Los subprogramas pueden ser de dos tipos: **funciones y procedimientos**.

Una **función** va a ser un subprograma que **nos devuelve un dato de tipo estándar**, y un **procedimiento** es un subprograma que puede **devolver además datos de tipo estructurado**. En lenguaje C solo existen las funciones.

La división de un programa en unidades más pequeñas o funciones presenta –entre otras– las ventajas siguientes:

1. **Modularización**. Cada función tiene una misión muy concreta, de modo que nunca tiene un número de líneas excesivo y siempre se mantiene dentro de un tamaño manejable. Además, una misma función puede ser llamada muchas veces en un mismo programa, e incluso puede ser reutilizada por otros programas. Cada función puede ser desarrollada y comprobada por separado.

2. **Ahorro de memoria y tiempo de desarrollo.** Cuando se ejecuta una función se dispone de un **espacio de memoria propio**, independiente del programa principal y además este espacio se libera cuando deja de utilizarse la función.

En cuanto al tiempo de desarrollo, en la medida en que una misma función es utilizada muchas veces, el número total de líneas de código del programa disminuye, y también lo hace la probabilidad de introducir errores en el programa.

3. **Independencia de datos y ocultamiento de información.** Una de las fuentes más comunes de errores en los programas son los *efectos colaterales* o perturbaciones que se pueden producir entre distintas partes del programa. Es muy frecuente que al hacer una modificación para añadir una funcionalidad o corregir un error, se introduzcan nuevos errores en partes del programa que antes funcionaban correctamente. Una función es capaz de mantener una gran independencia con el resto del programa, manteniendo sus propios datos y definiendo muy claramente **la interfaz o comunicación** con la función que la ha llamado y con las funciones a las que llama, y no teniendo ninguna posibilidad de acceso a la información que no le compete.

En definitiva las funciones se escriben solamente una vez, pero pueden ser referenciados en diferentes puntos de un programa de modo que se puede evitar la duplicación del código y son independientes del programa principal ya que los subprogramas o módulos pueden ser verificados independientemente del resto del programa lo cual facilita considerablemente la localización de un error. Los programas desarrollados de este modo son normalmente también más fáciles de comprender, ya que la estructura de cada unidad de programa puede ser estudiada independientemente de las otras unidades de programa.

Por ejemplo retomemos el ejemplo del problema del cálculo de la superficie (área) de un rectángulo. Este problema se puede dividir en tres subproblemas:

Subproblema 1: entrada de datos de altura y base.

Subproblema 2: cálculo de la superficie.

Subproblema 3: salida de resultados.

El algoritmo correspondiente que resuelve los tres subproblemas es:

leer datos (altura, base) {entrada de datos}

área = base * altura { cálculo de la superficie }

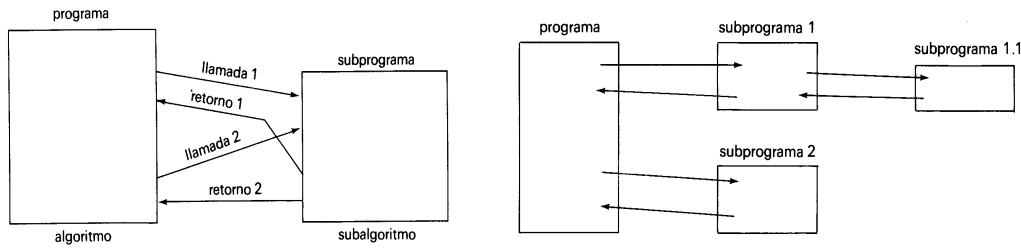
escribir base, altura, área { salida de resultados}

El problema principal se soluciona por el correspondiente programa o algoritmo principal y la solución de los subproblemas mediante subprogramas o funciones.

Un subprograma puede realizar las mismas acciones que un programa principal como aceptar datos, realizar algunos cálculos y devolver resultados. Un subprograma, sin embargo, se utiliza por el programa principal para un propósito específico.

El subprograma recibe datos desde el programa principal y le devuelve resultados. Se dice que el programa principal **llama** o **invoca** al subprograma.

Cada vez que el subprograma es llamado, el control retorna al lugar de donde fue hecha la llamada. Un subprograma puede llamar a su vez a otros subprogramas; esto queda representado en los gráficos siguientes:



Funciones

Una función está asociada con un **identificador o nombre**, que se utiliza para referirse a ella desde el resto del programa. En toda función hay que distinguir entre su **prototipo o declaración**, su **código o implementación** y su **llamada**. Para explicar estos conceptos hay que introducir los conceptos de **valor de retorno** y de **argumentos**.

De forma general:

tipo devuelto **nombre_función** (arg1 , arg2 ,....)

Una función es un conjunto de código que **puede o no** tomar o recibir uno o más valores llamados **argumentos** y produce o no, un valor denominado **resultado o valor de retorno**.

