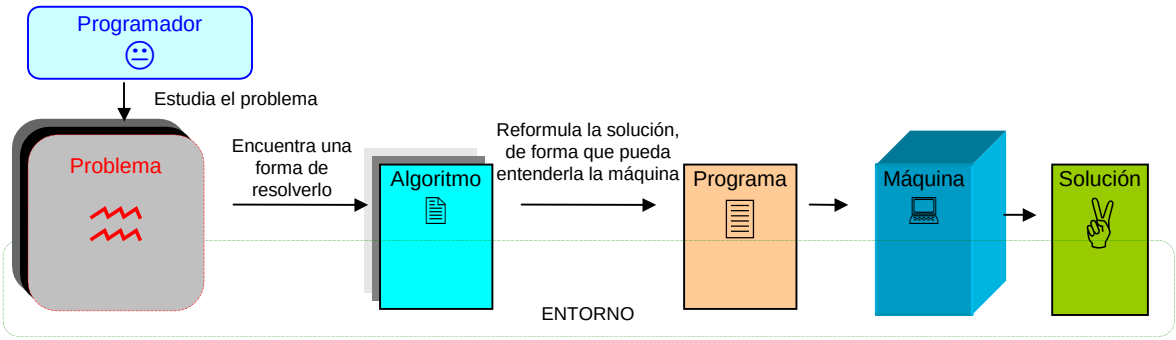
1 INTRODUCCIÓN

1.1 CONCEPTO DE PROGRAMACIÓN EN UN SENTIDO AMPLIO

Una definición de programación puede ser:
Programar es preparar la resolución de un problema para que pueda ser ejecutada por una máquina.



Elementos que intervienen en la programación:

Programador. Es el sujeto encargado de identificar el problema, buscar una forma de resolverlo, y darle a la máquina las instrucciones adecuadas para que ésta lo resuelva.

Procesador. Es el instrumento utilizado. Puede ser cualquier entidad (no necesariamente una máquina, también puede ser un ser vivo) capaz de entender un enunciado y ejecutar la tarea indicada. Tiene la propiedad de ser programable si puede aceptar diferentes enunciados, y cada enunciado sirve para resolver o realizar una tarea diferente.

Entorno. Es el conjunto de objetos que utilizan el programador y el procesador para la realización de la tarea, desde el principio hasta el fin.

Acción. Es un suceso que modifica el entorno. Puede ser:
Acción primitiva. Su enunciado lo entiende el procesador, y lo puede ejecutar directamente
Acción no primitiva.No la entiende el procesador.

Algoritmo. Es la descripción de la resolución de un problema.

Programa. Es el resultado de codificar el algoritmo en un conjunto de órdenes que pueden ser entendidas por el procesador. El programa depende no sólo del problema tratado, sino también del procesador utilizado. Por ejemplo: tengamos los procesadores P1, P2, y el programa PRG1 con el que P1 realiza la tarea T. Si en PRG1 existe una orden O que es primitiva para P1 pero no para P2, debemos reelaborar PRG1 descomponiendo O en acciones O1,..., ON que sí sean primitivas para P2. De esta forma se obtiene el programa PRG2, es decir, hay una tarea T y dos programas PRG1, PRG2 para realizarla, uno con cada procesador.

Objetos que puede haber en el entorno

Constantes. Son objetos que tienen un valor inalterable de principio a fin.
Variables. Son objetos cuya utilidad es almacenar valores que pueden cambiar por efecto de alguna instrucción del programa. Las variables tienen:
Nombre. Sirve para identificar la variable en el programa.
Tipo. Es la clase de valor que puede almacenar.

1.2 ANÁLISIS DESCENDENTE

Es el proceso de descomponer una tarea compleja, acción no primitiva, en una secuencia de acciones primitivas. También puede verse como la descomposición de un problema complejo en subproblemas más fáciles de resolver.

Ejemplo:
Para resolver un problema dado P con un procesador, se propone una tarea T. Puede suceder que T no sea una acción primitiva del procesador. Para que sea entendida, debemos descomponerla en otras más simples que sean acciones primitivas. Supongamos que descomponemos T en T1, T2, T3 y T4.
Solución del problema:

- Realiza la tarea T1
- Realiza la tarea T2
- Realiza la tarea T3
- Realiza la tarea T4

Puede suceder que T2 aún no sea una primitiva para el procesador. Debemos descomponerla en otras que sí lo sean. Supongamos que se descompone en las primitivas T21 y T22. Ahora:
Solución del problema:

- Realiza la tarea T1
- Realiza la tarea T21
- Realiza la tarea T22
- Realiza la tarea T3
- Realiza la tarea T4

Veamos un ejemplo más concreto en el que entran en juego todos estos elementos:
Supongamos que el problema que queremos resolver es calcular el producto de los tres primeros números naturales: 1*2*3. La forma de resolver el problema dependerá de las características del procesador de que se disponga, es decir, dependerá del conjunto de acciones primitivas que pueda entender y ejecutar. Supongamos que tiene estas facultades:

1) Sabe asignar un valor a una variable. La operación de asignación se expresa así:

```
NombreVariable := Valor ;
```

2) Sabe multiplicar el contenido de dos variables V1 y V2, y almacenarlo en la primera.
Esta operación se representa así:

```
V1 := V1 * V2 ;
```

3) Tiene en su memoria capacidad para tres variables, llamadas v1, v2 y v3

Una primera aproximación puede ser:
Solución del problema:

- Multiplica 1 * 2 * 3

Esta no es una acción primitiva. Podemos descomponerla en:
Solución del problema:

- Almacena el valor 1 en v1
- Almacena el valor 2 en v2
- Almacena el valor 3 en v3
- Multiplica v1 *v2 * v3

La última acción aún no es primitiva. Debemos seguir descomponiendo
Solución del problema:

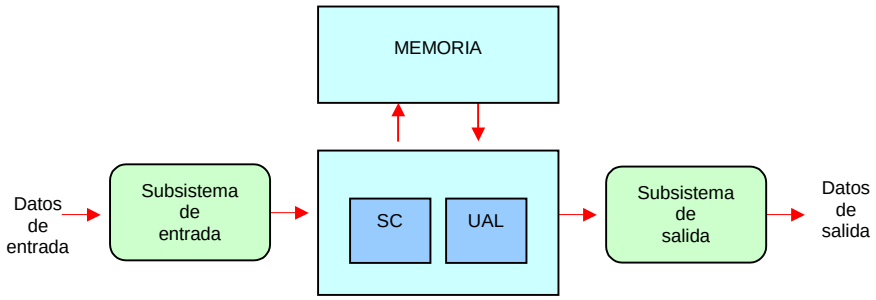
- Almacena el valor 1 en v1
- Almacena el valor 2 en v2
- Almacena el valor 3 en v3
  - Multiplica v1* v2 y el resultado almacénalo en v1
  - Multiplica v1* v3 y el resultado almacénalo en v1

La codificación del algoritmo en instrucciones comprensibles para el procesador sería:

```
v1 := 1 ;
v2 := 2 ;
v3 := 3 ;

v1 := v1 * v2 ;
v1 := v1 * v3 ;
```

Al acabar la ejecución del programa, la solución está contenida en la variable v1

1.3 ARQUITECTURA BÁSICA DE UN PROCESADOR PROGRAMABLE



La arquitectura más seguida para construir un procesador programable es la propuesta por John von Neumann; el procesador lo forman sólo tres elementos:

La **memoria**. Es donde se almacenan:

- Las instrucciones que componen el programa
- Los datos necesarios para su ejecución
- Los resultados generados por dicha ejecución

La **Unidad Aritmético Lógica** (U.A.L.) descifra las instrucciones del programa almacenado, y se encarga de su ejecución.

La **Sección de Control**. Marca la secuencia de señales que se deben enviar a los restantes elementos para que cada uno opere en el momento adecuado. Para ello lleva un reloj interno y un registro que contiene la dirección donde está almacenada la instrucción que se va a ejecutar inmediatamente.

