



7. Metodología de la programación

Introducción:

Antes de programar hay que analizar el proceso del problema y el planteamiento que le vamos a dar. Después resolveremos el problema mediante un algoritmo. Programar consiste en decir como se desarrolla el algoritmo y dar un resultado final.

Problema real → Enunciado → Algoritmo → Programa

El Pascal es un lenguaje con fundamento algorítmico y el Pascal en cada máquina es distinto y por ello es necesario que se establezca un lenguaje standard de Pascal para que funcionen en todas las máquinas.

7.1 Notación algorítmica:

Hay dos maneras de meter los datos (objetos): *constantes* y *variables*. Los objetos pueden ser de distinto tipo:

- a.- Tipo numérico: Entero (Z) sin decimales (32)
Real (R) con decimales
- b.- Tipo carácter: una única letra imprimible del código ASCII ('a', ~~32~~)
- c.- Tipo string: Es una secuencia de caracteres que se trata como un solo dato.
- d.- Tipo lógico o booleano: Pueden contener los valores de falso o verdadero

El tipo de variable determina el rango de valores que podemos almacenar en ella.

Variables y constantes

Los tipos de datos que manejaremos en nuestro programa pueden ser de dos clases: *variables* o *constantes*.

Como su nombre lo indica las variables pueden cambiar a lo largo de la ejecución de un programa, en cambio las constantes serán valores fijos durante todo el proceso.

Un ejemplo de una variable es cuando vamos a sumar dos números que serán introducidos por el usuario del programa, éste puede introducir dos valores cualesquiera y no sería nada útil restringirlo a dos valores predefinidos, así que dejamos que use los valores que el necesite sumar.



Ahora, si nuestro programa de operaciones matemáticas va a utilizar el valor de PI para algunos cálculos podemos definir un identificador PI con el valor de 3.1415926 constante, de tal forma que PI no pueda cambiar de valor, ahora en lugar de escribir todo el número cada vez que se necesite en nuestro programa, solo tenemos que escribir PI.

Las variables y constantes pueden ser de todos los tipos vistos anteriormente: numéricos tanto enteros como reales, caracteres, cadenas de caracteres, etc.

Comentarios

Es posible introducir comentarios en nuestro programa que sirvan únicamente para mejorar la comprensión del código fuente.

Un comentario no es tomado en cuenta al momento de la compilación del programa y es de enorme importancia al momento de crearlo, modificarlo o mantenerlo.

Existen dos formas de colocar comentarios en un programa de Turbo Pascal, entre llaves: {Comentario} o entre paréntesis y asteriscos: (*Comentario*).

Operaciones con variables:

Se pueden hacer dos tipos de operaciones:

Las operaciones aritméticas se aplican a tipos numéricos y son por orden de prioridad:

**mod (resto de la div), div, /,*
+,-
>,<,<=,>=,=,<> (distinto)**

Ej: 3+2*5>20 es 13>20 y es F

Mezcla de valores de distintos tipos numéricos:

Para la suma, la resta y el producto:

Operando Izquierda	Operando Derecha	Resultado
Real	Real	Real
Integer	Integer	Integer
Real	Integer	Real
Integer	Real	Real



Para la división:

Operando Izquierda	Operando Derecha	Resultado
Real	Real	Real
Integer	Integer	Real
Integer	Real	Real
Real	Integer	Real

Para mod:

Operando Izquierda	Operando Derecha	Resultado
Integer	Integer	Integer
Real	Real	Illegal
Integer	Real	Illegal
Real	Integer	Illegal

Las operaciones lógicas se aplican a todo y son por orden de prioridad:

NOT, AND, OR

Dos órdenes con igual prioridad en VMS es de izq. a derecha.

Tabla de operaciones lógicas:

p	q	NOT p	p OR q	p AND q
V	V	F	V	V
V	F	F	V	F
F	V	V	V	F
F	F	V	F	F

Un predicado es cualquier expresión que se evalúa a verdadero o falso.

[[p AND (NOT q)] OR r], [[7+(3/8)]<3]



7.2 CLASES DE INSTRUCCIONES: (en pseudocódigo)

Asignación: Externa (se lee un valor del dato por teclado)

leer <variable> , Ej: leer (x)

Interna: <variable>:=<variable> , Ej: y:=x

Escritura: escribir <variable o cte.> , Ej: escribir x

escribir "hola"

escribir pi

Condicional: si <condición>

entonces <secuencia de acciones 1>

sino < secuencia de acciones 2> {opcional}

finsi {termina el condicional}

Ejemplo: Dados tres números por teclado A,B,C, imprimir el mayor en pantalla.

Comienzo

leer (A,B,C);

si (A>B)

entonces si (A>C)

entonces escribir A;

sino escribir C;

finsi

sino si (B>C)

entonces escribir B;

sino escribir C;

finsi

finsi

Fin

Condicional generalizado:

según <variable> hacer

<valor1>:< secuencia de acciones 1>;

<valor2>:< secuencia de acciones 2>;

.....

<valor_n>:< secuencia de acciones n>;

finsegun

Ejemplo:

Comienzo

leer (x);



según <x> hacer

0..4:escribir "frío";

5,6:escribir "templado"

sino: escribir "calor"

finsegun

Fin

**->Construir un programa que según la tabla, un semáforo regule la duración de el tiempo de luz en verde.

Flujo/Tiempo	Escas o 0-300	Medio 301- 500	Alto >500
lluvia	30	50	60
nublado	30	45	50
soleado	30	45	45

Solución:

Comienzo

leer (tiempo,flujo);

si flujo=escaso entonces t-apertura=30

sino si flujo=medio entonces

si tiempo=lluvia entonces t-apertura=50

sino t-apertura=45

fsi

sino

si tiempo=lluvia entonces t-apertura=60

sino si tiempo=nubes entonces t-apertura=50

sino t-apertura=45

fsi

fsi

fsi

fsi

Terminar

Estructuras Iterativas¹

1.- Mientras <predicado> hacer <acciones>

F. mientras

Puede que nunca ejecute la acción.

¹ Las acciones no pueden modificar el predicado pq podría hacerse infinito. (bucle infinito).



2.- Repetir
 <acciones>
Hastaque <predicado>

Se realiza el bucle al menos 1 vez.

Ej: Escribir 1 a 10

i=1			i=0		
<u>mientras</u>	1<=10	<u>hacer</u>	<u>mientras</u>	i < 10	<u>hacer</u>
<u>escribir</u>	i		i=i+1		
i=i+1			<u>escribir</u>	i	
<u>fmientras</u>			<u>fmientras</u>		

i=1
Repetir
 escribir i
 i=i+1
hastaque i < 10
 Repetir se puede hacer equivalente a Mientras

Mientras:

si <predicado> entonces
 Repetir <acciones>
 hastaque NO <predicado>
fsi

Repetir:

<acciones>
mientras NO <predicado> hacer <acciones>
fmientras

Ejemplo: Órdago

Barajar un nº determinado de cartas, mirar-carta, hay-cartas cierto o falso

1.- Contar el nº de cartas (nº de sotas)



Barajar

cont=0

cont sotas=0

Mientras hay cartas hacer

mirar carta

cont=cont+1

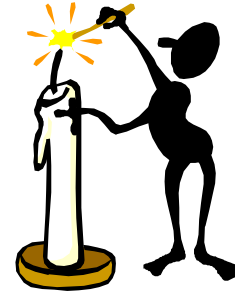
si carta=sota entonces

cont sotas=contsotas

finsi

fmientras

escribir cont, contsotas



2.- Contar el nº de veces que aparecen 2 reyes seguidos

mientras hay-cartas hacer

mirar carta1

mirar carta2

si carta1=rey y carta2=rey entonces

cont=cont+1

fsi

fmientras²

PROGRAMACIÓN EN PASCAL

{ } o (**) comentario dentro del programa.

; separa sentencias.

El programa siempre acabará con END. .

8. Estructura standard de un programa :

```
PROGRAM nombre_del_programa (input, output);
```

```
VAR {declara las variables del programa}
```

```
PROCEDURE {declaran procedimientos del programa}
```

² Esta mal, no contaria 1R, R3



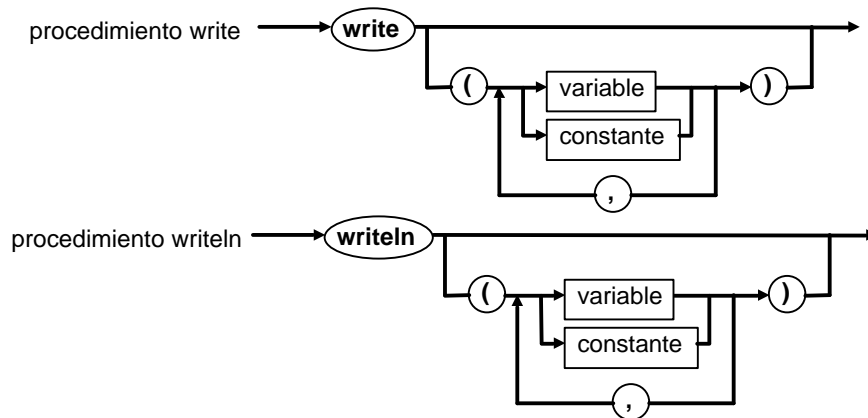
FUNCTION {declara funciones} ...

BEGIN
{cuerpo del programa}
end.

8. 1. Entrada/Salida de datos por consola a:

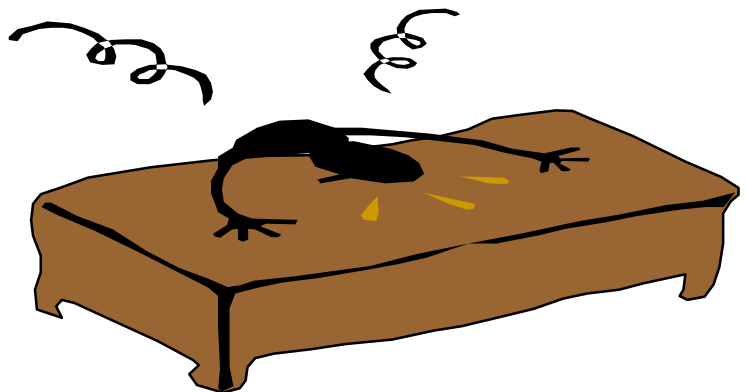
8. 1. 1. Salida por consola a: $\left\{ \begin{array}{l} \text{WRITELN} \\ \text{WRITE} \end{array} \right.$

El Pascal nos proporciona dos procedimientos para mostrar resultados por la salida estándar. Estos son write y writeln.



LA SENTENCIA WRITELN:

Visualiza resultados por pantalla. Al finalizar, da un salto a la siguiente línea para visualizar el siguiente resultado. Su sintaxis es: `WRITELN (lista_de_salidas)`. La `lista_de_salidas` es una serie de elementos separados por `,` (coma).





Ejemplo:

a :=58;

WRITELN ('Eso equivale a 'a,' centímetros');

Cadena

Cadena

En pantalla: Eso equivale a 58 centímetros

La sentencia WRITELN sin listas de salidas:

EL resultado en pantalla es una línea en blanco porque salta a la siguiente línea.

La sintaxis es: WRITELN;

La sentencia WRITE:

Visualiza resultados en pantalla pero al finalizar no pasa a la siguiente línea para visualizar el nuevo resultado.

La salida con formato:

- **Formato para valores enteros:** La sintaxis es: WRITELN (Expresión_INTEGER: C). Indica que reserva C posiciones para colocar el valor, ajustado a la derecha.
- **Formato para caracteres:** La sintaxis es: WRITELN (Expresión_CHAR: C). Indica que reserva C posiciones para colocar el valor, ajustado a la derecha.
- **Formato para reales:** La sintaxis es: WRITELN (Expresión_REAL: Longitud: N°_decimales).

Reglas:

1. Si no hay código de formato, se muestra en notación científica.
2. Si el número no cabe, también se muestra en notación científica.
3. Si no se pone número de decimales, se muestra con la longitud dado y los decimales que quepan.
4. Si sobran espacios, los rellena con espacios en blanco por delante.
5. Los decimales se redondean.

Ejemplo de un programa:

```
PROGRAM escribe5 (INPUT, OUTPUT);
```

```
VAR
```

```
    R: REAL;
```

```
BEGIN
```

```
    R := 3.14159;
```



```

WRITELN (R);
WRITELN (R:8:6);
WRITELN (R:8:5);
WRITELN (R:8:4);
WRITELN (R:8:3);
WRITELN (R:8:2);
WRITELN (R:8:1);
WRITELN (R:8:0)

```

END.

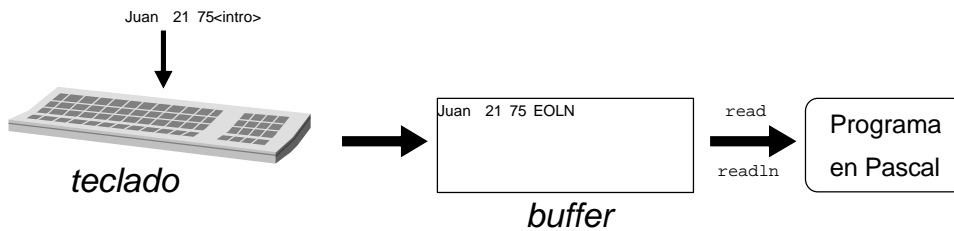
En pantalla:

```

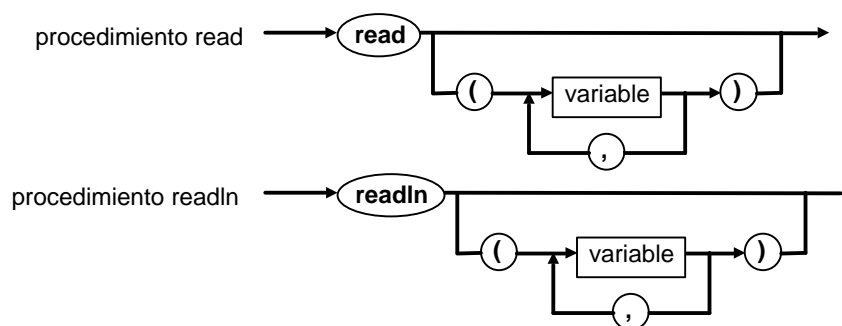
3.1415900000E + 00
3.141590
 3.14159
   3.1416
    3.142
     3.14
      3.1
       3

```

8. 1. 2. Entrada por teclado: READLN



Los procedimientos de lectura que nos proporciona el Pascal son dos: read y readln. Su sintaxis para la entrada estándar es la siguiente.



La sentencia READLN:

Solicita datos al usuario. Su sintaxis es: READLN(lista_de_variables). La lista_de_variables tiene que estar separada por ','. Al finalizar de leer todas las variables salta a la siguiente línea.



La sentencia READ:

Solicita datos al usuario. Su sintaxis es: READ(lista_de_variables). La lista_de_variables tiene que estar separada por ','. Al finalizar de leer todas las variables no salta a la siguiente línea. La sentencia READLN(Ch1, Ch2, Ch3, Cantidad); se puede escribir como READ(Ch1, Ch2, Ch3, Cantidad); READLN;

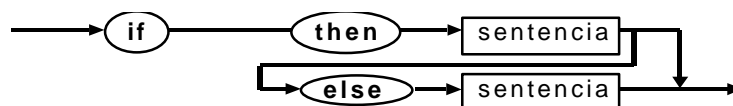
Ejemplo:

VAR

A, B, C, D: INTEGER;

Sentencia	Datos	Valores leídos
READ(A, B); READ(C, D);	123 456 789 <eoln> 987 654 321 <eoln>	A := 123 , B := 456 C := 789 , D := 987
READLN(A, B); READ(C, D);	123 456 789 <eoln> 987 654 321 <eoln>	A := 123 , B := 456 C := 987 , D := 654
READLN(A); READLN(B); READLN(C);	123 456 789 <eoln> 987 654 321 <eoln> <eoln>	A := 123 B := 987 -

8.2 Composicion secuencial .



If-Then-Else

Cada else va asociado a cada if inmediatamente anterior y la sentencia anterior al else no lleva ;.

if <condición> then <sentencia>
else <sentencia>;

<pre>if a<>0 then c:=b/a else c:=b*a;</pre>	<pre>if a<>0 then begin writeln('a,b,c'); c:=b/a; end else c:=b*a;</pre>
---	--



8.3 Tratamiento Secuencial.

En este apartado se introduce una metodología que permite diseñar algoritmos de naturaleza iterativa, es decir, algoritmos en los que una determinada acción se repite un cierto número de veces, eventualmente ninguna.

Concretamente, hay que conseguir

- la coherencia de la acción con el estado inicial (en el que se cumple la condición de continuación si se trata de una composición de tipo *mientras*).
- la coherencia en el encadenamiento de dos ejecuciones sucesivas de la misma acción (el estado final de la primera, si se cumple la condición de continuación, debe servir de estado inicial de la segunda).
- que el estado final de la última ejecución de la acción (en el que se cumple la condición de terminación) represente la solución del problema.