Así, por ejemplo

$$f(x) = \frac{x}{1+x^2}$$

donde f es el **nombre** de la función y x es el argumento, X es un **parámetro formal** utilizado en la definición de la función. Para evaluar f debemos darle un **valor** a x , con este valor se puede calcular el resultado. Con $x = 3$ se obtiene el valor 0.3

En este caso:

$$f(x, y) = \frac{x - y}{\sqrt{x} - \sqrt{y}}$$

es una función con dos argumentos. Sin embargo, solamente **un único valor** se devuelve desde la función para cualquier conjunto dado de argumentos.

Particularmente en Lenguaje C cuando vamos a utilizar una función dentro de un programa debe especificar en un primer momento lo que se denomina **prototipo de la función**, que esta compuesto por:

- tipo de dato devuelto (* ponemos **nada** si no devuelve ningún valor)
- nombre de la función
- **sólo** el tipo de los argumentos de entrada/salida (lo mismo que en *)

Así por ejemplo si queremos hacer una función que recoja como datos de entrada la base y la altura de un triángulo y nos calcule el área tendríamos el siguiente prototipo:

entero **Area** (entero , entero)

^_valor de salida ^_nombre función ^-----^----- argumentos

El concepto más próximo al usuario, que es el concepto de **llamada**. La llamada a la función es el **acto de utilizarla**. Los pasos son los siguientes:

- Si la función devuelve un valor se la llama asignando a una variable que recoja el resultado
- Se llama incluyendo el *nombre* de la función seguido de los *argumentos*, como una *sóla sentencia* del programa principal. Los argumentos se separan por comas.

Por ejemplo: `x= Area (3,5)`

Posteriormente cuando cerramos la llave del programa principal es cuando viene la **implementación del código** de la función:

```

Inicio
{
    real x;
    ....
    ....
    // llamada a la función
    x= Area (3,5);
    ....
    ....
} // fin programa principal

// Implementación de la función

real Area (entero base, entero altura)
{
    real resultado;
    ...
    resultado = base*altura ;
    ...
    devuelve resultado;
}

```

La primera línea de la definición es particularmente importante. La primera palabra **real** indica el tipo del valor de retorno. Esto quiere decir que el resultado de la función será un número real. Después viene el **nombre** de la función seguido de, entre paréntesis, la definición de los **argumentos** y de sus **tipos** respectivos. En este caso hay dos argumentos, **base** y **altura**, que son

ambos de tipo **entero**. A continuación se abren las llaves que contienen el código de la función. La primera sentencia declara la variable **resultado**, que es también de tipo real. Después vendrían las sentencias necesarias para calcular **resultado**. Finalmente, con la sentencia **devuelve** se retorna el **resultado**.

Si profundizamos ahora un poco en la estructura que tendría un programa con funciones, a grandes rasgos sería similar a este:

```

entero f (entero, entero)
Inicio
{
    entero x;
    real y;
    ....
    ....
    x= f (3,6);
    ...
    ...
}

entero f (entero a, b)
{
    entero z;
    ....
    devuelve z;
}

```

Podemos observar que **existen variables en el programa principal** y también hay otras distintas **en la implementación de la función**, por lo que explicamos ahora el concepto de **visibilidad o ámbito**.

AMBITO: VARIABLES LOCALES y GLOBALES

La parte del programa principal o función en que una variable se define y se puede utilizar o alterar su contenido se conoce como **Ámbito**.

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

- **variables locales**
- **variables globales**

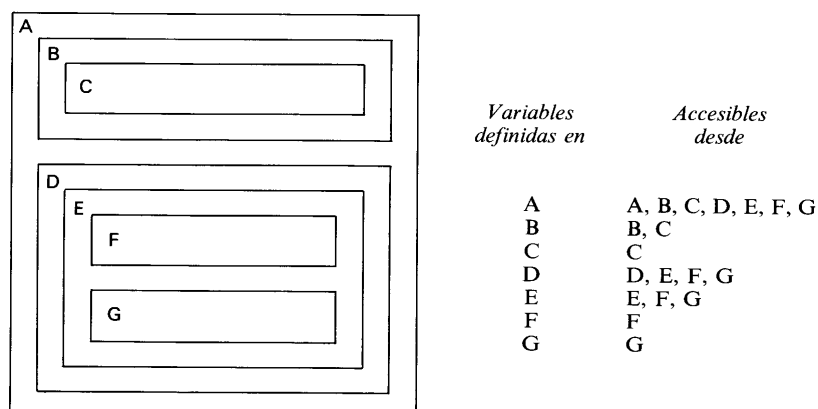
• Una **variable local** es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese subprograma y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. *El significado o visibilidad de una variable se confina a la función en el que está declarada.*

Cuando otro subprograma utiliza el mismo nombre se refiere a una posición diferente en memoria. Se dice que tales variables son **locales al subprograma** en el que están declaradas.

• Una **variable global** es aquella que está declarada para el programa principal. El uso de variables locales tiene muchas ventajas. En particular, hace a las funciones independientes y se establece la comunicación entre el programa principal y los subprogramas a través de la lista de parámetros o argumentos.

Una variable local a un subprograma no tiene ningún significado en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros subprogramas, es decir, no pueden utilizar este valor.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros. La figura muestra un esquema de un programa con diferentes subprogramas con variables locales y otras globales, aquí se muestra el ámbito de cada definición.