La interacción entre el programa almacenado y los recursos físicos del procesador la realiza un programa especial que se almacena y empieza a ser ejecutado al poner en funcionamiento el procesador. Este programa se llama **sistema operativo**, y además de coordinar las acciones de los elementos de que consta el procesador, gestiona otros programas que puede haber almacenados en memoria, ejecutándose de forma alternada.

1.4 PROCESADORES DIGITALES. LENGUAJES DE PROGRAMACIÓN

La gran mayoría de los procesadores programables que se usan actualmente están implementados en su interior con circuitos electrónicos digitales. De ahora en adelante los procesadores programables que trataremos serán únicamente los digitales, los cuales llamaremos computadores u ordenadores. En los circuitos digitales los voltajes de los nodos claves (llamados biestables) sólo pueden tener dos valores: 0 ó 5 voltios (en realidad no exactamente 0 ó 5 voltios, sino valores en entornos de estos voltajes). Con un conjunto de *n* biestables se puede representar un número de *n* cifras en base 2. (A un valor de voltaje del biestable le corresponde el 0 y al otro el 1). También es posible almacenar y procesar internamente caracteres alfabéticos , numéricos y tipográficos, si los codificamos asociando un número binario a cada carácter. Un código muy usado en informática es el ASCII.

Por todo esto, el único lenguaje que entiende el ordenador es este formado por ceros y unos, llamado **lenguaje máquina**, que se corresponde con la base binaria de numeración. Las características de los programas escritos en este lenguaje son:

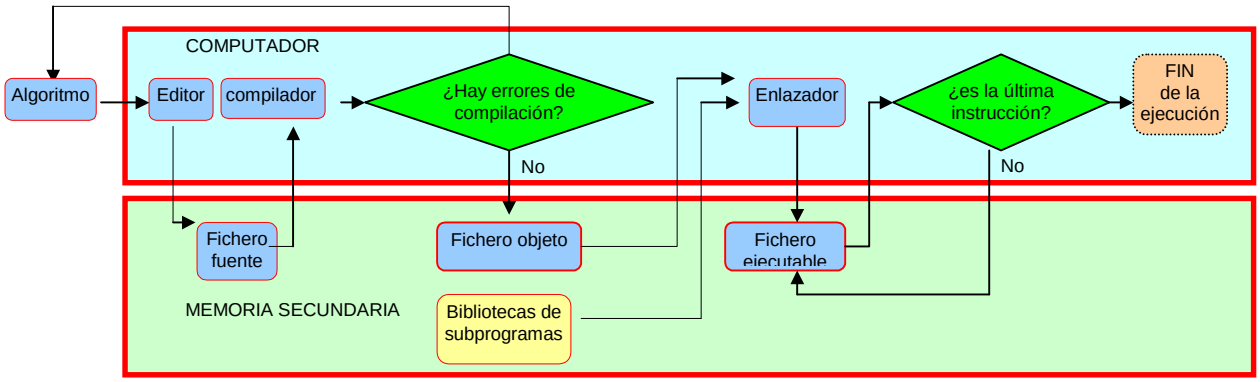
- No es fácil identificar una cifra formada por una larga ristra de ceros y unos, sin hacer los cambios de conversión de base.
- Si por error se altera un dígito, es muy difícil encontrar el error.
- Codificar un algoritmo escrito en lenguaje humano a lenguaje máquina es una tarea muy tediosa .

Para evitar todos estos inconvenientes, se idearon los **lenguajes de programación de alto nivel**, que son conjuntos de instrucciones codificadas con los caracteres numéricos, aritméticos y del alfabeto latino. Por supuesto, se necesita un programa intermedio que traduzca el lenguaje de alto nivel a lenguaje máquina. Este programa traductor usualmente se llama **compilador**.

Cualidades de los lenguajes de alto nivel:

- Tienen estructuras de cálculo y control de decisión muy semejantes a las del razonamiento humano.
- Sus reglas sintácticas son muy estrictas, por lo que evitan ambigüedades e imprecisiones.
- Portabilidad: Dos ordenadores con diferente construcción interna tienen diferentes lenguajes máquina, pero una vez instalado el compilador adecuado en cada ordenador, ambos pueden usar el mismo programa escrito en lenguaje de alto nivel.

1.5 PROCESO DE LA CREACIÓN DE UN PROGRAMA COMPILADO DE ORDENADOR



Una vez que tenemos planteado el algoritmo, lo codificamos en un lenguaje de alto nivel (en nuestro caso en Modula-2) utilizando un **editor de texto**. Un editor de texto es un programa que graba en la memoria secundaria un archivo de caracteres ASCII. El producto de la edición es el **archivo fuente**.

A continuación el compilador intenta traducir el fichero fuente a lenguaje máquina. Si en la etapa de edición no se han observado todas las reglas sintácticas del lenguaje de alto nivel, hay errores de compilación. Se debe corregir el archivo fuente con el editor, y volver a intentar la compilación hasta que el compilador no encuentre errores.

En ese momento crea el **archivo objeto.** Este archivo está escrito en lenguaje máquina, pero aún no es un programa completo y listo para ser ejecutado. Necesita la incorporación de algunos módulos donde están predefinidas acciones no primitivas, pero que por ser utilizadas con mucha frecuencia, se codifican y almacenan aparte para ser utilizadas por casi todos los programas. Estos módulos son las bibliotecas de subprogramas.

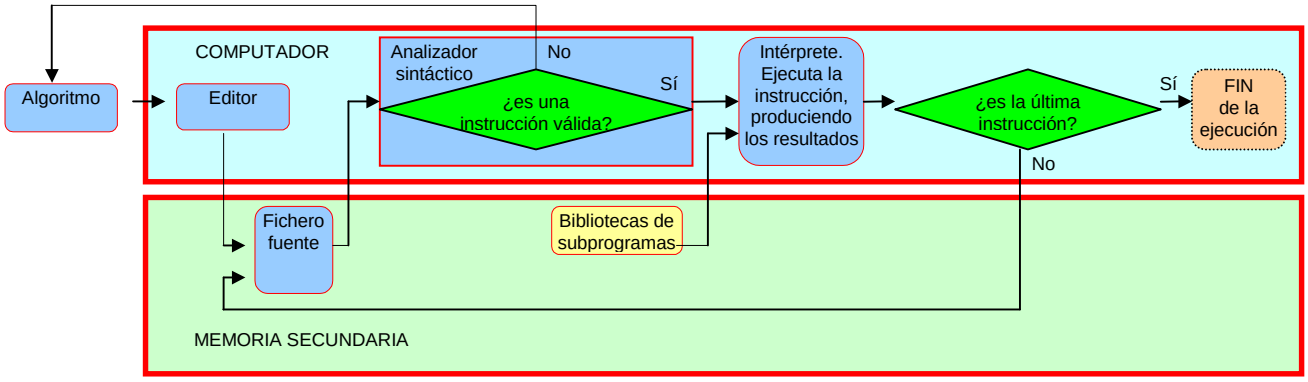
El encargado de realizar estas incorporaciones es el **enlazador**. El producto de la etapa del enlazado es el **archivo ejecutable**.

Para ejecutar un programa lo invocamos escribiendo su nombre y pulsamos [INTRO]. En ese momento se carga en la memoria principal y comienza su ejecución.

En Modula-2 la extensión de cada archivo indica su tipo :

Extensión	Tipo de archivo
.MOD	..... Fuente
.M2O	..... Objeto
.EXE	..... Ejecutable

1.6 PROCESO DE LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA INTERPRETADO DE ORDENADOR



Las diferencias son:

- No se generan ficheros objeto ni ejecutable.
- Las instrucciones las ejecuta (previamente se analiza si son válidas) el intérprete una a una, directamente desde el fichero fuente.

1.7 CUALIDADES DESEABLES EN TODO PROGRAMA

Corrección. Para estar satisfechos con un programa, es imprescindible que funcione bien, que realice la tarea para la que fue concebido. Evidentemente, antes de empezar a diseñar un programa, debemos tener bien especificada cuál es esa tarea. Para comprobar su corrección podemos hacer una serie de pruebas ejecutándolo con diferentes datos de entrada. Si en alguna de estas ejecuciones no se obtienen los resultados previstos, el programa no funciona correctamente. En el caso contrario no podemos asegurar nada. Podría tener algún fallo que se pusiera de manifiesto con un conjunto de datos no experimentado.