Un conjunto finito de objetos está organizado en forma de secuencia si o bien

- a) es el conjunto vacío, y en este caso diremos que se trata de la *secuencia vacía*, o bien
- b) es un conjunto no vacío y además es posible definir las nociones de
 - **primer elemento de la secuencia.** Se trata de un elemento del conjunto que se distingue de los demás. El acceso a este elemento permite el acceso posterior a todos los demás elementos de la secuencia.
 - **relación de sucesión entre los objetos.** Cada uno de los objetos de la secuencia, salvo el *último* o *elemento final*, precede a otro elemento (su *siguiente*). Esta relación de sucesión permite acceder a todos los elementos de la secuencia. Comenzando por el primero, para el que se dispone de un modo de acceso particular, se puede ir accediendo desde cada uno de ellos al siguiente hasta llegar al último.
 - **caracterización del fin de la secuencia.** Se dispone de un indicador que señala la presencia del *último elemento* y que permite detener el recorrido de la secuencia.

Desde el punto de vista del diseño de algoritmos, una información está organizada en forma de secuencia si es posible



- detectar que la secuencia es vacía, es decir, que no hay información que tratar, o bien,
- si no es vacía sabemos
 - 1.- cómo acceder al *primer elemento*,
 - 2.- cómo pasar de un elemento al *siguiente*, y
 - 3.- cómo detectar la presencia del *último elemento*.

8. 3. 1. Esquemas de recorrido y búsqueda en secuencias

Vamos a dar unos patrones o esquemas de programa que permiten resolver todos los problemas de naturaleza iterativa. Estos esquemas son como plantillas que deben ser cubiertas con los datos adecuados para el problema concreto de que se trate.

Siguiendo el texto proponemos tres esquemas distintos que resuelven otros tantos tipos de problemas iterativos diferentes. El programador que utilice este método de tratamiento secuencial debe

- 1.- Clasificar su problema en uno de estos tres grandes grupos
- 2.- Rellenar la plantilla o esquema, es decir
 - 2.1.- Realizar la máquina de secuencias
 - 2.2.- Realizar el tratamiento concreto que se debe aplicar a los elementos de la secuencia para resolver el problema

8. 3. 1. 1. Esquemas de recorrido

Resuelven problemas en los que cada uno de los elementos de la secuencia se trata o procesa para obtener el resultado. Se distinguen dos tipos de problemas:

- 1.- Aquellos en los que todos los elementos de la secuencia se tratan igual

```

esquema de recorrido número 1 es
{ recorre una secuencia no vacía tratando de
distinto modo al elemento final }
  iniciar_adquisicion;
  iniciar_tratamiento;
  obtener_elemento_siguiete;
  mientras elemento_no_final hacer
    { se han tratado todos los elementos que
    preceden al elemento en curso que no es el final
    }
    tratar_elemento
    obtener_elemento_siguiete;
  finmientras;
  tratar_elemento_final;
  finalizar_tratamiento
fin.

```



excepto el último que recibe un tratamiento diferente (esquema número 1)

2.- Aquellos en los que todos los elementos de la secuencia, incluido el último, se tratan de la misma forma (esquema número 2).

Los esquemas de recorrido correspondientes a estas dos clases de problemas son los que se muestran en las Figuras 4 y 5 respectivamente. En estos esquemas el tratamiento de los elementos de la secuencia viene descrito por las acciones de la Figura 6.

```
esquema de recorrido número 2 es  
{ recorre una secuencia no vacía  
  tratando igual a todos sus elemetos }  
  iniciar_adquisicion;  
  iniciar_tratamiento;  
  repetir  
    { se han tratado todos los elementos hasta el  
      elemento en curso que no es final, o bien se  
      acaba de iniciar el tratamiento }  
    obtener_elemento_siguiete;  
    tratar_elemento  
  hastaque elemento_final;  
  finalizar_tratamiento  
fin.
```

Figura 5

En este punto es importante recordar que los esquemas considerados solamente permiten tratar secuencias no vacías, con lo cual siempre es posible obtener el primer elemento que, eventualmente, puede ser el último.

tratar_elemento	tratar_elemento_final	iniciar_tratamiento
Describe el tratamiento en curso, es decir, lo que se hace con cada elemento (salvo el último en el esquema número 1)	Es lo que se hace con el último elemento en el esquema número 1	Describe el conjunto de acciones previas que deben realizarse para asegurar el correcto encadenamiento de los tratamientos en curso y final (para que exista coherencia y que el resultado sea el previsto)

Figura 6



8. 3. 1. 2. Esquemas de Búsqueda Asociativa

Se trata de un recorrido especial en el que se recorre la secuencia sin hacer nada con sus elementos, hasta encontrar un elemento que cumpla una determinada propiedad, o bien hasta llegar al final de la secuencia. El esquema de programa asociado es el que se muestra en la Figura 7.

```

esquema de búsqueda asociativa es
{ recorre una secuencia que no puede ser vacía hasta encontrar un elemento
que cumpla una propiedad o hasta llegar al final}  iniciar_adquisicion;
  obtener_elemento_siguiete;
  mientras elemento_no_hallado y elemento_no_final hacer
    obtener_elemento_siguiete
  finmientras;
  si elemento_hallado
  entonces tratar_elemento_hallado
  sino      tratar_ausencia
  finsi;
fin.
    
```

Figura 7

El significado de las acciones y predicados que aparecen en este esquema y que no están en ninguno de los anteriores es el que se muestra en la Figura 8

tratar_elemento_hallado	tratar_ausencia	elemento_hallado
Es lo que hay que hacer si se encuentra un elemento que cumpla la propiedad	Es lo que se hace si se llega al final sin haber encontrado un elemento que cumpla la propiedad	Es el predicado que nos indica si un elemento cumple la propiedad buscada

Figura 8

Resumen de los 3 esquemas generales distintos para tratamientos secuenciales.

Esquema 1:

Inicializar Tratamiento
Inicializar Adquisición
Obtener_1º_elemento



Mientras no_último_elemento
 Tratar elemento
 Obtener siguiente_elemento

Fmientras
Tratar ultimo elemento
Finalizar Tratamiento

El último elemento tiene diferente tratamiento del resto.

Esquema 2:

Inicializar Tratamiento
Inicializar adquisición
Repetir
 Obtener elemento
 Tratar elemento
Hasta último_elemento
Finalizar tratamiento

Esquema 3: esquema de búsqueda

Inicializar Adquisición
Obtener primer elemento
Mientras no_último_elemento y elemento_no_encontrado hacer
 Obtener siguiente_elemento
Fmientras
Si elemento_encontrado
entonces Tratar_presencia
sino tratar_ausencia
finsi

Ejemplo1: Dado un nº entero, imprimir en pantalla su imagen especular. (Todos los elementos tienen el mismo tratamiento).

El programa se realizará con divisiones entre 10 y cogiendo el resto. Los elementos de cada secuencia son el resto y el cociente de cada división.

```
PROGRAM invertir (input, output);  
  
VAR num, digito:integer;
```



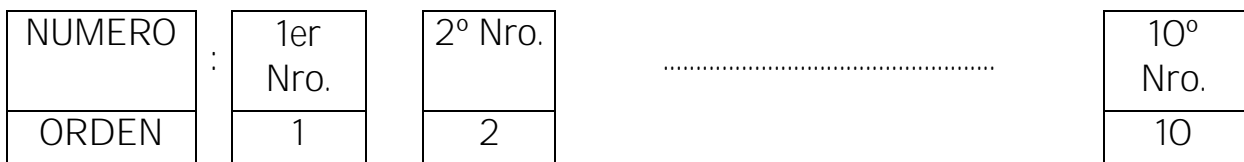
```

BEGIN
  {Inicializar}
  READLN (num); {Inicio de adquisición}
  REPEAT
    {Obtener elemento}
    Digito:=num mod 10;
    num:=num div 10;
    {Tratar elemento}
    WRITE('Dígito');
  UNTIL (num:=0);
  {Finalizar adquisición}
END.
    
```

Ejemplo2: Se trata de diseñar un algoritmo para controlar el acceso de un usuario a un cajero automático. El algoritmo debe leer una secuencia de 10 números como máximo y debe compararlos con un valor SECRETO. Si un número coincide con SECRETO el usuario puede acceder (y no debe introducir más números). Si, por el contrario, después de introducir 10 números ninguno de ellos coincide con SECRETO, el acceso es denegado.

Principio de Resolución: Hay que ir leyendo los números y comparándolos con el SECRETO tal y como indica el enunciado. Un detalle importante es que hay que llevar cuenta del ORDEN del número que se introduce para saber si llegamos al décimo.

Secuencia: Estará formada por pares de elementos de la forma



Con cada uno de los elementos hay que comprobar si el NUMERO coincide con el SECRETO, o bien si el ORDEN es 10 para detectar el último. La secuencia no es vacía ya que contiene exactamente 10 elementos.

Esquema: Teniendo en cuenta las características de la secuencia se trata de una búsqueda asociativa donde la propiedad buscada es que el NUMERO coincida con SECRETO

Algoritmo: Sustituyendo en el esquema anterior cada una de las acciones y predicados con los correspondientes a este problema, el algoritmo queda finalmente como



```
accion control de acceso es
{iniciar_adquisicion}
  ORDEN:=0;
{obtener_elemento_siguiete}}
  leer NUMERO;
  ORDEN:=ORDEN+1;
mientras NUMERO ≠ SECRETO {elemento_no_hallado} y
  ORDEN<10 {elemento_no_final} hacer
  {obtener_elemento_siguiete}
  leer NUMERO;
  ORDEN:=ORDEN+1
finmientras;
si NUMERO = SECRETO {elemento_hallado}
entonces escribir "acceso permitido" {tratar_elemento_hallado}
sino      escribir "acceso denegado" {tratar_ausencia}
finsi;
fin.
```

8. 3. 2. Tratamiento de secuencias eventualmente vacías

En esta sección vamos a generalizar los esquemas de programa anteriores para que sean capaces de tratar con secuencias que eventualmente pueden ser vacías. En principio lo único que hay que añadir a los esquemas vistos, tanto de recorrido como de búsqueda asociativa, es la capacidad de detectar que la secuencia es vacía antes de intentar acceder al primer elemento de la misma. Parece lógico que el momento oportuno para realizar esta comprobación es justo después de iniciar la adquisición, pero antes de obtener elementos. Así, lo que necesitamos con respecto a los esquemas anteriores es poder definir el predicado *secuencia_vacia* cuya especificación será la que se muestra en la Figura 9.

secuencia_vacia
Produce el valor cierto cuando la secuencia es vacía. En otro caso devuelve falso. Debe evaluarse después de la acción <i>iniciar_adquisicion</i> .

Figura 9

Si comparamos el predicado *secuencia_vacia* con el predicado *ultimo_elemento*, aunque en principio fueron concebidos para detectar situaciones distintas, en realidad tienen una utilidad muy parecida. Los dos son ciertos cuando no quedan elementos por tratar (porque no los hay, o bien porque ya se obtuvo el último). Y los dos deben ser evaluados después de la acción *iniciar_adquisición*. Así, podemos refundirlos en uno solo que nos permitirá



detectar las dos situaciones y que denominaremos *fin_secuencia*. Su especificación se muestra en la Figura 10.

fin_secuencia
Produce el valor cierto si no quedan elementos por recorrer. En otro caso devuelve falso. Debe evaluarse después de la acción <i>iniciar_adquisicion</i> .

Figura 10

A continuación indicamos cómo se transforman los esquemas introduciendo la posibilidad de tratar con las secuencias vacías.

8. 3. 2.1. Esquemas de recorrido de secuencias eventualmente vacías

El esquema de recorrido número 1 que recorre secuencias tratando de distinto modo al último elemento queda como se indica en la Figura 11. Por su parte el esquema número 2 que trata igual a todos los elementos queda en principio tal y como se indica en la Figura 12, pero se puede transformar de forma evidente en el que se muestra en la Figura 13 que es más claro.

```

esquema de recorrido número 1 es
{ recorre una secuencia que puede ser vacía tratando de distinto modo
al elemento final }
  iniciar_adquisicion;
  iniciar_tratamiento;
  si no fin_secuencia
  entonces obtener_elemento_siguiete;
            mientras no fin_secuencia hacer
              { se han tratado todos los elementos que preceden al
                elemento en curso que no es el final }
              tratar_elemento;
              obtener_elemento_siguiete
            finmientras;
            tratar_elemento_final;
  finsi;
  finalizar_tratamiento
fin.

```

Figura 11



Esquema de recorrido número 2 es

{ recorre una secuencia que puede ser vacía tratando igual a todos sus elementos }

iniciar_adquisicion;
iniciar_tratamiento;

si no fin_secuencia

entonces repetir

{ se han tratado todos los elementos hasta el elemento en curso que no es final, o bien se acaba de iniciar el tratamiento }

obtener_elemento_siguiete;
tratar_elemento

hastaque fin_secuencia

finsi;

finalizar_tratamiento

fin.

Figura 12

esquema de recorrido número 2 es

{ recorre una secuencia que puede ser vacía tratando igual a todos sus elementos }

iniciar_adquisicion;
iniciar_tratamiento;

mientras no fin_secuencia **hacer**

obtener_elemento_siguiete;
tratar_elemento

finmientras;

finalizar_tratamiento

fin.

Figura 13



8.3.2.2. Búsqueda Asociativa sobre secuencias que pueden ser vacías

En principio el esquema de búsqueda asociativa transformado para tratar secuencias vacías debe quedar como se indica en la Figura 14. Es decir, si la secuencia no es vacía se hace lo mismo que con el esquema de la Figura 7, y si es vacía se realiza la acción *tratar_ausencia* ya que no hemos encontrado un elemento con la propiedad buscada.

```

esquema de búsqueda asociativa es
{ recorre una secuencia eventualmente vacía hasta encontrar el primer elemento que cumpla
una propiedad, o bien hasta llegar al final }
  iniciar_adquisicion;
  si no fin_secuencia
  entonces obtener_elemento_siguiente;
           mientras elemento_no_hallado y no fin_secuencia hacer
             obtener_elemento_siguiente;
           finmientras;
           si elemento_hallado
           entonces tratar_elemento_hallado
           sino      tratar_ausencia
           finsi
  sino tratar_ausencia
fin.

```

Figura 14

No obstante, el esquema de la Figura 14 se puede simplificar si somos capaces de diseñar el predicado *elemento_hallado*, que en principio solamente se puede evaluar cuando tenemos un elemento en curso, para que se pueda evaluar también antes de obtener el primer elemento (lógicamente para dar el valor *falso*). Es decir, de forma que responda a la especificación que se muestra en la Figura 15.

elemento_hallado
Produce el valor <i>cierto</i> cuando tenemos un elemento en curso que cumple la propiedad buscada. En otro caso, es decir, cuando el elemento en curso no cumple la propiedad, o bien cuando todavía no tenemos elemento en curso, devuelve <i>falso</i> .

Figura 15



El esquema transformado, teniendo en cuenta la nueva especificación del predicado *elemento_hallado* de la Figura 15 es el que se muestra en la Figura 16.

```
esquema de búsqueda asociativa es  
{ recorre una secuencia eventualmente vacía hasta encontrar el primer elemento que cumpla  
una propiedad, o bien hasta llegar al final }  
  iniciar_adquisicion;  
  mientras elemento_no_hallado y no fin_secuencia hacer  
    obtener_elemento_siguiente;  
  finmientras;  
  si elemento_hallado  
  entonces tratar_elemento_hallado  
  sino      tratar_ausencia  
  finsi  
fin.
```

Figura 16

En muchas ocasiones no será posible definir el predicado *elemento_hallado* de acuerdo con la especificación de la Figura 15. En este caso el esquema se puede transformar un poco más en otro equivalente que utiliza una variable lógica *h*, con el valor inicial *falso*, para salvar el problema. Esta nueva variante de la búsqueda asociativa es la que se muestra en la Figura 17. Esta última versión es claramente menos elegante que la anterior, pero tiene la ventaja de que si el coste de evaluar *elemento_hallado* es grande resulta más eficiente.

```
esquema de búsqueda asociativa es  
{ recorre una secuencia eventualmente vacía hasta encontrar el primer elemento que cumpla  
una propiedad, o bien hasta llegar al final }  
  iniciar_adquisicion;  
  h:=falso;  
  mientras no h {elemento_no_hallado} y no fin_secuencia hacer  
    obtener_elemento_siguiente;  
    h:=elemento_hallado;  
  finmientras;  
  si h {elemento_hallado}  
  entonces tratar_elemento_hallado  
  sino      tratar_ausencia  
  finsi  
fin.
```

Figura 17



Ejemplo3: Calcular $\sum_{i=N}^M i$, siendo N y M números enteros cualesquiera. Si $M < N$ no se debe sumar nada, el resultado será 0.

Principio de Resolución: Si $N \leq M$ hay que recorrer los números desde N a M y cada uno de ellos se acumula en una variable SUMA. Si $N > M$ no hay números que sumar y el resultado debe ser 0.