Pasamos ahora a estudiar qué ocurre en la llamada a una función.

Paso de parámetros

Existen diferentes métodos para la transmisión o paso de los parámetros a subprogramas. Es preciso conocer el método adoptado por cada lenguaje, ya que no siempre son los mismos. Dicho de otro modo, un mismo programa puede producir diferentes resultados bajo diferentes **sistemas de paso de parámetros** .

Los parámetros pueden ser clasificados como

entradas: se proporcionan valores desde el programa que los llama y que se utilizan dentro del subprograma.

salidas: las salidas producen los resultados del subprograma, este devuelve un valor calculado por dicha función.

entradas/salidas: un solo parámetro se utiliza para mandar argumentos a un programa y para devolver resultados.

Los métodos mas empleados para realizar el paso de parámetros son:

-**paso por valor** (también conocido por *parámetro valor*).

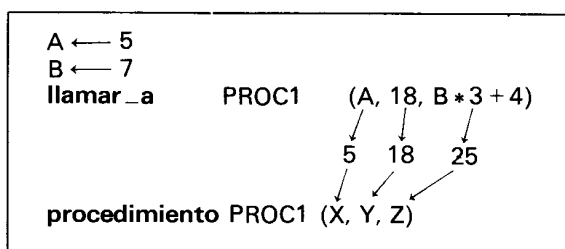
-**paso por referencia o dirección** (también conocido por *parámetro variable*).

Paso por valor

Los parámetros se tratan como variables locales y los valores iniciales se proporcionan **copiando los valores** de los correspondientes argumentos.

Los parámetros locales a la función **reciben** como valores iniciales los valores de los parámetros actuales y con ello se ejecutan las acciones descritas en el subprograma.

Aquí se muestra el mecanismo de paso por valor de un procedimiento con tres parámetros.



Aquí podemos ver que se realiza una **copia** de los valores de los argumentos en la llamada al procedimiento en las variables X, Y, Z , pero en

ningún caso los valores de A y B se verán modificados.

Por ejemplo:

```

entero f (entero);    //Prototipo de la función f
Inicio
{
    entero x, y;
    Leer (x);
    x=x+1;
    y= f(x);
    Escribe (x,y)
}

entero f (m)
{
    entero x;
    x=m;
    x=x*x;
    m=x*x;
    devuelve (m);
}
    
```

Si en la ejecución de este programa damos a x el valor 5, la siguiente línea lo incrementa en 1 y llamamos a la función con f(6). En la llamada a la función se hace una **copia** del valor de la variable **x** del programa principal en la variable **m** de la función. Una vez dentro de la función se declara otra variable **x** que es distinta a la del prog. Ppal., siendo después asignada al valor

de m, luego hacemos el cuadrado, lo asignamos a m y retornamos el valor. Aquí finaliza la llamada a la función. En el prog. ppal escribimos los valores de x e y dando como resultado x=6 e y=36. Es muy importante comprender este mecanismo.

Paso por referencia (dirección) o variable

Los parámetros por variable se deben definir en la cabecera del subprograma. En pseudocódigo de C deben ir precedidos por la palabra **dir**, operador que nos indica la dirección de memoria física que ocupa esa variable.

Se utiliza el paso por variable cuando el subprograma debe modificar el contenido de una variable del prog. ppal o devolver algún valor más, recordemos que una función solo devuelve un valor directamente.

Por ejemplo si la función anterior la modificamos así:

```

entero f (dir entero); //Prototipo de la función f
Inicio
{
    entero x, y;
    Leer (x);
    x=x+1;
    y= f(dir x);
    Escribe (x,y)
}

entero f ( dir m)
{
    entero x;
    x=m;
    x=x*x;
    m=x;
    devuelve (2x);
}

```

Si llegamos a la llamada con f(6), al ser un paso por variable o dirección, significa que la variable **m** de la función f tiene la dirección de memoria de la variable **x** lo que significa que cualquier cambio del valor de m se verá reflejado en x. Por tanto dentro de la función hacemos x=6, x al cuadrado (36), m=36 y devolvemos 2x, con lo que en el prog. ppal escribiremos x=36 e y=72.

Ejemplo, definición de la función **area_circulo (i , j)** , donde **i** es el radio y **j** será el área y la función devolverá un 1 si todo ha ido bien.

```
entero area_circulo (i, dir j) // Prototipo de la función
Inicio
{   Leer (radio)
    m= area_circulo (radio, dir y)
    si (m) entonces escribe ("Calculo correcto, área = ",a);
}

entero area_circulo (i, dir j )
{ si (i>0) entonces {   j=3.1416 * i * i;
                      devuelve (1);
                      }
  sino devuelve (0);
}
```

Si el radio es 2, en la llamada se copia en **j** la dirección de **y**, mientras que en **i** se copia el valor de radio. Al ser **j** una variable por valor o dirección su contenido se vera reflejado en la variable **y** del prog. ppal.

Bibliografía utilizada:

- Fundamentos de la programación. Algoritmos y estructuras de datos
Cap. 4 y 5 Ed. McGraw Hill
- Programación en Pascal a través de Pseudocódigo