En los grandes proyectos de programación son habituales las tareas de mantenimiento, que constan de pruebas, correcciones, mejoras y ampliaciones después de haberlo puesto a disposición de los clientes.

Claridad. Para facilitar las tareas de mantenimiento, es conveniente que las descripciones del programa, su documentación, y su redacción sean claras e inteligibles para otras personas, o incluso para el mismo autor, pues después de un cierto tiempo puede haber olvidado los detalles del programa.

Eficiencia. Un programa eficiente es el que aprovecha bien los recursos del ordenador. Los criterios de eficiencia son:

- Ahorro de memoria
- Velocidad de ejecución

El primer objetivo, obviamente es la corrección. Los objetivos de claridad y eficiencia habitualmente son antagónicos. Si queremos que un programa esté estructurado y redactado con más claridad, lo más probable es que necesite más memoria, y por consiguiente sea menos eficiente. Si los recursos materiales de un sistema (memoria física y velocidad del procesador) son limitados, el objetivo de la eficiencia prevalece sobre el de la claridad. Esto ocurría en los primeros tiempos de la Informática . Los programas eran fruto más de una elaboración artesanal que de una metodología bien estructurada de diseño. Actualmente lo que prima es la claridad.

2 PRIMEROS CONCEPTOS DE MODULA-2

2.1.- TIPOS DE DATOS EN MODULA-2

Presentamos aquí algunos de los tipos de datos (para variables y constantes). El resto de tipos predefinidos se irán presentando cada uno en su momento.

2.1.1.- Tipo INTEGER

Los valores de este tipo se corresponden con el concepto matemático de números enteros, tanto negativos como positivos.

El menor y el mayor de los números enteros que se pueden utilizar depende de cada compilador. Esto es debido a que cada compilador puede usar un número diferente de bits para representar un INTEGER. Si un compilador usa *n* bits para representar los enteros, como un bit lo dedica para codificar el signo, quedan n-1 bits para codificar el número, resultando:

Nos negativos : desde -1 hasta -2^{n-1}
Nos positivos : desde 1 hasta $2^{n-1} - 1$, pues una de estas combinaciones se reserva para el cero

Si por ejemplo n=16 , los INTEGER posibles están entre -32768 y 32767

Más adelante veremos un método que proporciona el lenguaje Modula-2 para saber cuáles son esos valores mínimo y máximo en el compilador que estemos usando.

Las operaciones que se pueden realizar con los datos INTEGER son las que permite este conjunto de operadores:

+	Suma de enteros
-	Resta de enteros
*	Multiplicación de enteros
DIV	División de enteros (Trunca la parte fraccionaria)
MOD	Resto de la división entera (El resultado siempre es un entero)
+	Identidad de un entero
-	Cambia de signo un entero

2.1.2.- Tipo CARDINAL

Se corresponde con los números enteros no negativos. Como para representarlos no se necesita dedicar uno de los bits para el signo, si el compilador los representa con n bits, el rango de números CARDINAL alcanzado es desde 0 hasta 2^n-1 .

Las operaciones posibles con estos datos son las mismas que con los INTEGER, pero sólo se pueden realizar entre parejas de CARDINAL, nunca entre un INTEGER y un CARDINAL.

2.1.3.- Tipo REAL

Coincide con el concepto matemático de número real. Los compiladores representan y procesan internamente los números reales mediante una mantisa y un exponente.

El número de bits empleado para codificar la mantisa determina la precisión (diferencia mínima entre dos valores reales representables).

El número de bits empleado para codificar el exponente determina el rango posible abarcado.

Hay dos notaciones posibles para escribir un número real:

1ª.- Notación habitual: Secuencia de dígitos del 0 al 9, entre los cuales hay un punto que separa la parte entera de la fraccionaria. La parte entera no puede ser vacía, y puede estar precedida de los signos (+) o (-). La parte fraccionaria puede estar vacía.

2ª.- Notación científica: Consta de la secuencia :

Mantisa (un número entero)
Letra E
Exponente (un número entero)

Las operaciones posibles con los datos REAL son las que permite este conjunto de caracteres:

+	Suma de reales
-	Resta de reales
*	Multiplicación de reales
/	División de reales
+	Identidad de un real
-	Cambia de signo un real

2.1.4.- Tipo CHAR

Un dato de tipo char puede tomar como valor cualquiera de los caracteres disponibles en el ordenador. Estos caracteres suelen codificarse según la tabla ASCII.

En Modula-2 un carácter se representa entre comillas (") o entre apóstrofos (').

Ejemplos: `'a'`, `"Ñ"`, `"¬"`

2.1.5.- Tipo STRING

Un dato de tipo STRING es una ristra de caracteres. Estas ristras van encerradas entre comillas o entre apóstrofos.

Ejemplos:

```
"¡ Hola !"
'Esto es una ristra de caracteres'
"El resultado es: "
"Vamos p'alante"
'Me dijo: "Iré mañana"'
```

Notas:

- Si una ristra incluye en su interior apóstrofos, sólo puede encerrarse entre comillas, y viceversa.
- Aunque un valor de tipo carácter se presenta de la misma forma que una ristra de un único carácter, el compilador procesa de forma diferente estos dos datos.
- Es posible tener una ristra vacía, sin ningún carácter.

2.2.- CLASES DE EXPRESIONES

Con cada expresión o sentencia de Modula-2 le indicamos al ordenador una acción que debe realizar. Las sentencias se separan por punto y coma (;), por lo que la última sentencia de cada bloque no necesita terminar con punto y coma, pero también puede llevarla.

Veamos algunas de las clases de expresiones.

2.2.1.- Aritméticas:

Son una combinación de operaciones matemáticas realizadas con variables y ó constantes. No todos los operadores que aparecen en una expresión aritmética tienen la misma prioridad de evaluación. Los operadores se clasifican en dos clases según la prioridad:

1ª.- Operadores multiplicativos: *, /, DIV, MOD
2ª.- Operadores aditivos: +, -

Si en una expresión aritmética aparecen varios operadores de un mismo nivel, se ejecutan las operaciones asociadas de izquierda a derecha. El orden de evaluación siempre puede cambiarse con el uso de paréntesis. Se evalúan en primer lugar las operaciones encerradas entre paréntesis.

Ejemplos:

```
 4 + 3 * 5 - 1
(4 + 3) * 5 - 1
-3 + 7 DIV 2 * 2
```

2.2.2.- Declaración de variables

Empieza con la palabra clave VAR, sigue el nombre de la variable, dos puntos (:), y el tipo, terminando en punto y coma (;).

Ejemplos:

```
VAR Peso : REAL ;
```

Pueden declararse juntas varias variables del mismo tipo, separadas por comas (,):

```
VAR
     Base, Altura : REAL ;
      DiaSuma : INTEGER ;
     EstadoCivil : CHAR ;
```

2.2.3.- Declaración e inicialización de constantes

Forma de hacerlo:

```
CONST NombreDeLaConstante = ValorConstante ;
```

Ejemplos :

```
CONST Radio = 4 ;
CONST
      Pi = 3.1416 ;
      Diametro = 2 * Radio ;
```

Los nombres de las constantes son sustituidos por su valor en el momento de la compilación. Ese valor debe ser conocido antes de empezar a ejecutarse el programa. Por tanto, si para inicializar una constante se utiliza una expresión aritmética, en esta expresión sólo puede haber constantes conocidas, no variables. Los valores de las varibles no son conocidos antes de empezar a ejecutarse.

2.2.4.- Asignación de un valor a una variable

Formas de hacerlo:

```
NombreDeVariable := NuevoValor ;
```

Ejemplos :

```
Area := 1.0 ;
Area := Base * Altura ;
EstadoCivil := 'S' ;
```

En general, para que una sentencia de asignación no sea errónea, el tipo de la variable y el valor asignado deben ser compatibles.