Secuencia: En este caso se trata de la secuencia de números enteros

$$N, N+1, \dots, M$$

Que puede ser vacía si $N > M$. A todos los elementos se les aplica el mismo tratamiento con lo cual tenemos que utilizar el siguiente

Esquema: El que permite recorrer secuencias eventualmente vacías haciendo lo mismo con todos sus elementos, es decir, el número 2 modificado.

Algoritmo: Sustituyendo las acciones y predicados en el esquema con las del problema, queda como:

```

accion sumar N+...+M es
  {iniciar_adquisicion}
  leer N,M;
  i:=N-1;
  {iniciar_tratamiento}
  SUMA:=0;
  mientras no i≥M {no fin_secuencia} hacer
    i:=i+1; {obtener_elemento_siguiete}
    SUMA:=SUMA+i {tratar_elemento}
  finmientras;
  escribe SUMA {finalizar_tratamiento}
fin.

```

Ejemplo4: Comprobar si un número $N \geq 3$ es primo.

Principio de Resolución: Una estrategia clara (y poco eficiente) para comprobar si un número es primo consiste en calcular su división entera entre los números: 2, 3, ..., N div 2. Si en algún caso el resto es 0 entonces el número no es primo, en caso contrario es primo.

Secuencia: En este caso se trata de la secuencia de números enteros

$$2, 3, \dots, N \text{ div } 2$$

con $N \geq 3$. La secuencia puede ser vacía (si $N=3$). Con cada elemento hay que comprobar si cumple la propiedad de ser divisor de N. Si se encuentra uno con esta propiedad ya podemos parar y concluir que el número N no es primo.



Esquema: Está claro que se trata de una búsqueda asociativa sobre una secuencia que puede ser vacía. Elegimos la última versión (con la variable h) ya que no resulta fácil definir el predicado *elemento_hallado* de acuerdo con la especificación de la Figura 15.

Algoritmo: Sustituyendo las acciones y predicados en el esquema con las del problema, queda como:

```
accion ¿es primo N? es
  leer N; i:=1; {iniciar_adquisicion}
  h:=falso;
  mientras no h {elemento_no_hallado} y i < (N div 2) {no_fin_secuencia} hacer
    i:=i+1; {obtener_elemento_siguiete}
    h:=(N mod i) = 0 {elemento_hallado}
  finmientras;
  si h {elemento_hallado}
  entonces escribir "el numero no es primo" {tratar_elemento_hallado}
  sino      escribir "el numero es primo" {tratar_ausencia}
  finsi
fin.
```

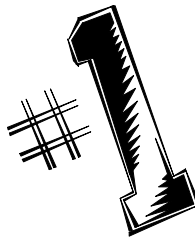


9. La Abstracción procedimental

1. Refinamientos sucesivos y diseño descendente:

Un programa cualquiera se le puede dividir en subprogramas más pequeños (módulos) que realicen las distintas tareas del programa principal. A su vez, a cada módulo se le puede dividir de nuevo en módulos independientes. A esto se le llama *diseño descendente*. Estos subprogramas independientes pueden ser de dos tipos:

- Los procedimientos
- Las funciones



2. Los procedimientos:

Los procedimientos son partes del programa declaradas después de las variables bajo el nombre del procedimiento, con el fin de sí se desea repetir una parte del algoritmo se hace una llamada al nombre del procedimiento, y así no repetir la implementación.

2.1 Procedimientos sin parámetros:

La *estructura* de estos procedimientos es:

```
PROCEDURE nombre_procedimiento(Param_formales); (* Cabecera *)
Sección_declaraciones_locales (* Constantes, variables, otros procedimientos. Es opcional *)
Cuerpo_del_procedimiento:(*Sentencias en PASCAL que corresponden a ese procedimiento*)
    BEGIN
    ...
    END; ← ojo
```

Llamada a un procedimiento: Si no tiene parámetros reales⁵, la llamada es simplemente el nombre del procedimiento.

Ejemplo: Programa que escriba 'HOLA MAMA' en letras hechas por nosotros.

```
PROGRAM mensaje (INPUT, OUTPUT);
    PROCEDURE imprime_H;
```

⁵ La relación entre los p. formales y reales está en el orden de llamada.



```
BEGIN
  WRITELN ( '*      *' );
  WRITELN ( '*****' );
  WRITELN ( '*      *' )';
  WRITELN ( '*      *' );
  WRITELN
    END;
PROCEDURE imprime_O;6
  ...
PROCEDURE imprime_L;
  ...
PROCEDURE imprime_A;
  ...
PROCEDURE imprime_M;
  ...
BEGIN (*p.p.*)
  Imprime_H;
  Imprime_O;
  Imprime_L;
  Imprime_A;
  Imprime_M;
  Imprime_A;
  Imprime_M;
  Imprime_A
END. (* p.p. *)
```

Ejemplo:

```
PROGRAM incrementar (INPUT, OUTPUT);
VAR
  Num: INTEGER;
PROCEDURE incrementar;
  BEGIN 7
    Num := num +1
  END;
BEGIN (* p.p. *)
  WRITELN ('Introduzca un número');
  READLN (num);
  Incrementar;
  WRITELN (num)
END. (* p.p.*)
```

2.2 Los procedimientos con parámetros:

⁶ Los procedimientos son análogos a la construcción del anteriormente implementado.

⁷ El uso de variables globales en procedimientos no es aconsejable ya que no serviría ese procedimiento para otros programas.



Los parámetros van a servir como control entre procedimientos y programas/otros procedimientos. Este tipo de procedimientos necesita parámetros, y estos pueden ser de dos tipos:

1. Parámetros por valor: Pueden ser variables o constantes que no cambian su valor en el cuerpo del procedimiento. Son sólo datos de entrada.
2. Parámetros por variable o referencia: Son variables que en el cuerpo del procedimiento puede cambiar su valor. Son datos de entrada y salida.

La *estructura* de este tipo de procedimientos es:

```
PROCEDURE nombre_procedimiento (lista_parámetros_formales);
  Sección_declaraciones_locales
      BEGIN
      Cuerpo_del_procedimiento
      END;
```

La lista de parámetros formales incluye todos los parámetros que se necesitan del programa principal. La sintaxis de cada parámetro dentro del paréntesis es: *Nombre_Parámetro: tipo de parámetro*. Si es un parámetro por variable, delante del nombre del parámetro se coloca la palabra **VAR**. Si hay más de un parámetro, estos van separados por punto y coma (;). Pero si hay varios del mismo tipo y clases se puede poner: VAR si son por variable, juntos los nombres de cada parámetro separados por comas (,) y después dos puntos (:) y el tipo de parámetro.

Ejemplos:

- Por valor:

```
Parámetro1: tipo_parámetro
Parámetro1, Parámetro2, Parámetro3 , ... , ParametroN : tipo_parámetro
```

- Por variable:

```
VAR Parámetro1: tipo_parámetro
VAR Parámetro1, Parámetro2, Parámetro3 , ... , ParámetroN: tipo_parámetro
```

Llamada a procedimientos con parámetros: Al requerir el procedimiento parámetros, la llamada es:

Nombre_procedimiento (lista_argumentos);



La lista de argumentos debe tener tantos argumentos como parámetros requiera el procedimiento. El orden de los argumentos influye, es decir -al primer argumento se le asigna el primer parámetro, al segundo argumento se le asigna el segundo parámetro y así sucesivamente-. El tipo de argumento debe coincidir con el tipo de parámetro al que esté asociado, por ejemplo -si un parámetro es del tipo INTEGER no se le puede asignar un argumento de tipo CHAR-. Si un parámetro es por variable el argumento debe ser una variable y no una constante o una expresión.

Ejemplo:

Dados los procedimientos en un programa:

```
PROCEDURE prueba1 (VAR num: INTEGER);  
PROCEDURE prueba2 (VAR p1: CHAR; p2, p3: REAL; VAR p4: INTEGER; p5, p6: CHAR);
```

Dadas la declaración de variables de dicho programa:

```
VAR  
  Ch1, ch2: CHAR;  
  NumReal: REAL;  
  NumEntero: INTEGER;
```

Llamadas válidas a los procedimientos:

```
Prueba1 (numEntero);  
Prueba2 (ch1, 35, numReal, numEntero, 'a', 'z');
```

Llamada no válida:

```
Prueba1 (8);
```

Cuando se concluye la ejecución del procedimiento las variables utilizadas en el procedimiento quedan con un valor desconocido y los dobles nombres de variables, que surgen al asignar argumentos a parámetros, desaparecen, quedando la variable del programa principal (argumento) con el valor surgido del doble nombre y la variable del procedimiento (parámetro) desconocida.

Ejemplo:

```
PROGRAM productos (INPUT, OUTPUT);  
CONST  
  Desc = 15;  
VAR  
  Unidades, precio: INTEGER;  
  Total, cantDesc: REAL;  
PROCEDURE descuento (VAR cantidad, descuento: REAL; porciento: INTEGER);  
  BEGIN  
    Descuento := cantidad * porciento/100;  
    Cantidad := cantidad - descuento;  
  END;  
BEGIN (* p.p. *)
```



```

WRITELN ('Introduzca el número de unidades');
READLN (unidades);
WRITELN ('Introduzca el precio');
READLN (precio);
Total := precio * unidades;
IF (unidades > 5) THEN descuento (total, cantDesc, desc)
    ELSE cantDesc := 0;
WRITELN ('Total: ',total,' Descuento: ',cantdesc)
END. (* p.p. *)

```

Ejemplo:

```

PROGRAM incrementarNúmero (INPUT, OUTPUT);
VAR
    Num: INTEGER;
PROCEDURE incrementar (VAR numero: INTEGER);
    BEGIN
        Numero := numero + 1;
    END;
BEGIN (* p.p. *)
    WRITELN ('Introduzca un número');
    READLN (num);
    Incrementar (num);
    WRITELN (num)
END. (* p.p. *)

```

Ejemplo:

```

PROGRAM calculaArea (INPUT, OUTPUT);
VAR
    RadioCirculo, resultado: REAL;
PROCEDURE areaCirculo (radio: REAL; VAR area: REAL);
    CONST
        Pi = 3.1416;
    BEGIN
        Area := pi * SQR(radio)
    END;
BEGIN (* p.p. *)
    WRITELN ('Introduzca el radio');
    READLN (radioCirculo);
    IF (radioCirculo > 0) THEN
        BEGIN
            AreaCirculo (radiocirculo, resultado);
            WRITELN (resultado:6:2);
        END
    END;
END. (* p.p. *)

```

Ejemplo:

```

PROGRAM intercambiar (INPUT, OUTPUT);
VAR
    valor1, valor2: REAL;
PROCEDURE intercambiar (VAR A, B: REAL);
    VAR
        Aux: REAL;
    BEGIN
        Aux := A;
        A := B;
        B := Aux
    END;

```



```
END;
BEGIN (* p.p. *)
WRITELN ('Introduzca el valor de A');
READLN (valor1);
WRITELN ('Introduzca el valor de B');
READLN (valor2);
Intercambiar (valor1, valor2);
WRITELN ('Valores intercambiados. Valor1: ',valor1, ' Valor2: ',valor2)
END. (* p.p. *)
```

Ejemplo:

```
PROGRAM sumaEnteros (INPUT, OUTPUT);
VAR
    N, suma: INTEGER
    Media: REAL;
PROCEDURE sumaMedia (num: INTEGER; VAR suma: INTEGER; VAR media2: REAL);
VAR
    I: INTEGER;
BEGIN
    Suma := 0;
    FOR I := 1 TO num DO suma := suma + I;
    Media2 := suma / num;
END;
BEGIN (* p.p. *)
WRITELN ('Introduzca el último número');
READLN (N);
IF (N > 0) THEN
    BEGIN
        SumaMedia (N, suma, media);
        WRITELN (suma:6, media:6:2)
    END
END. (*p.p. *)
```

3. Las funciones:

Las funciones son análogas a los procedimientos, en el sentido de que son fragmentos de programas, pero se diferencian en que las funciones devuelven un valor y los procedimientos ejecutan unas determinados comandos como si fuesen parte del programa principal.

La *estructura* de las funciones es:

```
FUNCTION nombre_función (lista_parámetros): tipo_resultado;
Sección_declaraciones_locales
Cuerpo_de_la_función
    BEGIN
        ...
    END; ← ojo
```



La función devuelve un único valor, y por esto no tiene sentido utilizar parámetros por referencia o variable en la lista de parámetros. Esta lista de parámetros es opcional. En el cuerpo de la función tiene que haber una sentencia que asigne al nombre de la función un valor del tipo del resultado.

Llamada a funciones: La llamada a la función es simplemente el nombre de la función con sus argumentos entre paréntesis. Al devolver un resultado se puede utilizar la función en cualquier sitio que se espere un valor del tipo del resultado de la función.

Ejemplos:

```
X := nombre (lista_argumentos);      (* asignación *)
IF (nombre (lista_argumentos) = 4) THEN ... (* expresión *)
```

Ejemplo:

```
PROGRAM verifica (INPUT, OUTPUT);
VAR
    Car: CHAR;
FUNCTION esDigito (ch: CHAR): BOOLEAN;
    BEGIN
        EsDigito := (ch >= '0') AND (ch <= '9')
    END;
BEGIN (* p.p. *)
    WRITELN ('Introduce un carácter');
    READLN (car);
    IF esDigito (car) THEN WRITELN ('Es carácter')
    ELSE WRITELN ('No es carácter')
END. (* p.p. *)
```

Ejemplo:

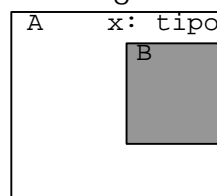
```
PROGRAM diasDelMes (INPUT, OUTPUT);
VAR
    Mes: INTEGER;
FUNCTION diasMes (i: INTEGER): INTEGER;
    BEGIN
        CASE i OF
            1,3,5,7,8,10,12: diasMes := 31;
            4,6,9,11: diasMes:= 30;
            2: diasMes:= 28;
        END;
    END;
BEGIN (* p.p. *)
    WRITELN ('Introduzca mes');
    READLN (mes);
    IF (mes < 1) OR (mes > 12) THEN WRITELN ('ERROR')
    ELSE WRITELN ('El número de días es ',diasMes(mes))
END. (* p.p. *)
```

4. Alcance y visibilidad (Ambito) de los identificadores:

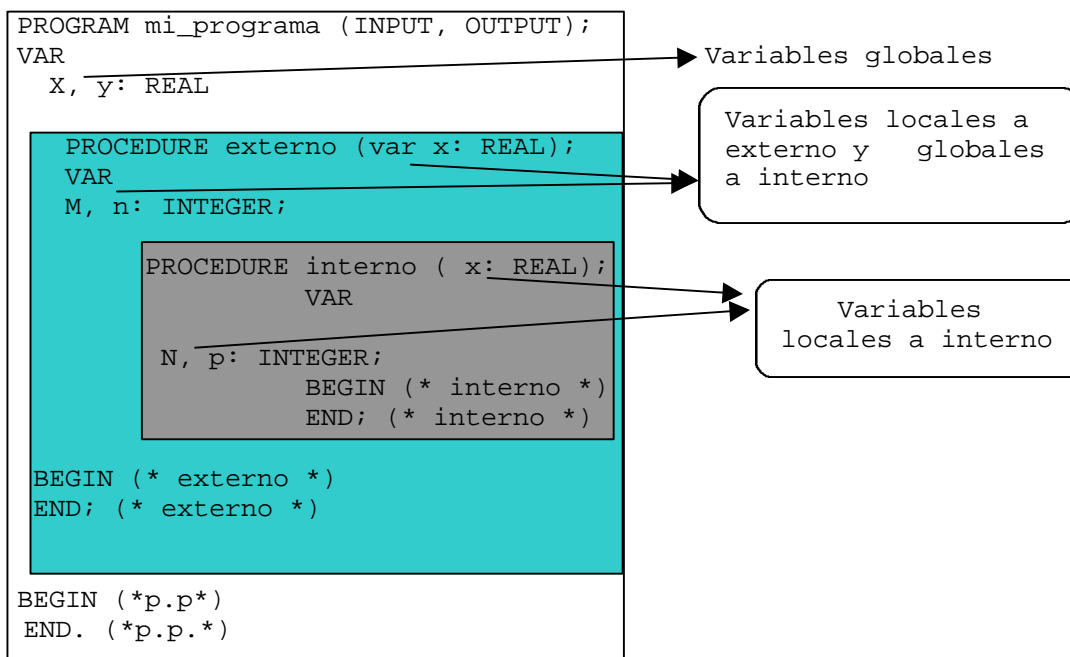


4.1 Tipos de identificadores:

- **Identificadores globales:** Se declaran en la sección de declaración locales del programa principal. Son variables, constantes y los procedimientos de primer nivel.
- **Identificadores locales a un subprograma:** Se declaran en el subprograma y sólo son visibles en ese subprograma o procedimiento.
- **Identificadores no locales:** Se produce cuando hay anidamiento de procedimientos. Sí, por ejemplo, hay un procedimiento (de primer nivel) con sus variables locales y dentro de el también hay otro procedimiento (de segundo nivel), para este segundo las variables del primero son no locales.



Ejemplo:



4.2 Alcance y visibilidad:

4.2.1 Alcance

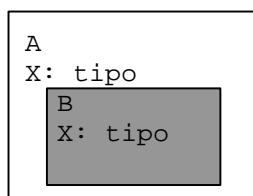
Alcance de un identificador es la zona del programa donde se puede referenciar dicho identificador.



Regla de alcance: Alcance de un identificador es el bloque (programa, procedimiento o función) donde esté y los bloques anidados que se encuentren por debajo de la declaración del identificador.

4.2.2 Visibilidad:

Visibilidad es la zona de programa donde puede referenciarse un identificador por no estar oculto.



x de A: si tiene alcance
x de A: no es visible.

4.3 Regla(s) de ámbito: ****VIP****

- El ámbito de definición de un identificador (vbles, módulos) definido en un módulo es el propio módulo y todos los módulos encerrados por él (anidados), excluyendo cualquier módulo en el que se redefine el mismo identificador.
- Un identificador está indefinido en cualquier punto fuera de su ámbito y entre el principio de su ámbito y su punto de definición.
- Un identificador tiene significado en cualquier punto dentro de su ámbito y después de su punto de definición.

Regla resumen de las tres anteriores:

- El ámbito de un identificador incluye todas las sentencias posteriores del bloque en el que se encuentra declarado y los bloques anidados en los que no se redeclara.

Ejemplo:

```
PROGRAM A (INPUT, OUTPUT);
VAR
    X1: INTEGER;
PROCEDURE B (X2: INTEGER);
VAR
    X3: INTEGER;
BEGIN (* B *)
    ...
END ;(* B *)
PROCEDURE C(X4:integer);
VAR
    X5: INTEGER;
PROCEDURE D(X6:integer);
VAR
```



```

                X7: INTEGER;
BEGIN (* D *)
    ...
END; (* D *)
BEGIN (* C *)
    ...
END; (* C *)
BEGIN (* A *)
    ...
END. (* A *)
    
```

Representar la accesibilidad de cada uno de los identificadores en cada uno de los bloques del programa.

DesdeBloque Identificador es	A	B	C	D
A	√	√	√	√
X1	√	√	√	√
B	√	√	√	√
X2	X	√	X	X
X3	X	√	X	X
C	√	X (€)	√	√
X4	X	X	√	√
X5	X	X	√	√
D	√		√	√
X6	X	X	X	√
X7	X	X	X	√
Mod acc.		B	B,C,D	D,C,B
Vbles acc.	X1	x1,x2,x3	x1,x4,x5	x1,x4,x5,x6,x7

El programa principal se puede llamar a si mismo.

No se pueden llamar a módulos en el mismo nivel que no estén todavía definidos en el listado (desde B no se puede llamar a C pero si al revés).

Desde el programa principal se puede llamar a todo, aunque no se ponga en la tabla.



10. Tipos Estructurados de Datos I.

Los tipos estructurados de datos son tipos de variables formado por combinaciones de otros tipos de datos y se dividen en homogéneos (todos del mismo tipo) como una cadena de caracteres, y heterogéneos (combinan componentes de distinto tipo).

El comienzo del estudio de los tipos estructurados de datos empieza en el punto 10.4, antes estudiaremos tipos de datos simples definidos por el usuario.

10.1 Tipos de Variables.

Hasta ahora unos tipos de variables sencillos como integer, real o boolean. Hay otros tipos de variables que son definidos por el usuario como subrangos, enumeraciones, etc.

Aquí representamos un cuadro completo de TIPOS DE VARIABLES:

Puntero	Estructurados	Simples		
	Array o vector	Real	Indefinidos	Definidos
	Registro		Integer	Subrangos
	Conjunto		Char	Enumeraciones
	Ficheros		Boolean	

- Hay dos formas de definir variables:
 - a) Implícita: como hasta ahora, VAR x,y:integer;
 - b) Explícita: se define el tipo de variable y a continuación de declara la variable.
 TYPE entero=integer;
 VAR x:integer; y:entero;

Los dos ejemplos anteriores son equivalentes como el siguiente:

```

VAR  anho :1900..1999;
      variable      TIPO
TYPE tipoanho=1900..1999;
VAR  anho :tipoanho;
      variable      TIPO
    
```

Ojo, a la variable se le puede asignar un valor pero a los tipos NO.



- Razones del uso de distintas formas de definir una variable:

1. Reutilizar código.

Se le puede poner un TYPE a muchas variables y a la hora de hacer un cambio es mucho más cómodo cambiar el rango una sola vez, que muchas veces en cada una de las variables que utilicen el mismo rango.

2. Hay variables que sólo admiten declaración explícita que son:

- a) Declaración de parámetros formales.
- b) Declaración de tipos de valores devueltos por una función.

```
FUNCTION cod_año(año:1900..1999):'A'..'Z';    ← MAL
```

```
FUNCTION cod_año(año:tipoaño):tipomayuscula;  
                    TIPO                TIPO
```

10.2 Subrangos

Permite al usuario definir un "rango" de valores sobre un tipo base dando el extremo superior e inferior. Se define a partir de un ordinal estableciendo unos límites inferior y superior para el tipo subrango.

Declaración:

```
Type  
    Subrango =liminf .. limsup ;  
Var  
    s : Subrango ;
```

Los tipos bases son los indefinidos y enumeraciones pero los reales no, porque tienen que establecer una biyección con los naturales. Las variables de un tipo subrango admiten las mismas operaciones que el ordinal del cual proceden. También resulta posible asignar a una variable de tipo subrango otra que ha sido declarada como perteneciente al tipo ordinal del cual se deriva el subrango. Los subrangos se utilizan para dar una mayor legibilidad a los programas.

- Hay dos formas de definir rangos.
 - Implícita: VAR x:1900..1999; y:'A'..'Z'; E:False..True; ← Ojo.
 - Explícita: TYPE tipovocal=('a'..'u');
VAR x:tipovocal;



Su utilidad consiste en el chequeo automático de rangos, es decir, cuando se sale del rango sale un mensaje de error y se para el programa.

10.3 Enumeraciones

Permiten definir al usuario tipos de escalares no numéricos (tienen que estar ordenados, sólo funcionan internamente y cada valor es indivisible).

- Hay dos formas de declarar las enumeraciones:
 - Implícita: `VAR pareja:(sota,as); {sota<as}`
 - Explícita: `TYPE par=(sota,as);
VAR pareja:par;`
- Con las enumeraciones se pueden utilizar unas funciones del Pascal:
 - a) Ord: devuelve la posición de un valor en la ordenación de un tipo de datos ordinal.
 - b) Pred: devuelve el predecesor único de un valor de un tipo de datos ordinal.
 - c) Succ: devuelve el sucesor único de un valor de un tipo de datos ordinal.
 - d) Chr: devuelve el carácter de una posición ordinal.

Ejemplo:

`Ord('B') → 66; ord(false) → 0; succ('B') → 'C'; pred('A') → @;`

Las variables de tipo enumerado tienen un funcionamiento interno, es decir, no es posible leer directamente valores de tipo enumerado ni desde teclado, ni desde un archivo de texto. Tampoco se pueden escribir en pantalla ni en archivos de texto (secuenciales) variables de este tipo. Solamente se pueden utilizar para manipulación interna de datos.

Podemos enumerar los días de la semana pero no podemos introducir o escribir un valor definido directamente en el programa sino de una manera codificada, traduciéndola después a uno de los valores definidos por el usuario. Para esto se suele utilizar la sentencia CASE-OF. La única excepción son los valores booleanos que pueden imprimirse.

El valor ordinal de un dato de este tipo se corresponde con la posición del identificador en la declaración del tipo, y el 1º tiene valor ordinal 0. Las funciones Pred y Succ quedan indefinidas cuando se trata del 1º o último identificador de la lista respectivamente.



Ejemplo:

```

Type
  Marcas = (seat, fiat, renault, citroen) ;
Var
  m : Marcas ;
  k : 4 .. 7; (* auxiliar *)
Begin
  Writeln('del 0 al 3')
  Readln(m);
  Case k of (*no imprime nada en pantalla con write*)
    4 : m:=seat ;
    5 : m:=fiat ;
    6 : m:=renault ;
    7 : m:=citroen ;
  End;
  Case m of (*Valores de m, aunque se llamen seat...*)
    0 : Writeln('seat') ;
    1 :Writeln('fiat') ;
    2 :Writeln('renault') ;
    3 :Writeln('citroen') ;
  End;
End.

```

10.4 Vectores o arrays. (**VIP**)

El ejemplo más típico es el de una cadena de caracteres que es una combinación de componentes de tipo char.

Cualquier vector en el que en cada elemento contenga el mismo tipo de datos son homogéneos y si en cada campo del vector hay distintos tipos de datos es heterogéneo.

Ejemplo:

Nombre	DNI	Teléfono
--------	-----	----------

Nombre	Nombre	Nombre
DNI	DNI	DNI
Teléfono	Teléfono	Teléfono

El primero es heterogéneo y el segundo homogéneo.

10.4.1 El Array Unidimensional.

Está formado por una combinación de n elementos del mismo tipo.

'a'	'b'	'n'
-----	-----	----	----	-----



Se llama componente a la información de cada casilla y el tipo de la información que en ella se le puede meter es toda variable que se pueda definir.

Se llama índice a la posición que ocupa cada casilla o componente en el array (1..n) y el tipo índice tiene que ser de tipo escalar numerable y finito (char, integer, boolean, subrangos, enumeraciones).

`nombre:array [1..5] of integer ;`
tipo índice
tipo del componente

1945	711	1977	2026	1492
------	-----	------	------	------

Hay dos formas de definir un array:

- ✓ Implícita: `VAR nombre : array [1..5] of integer;`
- ✓ Explícita: `TYPE tipomonbre : array [1..5] of integer;`
 `VAR nombre,n1,n2: tiponombre;`

Para leer, escribir los arrays se utiliza el bucle FOR (son estructurados, no se pueden leer o escribir directamente por teclado):

```
FOR i:=1 TO 5 DO READ(nombre[i]);
FOR i:=1 TO 5 DO WRITELN(nombre[i]);
```

Para asignación se procede de la siguiente manera: se indica el nombre del array seguido entre corchetes de la posición que queremos asignar un valor y después la notación habitual.

```
Nombre[3]:=1977;
Nombre[4]:=1492+Nombre[3];
```

NOTA: la diferencia entre nombre[i] y nombre(i) es que el primero es una variable de tipo array y el segundo es una llamada a un procedimiento o función con i como argumento.

Ejemplos de programas con arrays unidimensionales:

```
Program sueldo_semanal (input,output);
const max=5000000;
type dinero=1..max;
   tipodia=(lunes,martes,miercoles,jueves,viernes);
   tiposueldo=array[tipodia] of dinero;(*Patrón de array*)

var sueldo:tiposueldo;
    sueldo2:tiposueldo;
    sueldototal:integer;
    i:tipodia; (*Para utilizar en el bucle FOR, mismo tipo que el array*)
```




```
(*****
TIPOSUELDO es un patrón de vector para un grupo de 5 componentes cuyo
tipo es un subrango llamado DINERO. El tipo de índice es una enumeración
de los días de la semana. Cada componente del vector puede tener un valor
entero entre 1 y max.
SUELDO es el vector definido para el programa de tipo TIPOSUELDO.
*****)
begin (*pp*)
sueldototal:=0;
for i:=lunes to viernes do readln(sueldo[i]);
for i:=lunes to viernes do write(' ', sueldo[i]);
for i:=lunes to viernes do sueldototal:=sueldototal+sueldo[i];
sueldo2:=sueldo; (*un array puede ser asignado por otro del mismo tipo*)
writeln;
writeln(sueldototal);
(*****
En el programa leemos el sueldo que recibe cada día, lo escribimos en
pantalla y después sumamos todo lo que gana a la semana.
*****)
end.
```

```
Program producto_escalar (input,output);
const dim=3;
type tipodim=1..dim; (*subrango*)
tipovector=array[tipodim] of real;
var v1,v2:tipovector;
procedure leer( var x:tipovector);
var i:tipodim;
begin
for i:=1 to dim do read(x[i]);
end;
function pe (x,y:tipovector):real;
var aux:real;
i: tipodim;
begin
aux:=0;
for i:=1 to dim do aux:=aux+x[i]*y[i];
pe:=aux;
end;
(*****
Se definieron dos vectores reales, un procedimiento para leerlos y una
función que calcula el producto escalar.
*****)
begin
leer(v1);
leer(v2);
writeln(pe(v1,v2):6:2);
end.
```

```
Program cuenta_minusculas (input,output);
(*****
Contar con que frecuencia salen las letras minusculas en un texto de
50 caracteres e imprimirlo en pantalla.
*****)
type tipoindice='a'..'z';
tipovector=array[tipoindice] of integer;

var i:tipoindice; ch:char;
```

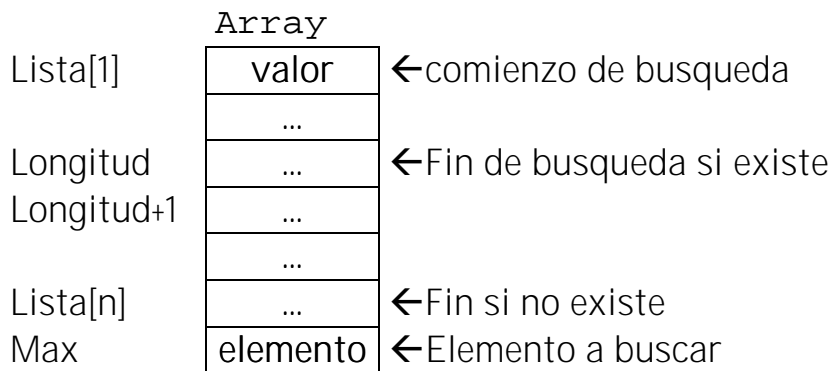


```

k:1..50;
cadena:tipovector;
(*****
Aunque sean iguales, no es lo mismo a la hora de dar el error: ← ojo
for i:='a' to 'z' do readln(cadena[i]); que
for ch:='a' to 'Z' do readln(cadena[i]);
Los dos tienen su sintaxis correcta y compilan pero si están mal, el
primero da el error en la compilación y el segundo compila y lo da en
la ejecución porque no hay posiciones en el vector para esos valores;
lo que hace el programa mucho menos elegante y pobre.
*****
(*****
(*****
El programa se basa en crear un vector del tamaño de las letras del
abecedario (índice) y en él ponemos un contador para cada letra
*****
begin
for i:='a' to 'z' do cadena[i]:=0; (*contador a 0*)
for k:=1 to 50 do
begin
read(ch);
if ch in ['a'..'z'] then cadena[ch]:=cadena[ch]+1;
end;
for i:='a' to 'z' do writeln(i,':', cadena[i]/50:4:2, ' veces por uno.');
```

Ejemplo 1 de uso de Arrays: búsqueda secuencial de un elemento:

Buscamos un elemento entre una lista aleatoria introducida y nos dice si el valor se encuentra en ella o no. Y para ello utilizamos un array como se dice a continuación.