2.2.5.- Llamadas o invocaciones a subprogramas

Se hacen escribiendo el nombre del subprograma, y entre paréntesis los datos que éste necesita para ser ejecutado. Estos datos se conocen como **argumentos** o **parámetros**

Identificador. Es el nombre de variables, constantes, subprograma o definición de tipo.

Reglas de Modula-2 para los identificadores:

. El primer carácter debe ser alfabético
. Los caracteres alfabéticos posibles son sólo las 26 letras del alfabeto inglés.
. Modula-2 diferencia mayúsculas de minúsculas. (Las variables `NomVar` y `nomvar` no son la misma)
. Pueden usarse dígitos del 0 al 9 en cualquier lugar, excepto al comienzo del nombre
. No pueden ser usados los espacios en blanco ni los signos de puntuación intercalados
. No se pueden utilizar las palabras reservadas del lenguaje

2.3.- OPERACIONES DE ESCRITURA

Las operaciones de escritura tienen como objetivo dar a conocer al mundo exterior al ordenador algún dato. Este dato generalmente es el resultado producido por el procesamiento interno de otros datos.

En Modula-2 las operaciones de escritura se realizan mediante subprogramas:

`WriteInt( x1, m1 );`	Escribe el INTEGER	contenido en la variable	`x1`
`WriteCard( x2, m2 );`	Escribe el CARDINAL	contenido en la variable	`x2`
`WriteReal( x3, m3 );`	Escribe el REAL	contenido en la variable	`x3`

En los tres casos, el segundo argumento (m1, m2 ó m3) indica el número mínimo de caracteres con que es presentado el primero. Si éste tiene menos de los mínimos, es rellenado por delante con espacios en blanco.

`Write( Letra );`	Escribe un carácter
`Write( Ristra );`	Escribe una ristra de caracteres
`WriteLn;`	Provoca un salto de línea. No lleva argumentos.

Los argumentos pueden ser variables, constantes o expresiones

2.4.- PRIMER PROGRAMA

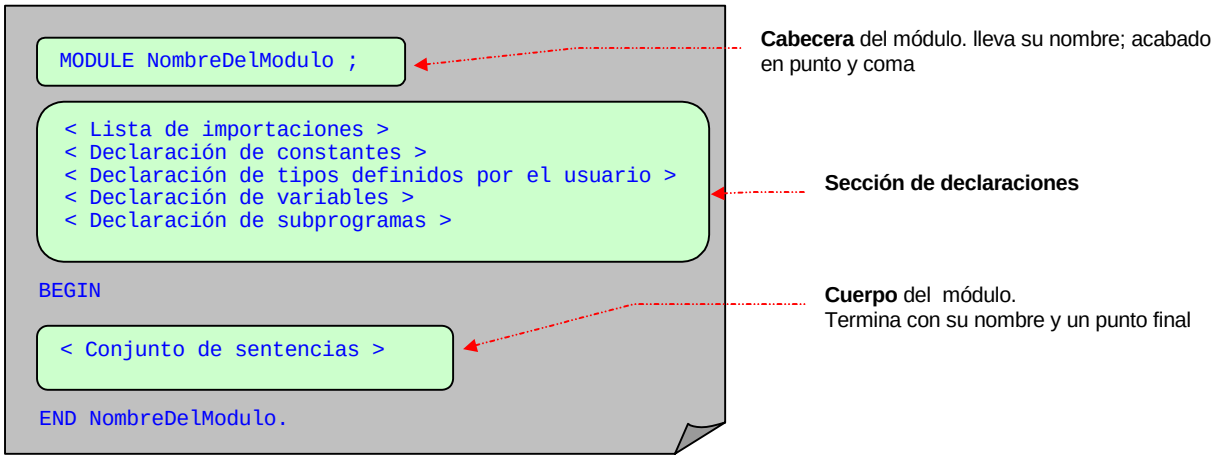
Estamos ya en condiciones de escribir y ejecutar el primer programa en Modula-2. Habitualmente cuando se está aprendiendo un lenguaje lo primero que se aprende es a saludar. Si este lenguaje es de programación, el primer programa suele ser uno que escriba un saludo en la pantalla.

```
MODULE EscribirHola;
    FROM InOut IMPORT WriteString, WriteLn;
BEGIN
    WriteString( "¡Hola, amigos!" );
    WriteLn ;
END EscribirHola.
```

2.5.- OPERACIONES DE LECTURA DESDE EL TECLADO

Son las inversas a las de escritura. También se realizan con llamadas a subprogramas. Leen un dato tecleado y lo almacenan en la variable que se indica en el argumento del subprograma.

OPERACIÓN	TIPO DEL DATO LEÍDO	BILIOTECA DONDE ESTÁ
`Read( C );`	carácter	InOut
`ReadInt( n );`	entero	InOut
`ReadCard( z );`	entero positivo	InOut
`ReadReal( R );`	número real	ReadInOut



La lista de importaciones indica al enlazador:

1º Los subprogramas que debe importar para enlazar con el módulo actual. Sus nombres aparecen tras la palabra IMPORT, separados por comas y un punto y coma tras el último.

2º El módulo desde donde debe importarlos. Su nombre aparece tras la palabra FROM

Si no todos los subprogramas necesarios están en el mismo módulo, se hacen varias listas de importaciones, una para cada módulo.

Inclusión de comentarios

Un comentario es un texto encerrado entre las parejas de símbolos:

(* al comienzo del texto, y
*) al final.

El compilador ignora los comentarios, no intenta traducirlos. Pero cualquier persona que lea el archivo fuente puede leerlos. Y si están hechos con buen criterio, pueden ayudarle a entender lo que hace un determinado bloque de programa.

Podemos transcribir a Modula-2 el ejemplo en el que se calculaba el producto de los tres primeros número enteros:

```
MODULE Factorial23 ; (* Calcula el producto de los 3 primeros números enteros *)

    FROM InOut IMPORT WriteString, WriteInt, WriteLn ;

    VAR
        V1, V2, V3 : INTEGER ;

    BEGIN

        V1 := 1 ;
        V2 := 2 ;
        V3 := 3 ;

        V1 := V1 * V2 ;
        V1 := V1 * V3 ;

        WriteString( "El producto de los 3 primeros números enteros es: " );
        WriteInt( V1, 2 ) ;
        WriteLn ;

    END Factorial23.
```

2.7.- PRINCIPIOS BÁSICOS DE LÓGICA BOOLEANA APLICADOS A MODULA-2

La lógica booleana estudia los objetos cuyo valor sólo puede ser uno cualquiera de los dos que hay en el conjunto { FALSO, CIERTO }, y de las operaciones que pueden realizarse con estos objetos.

2.7.1.- Tipo BOOLEAN

En Modula-2 hay un tipo de datos pensado especialmente para las variables que almacenan estos objetos: es el tipo BOOLEAN. Las variables o constantes declaradas de este tipo sólo pueden tener uno de estos dos valores: o FALSE o TRUE.

2.7.2.- Operadores de Relación

Sirven para construir los enunciados booleanos. Esta es la lista:

=	Igual
<> ó #	Distinto
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Ejemplos:

4 = 4	Es un enunciado cuyo valor es TRUE
4 < 4	Es un enunciado cuyo valor es FALSE
4 <= 4	Es un enunciado cuyo valor es TRUE
4 <> 4	Es un enunciado cuyo valor es FALSE

Los operandos utilizados a cada lado de un operador de relación pueden ser contantes, variables u otras expresiones, pero deben ser del mismo tipo. (¡No se pueden comparar sandías con melones!).

Siempre que se hace una comparación con un operador de relación, el operador devuelve un valor de tipo BOOLEAN que podemos asociar a una variable de este tipo.

Ejemplo:

```
VAR
     x : INTEGER ;
     B : BOOLEAN ;

BEGIN
     ReadInt( x) ;
     B := x < 100 ;
```

2.7.3. - Operadores booleanos

Se utilizan para combinar varios enunciados, formando un enunciado compuesto.