El programa es:

```
const max= 7;

type tipoelemento=1..5000;

    tipolista=array [1..max] of tipoelemento;

var elemento:tipoelemento;
    lista:tipolista;
    longitud,indice:integer;
    encontrado:boolean;
procedure Buscar(Lista: (*array en el que se busca*)
                 tipolista;
                 Elemento: (*valor que se busca*)
                 tipoelemento;
                 Longitud: (* Tamaño de la lista*)
                 integer;
                 var indice: (*posicion del valor si esta*)
                 integer;
                 var encontrado: (*verdad si esta el valor*)
                 boolean);

(*****
Se busca en la lista un elemento. Si se encuentra elemento, encontrado se
pone a Verdad e indice da su posición. Si no, encontrado da falso
e indice es longitud+1. Pag 486.
*****)
begin (*buscar*)
Encontrado:=False;
Indice:=0;
while (indice<=longitud-1)and(encontrado=false) do (*hasta 5 y false*)
    begin
        indice:=indice+1;
        if elemento = lista[indice] then encontrado:=true;
        end;
if encontrado=false then indice:=indice+1;
end;

begin (*pp*)
indice:=0;encontrado:=false;
Writeln('Introduce la lista (6)');
for indice:=1 to 6 do Readln(lista[indice]);
writeln('Introduce elemento a buscar');
Readln(lista[7]);
buscar(lista,lista[7],6,indice,encontrado);
Writeln(encontrado,' y el valor es:',lista[indice]:4);
end.
```



Ejemplo 2 de uso de Arrays: ordenación de una lista:

Dada una lista, hacemos un programa que los ordena:

1492	711	711	711	711
711	1492	1312	1312	1312
1977	1977	1977	1492	1492
1312	1312	1492	1977	1945
1945	1945	1945	1945	1977

```

const max=5;

type tipoelemento=1..5000;

    tipolista=array[1..max] of tipoelemento;

var lista:tipolista;
    longitud:integer;

procedure ordenacion(var lista: (*array a buscar*)
                    tipolista;
                    longitud: (*numero de elementos del array*)
                    integer);

( *****
Ordenacion de una lista en orden ascendente. Pag 491
Una lista no ordenada con longitud valores es la entrada.
La misma lista pero ordenada en orden ascendente es la salida
***** )

var temp:tipoelemento; (* variable temporal*)
contpasos, (*var de control del bucle*)
contposiciones, (* var de control del bucle*)
minindice: (* indice del minimo hasta ese momento*)
    integer;

begin
for contpasos:=1 to longitud-1 do
( *****
lista[1]..lista[contpasos-1] esta ya ordenado en orden ascendente
y contpasos-1 es el indice del valor mayor ordenado y menor que
de los desordenados
***** )
    begin
        minindice:=contpasos;
( *****
Encuentra el indice de la componente mas pequeña de la lista
***** )
        for contposiciones:=contpasos+1 to longitud do
            if lista[contposiciones]<lista[minindice] then
                minindice:=contposiciones;
( *****
Intercambia lista[indice] y lista[contpasos]
***** )
        temp:=lista[minindice];

```



```
lista[minindice]:=lista[contpasos];
lista[contpasos]:=temp;
end;
end;

begin
Writeln('Introduce ',max:2,' elementos de 1 a 5000.');
```

```
for longitud:=1 to max do Readln(lista[longitud]);
longitud:=5;
ordenacion(lista,longitud);
writeln;writeln;
for longitud:=1 to max do Writeln(lista[longitud]);
end.
```

Ejemplo 3 de uso de Arrays: búsqueda secuencial de un elemento en una lista ordenada:

```
const max= 7;

type tipoelemento=1..5000;
      tipolista=array [1..max] of tipoelemento;

var elemento:tipoelemento;
    lista:tipolista;
    longitud,indice:integer;
    encontrado:boolean;

procedure Busca_ord(Lista: (*array en el que se busca*)
                    tipolista;
                    Elemento: (*valor que se busca, está en longitud+1*)
                    tipoelemento;
                    Longitud: (* Tamaño de la lista*)
                    integer;
                    var indice: (*posicion del valor si esta*)
                    integer;
                    var encontrado: (*verdad si esta el valor*)
                    boolean);

(*****
Se busca en la lista un elemento. Si se encuentra elemento, encontrado se pone a
Verdad e indice da su posición. Si no, encontrado da falso.
La lista esta ordenada en orden ascendente y longitud menor que max
Pag 493.
*****)

var stop:boolean; (*verdad cuando termina la búsqueda*)

begin
indice:=1; stop:=false;
lista[longitud+1]:=elemento;
(*Se sale del bucle cuando se encuentra el valor o no existe*)
while not stop do
(*elemento no esta en lista[1]..lista[indice-1]*)
if elemento > lista[indice]
(*elemento no esta en lista[1]..lista[indice]*)
then indice:=indice+1
else (*no se encuentra o no esta en la lista*)
stop:=true;
```



```
(*Determina si se ha encontrado o no el elemento*)
encontrado:=(indice<>longitud)and(elemento=lista[indice]);
end;

begin (*pp*)
Writeln('Introduce la lista ordenada ascendentemente (7)');
for indice:=1 to 6 do Readln(lista[indice]);
writeln('Introduce elemento a buscar');
Readln(lista[7]);
Busca_ord(lista,lista[7],6,indice,encontrado);
Writeln(encontrado);
end.
```

Ejemplo 5 de uso de Arrays: búsqueda binaria en lista ordenada:

```
const max= 7;

type tipoelemento=1..5000;
      tipolista=array [1..max] of tipoelemento;

var elemento:tipoelemento;
    lista:tipolista;
    longitud,indice:integer;
    encontrado:boolean;

procedure Busca_binaria(Lista: (*array en el que se busca*)
                        tipolista;
                        Elemento: (*valor que se busca*)
                        tipoelemento;
                        Longitud: (* Tamaño de la lista*)
                        integer;
                        var indice: (*posicion del valor si esta*)
                        integer;
                        var encontrado: (*verdad si esta el valor*)
                        boolean);

(* *****
Se busca en la lista un elemento dividiendo la lista ordenada por la mitad,
determina si esta en la mitad superior o inferior y repite la operacion
sucesivamente hasta que lo encuentra o no.
Es un procedimiento mas rapido que en la anterior porque
la lista se corta por la mitad cada vez que se realiza una iteración.
***** *)

var Primero, (*limite inf de la lista*)
    Ultimo, (*limite superior de las lista*)
    Mitad: (*indice mitad*)
        integer;

begin
Primero:=1;Ultimo:=Longitud;encontrado:=false;
while (ultimo>=primero)and(encontrado=false) do
begin
mitad:=(primero+ultimo) div 2;
if elemento < lista[mitad]
(*Elemento no esta en la lista superior*)
then ultimo:=mitad-1
else if elemento > lista[mitad]
(*Elemento no esta en la lista inferior*)
```



```
        then primero:=mitad+1
        else encontrado:=true;
    end;
indice:=mitad
end;

begin
writeln('Mete ',max,' elementos ordenados de mayor a menor.');
```

```
for longitud:=1 to max do Readln(lista[longitud]);
Writeln('Mete el elemento a buscar');
Read(elemento);
busca_binaria(lista,elemento,max,indice,encontrado);
writeln(encontrado);
if encontrado=true then Writeln(lista[indice]);
end.
```

10.4.2 Arrays Bidimensionales

Son una colección de componentes, todas del mismo tipo, estructuradas en dos dimensiones (como matrices). A cada componente se le accede mediante un par de índices que representan la posición de la componente dentro de cada dimensión.

Un array multidimensional se define exactamente igual que uno unidimensional, sólo que deben darse tantos tipos índices como dimensión tenga el array. Además de todo eso, se puede definir a partir de arrays unidimensionales. Las siguientes formas que se ponen a continuación son equivalentes:

- fila=array [1..10] of integer;
matriz=array [1..3] of fila;
- matriz=array[1..3] of array [1..10] of integer;
- matriz=array [1..3,1..10] of integer;

Las siguientes formas de acceder a cada componente es igual:

matriz[2][4]≡matriz[2,4]

Ejemplo: Procedimineto para multiplicar matrices.

```
(***** A(n,m) x B(m,v) = C(n,v) *****)
const n=3; m=5; v=4;
```

```
type MA=array [1..n,1..m] of integer;
      MB=array [1..m,1..v] of integer;
      MC=array [1..n,1..v] of integer;
```

```
var A:MA;B:MB;C:MC;
```

```
Procedure multiplica(A:MA;B:MB; var C:MC);
var i,j,k,suma:integer;
begin
```



```

for i:=1 to n do      (*Para operar con elementos se utilizan bucles anidados*)
  for j:=1 to v do
    begin
      suma:=0;
      for k:=1 to m do
        suma:=suma+A[i,k]*B[k,j];
      C[i,j]:=suma;
    end;
  end;
end;

```

Ejemplo: Contar votos por distritos de varios candidatos.

```

const max=100000000;
type rangovotos=0..max;
   rangodistritos=(andalucia,asturias,galicia,madrid);
   rangocandidatos=(PP,PSOE,IU);
   contvotos=array[rangodistritos,rangocandidatos] of rangovotos;

var votos:contvotos;

procedure def_distrito(distritos:rangodistritos);
(******Permite imprimir los distritos******)
begin
  case distritos of
    andalucia:write('Andalucia');
    asturias:write('Asturias');
    galicia:write('Galicia');
    madrid:write('Madrid');
  end;
end;

procedure def_candidato(candidatos:rangocandidatos);
(******Permite imprimir los candidatos******)
begin
  case candidatos of
    PP:write(' PP ');
    PSOE:write(' PSOE ');
    IU:write(' IU ');
  end;
end;

procedure votoscero(var votos:contvotos);
(******
   Pon el array de votos a 0 mediante bucles anidados
******)
var distritos:rangodistritos; candidatos:rangocandidatos;

begin
  for distritos:=andalucia to madrid do
    for candidatos:=PP to IU do votos[distritos,candidatos]:=0;
  end;
end;

procedure meter_votos(var votos:contvotos);
(******
   Metemos valores en cada casilla de votos
******)
var distritos:rangodistritos; candidatos:rangocandidatos;

begin
  for distritos:=andalucia to madrid do

```




```
for candidatos:=PP to IU do
begin
write('Numero de votos para ');
def_candidato(candidatos); write(' en ');
def_distrito(distritos); write(' : ');
Readln(votos[distritos,candidatos]);writeln;
end;
end;

procedure contar_votos(votos:contvotos);
(*****
Sumamos los votos de cada candidato en un array unidimensional
e imprimimos los votos totales y por distrito en pantalla
*****)
var votos_totales:array [rangocandidatos] of rangovotos;
    distritos:rangodistritos; candidatos:rangocandidatos;

begin
for candidatos:=PP to IU do votos_totales[candidatos]:=0; (*a cero*)
for distritos:=andalucia to madrid do (*suma de votos totales*)
for candidatos:=PP to IU do
votos_totales[candidatos]:=votos[distritos,candidatos]+votos_totales[candidatos];
Writeln('*** Los votos totales son: '); (*Resultados*)
for candidatos:=PP to IU do
begin
def_candidato(candidatos);
Writeln(' : ', votos_totales[candidatos]:7,' votos. ');
end;
writeln;
Writeln('Los votos obtenidos por distritos : ');
for distritos:=andalucia to madrid do
for candidatos:=PP to IU do
begin
Write(' El candidato ');
def_candidato(candidatos); write(' en ');
def_distrito(distritos); write(' tuvo ');
writeln(votos[distritos,candidatos]:7,' votos. ');
end;
end;

begin (*pp*)
(*****
En este programa se cuenta los votos de cierto numero de candidatos en
cierto numero de distritos en unas elecciones.
Para contar los votos se crea un array bidimensional cuyos indices estan
definidos por los candidatos y los distritos que concurren a las elecciones.
*****)
votoscero(votos);
meter_votos(votos);
contar_votos(votos);
end.
```



10.5 Conjuntos.

Son un tipo estructurado de datos que pueden ser comparados a arrays unidimensionales booleanos: cada *posible* componente del conjunto es el índice y lo que indica que pertenece al conjunto es la componente de cada índice (true: pertenece, false: no pertenece).

```
TYPE vocales=('a','e','i','o','u');
VAR conjunto:array [vocales] of boolean;
```

Cuando se define un conjunto se tienen que establecer todos sus *posibles* elementos y dentro del programa asignamos que elementos de los anteriores pertenecen al conjunto.

Se definen así:

```
VAR <nombre_variable> : set of <tipo_base>;
VAR vocales : set of ('a','e','i','o','u');
```

Vocales es un conjunto y sus *posibles* elementos son las vocales.

Se asignan así:

```
vocales:=[]; {conjunto vacio}
vocales=['a']; { con un elemento}
vocales=['a','e']; {con dos elementos}
vocales:=['i']; {conjunto con el elemnto i}
```

Estas operaciones de asignación no son acumulativas, es decir, los elementos del conjunto son los de la última asignación. Para realizar operaciones con elementos y conjuntos existen unos operadores:

Union	+
Intersección	*
Diferencia	-
Igualdad	=
Incluido	<=
Incluye	>=

La operación **cardinal** (decir el número de elementos comunes a dos conjuntos) no existe en Pascal standard y se utiliza la expresión:

<elemento> in <conjunto> que da TRUE o FALSE.



Ejemplo de uso de conjuntos : La Bonoloto.

```
(*****
Programa que compara una apuesta de la bonoloto con la combinación ganadora
y da como resultado el número de aciertos en cada boleto por conjuntos.
*****)
const max=49;
      numap=5;
      cruces=6;

type numeros=1..max; (*rango de números en cada boleto*)
      apuesta=1..cruces; (*rango de cruces en cada boleto*)
      tipoboleto=set of numeros; (*conjunto de números en cada boleto*)
      tipoboletos=array [1..numap] of tipoboleto; (*todas las apuestas*)

var boletos:tipoboletos; (*toda la apuesta*)
      ganadora:tipoboleto; (*combinación ganadora*)
      j:1..7; (*aux*)

procedure meter_datos(var boleto:tipoboleto);
(*****
Mete las combinaciones en cada boleto de la apuesta y rellena la combinación
ganadora para poder compararla después.
*****)
var i:apuesta;
      k:numeros;

begin
boleto:=[];
if j<>7 then Writeln('Numeros para el boleto n§',j:1,' :')
      else Writeln('Numeros del boleto ganador:');
for i:=1 to cruces do
begin
Readln(k);
boleto:=boleto+[k]; (*se añaden elementos al conjunto*)
end;
end;

procedure contar_aciertos(boleto, ganadora:tipoboleto);
(*****
Compara cada boleto con la combinación ganadora y establece el número
de aciertos en cada boleto.
*****)
var k:numeros; temp:tipoboleto;
      i:0..6;

begin
i:=0;temp:=[];writeln;
temp:=ganadora*boleto;(*Intersección de los dos conjuntos*)
for k:=1 to max do if k in temp then i:=i+1; (*Cardinal de los conjuntos (VIP)*)
Write('El boleto n§',j:1,' ha tenido ',i:1,' aciertos. ');
Case i of
0:Writeln(' Paquete');
1:Writeln('Estas más flojo que una tosa de Paco Clavel. ');
2:Writeln('Dedicate a otra cosa');
3:Writeln('Pse, bastante flojo. ');
4:Writeln('Solo da para pipas. ');
5:Writeln('Casi, casi. ');
6:Writeln('Has acertado de pleno, recuerda que soy tu amigo. ');
end;
```



```
end;

begin (*pp*)
(*****
Una apuesta entera consta de un maximo de 6 boletos. Definimos un array
unidimensional en el que cada componente sea un conjunto de TIPOBOLETO.
Comparamos cada componente con la combinacion ganadora.
*****)
for j:=1 to numap do boletos[j]:=[]; ganadora:=[];
for j:=1 to numap do meter_datos(boletos[j]); j:=7;
meter_datos(ganadora);
for j:=1 to numap do contar_aciertos(boletos[j],ganadora);
end.
```

El tipo base de un conjunto ha de ser un ordinal(tipo escalar) y cada elemento es único en el conjunto. Los valores ordinales de los elementos han de estar entre 0..255: char, rangos de enteros, booleanos,enumeraciones...

Las siguientes declaraciones son ilegales :

```
Conjunto = SET of integer ;
Conjunto = SET of 1997..2000 ;
```

Las siguientes declaraciones son legales :

```
Conjunto = SET of char ;
Conjunto =SET of 'A'..'Z' ;
```

Aunque para facilitar la legibilidad se suele escribir :

```
Type
Letras = 'A'.. 'Z' ;
Conjunto = SET of Letras ;
```

NOTA: Podemos simular conjuntos mediante arrays de booleanos :

```
ConjuntoArr = ARRAY [1..6] of boolean ; { aprox. como Conjunto = SET of 1..6 ;}
ConjuntoArr1 = ARRAY [1..6] of boolean ;
ConjuntoArr2 = ARRAY [1..6] of boolean ;
```

Para introducir un elemento :

```
ConjuntoArr[5] :=True ; {se mete el elemento 5 en el conjunto}
```

Conjunto vacio: (* como si fuera ConjuntoArr:=[] *)

```
for k:=1 to 6 do conjuntoArr[k]:=False;
```

Intersección de conjuntos: (* ConjuntoArr:=ConjuntoArr1 * ConjuntoArr2 *)

```
for k:=1 to 6 do ConjuntoArr[k]:=ConjuntoArr1[k]ANDConjuntoArr2[k];
```

En los conjuntos, no puede haber repeticiones en los elementos (solo habrá un elemento de cada).



```
Type
  Letras='A'..'Z' ;
  Conjunto=SET of Letras ;
  Arr= ARRAY['A'..'Z'] of boolean ;

Var
  c :conjunto ;
  a :arr ;
  i :integer ;
  car :char ;

Begin (*pp*)

  {usando conjuntos}
  c :=[ ] ; { se asigna conjunto vacío, es obligatorio}
  Readln (car) ;
  c :=c+[car] ; {elemento c UNION conjunto car}{[car] = conjunto del
contenido de car}

  {usando arrays}
  For car :='A'..'Z' do ;
    a[car] :=False ; {inicialización del conjunto}
  Readln[car] ;
  a[car] :=True ; {introducido elemento en conjunto}
End.
```

Para ver si un elemento está en un conjunto :

```
{usando conjuntos}
For car :='A'..'Z' do
  If car IN c then {comprueba que esté en el conjunto}
    Writeln(car) ;

{usando arrays}
For car :='A'..'Z' do
  If a[car] then {comprueba que esté en el conjunto}
    Writeln(car) ;
```

Ejemplos de Tratamiento y Operaciones con conjuntos :

- Definición de tipos :

```
Const
  elementos=7;{número máximo de elementos del conjunto}

Type
  tipobase=char;
  Conj=SET of tipobase;{tipobase=simple y ordinal con valores ordinales
entre 0..255}
```

- Declaración de variables :

```
Var
  c :Conj;8
  car :char ;{tipobase}{variable}
  k:1..elementos;

Begin
  {hay que INICIALIZAR siempre}
  c :=[ ] ;{inicializamos a conjunto vacío o c :=['A'..'Z'] ; inicializa a
conjunto universal }
```

⁸ También es posible la forma : c : SET of 'A'..'Z' ; pero tiene poca utilidad, y en programación modular, para el paso por variable, se necesita haber declarado el tipo.



```
k:=1;
```

- ASIGNACIÓN :⁹

```
c := ['A'..'D'] ;
{también se puede así → c := ['A'..'D', '5', car], pero se utiliza poco}10
```

- INTRODUCCIÓN DE ELEMENTOS :

```
{leemos variable de tipobase}
Repeat
  Readln(car) ; k:=k+1;

  c :=c + [car]11 ;12           {car → Elemento}
Until (car>='A') AND (car<='Z')AND(k=elementos);
```

Una vez introducidos los datos podemos visualizarlos.