Operador AND. Actúa según se muestra en su tabla de verdad:

Enunciado A	Enunciado B	(Enunciado A) AND (Enunciado B)
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Operador OR. Actúa según se muestra en su tabla de verdad:

Enunciado A	Enunciado B	(Enunciado A) OR (Enunciado B)
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Operador NOT. Actúa sobre un único operando, invirtiendo su valor

Enunciado A	NOT(Enunciado A)
FALSE	TRUE
TRUE	FALSE

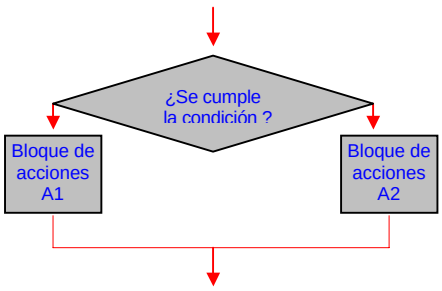
3 CONTROL DEL FLUJO

3.1.- INTRODUCCIÓN

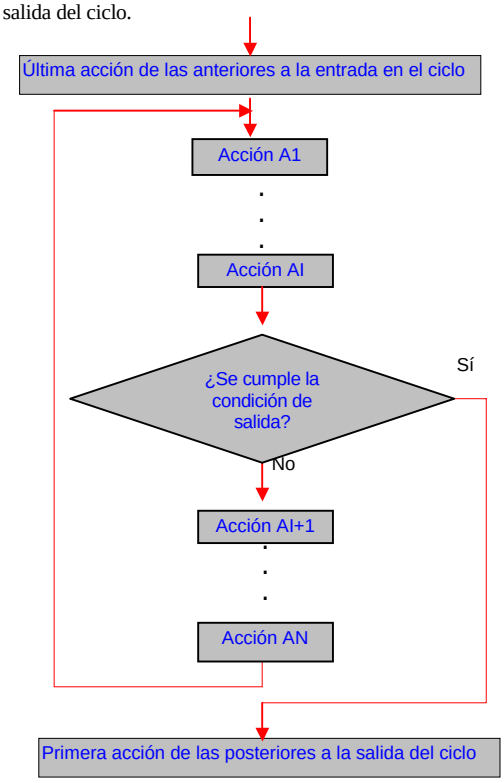
Cada programa de los vistos hasta ahora se componía de una secuencia de sentencias que se realizaban siempre en el mismo orden, en todas las ejecuciones del programa.

Esta es una situación poco realista, pues en la mayoría de las ocasiones nos encontraremos con alguna de estas dos circunstancias:

1ª Existe un condicionante según el cual, se cumpla o no, lo que hay que realizar es un bloque de acciones u otro.



2ª Existe un bloque de acciones que hay que repetir de forma cíclica hasta que se cumpla alguna condición de salida del ciclo.



3.2.- RAMIFICACIONES DE CONTROL

3.2.1.- Construcción IF THEN ELSE.

Su expresión más simple es:

```
IF  < Condición > THEN
    < Bloque 1 de sentencias >
ELSE
    < Bloque 2 de sentencias >
END ;
```

Esta construcción influye de esta manera en la selección de acciones que se han de realizar:

Si la Condición tiene el valor TRUE
- . Se ejecuta el Bloque 1 de sentencias situado entre THEN y ELSE
- . Se continúa con la sentencia que haya tras la línea de END

Si la Condición tiene el valor FALSE
- . Se ejecuta el bloque 2 de sentencias
- . Se continúa con la sentencia que haya tras la línea de END

Ejemplo:

En este programa el ordenador conoce un número (almacenado en una constante) y el usuario intenta adivinarlo. Sólo tiene una oportunidad. Si lo acierta, recibe un mensaje, y si falla, recibe otro diferente.

```
MODULE Adivina1 ;
    FROM InOut IMPORT WriteLn, ReadInt, WriteString ;

    CONST
        NumeroOculto = 37 ;
    VAR
        Conjetura : INTEGER ;
BEGIN
    WriteString("Intente adivinar el número que estoy pensando ") ;
    WriteLn ;
    WriteString("Dígame un número entre 1 y 100") ;
    WriteLn ;
    ReadInt( Conjetura ) ;
    WriteLn ;

    IF Conjetura = NumeroOculto THEN
        WriteString("¡ Vaya suerte, ha acertado !" );
    ELSE
        WriteString("Lo siento, ha fallado" );
    END ;

    WriteLn ;
END Adivina1.
```

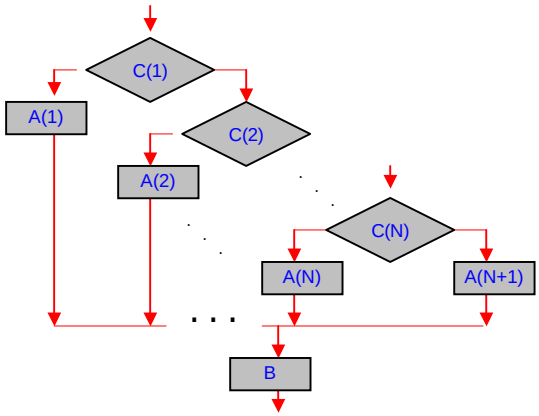
3.2.2.- Esquema de selección en cascada

En cada bifurcación C(I) se hace la siguiente selección:

Si se cumple C(I)
- Se realiza el bloque de acciones A(I)
- Se continúa con la acción B

Si no se cumple C(I)
Si I< N
- Se pasa a examinar la condición C(I+1)

Si I = N
- Se realiza el bloque de acciones A(N+1)
- Se continúa con la acción B



El resultado es que se examina cada condición sólo si no se ha cumplido ninguna de las anteriores.

En Modula-2 hay dos formas de codificar este esquema:

3.2.2.1.- Esquema de IF THEN ELSE anidadas

```
IF   C( 1 )    THEN  A( 1 )
ELSE
    IF    C( 2 )   THEN    A( 2 )
    ELSE
       .
         .
           .
            IF   C( N )    THEN    A( N )
            ELSE   A( N+1 )
            END  ;    (*  Del   IF  N-ésimo  *)
           .
         .
       .
    END ;       (*  Del   IF  2-ésimo  *)
END ;       (*  Del   IF  1-ésimo  *)
```

Cada END cierra el IF inmediatamente anterior que queda sin cerrar.

Ejemplo :

En este programa el ordenador pide un número entero entre 1 y 100 al usuario y le dice el rango en que está comprendido.

```
MODULE Rangos1 ;
    FROM InOut IMPORT WriteLn, ReadInt, WriteString ;
    VAR
        NumeroLeido : INTEGER ;
BEGIN
    WriteString("Dígame un número entre 1 y 100") ;
    WriteLn ;
    ReadInt( NumeroLeido ) ;
    WriteLn ;

    IF NumeroLeido < 25 THEN
          WriteString("Es menor que 25" );
    ELSE
        IF ( NumeroLeido >= 25 ) AND ( NumeroLeido < 50 ) THEN
             WriteString("Está  en el intervalo [25, 50)" );
        ELSE
            IF ( NumeroLeido >= 50 ) AND ( NumeroLeido <= 75 ) THEN
                 WriteString("Está en el intervalo [50, 75]" );
            ELSE WriteString("Es mayor que 75" );
            END ;
        END ;
    END ;

    WriteLn ;
END Rangos1.
```

3.2.2.2.- Esquema IF THEN ELSIF

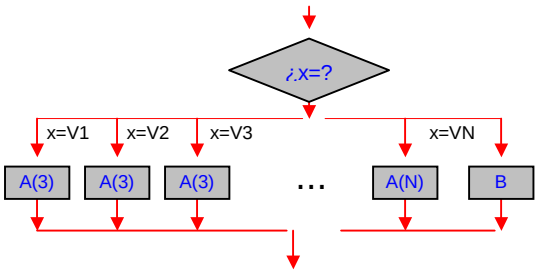
Codifica de una forma más simple y clara la selección en cascada:

```
IF     C( 1 )    THEN A( 1 ) ;
ELSIF  C( 2 )    THEN A( 2 ) ;
ELSIF  C( 3 )    THEN A( 3 ) ;
    .
    .
    .
ELSIF  C( N )    THEN  A( N ) ;
ELSE                   A( N+1 ) ;
END ;
```

Ejemplo:
Versión del anterior programa, ahora utilizando ELSIF

```
MODULE Rangos2 ;
    FROM InOut IMPORT WriteLn, ReadInt, WriteString ;

    VAR
        NumeroLeido : INTEGER ;
BEGIN
    WriteString("Dígame un número entre 1 y 100") ;
    WriteLn ;
    ReadInt( NumeroLeido ) ;
    WriteLn ;

    IF NumeroLeido < 25 THEN
          WriteString("Es menor que 25" );
    ELSIF ( NumeroLeido >= 25 ) AND ( NumeroLeido < 50 ) THEN
          WriteString("Está en el intervalo [25, 50)" );
    ELSIF ( NumeroLeido >= 50 ) AND ( NumeroLeido <= 75 ) THEN
          WriteString("Está en el intervalo [50, 75]" );
    ELSE
          WriteString("Es mayor que 75" );
    END ;

    WriteLn ;
END Rangos2.
```

3.2.3.- Ramificaciones de Clasificación



V1, V2, V3, ... , Vn son cada uno un conjunto de valores posibles de la variable x
A1, A2, A3, ... , AN y B son bloques de acciones

Esta estructura determina que el flujo de operaciones sea de la siguiente manera:

- Se examina el valor de la variable x
  Si coincide con alguno de los elementos del conjunto V1 de valores,
  ☞ se ejecuta el bloque de acciones A1

  Si coincide con alguno de los elementos del conjunto V2 de valores,
  ☞ se ejecuta el bloque de acciones A2

  .
  .
  .

  Si coincide con alguno de los elementos del conjunto VN de valores,
  ☞ se ejecuta el bloque de acciones AN

  Si no se ha ejecutado ninguno de los anteriores bloques de acciones
  ☞ se ejecuta el bloque de acciones B

```
CASE   <  Expresión  >  OF
            < Elementos  de V1 > :   A1    |
            < Elementos  de V2 > :   A2    |
            < Elementos  de V3 > :   A3    |
                        .
                        .
                        .
            < Elementos  de V3 > :   AN
ELSE     B
END ;
```

Los tipos de la variable x pueden ser INTEGER, CARDINAL, CHAR, enumerado, o subrango. Hay que cuidar que a un mismo valor no le pueda corresponder más de un mismo bloque de acciones. En algunas ocasiones no se debe realizar nada si no se cumple la última condición. Por eso, se puede omitir el ELSE de las estructuras IF simple, de los IF anidados, del ELSIF y del CASE.

Ejemplo:

```
MODULE Rangos3 ;
    FROM InOut IMPORT WriteLn, ReadInt, WriteString ;

    VAR
        NumeroLeido, Caso : INTEGER ;
BEGIN
    WriteString("Dígame un número entre 1 y 100") ;
    WriteLn ;
    ReadInt( NumeroLeido ) ;
    WriteLn ;

    Caso := NumeroLeido DIV 25 ;

    CASE Caso OF
        0 : WriteString("Es menor que 25" )              |
        1 : WriteString("Está en el intervalo [25, 50)" ) |
        2 : WriteString("Está en el intervalo [50, 75)" ) |
        3 : WriteString("Es mayor o igual que 75" )
    ELSE   (* No se realiza nada, se podría omitir *)
    END ;

    WriteLn ;
END Rangos3.
```

3.2.4.- CASE versus ELSIF

La estructura CASE aporta más claridad a la lectura de un programa. Como norma general, se recomienda usarla cuando se puede identificar una unidad lógica en la que haya unos casos muy interrelacionados, y cada caso tenga visiblemente asociada una acción o bloque de acciones.

Hay ocasiones en que no se puede utilizar CASE y sí ELSIF:

1ª. Cuando la condición se deduce a partir de datos almacenados en variables:

```
VAR    A, B, C, D : INTEGER ;
BEGIN
     IF     A = B THEN  < Acciones  A1 > ;
     ELSIF  A = C THEN  < Acciones  A2 > ;
     ELSIF  A = C THEN  < Acciones  A3 > ;
    END ;
END
```

2ª. Cuando la condición no se expresa con igualdades estrictas.

```
IF     A < 10  THEN   < Acciones A1 > ;
ELSIF  A = 10  THEN   < Acciones A2 > ;
ELSIF  A > 10  THEN   < Acciones A3 > ;
END ;
```

3.3.- ITERACIONES

En un tema anterior hemos hecho un programa para calcular el factorial de 3 . Aquel programa tenía muchas limitaciones. Entre otras:
- Sólo servía para calcular el factorial de 3.
- Necesitaba tres variables.
- Tenía un fragmento de código que se repetía varias veces, una vez con cada variable.

Si quisieramos calcular el factorial de un número N muy grande, con aquel algoritmo, necesitaríamos N variables, y habría un framento de código que se repetiría N veces.

Podemos mejorarlo si aprovechamos la existencia de esas iteraciones. De hecho, la potencia y utilidad de los ordenadores se pone de manifiesto cuando hay que realizar una misma tarea un número muy elevado de veces. Estas iteraciones las hacen en los bucles.

Bucle: Es una construcción en la que hay un bloque de acciones que se repiten de forma cíclica hasta que se cumple una determinada condición.

En Modula-2 hay cuatro esquemas de bucle:

3.3.1.- Bucle FOR

Se usa cuando el número de repeticiones es conocido previamente a la entrada en el bucle. Este número lo indica una variable llamada **índice del bucle**. Implementa este algoritmo:

Desde que el índice del bucle I tiene su valor inicial Ii hasta que alcanza su valor final If :
- se realiza el bloque de sentencias B
- se le añade a I un incremento constante

Tiene esta sintaxis:

```
FOR   < I := Ii >   TO   < If >  BY  < Incremento > DO
       < Bloque de sentencias  B >    (* Cuerpo del bucle *)
END ;
```

Ejemplo:
Programa que calcula el factorial de un número cualquiera N

```
MODULE Factor2 ;
    FROM InOut IMPORT ReadCard, WriteLn, WriteCard, WriteString ;
    VAR
        Acumulador, Contador, NumeroLeido : CARDINAL ;
    BEGIN
        WriteString( "Dígame el número cuyo factorial quiere calcular: " );
        ReadCard( NumeroLeido ) ;
        Acumulador := 1 ;

        FOR Contador := 2 TO NumeroLeido BY 1  DO
            Acumulador := Acumulador * Contador ;
        END ;

        WriteLn ;
        WriteString( "El factorial de " );
        WriteCard( NumeroLeido, 2 ) ;
        WriteString( " es " );
        WriteCard( Acumulador, 3 ) ;
        WriteLn ;
    END Factor2.
```

Comentarios:
- La variable `I` utilizada como índice de bucle sólo puede ser INTEGER o CARDINAL.
- Puede omitirse la cláusula BY si el incremento es +1
- Si el incremento es positivo e `Ii` > `If` , no se realiza el bloque de sentencias
- Si el incremento es negativo e `Ii` < `If` , no se realiza el bloque de sentencias
- Aunque puede modificarse el índice de bucle dentro del cuerpo del bucle, no es recomendable.

3.3.2.- Bucle WHILE

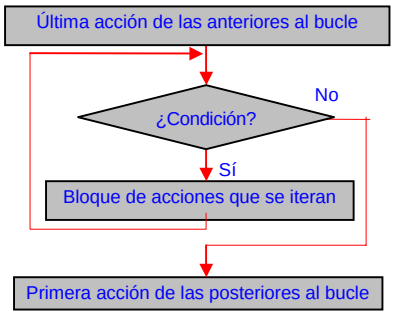
Implementa este algoritmo :

Mientras se cumpla la condición C
☞ Se ejecuta el bloque de acciones B

Tiene esta sintaxis:

```
WHILE  < Condición C >  DO
       < Bloque de acciones B > ;
END ;
```

Es decir, tiene este diagrama de flujo :



La comprobación de salida se hace al comienzo de cada iteración. Si la condición nunca es cierta, el cuerpo del bucle no se ejecuta nunca.