- VISUALIZACIÓN DE ELEMENTOS :

Se puede realizar mediante un bucle for :

```
For car := 'A' to 'Z' do
  If car IN c then 13
    Writeln(car) ;
```

Aunque es más conveniente un while (por si sólo hay 2 elementos en el conjunto) :

```
car := 'A' ;
caux :=c ;{para no vaciar el conjunto inicial lo metemos en una variable
auxiliar}
While (car<='Z') AND (caux < > [ ] then
Begin
  If car IN caux then
  Begin
    Writeln(car) ;
    caux :=caux - [car] ; {diferencia de conjuntos}
  End ;
  If caux < > [ ] then
    car := Succ(car) ;           {siguiente elemento}
End ;
{como car es de tipo char, habrá sucesor. Si fuese de tipo enumerado, podría no
haberlo}
```

⁹ Para dar valores a variables de tipo conjunto (que no pueden ser leídos desde teclado o escritos por pantalla), debemos hacerlo elemento a elemento, para lo que usamos la variable tipobase.

¹⁰ Las variables han de ser de tipo base.

¹¹ [car] → Conjunto con el elemento.

¹² Unión entre conjuntos.

¹³ **el operador IN admite elemento de distinto tipo.** Lo usamos para saber si car pertenece a c.



Operadores con conjuntos :

- + Unión de conjuntos.
- Diferencia de conjuntos.
- * Intersección de conjuntos.
- IN Pertenencia de elemento en un conjunto.
- >= Un conjunto incluye a otro conjunto.
- <= Un conjunto es incluido en otro conjunto.

Prioridad de operadores :

- NOT
- /, *, AND
- -, +, OR
- >, <=, <, >=, =, <>, IN

Simulación de conjuntos :

- Mediante conjunto de enumerados (ya que un conjunto no puede tener cadenas o más de 255 elementos).
- Mediante arrays .

10.6 Los Registros.

Es un tipo de variable estructurada que admite almacenar datos de distinto tipo. Como todas las variables estructuradas admiten asignaciones completas y accesos puntuales a un solo tipo de dato del registro.

A.- Declaración

Esta estructura se declara en tipos con la sintaxis siguiente:

```
TYPE  
Identificador_del_Tipo = RECORD  
    lista de variables con sus tipos(Campos)14  
END;
```

Ejemplo:

```
TYPE  
    TipoFecha = Record (*1º Registro*)  
        dia: integer;  
        mes:1..12;  
        año:integer;  
    END;
```

¹⁴ Campos: Cada variable contenida en un Registro.



```

TipoMeses = Array [1 .. 12] of integer;
  TipoEmpleado = Record      (*2º Registro*)
    nombre: Array [1 .. 30] of char;
    apellido1: Array [1 .. 30] of char;
    apellido2: Array [1 .. 30] of char;
    fechaingreso: TipoFecha;
    sueldo: integer;
    trabajosrealizados: TipoMeses;
  END;

  TipoEmpleados = Array [1 .. 100] of tipoempleado;

VAR
  Meses: TipoMeses;
  DiaActual: TipoFecha;
  Empleado : TipoEmpleado ;
  TodosEmpleados : TipoEmplaados;

```

En este ejemplo se ve como se puede declarar un tipo simple de estructura como es la de la variable "DiaActual" que es una estructura que contiene 3 datos enteros que son el día, mes y año.

Una estructura un poco más compleja es la que tiene la variable "Empleado", ya que tiene dentro de si otra estructura del tipo "TipoFecha" y también tiene un array dentro del tipo "TipoMeses" que tiene 12 posiciones numéricas dentro.

Mas Difícil todavía, variable ¡ " TodosEmpleados " ! que puede tener almacenados los datos de 100 personas de la forma de la estructura "TipoEmpleados".

B.- Referenciación de posiciones de una estructura:

Esto se hace mediante el operador "." (Punto) de la siguiente forma:

IdentificadorVariable . ParteEsturctura [. ParteEstructura . etc..]

Ejemplos (Utilizamos las estructuras anteriormente declaradas y haremos un repaso a los arrays)

Variable "DiaActual" (es una estructura)

Si queremos referenciar el día se pone: DiaActual.dia

Si queremos almacenar que estamos en el mes 12 : DiaActual.mes:=12;

Para imprimir el día se pone:

Write(DiaActual.dia, "/", DiaActual.mes, "/", DiaActual.año);



Si quieres pasar la estructura entera a un procedimiento se pone:

```
PonerLaFecha( DiaActual );  
Procedimiento PonerLaFecha( F : TipoFecha );
```

Si quieres pasar un elemento a un procedimiento se pone:

```
Día( DiaActual.dia );  
Procedimiento Día( d : entero );
```

Variable "Meses" es un array del tipo "TipoMeses"

Para Referenciar a cualquier posición se pone: Meses[número]

Donde número tiene un rango de 1 a 12

Variable "Empleado" tiene una estructura de tipo "TipoEmpleado"

-Si queremos saber el nombre o apellidos y sueldo se pone:

```
Empleado.nombre ⇒ que es de tipo "Cadena de caracteres"  
Empleado.apellido1 ⇒ que es del mismo tipo  
Empleado.sueldo ⇒ que es de tipo "Entero"
```

-Si queremos saber datos de la fecha de ingreso se pone:

```
Empleado.fechaingreso.dia ⇒ de tipo "Entero"  
Empleado.fechaingreso.mes ⇒ del mismo tipo  
Empleado.fechaingreso ⇒ Esto sería de tipo "TipoFecha"
```

- Si queremos saber los trabajos realizados en algún mes se pone:

```
Empleado.trabajosrealizados[ número ] ⇒ de tipo "entero"  
Donde número tiene un rango de 1 a 12  
Empleado.trabajosrealizados ⇒ es de tipo "TipoMeses"
```

Variable "TodosEmpleados" de tipo "TipoEmpleados"

-Si quieres referenciar todos los datos de un empleado se pone:

```
TodosEmpleados[ número ] ⇒ de tipo "TipoEmpleado"
```

- Todo lo demás se hace igual con lo de antes

- Si quieres saber el sueldo del trabajador 30 se pone:

```
TodosEmpleados[ 30 ].sueldo ⇒ de "entero"
```

- Si quieres saber el día de ingreso del trabajador 50

```
TodosEmpleados[ 50 ].fechaingreso.dia
```

-Si quieres saber los trabajos del mes 1 del trabajador 20

```
TodosEmpleados[ 20 ].trabajosrealizados[ 1 ]
```



-Si quieres almacenar 25 trabajos en el mes 7 del empleado 9 se pone:

```
TodosEmpleados[9].TrabajosRealizados[7]:=25;
```

Este tipo de variable admite la sentencia WITH que permite acceder a Campos sin poner el nombre del registro cada vez que se accede a Campos de un mismo registro.

Ejemplo:

```
With DiaActual do
  Begin
    Año:=1998;
    Readln(Dia);
    Writeln(Año);
  End;
```

Hay que evitar que una variable tenga el mismo nombre que un campo de un registro para evitar confusiones.

Como se ve para acceder a campos se hace por sus nombres, no por índices.

Ejemplo de uso de Registros : Comparar dos fechas.

```
type Tipofecha = Record
  dia:1..31;
  mes:1..12;
  anyo:0..2100;
End;
Relacion=(Antes,Igual,Despues);

(*****
  Compara Fecha1 con Fecha2 campo a campo
  *****)

function Comparar( fecha1, fecha2: Tipofecha):Relacion;

Begin
  if fecha1.anyo < fecha2.anyo
  then
    comparar:=antes
  else
    if fecha1.anyo > fecha2.anyo
    then
      comparar:=despues
    else
      if fecha1.mes < fecha2.mes
      then
        comparar:=antes
      else
        if fecha1.mes > fecha2.mes
        then
          comparar:=despues
        else
```



```
if fecha1.dia < fecha2.dia
then
  comparar:=antes
else
  if fecha1.dia > fecha2.dia
  then
    comparar:=despues
  else
    comparar:=igual;
```

End;

10.7 Archivos

Los archivos son una colección secuencial de elementos del mismo tipo (\neq array), tratados como estructura de datos, almacenados en un dispositivo de almacenamiento externo permanente (físico), en la que los elementos se encuentran almacenados de forma secuencial.

Los archivos no pueden tener una asignación directa de la forma $a:=b$, hay que hacerlo de forma indirecta.

Tipos de Acceso a la información almacenada en un fichero:

- ✓ Secuencial: (viene de las cintas magnéticas).
- ✓ Aleatorio o directo: se accede directamente al dato mediante un índice o dirección.

En el Pascal standard sólo hay acceso secuencial cosa que no ocurre en el TurboPascal que admite los dos tipos de acceso.



Llegado a este punto conviene distinguir entre los dos clases de ficheros con los que se trabaja al programar:

- ✓ Fichero físico: información permanente y que no se borra terminar el programa (ej: fichero que se guarda en el disco).
- ✓ Fichero lógico: Copia del fichero físico en memoria y sólo existe durante la ejecución del programa.

A la hora de programar hay que vincular el fichero físico a un fichero lógico para que lo utilice el programa.

❖ Operaciones con ficheros:



- ✓ Declaración de la variable:


```
TYPE      Tipofichero = FILE OF <tipo_de_variable>;
VAR      fib : Tipofichero; (es la copia de un fichero del disco.)
```

- ✓ Vinculación del fichero físico con el lógico:

- ✓ En el Pascal standard:

```
PROGRAM prueba (input, output, alumnos, fib);15
VAR alumnos, fib : file of real;16
```

- ✓ En TurboPascal:

```
Program prueba (input, output);17
VAR x1, x2:file of real;
```

```
Begin
    Assign (x1, 'alumnos.dat');18
    Assign (x2, 'fib.dat');
```



γ Para sacar una copia de un fichero se saca cada elemento secuencialmente y se le añade de la misma forma al otro fichero.

γ Los tipos de elementos de un fichero pueden ser de cualquier tipo menos de tipo FILE.

❖ Operaciones sobre ficheros:

- ❖ Lectura de un fichero:

```
(1ª vez) RESET (fib);19
(n veces) READ(fib,<vble>);
```

- ❖ Escritura

```
(1ª vez) Rewrite(fib);20
(n veces) Write(fib,vble);
```

¹⁵ fichero/s físico/s sin extensión.

¹⁶ En VAR ficheros lógicos: tienen que tener el mismo nombre que el declarado en program.

¹⁷ es opcional ponerlos aquí (con extensión)..

¹⁸ se asigna un fichero lógico a fichero físico con extensión.

¹⁹ abre el fichero lógico en modo lectura y el puntero al cominezo de fichero.

²⁰ abre el fichero lógico en modo de escritura al cominezo del fichero y borra el contenido del fichero.



Ejemplo de declaración de un fichero como variable :

```
Program fichas_de_alumnos (input,output,Alumnos);  
  
TYPE      Tficha=Record  
           nombre,  
           apellido1,  
           apellido2: array [1..20] of char;  
           edad,  
           peso: integer;  
End;  
  
           FichFichas = FILE OF Tficha;  
VAR       Alumnos: FichFichas;
```

El VAX/Pascal posee la instrucción (open/close) que permite abrir ficheros de cualquier nombre y tipo. La función booleana que detecta si se ha alcanzado el final del fichero en modo lectura es EOF (End Of File).

Los ficheros pueden ser de dos tipos que veremos más adelante con más detalle:

- ✓ Binarios: ficheros compuestos de unos y ceros sin saltos de líneas. En este tipo de ficheros no existe Readln y Writeln porque no hay saltos de línea.
- ✓ Texto: La información sólo se puede leer y escribir de forma secuencial (acceso secuencial). Se consideran formados por una serie de líneas las cuales a su vez están formadas por una serie de caracteres. La longitud de éstas líneas no puede exceder 127 caracteres.

En Pascal, los archivos de texto vienen definidos como de tipo 'text'. Para Pascal, la pantalla y el teclado son archivos de tipo texto asociados con la E/S estándar del sistema operativo.

- Teclado : Input.
- Pantalla : Output.

El Pascal estándar obliga a especificar el tipo de archivo, pero Turbo Pascal no.



10.7.1 Tratamiento secuencial de ficheros:

El siguiente esquema muestra la secuencia de acciones al tratar cada elemento de un fichero.

```

Esquema de recorrido es
{ recorre una secuencia que puede ser vacía }
Reset (f)  { iniciar_tratamiento; }
Mientras not Eof(f) {fin_de_fichero} hacer
    { se han tratado todos los elementos que preceden al
    elemento en curso que no es el final }

Begin
Read (f, x) { leer elemento y posicionar en elemento_siguiete }
Tratar x; { tratar_elemento }
End;
Finmientras;
Finalizar_tratamiento;

```

Figura 18



γ Siempre hay que comprobar que el fichero no esté vacío y con el esquema anterior nos aseguramos de que así es. El siguiente esquema muestra el error de no comprobar de que el fichero esté vacío, además de no tratar el último elemento en el bucle.

```

esquema de recorrido del esquema equivocado (error grave) es
Reset (f)  { iniciar_tratamiento; }
Read(f,x) {Lee 1º elemento}
mientras not Eof(f) {fin_de_fichero} hacer
    { se han tratado todos los elementos que preceden al
    elemento en curso que no es el final }

Begin
Tratar x; { tratar_elemento }
Read (f, x) { leer elemento y posicionar en elemento_siguiete }
End;
finmientras;
Tratar x; { tratar último elemento }
Finalizar_tratamiento;

```

Figura 19

Los booleanos no imprimen en pantalla. Se puede leer o imprimir un registro entero en pantalla pero no sucede lo mismo cuando metemos los valores del registro por teclado. Los ficheros en procedimientos deben llevar VAR tanto si se modifican como si no.



Ejemplo de usos de archivos en un programa.