Ejemplo:
Supongamos que buscamos el menor número entero *n* tal que su factorial *n!* es mayor o igual que un tope dado. (*n!* ≤ Tope).

```
MODULE TopeFact ;
    FROM InOut IMPORT WriteLn, WriteInt, ReadInt, WriteString ;
    VAR
        Acumulador, Tope, Contador : INTEGER ;
BEGIN
    WriteString("Dígame el tope del factorial:  ") ;
    ReadInt( Tope ) ;
    WriteLn ;
    Acumulador := 1 ;
    Contador := 0 ;

    WHILE  Acumulador < Tope  DO
        Contador := Contador + 1 ;
        Acumulador := Acumulador * Contador ;
    END ;

    WriteString("El primer número entero cuyo factorial es mayor o igual que ") ;
    WriteInt( Tope, 3 ) ;
    WriteString(" es ") ;
    WriteInt( Contador, 3 ) ;
    WriteLn ;
    WriteString("Su factorial es ") ;
    WriteInt( Acumulador, 3 ) ;

END TopeFact.
```

3.3.3.- Bucle REPEAT

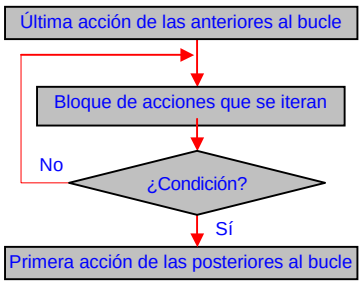
Implementa este algoritmo :

☞ Se ejecuta el bloque de acciones B
Hasta que sea cierta la condición C

Tiene esta sintaxis:

```
REPEAT
       < Bloque de acciones > ;
UNTIL  < Condición  C > ;
```

Es decir, tiene este diagrama de flujo :



La comprobación de salida se hace al final de cada iteración. Este bucle se usa cuando se necesita que la iteración se ejecute al menos una primera vez.

Ejemplo:
Programa que calcula la media aritmética de los números que se van introduciendo por el teclado. Cuando el usuario quiera parar, debe escribir un cero.

```
MODULE Media1;
FROM InOut     IMPORT Write, WriteCard, WriteLn, WriteString ;
FROM RealInOut IMPORT ReadReal, WriteReal ;

VAR
    Contador : CARDINAL ;
    Acumulador, NumeroLeido : REAL ;
BEGIN
    Acumulador := 0.0 ;
    Contador   := 0 ;

    REPEAT
          WriteLn ;
          WriteString("Deme el dato número ") ;
          WriteCard( Contador, 2 ) ;
          Write(':') ;
          ReadReal( NumeroLeido );
          Acumulador := Acumulador + NumeroLeido ;
          Contador   := Contador + 1 ;
    UNTIL ( NumeroLeido = 0.0 ) ;

    Contador := Contador - 1 ;   (* Para no contar la última lectura *)

    IF Contador > 0 THEN         (* Para evitar una división por cero *)
        Acumulador := Acumulador / FLOAT( Contador ) ;  (* Para que haya compatibilidad de tipos *)
    END ;

    WriteLn ;
    WriteString("La media aritmética es: ") ;
    WriteReal( Acumulador, 3 ) ;
    WriteLn ;
END Media1.
```

Comentarios:

- El número de datos sobre los que hay que realizar el promedio es uno menos que el número de datos leídos. El 0.0 último no debe tenerse en cuenta. Por este motivo hay que hacer una restauración del valor válido de la variable que se utiliza como contador.
- Para evitar una incompatibilidad de tipos, se ha utilizado el subprograma FLOAT, que convierte el argumento CARDINAL a REAL.

3.3.4.- Bucle LOOP

Implementa este algoritmo :

- Iterar "eternamente":
  - Ejecutar el bloque de acciones A
    Si se cumple la condición C de salida, terminar las iteraciones.
  - Ejecutar el bloque de acciones B

Tiene esta sintaxis:

```
LOOP
    < Bloque de acciones A > ;
    IF  < Condición  C >  THEN  EXIT END ;
    < Bloque de acciones B > ;
END ;
```

Es útil cuando la condición de salida está en el interior del cuerpo del bucle, y hay que salir cuando se han ejecutado una parte de las sentencias del cuerpo del bucle, pero no todas.

Puede haber varias cláusulas EXIT en diferentes puntos del cuerpo del bucle, cada una asociada a una condición de salida.

Ejemplo:

```
MODULE Media2;

FROM InOut     IMPORT Write, WriteCard, WriteLn, WriteString ;
FROM RealInOut IMPORT ReadReal, WriteReal ;

VAR
    Contador : CARDINAL ;
    Acumulador, NumeroLeido : REAL ;
BEGIN
    Acumulador := 0.0 ;
    Contador   := 0 ;

    LOOP
          WriteLn ;
          WriteString("Deme el dato número ") ;
          WriteCard( Contador, 2 ) ;
          Write('º') ;
          ReadReal( NumeroLeido );
          IF NumeroLeido = 0.0 THEN EXIT END ;
          Acumulador := Acumulador + NumeroLeido ;
          Contador   := Contador + 1 ;
    END ;

    IF Contador > 0 THEN         (* Para evitar una división por cero *)
        Acumulador := Acumulador / FLOAT( Contador ) ;  (* Para que haya compatibilidad de tipos *)
    END ;

    WriteLn ;
    WriteString("La media aritmética es: ") ;
    WriteReal( Acumulador, 3 ) ;
    WriteLn ;
END Media2.
```

Como en la última pasada por el bucle se sale sin incrementar la variable contadora, no hace falta restaurarla después de haber salido.

3.4.- ESTUDIO COMPARADO DE LOS CUATRO BUCLES

. El bucle LOOP EXIT es el esquema general de iteraciones. Todos los demás son casos particulares:
- Si su bloque de Acciones A está vacío, se reduce al bucle REPEAT UNTIL
- Si su bloque de Acciones B está vacío, se reduce al bucle WHILE DO

El esquema

```
WHILE  < Condición > DO
       < Acciones >
END ;
```

puede expresarse :

```
IF < Condición > THEN
     REPEAT
               < Acciones >
    UNTIL  <  NOT(Condición) >
END ;
```

El esquema

```
REPEAT
               < Acciones >
UNTIL  < Condición >
```

es equivalente a:

```
< Acciones >
WHILE  < NOT( Condición )  DO
    < Acciones >
END ;
```

El bucle FOR puede sintetizarse con cualquiera de los anteriores, por ejemplo:

```
FOR  x := ValorInicial  TO ValorFinal  BY Incremento  DO
       < Acciones >
```

si (Incremento > 0) es equivalente a:

```
x := Valor Inicial ;
WHILE  x  < ValorFinal  DO
       < Acciones >
       x := x + Incremento ;
END ;
```

y si (Incremento < 0) es equivalente a:

```
x := Valor Inicial ;
WHILE  x  > ValorFinal  DO
       < Acciones >
       x := x + Incremento ;
END ;
```

3.5.- BUCLES ANIDADOS

Dentro de cada bucle puede haber otro bucle de cualquier tipo.
Ejemplos : Este framento de programa escribe 1000 veces la ristra de caracteres "Hola"

```
FOR   i := 1  TO  100  DO
   FOR  j := 1  TO  10  DO
       WriteString( "Hola" ) ;
   END ;
END ;
```

La salida generada por este:

```
MODULE  Bucles ;
    FROM InOut IMPORT WriteLine, WriteString, WriteLn, WriteCard ;
    VAR Indice1, Indice2 : CARDINAL ;

BEGIN
    Indice1 := 1 ;
    WriteLn ;
    WriteLine("Este programa muestra el funcionamiento de los bucles anidados");

    WHILE Indice1 <= 3 DO
        WriteLn ;
        WriteString("Bucle externo. Pasada Nº ") ;
        WriteCard( Indice1, 1 ) ;
        INC( Indice1 ) ;

        FOR Indice2 := 1 TO 4 DO
            WriteLn ;
            WriteString("                    Bucle interno. Pasada Nº ") ;
            WriteCard( Indice2, 1 ) ;
        END ;
    END ;
END Bucles .
```

es:

```
Este programa muestra el funcionamiento de los bucles anidados

Bucle externo. Pasada Nº 1
                    Bucle interno. Pasada Nº  1
                    Bucle interno. Pasada Nº  2
                    Bucle interno. Pasada Nº  3
                    Bucle interno. Pasada Nº  4
Bucle externo. Pasada Nº 2
                    Bucle interno. Pasada Nº  1
                    Bucle interno. Pasada Nº  2
                    Bucle interno. Pasada Nº  3
                    Bucle interno. Pasada Nº  4
Bucle externo. Pasada Nº 3
                    Bucle interno. Pasada Nº  1
                    Bucle interno. Pasada Nº  2
                    Bucle interno. Pasada Nº  3
                    Bucle interno. Pasada Nº  4
```

Debe evitarse utilizar un mismo índice de bucle para varios bucles anidados.
Si hay varios bucles anidados, y uno de ellos es LOOP, sus EXIT pueden estar dentro de otros bucles. De ser así provocarán la salida de todos esos bucles interiores.