Crear una base de datos de alumnos con la nota de una asignatura. Se podrá crear una lista nueva, imprimir los datos de un alumno, la nota de un alumno o añadir información a fichero sin borrar lo que había.

```
program listado (input, output, alumnos,temporal);
(*****
Este programa gestiona una base de datos de alumnos que almacena
información en un registro el nº de matricula,Nombre,Apellidos,
Telefono,Nota de una asignatura y si entregó practicas.

Los registros se almacenan en un archivo (ALUMNOS.DAT).

El programa tiene la capacidad de crear,buscar,anyadir o ver
la nota.
*****
CONST Longnombre=20;
      max=7;

TYPE
  indiceNombre=1..LongNombre;
  tiponumero=array [1..max] of char;
  CadenaNombre= array [indicenombre] of char;
  TipoTelefono= Record
      CodigoArea: 0..999;
      Numero: tiponumero;
      End;      (*prefijo + numero*)

  Datos= Record
      Nombre:CadenaNombre;
      Apellido: Cadenanombre;
      telefono: Tipotelefono;
      matricula:integer;
      NotaTest:real;
      NotaP1:real;
      NotaP2:real;
      Practicas:boolean;
      end;
  ListaDatos= FILE of Datos;

VAR
  k,nmatricula:integer;
  alumnos, (*mismo nombre que el archivo real del disco duro sin ext.*)
  Temporal: (*archivo auxiliar*)
      ListaDatos;
(*****
  procedure leer_caracteres ( var cadena: Cadenanombre );
  (*Posibilita escribir caracteres en un array(termina en blanco)*)
  var b:integer; (*aux*)
  begin
    for b:=1 to Longnombre do cadena[b]:=' ';
    b:=1;
    repeat
      Read(cadena[b]);
```



```

        b:=b+1;
        until ((b=LongNombre)or(cadena[b-1]=' '));
    Writeln;
end;
(*****
procedure leer_telefono ( var cadena: tiponumero );
(*Escribe un numero en un array*)
var b:integer; (*aux*)
begin
    b:=1;
    for b:=1 to Longnombre do cadena[b]:=' ';
        repeat
            Read(cadena[b]);
            b:=b+1;
        until ((b=max)or(cadena[b-1]=' '));
    writeln;
end;

(*****
procedure altas (var f: listaDatos); (*El fichero alumnos se llama f aqui*)
(*Ojo: Crea el archivo alumnos.dat desde cero cada vez que se ejecuta!!!!*)
var alumno:Datos; (*Registro*)
    a:char; (*aux*)

begin
assign(f, 'alumnos.dat');    (*solo en Turbo Pascal*)
Rewrite(f); (*borra todo lo que habia y empieza a grabar*)
    with alumno do (*Dentro del Registro hacemos lo siguiente*)
        begin
            repeat
                Writeln('Nombre: '); leer_caracteres(nombre);
                Writeln('Apellido: '); leer_caracteres(Apellido);
                Writeln('Telefono: ');
                Writeln('Prefijo: '); Readln(telefono.CodigoArea);
                Writeln('Numero: '); leer_telefono(telefono.numero);
                Writeln('Nº de Matricula: '); Readln(Matricula);
                Writeln('Nota del Test: '); Readln(NotaTest);
                Writeln('Nota de 1º Problema: '); Readln(NotaP1);
                Writeln('Nota de 2º Problema: '); Readln(NotaP2);
                Writeln('entregó las Pr cticas? s/n '); Readln(a);
                if a='n' then practicas:=false
                    else practicas:=true;
                Write(f,alumno);
                Writeln('Desea meter a otro alumno? s/n'); Read(a);
            until a='n';
        end;
end;

(*****
procedure anyadir (var f,Temporal:ListaDatos);
var alumno:Datos;(*Registro*) a:char;(*aux*)
(*****
Como al grabar un archivo se borra todo lo anterior, creamos un archivo
temporal al que grabamos secuencialmente los datos y al finalizar el
volcado de datos continuamos grabando registros en el Temporal.
Cuando terminamos, hacemos el proceso inverso al archivo alumnos.dat
*****
begin
assign(f,'alumnos.dat');    (*sólo en Turbo Pascal*)
assign(temporal,'temporal.dat'); (*sólo en Turbo Pascal*)

```




```
Reset(f);
Rewrite(Temporal);
  repeat
    Read(f,alumno); Write(temporal,alumno);
  until Eof(f); (*hasta final de fichero de alumnos*)
with alumno do (*entonces sigue escribiendo registros en Temporal*)
  begin
    repeat
      Writeln('Nombre: '); leer_caracteres(nombre);
      Writeln('Apellido: '); leer_caracteres(Apellido);
      Writeln('Telefono: ');
      Writeln('Prefijo: '); Readln(telefono.CodigoArea);
      Writeln('Numero: '); leer_telefono(telefono.numero);
      Writeln('Nº de Matricula: '); Readln(Matricula);
      Writeln('Nota del Test: '); Readln(NotaTest);
      Writeln('Nota de 1º Problema: '); Readln(NotaP1);
      Writeln('Nota de 2º Problema: '); Readln(NotaP2);
      Writeln('entregó las Prácticas? s/n '); Readln(a);
      if a='n' then practicas:=false
        else practicas:=true;
      Write(Temporal,alumno);
      Writeln('Desea meter a otro alumno? s/n'); Read(a);
    until a='n';
  end;
close(Temporal); (*Cierra el archivo Temporal de modo escritura*)
Reset(Temporal); (*Proceso inverso al anterior*)
Rewrite(f); (*borra todo lo que habia y empieza a grabar*)
  repeat
    Read(Temporal,alumno); Write(f,alumno);
  until Eof(temporal);
end;

(*****)

procedure imprimir_nota (var f: ListaDatos; nmatricula:integer);
var alumno: Datos; a:char; encontrado:boolean;

begin
  assign(f, 'alumnos.dat'); (*solo en Turbo Pascal*)
  Reset(f);
  encontrado:=false;
  while (not encontrado)and(not eof(f)) do
    begin
      Read(f,alumno);
      if alumno.matricula=nmatricula then encontrado:=true;
    end;
  If encontrado then
    with alumno do
      Writeln('Nota : ', ((notatest+notaP1+notaP2)/3):4:2)
    else Writeln(' No encontrado. ');

end; (*fin de procedimiento*)

(*****)

procedure imprimir (var f: ListaDatos; nmatricula:integer);
var alumno: Datos; a:char; encontrado:boolean;

(*****)
procedure imprimir_caracteres ( var cadena: Cadenanombre );
```



```

var b:integer; (*aux*)
begin
  for b:=1 to Longnombre do write(cadena[b]);
end;
(*****)
procedure imprimir_telefono ( var cadena: tiponumero );
var b:integer; (*aux*)
begin
begin
  for b:=1 to max do write(cadena[b]);
end;
end;
(*****)

begin
assign(f,'alumnos.dat'); (*solo en Turbo Pascal*)
Reset(f);
encontrado:=false;
while (not encontrado)and(not eof(f)) do
begin
  Read(f,alumno);
  if alumno.matricula=nmatricula then encontrado:=true;
end;
If encontrado then
begin
  with alumno do
begin
  Writeln('Nombre: '); imprimir_caracteres(nombre);
  Writeln('Apellido: '); imprimir_caracteres(Apellido);
  Writeln('Telefono: ');
  Writeln('Prefijo: '); writeln(telefono.CodigoArea);
  Writeln('Numero: '); imprimir_telefono(telefono.numero);
  Writeln('Nº de Matricula: '); writeln(Matricula);
  Writeln('Nota del Test: '); writeln(NotaTest:5:3);
  Writeln('Nota de 1º Problema: '); writeln(NotaP1:5:3);
  Writeln('Nota de 2º Problema: '); writeln(NotaP2:5:3);
  if practicas=true then Writeln('Entregó las Pr cticas. ');
  else Writeln('No entregó las Pr cticas? ');
  Writeln('Nota : ', ((notatest+notaP1+notaP2)/3):4:3);
end;
end (*no lleva ; por tener else despues de la sentencia*)
else Writeln(' No encontrado. ');
end; (*fin de procedimiento*)

(*****)

begin (*PP*)

Writeln('Si quieres meter alumnos pulsa 0, buscar 1 , Nota 2 y anyadir 4');
Readln(k);
if (k=1)or(k=2) then
begin
  writeln(' Mete nº de Matricula: ');
  Read(nmatricula);
end;
Case k of
0:altas (alumnos);
1:imprimir(alumnos, nmatricula);
2:imprimir_nota(alumnos,nmatricula);

```



```
4: anyadir(alumnos, temporal);  
end;  
end.
```

10.7.2 Archivos de Texto.

Hasta ahora los archivos que habíamos visto son binarios, es decir que al editar el archivo en cuestión no podemos ver los datos que guardamos directamente. Pero sin embargo, hay otros tipos de archivos que además de poder ver directamente los datos en un editor, también podemos utilizar operaciones que no admiten los archivos binarios.

La información sólo se puede leer y escribir de forma secuencial (acceso secuencial). Se consideran formados por una serie de líneas las cuales a su vez están formadas por una serie de caracteres.



En los archivos de Texto tanto en escritura como en lectura se hace una conversión automática de los distintos tipos a tipo caracteres.

Declaración: **VAR alumnos: TEXT;**



Ojo: alumnos: ~~file of text;~~

El Pascal estándar obliga a especificar el tipo de archivo, pero Turbo Pascal no.

En un archivo, los registros hay que escribirlos campo a campo. Un registro se leerá de teclado campo a campo. (las variables de tipo registro se leen y escriben campo a campo). En un archivo de texto se pueden escribir tanto caracteres como números. Los datos no sufren una conversión automática a carácter al escribir en pantalla, pero se diferencian en memoria donde sufren la conversión.

Los ficheros de tipo texto son lentos trabajando con números, pero son útiles para pasar datos de un programa a otro. Éste tipo de archivos es el único ejemplo secuencial de Pascal (los archivos de texto), por lo que NO se pueden abrir a la vez para lectura y escritura, hay que cerrarlo antes.



Operaciones con archivos de texto.

Read(f,x);	en la posición que corresponde lee la variable x
Write(f,x);	en la posición que corresponde escribe la variable x
Reset(f);	Modo lectura de f en la 1ª posición.
Rewrite(f);	Modo escritura de f en la 1ª posición.
EOF(f);	función booleana de fin de fichero
Readln(f,x,y,z);	Lee 3 variables y un intro.
Writeln(f,x,y,z);	Escribe 3 variables y un intro.
Readln(f);	Lee un intro omitiendo lo que haya hasta él.
Writeln(f);	Escribe intro en el fichero.
EOLN(f);	Función booleana de fin de línea.

Debido a estas operaciones que tiene el archivo de texto no puede escribirse en él directamente tipos de variable estructuradas como Registros o Arrays,etc...

Packet Array: es un tipo de array que permite manejar una secuencia de caracteres directamente sin tener que señalar la posición a cada carácter. Permite la signación completa e incluso la ordenación o comparación de varias cadenas. En la declaración en vez de poner array se pone packet array.

Una ventaja es que en el VAX admite Write y Read y rellena con blancos lo que sobra.

Ejemplo de Fichero de Texto: Ordenación de una lista por el método de la burbuja.

```
program ordenacion_burbuja (input,output,archivo);
(*El programa consiste en ordenar un fichero de Texto que puede tener
hasta max numeros reales y graba el resultado en el mismo fichero.
```

En el fichero no se puede ordenar directamente así que pasaremos el fichero a un Array donde trabajaremos con los datos del fichero.

Además de **la condicion** del fin de fichero hay que tener en cuenta el **final de línea** puesto que **puede haber más de un numero por línea** y no debemos leer el final de líneas (return) al leer los numeros *)

```
const max=10;

type Tipovector=array [1..max] of integer;

var a:integer;
    Vector:Tipovector;
    archivo: Text;
```



```
procedure Meter_a_vector (var archivov1:Text; var Vector1:tipoVector);
(*Lee cada linea del fichero que tiene un numero y lo pone en un array que es
donde lo vamos a ordenar despues*)

var k,i:integer;

begin
  k:=0;
  Reset(archivov1);
  while not(eof(archivov1)and(k<max+1) do(*hasta fin de fichero o tamaño maximo*)
    begin
      k:=k+1;
      if eoln(archivov1) then Readln(archivov1)
      (*Puede haber mas de un numero por linea o ninguno*)
      else
        begin
          Read(archivov1,i);
          Vector[k]:=i;
          writeln(i:3);
        end;
      end;
    close(archivov1); (*cierra el archivo*)
  end;

procedure burbuja (var vector2:Tipovector);
(*Ordenamos el array de la siguiente manera durante max veces:
-Comparamos la posicion i y la i-1 del array desde la posicion max hasta la 2.
-Si esas dos posiciones estan desordenadas las ordenamos.
-Si i < i-1, entonces intercambiamos los valores. *)

var k,i,temp:integer;
begin
  for k:=1 to max do
    for i:=max downto 2 do (*ojo aqui, que es hacia atras*)
      if vector2[i] < vector2[i-1] then
        begin
          temp:=vector[i-1];
          vector[i-1]:=vector[i];
          vector[i]:=temp;
        end;
    end;
  end;

procedure Meter_a_archivo (var archivo3:Text; var Vector3:tipoVector);
(* Vuelve a meter los numeros pero esta vez ordenados en el fichero de Texto,uno
en cada linea *)

var k,i:integer;
begin
  k:=0;
  Rewrite(archivo3);
  repeat
    k:=k+1;
    Vector3[k]:=i;
    Writeln(archivo3,i); (*Escribe un numero en cada linea*)
    Writeln(i:3);
  until (k=max);
  close(archivo3);(*Cierra el archivo en modo escritura*)
end;

begin(*pp*)
  for a:=1 to max do vector[a]:=0; (*inicializamos el vector pq puede no llenarse*)
```



```
assign(archivo,'d:\a.txt'); (*esto solo en Turbo Pascal*)
Meter_a_vector(archivo,Vector);
Burbuja(Vector); (*ordenamos la lista*)
Meter_a_archivo(archivo,Vector);
end.
```

10.8 Recursividad:

La recursividad pretende ayudar al programador cuando trata de resolver problemas que por su naturaleza están definidos en términos recursivos.

Ejemplo : Factorial de un número

$$n! = n (n-1) !$$

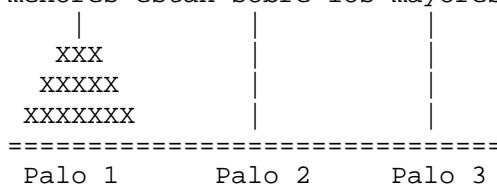
Se define un problema recursivamente cuando se resuelve en términos más sencillos del mismo problema. Puede ser :

- ✓ Directa : La recursividad es directa cuando un procedimiento se llama a sí mismo pasando cada vez a un paso más pequeño.
- ✓ Indirecta : La recursividad indirecta consiste en que un procedimiento llame a otro que a su vez vuelve a llamar al primero.

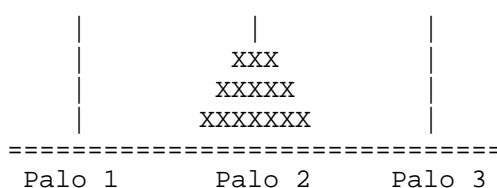
Cuando un procedimiento se llama de forma recursiva los parámetros por valor y las variables locales se almacenan en la pila. El procedimiento trabaja con esos valores y cuando sale, retorna al nivel anterior y recupera los valores de la pila del nivel anterior.

Ejemplo: Las Torres de Hanoi

Se tienen tres palos 1, 2 y 3 y n discos don un agujero en el medio y un diámetro diferente. Al comienzo del juego los discos están en el mismo palo 1 de forma que los discos menores están sobre los mayores.



El objetivo del juego es los discos al palo 2 desde el 1 y que éstos queden en la misma disposición que tenían al principio.





Las reglas que se han de cumplir son las siguientes:

- 1) Sólo se puede mover un disco cada vez
- 2) Los discos deben estar colocados siempre en uno de los palos
- 3) Nunca puede estar un disco mayor sobre uno menor.

```
Program hanoi (input,output,file)

var n,x,y,aux:integer;
    file:text; (*archivo de texto en el que grabamos las jugadas*)

procedure hanoi (n1,pinicial,pfinal,paux:integer);
begin
if n1=1 then writeln(f,'Lleva el disco del palo ',pinicial,' al',pfinal);
    else begin
        hanoi (n1-1,pinicial,paux,pfinal);
        writeln(f,'lleva el disco del palo ',pinicial,' al palo ',pfinal);
        hanoi(n1-1,paux,pfinal,pinicial)
    end
end;

begin (*PP*)
rewrite(f);
write('Discos?: ');
readln(n);
x:=1; (*primer palo*)
y:=3;  (*último palo*)
aux:=2; (*Segundo palo*)
hanoi(n,x,y,aux); (*Mete el nº de discos, y el numero de los palos*)
close(f); (*Cierra el archivo*)
end.
```

Explicación para 3 Discos:



Cuando se entra en el procedimiento en el PP los valores que se meten son 3 Discos, palo 1 como palo inicial, palo 3 como palo final y palo 2 como palo auxiliar. Hay que tener en cuenta que al llegar al final de una recursividad hay que ir hacia atrás restaurando los valores que teníamos pues se van recuperando y se va ejecutando todo el procedimiento que falta por ejecutar.

A continuación representamos las variables que metemos en el procedimiento principal teniendo en cuenta desde donde lo llamamos, pues el orden de las variables cambian cada vez que llamamos al procedimiento.



Nº Discos	P. inicial	P. Final	P. aux	Accion	Orden de ejec
3	1	3	2	Lleva de 1 a 3	4
Se mete en otro proced.					
2	1	2	3	Lleva de 1 a 2	2
1	1	3	2	Lleva de 1 a 3	1
1	3	2	1	Lleva de 3 a 2	3
Se mete en otro proced.					
2	2	3	1	Lleva de 2 a 3	6
1	2	1	3	Lleva de 2 a 1	5
1	1	3	2	Lleva de 1 a 3	7

En este otro esquema se ve mejor la secuencia de la recursividad:

Seguir Flechas:		Empieza -> 1132
3132 <-	2123 <-	-> 1321
	2231 <-	-> 1213
		-> 1132

Ejemplo 2 : Hallar el factorial de n.

```

program pr_factorial (input,ouput);
var Resultado,n:integer;

function factorial (n:integer):integer;
begin
  If n=0 then factorial:=1
  else Factorial:=n*Factorial(n-1);
end;

begin
Writeln('Numero?'); Readln(n);
Resultado:=Factorial(n);
Writeln(Resultado);
End.
    
```

Ejemplo 3: multiplica dos enteros

```

program programa_producto (input, output);
{multiplica dos enteros usando 2 versiones de una función recurrente}

type natural = 0 .. maxint;

var n, m, resultado: integer;
    
```




```
function producto_1 (n: natural; m:integer):integer;
{Necesita: 2 enteros, el primero un natural n (>= 0) y el segundo, m, un entero
cualquiera}
{Produce: el producto de ambos n * m}
begin
  if (n = 0) then producto_1 := 0
  else if (n = 1) then producto_1 := m
  else {n > 1, caso no trivial} producto_1 := m + producto_1 (n -1, m)
end;

begin {principal}
  write(' Escribe los numeros a multiplicar '); readln(n, m);
  writeln; writeln(' ***** Calculo de productos *****');
  writeln;
  writeln('      Resultado del Producto: ' producto_1(n,m));
writeln; writeln(' ***** Han finalizado los calculos *****');
end.
```

Ejemplo 4: Máximo Común Divisor

```
program mcd_recur(input,output);
uses crt; (*TP7*)
var num1,num2:integer;

function mcd(num1,num2:integer):integer;
begin
  if num2=0 then mcd:=num1
  else
  mcd:=mcd(num2,num1 mod num2);
end;

begin
  clrscr; (*TP7*)
  readln(num1);readln(num2);
  write(mcd(num1,num2));
end.
```

Ejemplo 5: parejas de conejos

```
program Fibonacci (input, output);
{Leonardo de Pisa, llamado también Leonardo Fibonacci, en uno de sus escritos
planteaba la pregunta: ¿Cuántas parejas de conejos se originan de 1 pareja inicial
en n meses?. Se supone que cada mes cada pareja llega a ser fértil 1 vez y que no
se muere ningún conejo. La respuesta es el n-ésimo término de la siguiente serie
que, desde entonces, se conoce con el nombre de serie de Fibonacci:
```

1,1,2,3,5,8,13,21,34, ...

y que tiene la siguiente definición recurrente:

```
fib(n) := if (n = 0) or (n = 1) then 1
          else fib(n-1) + fib(n-2).
```

Este programa calcula el término n-esimo de esta serie de Fibonacci usando la
recurrencia anterior}

```
var n, resultado: integer;

function fib (n:integer):integer;
{Necesita: un entero positivo n
Produce: el n-ésimo término de la serie de Fibonacci}
begin
  writeln('Calculo el término número ', n);
```



```

        if (n = 0) or (n = 1) then {caso trivial}      fib :=1
        else {caso no trivial} fib := fib(n-2) + fib(n-1)
end;

begin {principal}
  writeln('Escribe el entero positivo n para que calcule el término n-esimo de
  la');
  write(' serie de Fibonacci '); readln(n);
  writeln; writeln(' ***** Comienza el cálculo *****');
  resultado:= fib (n);
  writeln; writeln(' ***** Ha finalizado el cálculo *****');
  writeln('Resultado final: ', resultado)
end.

```

10.9 Punteros y Listas

ESTRUCTURAS DINÁMICAS DE DATOS.

Son aquellas en las que el espacio que ocupan en memoria se determinan en tiempo de ejecución a diferencia de las estáticas, en las que el espacio que ocupaban se establecían en tiempo de compilación. Una variable dinámica se crea en tiempo de ejecución, y necesita para su creación del uso de punteros.

Las variables de tipo puntero se representan $\square \rightarrow$ y almacenan la dirección de memoria donde se encuentra la variable dinámica apuntada y el contenido de dicha dirección de la variable.

Declaración: **VAR p:^integer ; {ejemplo cualquiera}**

P es de tipo puntero que sirve a una dirección de memoria y a su contenido.

Variable dinámica: variable creada durante la ejecución de un programa.

Variable referenciada: variable creada y accedida por una variable tipo puntero.

- Operaciones con punteros :

Asignación :

p :=NIL ; {valor nulo, indica que no apunta a nada.}
 p :=q ; {p apuntará a donde apunte q, la dirección claro. }



Crear la variable dinámica :

New(p) ; -> Reserva espacio para la variable dinámica. Con New(p) se crea la variable dinámica apuntada (se asigna una dirección a la variable). Almacena en p una dirección de memoria donde se pueden guardar datos de tipo entero

p^:=3; {con p apuntada} Asigna un contenido a la variable que se almacena en su dirección. Si no hay dirección la almacena en una aleatoria pisando lo que hay allí.

Dispose(p) ; Libera ese espacio de memoria ocupado por la variable dinámica apuntada . Conviene hacer después p:=NIL; Entre new y dispose se trabaja de forma segura.

Comparación :

Sólo acepta < > y = (nunca > o <) para del mismo tipo. Ej : If p < > q then...

Los punteros no pueden ser leídos de teclado ni verlos en pantalla. Las variables dinámicas admiten cualquiera de las operaciones que admitiría una variable de ese tipo.

Si se puede :

```
ReadLn(p^); { porque p^ es un entero }
WriteLn(p^);
```

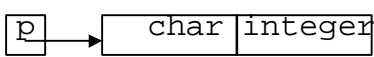
Nunca: **ReadLn(p)** ; { porque p es un puntero }

Las variables dinámicas (apuntadas) pueden ser de cualquier tipo :

Type

```
puntero = ^integer ;
         ^real ;
         ^char ;
         ^string ;
         ^array ;
         ^Registro ; {Registro se debe declarar antes que el puntero}
Con Registro = RECORD
    c1 : char ;
    c2 : integer ;
end ;
```

Si es puntero a variable de tipo registro :

```
Var p : puntero
Begin
    New(p) ; {  ← creada en tiempo de ejecución }
```



```

    ReadLn(p^.c1) ;
    ReadLn(p^.c2) ;

```

Otro caso: {estructura de datos enlazada}

```

    puntero=^Registro ;
    Registro=RECORD
        c1 :char ;
        c2 :puntero
    end ;

```

```

    Var p :puntero ;
    Begin
        New(p) ;
        New(p^.c2) ;
        ReadLn(p^.c1) ;{ no se puede leer p^.c2 porque es puntero}
        ReadLn(p^.c2^.c1) ;

```



En un procedimiento se puede modificar un puntero con o sin VAR porque se pasa por valor o referencia la dirección no el contenido y al asignar un valor borramos lo que había allí. *Distinguir dirección y contenido.*

Ejemplo:

```

Program prueba(input,output);
VAR p,q:^integer;
Begin
    New(p);
    P^:=5; {p=5}
    New(q);
    q^:=33; {p=5 y q=33}
    dispose(p);
    p:=q; { a p se asigna la direccion de q aunque está liberada}
    p^:=20; { en la direccion de q se asigna el valor 20}
End.

```

Ejemplo: Dada una lista de nombres en un archivo, se pide a ordenarlos por apellidos y grabarlos en el mismo fichero.

```

program ordenacion_con_punteros (input,output,lista);
(*Los registros del archivo Lista se ordena por apellidos y se reescriben
el Lista otra vez. Si el archivo no existe, lo creamos.*)

const max=1000; (*maximo nº de nombres a ordenar*)
type
    (*cadena=string ; en turbo pascal*)
    cadena=packed array [1..30] of char ;(*estandard*)
    Datos= record
        Nombre:Cadena;
        Apellidos:Cadena;
    end;

```



```
Rango=1..max;
Punteropersona=^Datos;(*Tipo de puntero*)
TipoLista=array [rango] of Punteropersona; (*Tipo de Array de punteros *)
Archivo_personas=file of Datos;

var
  persona:tipolista; (*array de punteros a los registros personales*)
  Longitud:integer; (*Numero de punteros valido de persona*)
  UnaPersona:Punteropersona; (*Punteros al registro temporal que se va a crear*)
  Indice:Rango;(*variable de control del bucle*)
  Lista:Archivo_personas; (*Archivo con los nombres*)

(*****)
procedure crear (var lista:archivo_personas);
(*Si el archivo no existe lo creamos con este procedimiento*)
var identidad:Datos; (*Registro*)
    a:char; (*aux*)

begin
  assign(lista, 'd:\lista.dat'); (*solo en Turbo Pascal*)
  Rewrite(lista); (*borra todo lo que habia y empieza a grabar*)
  with identidad do
    begin
      repeat
        Writeln('Nombre: '); Read(Nombre);Readln;
        Writeln('Apellido: '); Read(Apellidos);Readln;
        Write(lista,identidad);
        Writeln('Desea meter a otra persona? s/n'); Readln(a);
      until a='n';
    end;
end;

(*****)
procedure Ordenacion (var Lista:Tipolista; (*array de punteros*)
                      Longitud:integer); (*punteros totales en lista*)

(*Un array de longitud punteros se toma como entrada. El mismo array se devuelve
ordenado por apellidos. el algoritmo consiste en encontrar el valor minimo del
array y lo intercambia por el primer valor del array. Luego se intercambia el
siguiente
valor mas pequeño del array con la segunda posicion del array.
La posicion que se cambiar es aquella que contenga el valor minimo. *)

var Punterotemporal:punteropersona; (*utilizado en el intercambio*)
    contpasos, (*Variable de control del bucle*)
    contposiciones, (*Variable de control del bucle*)
    minindice: (*indice minimo hasta ahora*)
        rango;

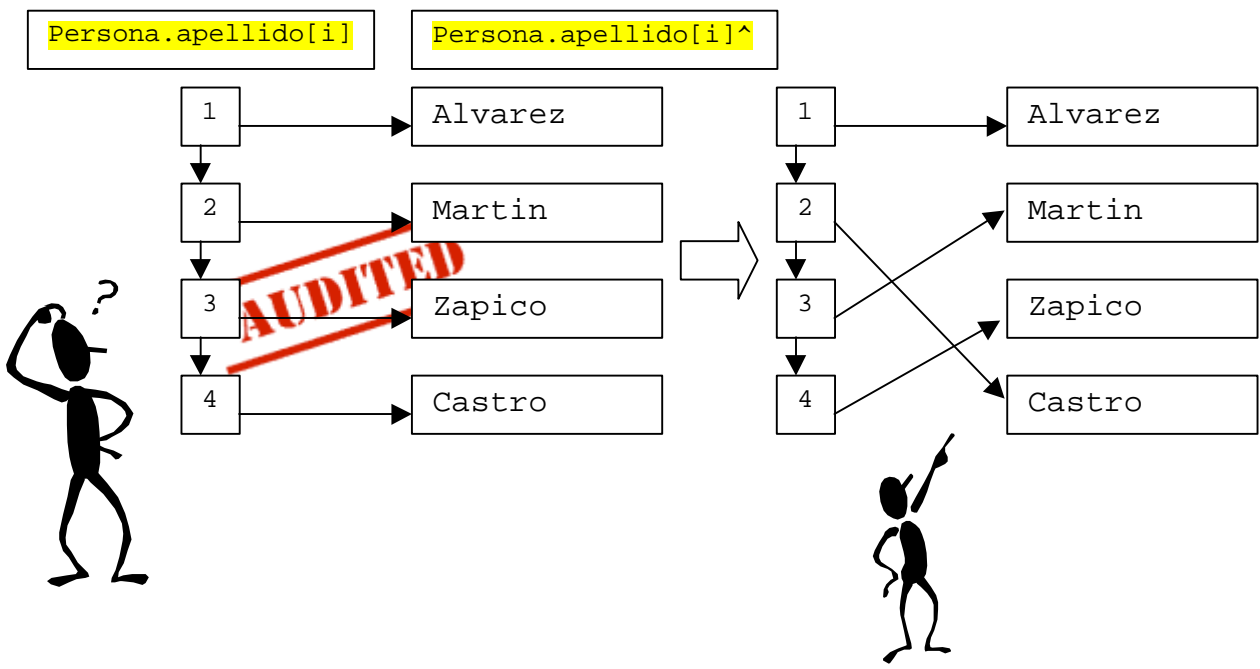
begin
  for contpasos:=1 to Longitud-1 do
    (*Los registros de lista[1] a lista[contpasos-1] estan ya ordenados
alfabeticamente*)
    begin
      minindice:=Contpasos;
      (*Encuentra el indice del puntero al 1º apellido sin ordenar*)
      for contposiciones:=Contpasos+1 to Longitud do
        if lista[contposiciones]^Apellidos < Lista[minindice]^Apellidos
            then minindice:=Contposiciones;
      (*Intercambia lista[minindice] por lista[cotpasos].
Lo unico que hace es cambiar la direccion del puntero a la memoria*)
```



```

Punterotemporal:=Lista[minindice];
Lista[minindice]:=lista[Contpasos];
Lista[contpasos]:=punterotemporal;
end;
end;
(*****
begin (*PP*)
Assign(lista,'d:\lista.dat'); (*solo en Turbo Pascal*)
Writeln('Teclea 0 para crear la lista y el resto para ordenarla.');
```

Como vemos en el ejemplo no se cambia en contenido de los punteros del array sino que redireccionamos la direccion de cada puntero del array al contenido correcto para tenerlos ordenados alfabeticamente.





Las estructuras dinámicas de datos pueden ser :

- Lineales (Listas, Pilas y Colas) {Las pilas y colas son tipos especiales de listas}

En una estructura dinámica lineal, cada elemento tiene un único elemento sucesor o siguiente y cada elemento y un único elemento predecesor o anterior.

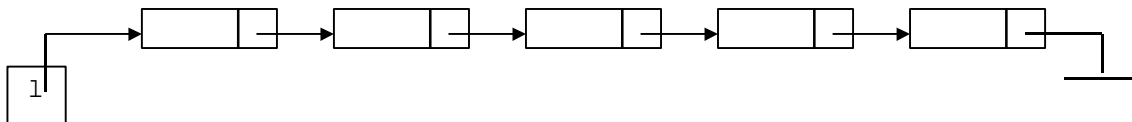
- No Lineales (Árboles y Grafos)

Las estructuras no lineales se caracterizan porque sus elementos pueden tener más de un elemento sucesor o siguiente (Árboles : un sólo predecesor y varios posibles sucesores). Los Grafos pueden tener más de un elemento predecesor (varios predecesores y varios sucesores).

LISTAS.



Una Lista es una colección o serie de datos del mismo tipo que se almacenan en la memoria del ordenador. En una lista cada elemento podrá tener un único sucesor o siguiente y un único antecesor. Además la inserción y eliminación de elementos se podrá efectuar en cualquier lugar de la lista. Como casos especiales de listas son las pilas y las colas, en las que la inserción o eliminación de elementos está restringida a efectuarse por un determinado lugar.



Las Listas pueden ser de varios tipos :



- Listas Contiguas :

En ellas, los elementos que las constituyen ocupan posiciones consecutivas en memoria. Se suelen implementar por medio de arrays.


- Listas Enlazadas :

Son las 'verdaderas listas'. Tienen la ventaja de que la inserción y eliminación de un elemento en cualquier lugar nunca obliga al



desplazamiento de los restantes elementos de la lista. En las listas enlazadas, los elementos no ocupan posiciones consecutivas de memoria, por lo que cada uno de ellos se ve obligado a almacenar la posición o dirección de memoria donde se encuentra el siguiente elemento de la lista. Para recorrer una lista enlazada hay que hacerlo siempre desde el principio de la lista (para seguir los punteros). Estas listas se representan con punteros en general, lo que consigue que la compilación en memoria pueda variar en la ejecución del programa.

Las listas, sin punteros, se simulan mediante arrays, lo que provoca el inconveniente de que el espacio reservado en memoria es constante (deja de ser una estructura dinámica para convertirse en estática).

Nodo: bloque que forma la estructura dinámica, el cual está formado por una componente (dato) y un puntero (el enlace) al siguiente nodo. 

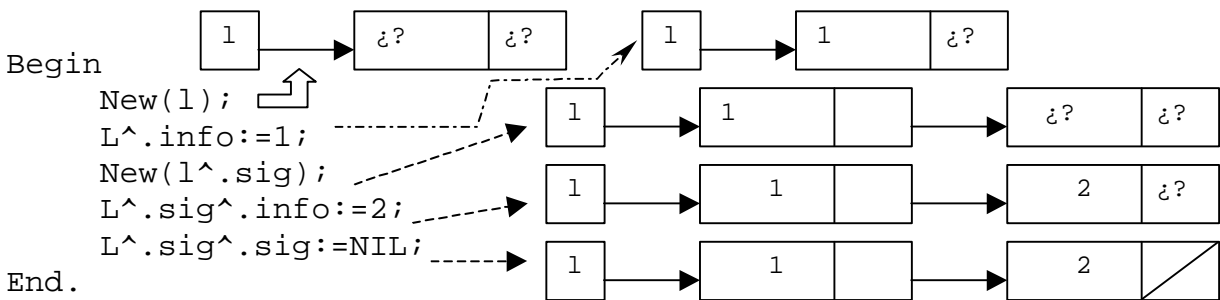
Lista enlazada: sucesión de nodos unos detrás de otros. La lista puede ser un solo nodo.

Con el siguiente Ejemplo vamos a ver como se manejan listas enlazadas y se podrá comprender mejor su estructura y funcionamiento.



Creacion de una lista l (ele) y operaciones:

```
Type      lista=^nodo;
         Nodo= record
             Info:integer; {puede ser otro tipo de variable}
             Sig:lista;
         End;
Var l:lista;
```





Llegado a este punto vemos la utilidad de la recursividad para operar con listas.

Todo el manejo de listas se basa en procedimientos recursivos. En los ejemplos de la página siguiente (es una fotocopia) se muestran operaciones con las listas mediante procedimientos. Estos procedimientos para aplicarlos en programas y en el examen podemos aplicarlos de frente sin tener que escribir el código del módulo.

De todas formas conviene saber su fundamento, que consiste en jugar con las direcciones de memoria crearlas, redireccionarlas o liberarlas según convenga.



Estos apuntes se acabaron de escribir el 9 de Junio de 1998.

Juan Ramón Álvarez Riera

<http://www2.drogas.org>

ICQ # 24246105

jrar@micorreo.net