Ejemplo:

```
LOOP
    REPEAT
        WHILE   CondicionWhile  DO
                . . .
                IF CondicionLoop THEN EXIT END ;
                . . .
        END ;
   UNTIL   CondicionUntil
 END ;
```

Si `CondicionLoop = TRUE`, el flujo del programa sale de todos los bucles, aunque `CondicionWhile` sea aún `TRUE` y aunque `CondicionUntil` sea aún `FALSE`.

Cuando se anidan varios bucles LOOP, cada EXIT está asociado al LOOP interno de todos los anteriores que no estén cerrados con un END. Ejemplo:

```
LOOP                (* Abre el nivel   1 *)
  ...
  ...EXIT...        (* Sale del nivel  1 *)
  LOOP              (* Abre el nivel   2 *)
    ...
    LOOP            (* Abre el nivel   3 *)
      ...
      ...EXIT...    (* Sale del nivel  3 *)
      ...
    END             (* Cierra el nivel 3 *)
    ...
    ...EXIT..       (* Sale del nivel  2 *)
    ...
  END               (* Cierra el nivel 2 *)
  ...
END                 (* Cierra el nivel 1 *)
```

Es evidente que un programa que abuse de este esquema se hace muy ilegible.

3.6.- RECOMENDACIONES PARA UNA ACERTADA ELECCIÓN DEL BUCLE

- Si se conoce previamente el número de iteraciones, es preferible, por su simplicidad, el FOR.
- Si el número de iteraciones es indeterminado, y la comprobación de salida debe hacerse al empezar, es preferible el WHILE.
- Si el número de iteraciones es indeterminado, y las acciones deben realizarse al menos una vez, es preferible el REPEAT.
- Si hay una serie de circunstancias, que cada una por separado deben provocar la salida, es preferible el LOOP, con una cláusula EXIT para cada circunstancia.
- Si se debe poder salir simultáneamente de varios bucles anidados, es preferible usar el LOOP
- Pero el bucle LOOP debe evitarse siempre que se pueda. Produce un código confuso, pues :
  - . la condición de salida puede estar en cualquier parte
  - . Rompe la relación una salida por cada entrada.

4 PROCEDIMIENTOS

4.1 INTRODUCCIÓN

En casi cualquier programa se puede apreciar que hay algún subprograma o bloque de código que se comporta como una unidad lógica. Sus sentencias están muy interrelacionadas entre sí.

Procedimiento Es un subprograma definido de forma separada del resto del programa.

Es aconsejable aislar alguna tarea en un procedimiento cuando:
- El programa es muy largo y complejo. Se gana legibilidad.
- Hay una tarea que se repite varias veces en diferentes puntos del programa.

Cada procedimiento aparece en tres lugares, de tres formas diferentes:

Definición. Es la especificación de su nombre y de las acciones que ejecuta.
Declaración. Si un programa utiliza un procedimiento, debe declararlo antes de utilizarlo.
Llamada. Es la invocación que se hace al procedimiento en una expresión, para ejecutarlo.

La tendencia general en los lenguajes de programación modernos es modularizar todas las rutinas en procedimientos separados, manteniendo en el módulo principal del programa exclusivamente una serie de llamadas a los procedimientos secundarios.

Según donde esté declarada, una variable puede ser:

Variable global. Declarada fuera de todo procedimiento.
Es accesible por todos los procedimientos.
Existe durante toda la ejecución del programa.
Variable local. Declarada dentro de un procedimiento.
Sólo es accesible por ese procedimiento, y por los que tenga anidados.
Se destruye al salir del procedimiento. No conserva su valor entre llamadas
Variable formal . Es la utilizada en la lista de argumentos en la definición de un procedimiento.
Variable estática. Es local en un procedimiento, pero no se destruye al terminar el procedimiento. Conserva su valor hasta la siguiente llamada.

Tipos de Procedimientos

Procedimiento puro. Cada vez que es llamado, procede a realizar una tarea, siempre la misma.
Procedimiento parametrizado Necesita que se le pase un conjunto de datos, llamados **argumentos.**
Procedimiento tipo función. Además de realizar una tarea, devuelve un dato al punto desde donde se hizo la llamada

4.2.- PROCEDIMIENTOS EN MODULA-2

Forma general de la definición de un procedimiento:

```
PROCEDURE  < Nombre del procedimiento > ( Lista de parámetros ) : < Tipo de dato devuelto > ;
        < Sección de declaraciones >
BEGIN
          < Bloque de acciones >
END       < Nombre del procedimiento >
```

4.2.1.- Procedimientos puros

Ejemplo:

```
MODULE TareasPuras ;
    FROM InOut IMPORT WriteLn, WriteString ;
    PROCEDURE Tarea1 ;
        BEGIN
            WriteLn ;
            WriteString("Esta frase se escribe desde el procedimiento Tarea1" ) ;
            WriteLn ;
        END Tarea1 ;
    PROCEDURE Tarea2 ;
        BEGIN
            WriteLn ;
            WriteString("Esta frase se escribe desde el procedimiento Tarea2" ) ;
            WriteLn ;
        END Tarea2 ;
    BEGIN
        Tarea1 ;
        Tarea2 ;
    END TareasPuras.
```

4.2.2.- Procedimientos parametrizados

En la definiciones y declaraciones la lista de argumentos va entre paréntesis. Cada nombre de variable va seguido de dos puntos (:) y de su tipo. Pueden agruparse varias variables de un mismo tipo separadas por comas. Las de tipo diferente se separan por punto y coma. Los tipos de los argumentos utilizados en la llamada y en su declaración deben coincidir con los utilizados en la definición del procedimiento.

```
MODULE TareaParametrizada ;
    FROM InOut IMPORT WriteLn, Write, ReadCard, Read, WriteString ;
    VAR
        x, y : CARDINAL ;
        LetraDeRelleno : CHAR ;

    PROCEDURE DibujaCuadrado( Longitud, Altura : CARDINAL ; Letra : CHAR ) ;
        VAR i, j : CARDINAL ;

        BEGIN
            FOR i := 1 TO Altura DO
                FOR j := 1 TO Longitud DO
                    Write( Letra ) ;
                END ;
                WriteLn ;
            END ;
        END DibujaCuadrado ;

    BEGIN
        WriteString("Vamos a dibujar un rectángulo" ) ;
        WriteLn ;
        WriteString("Deme la longitud del rectángulo " ) ;
        ReadCard( x ) ;
        WriteLn ;
        WriteString("Deme la altura del rectángulo " ) ;
        ReadCard( y ) ;
        WriteLn ;
        WriteString("Deme la letra con que lo rellenaré  " ) ;
        Read( LetraDeRelleno ) ;
        WriteLn ;
        DibujaCuadrado( x, y, LetraDeRelleno ) ;
    END TareaParametrizada.
```

4.2.3 Procedimientos tipo función

El tipo de dato devuelto se indica detrás de la lista de parámetros. Este dato puede asignarse a una variable de ese mismo tipo.

Ejemplo:
Definimos un procedimiento tipo función que calcula la potencia de un número real elevado a un exponente entero positivo.

```
MODULE Potencias ;
    FROM InOut IMPORT ReadInt, WriteLn, WriteInt, WriteString ;
    FROM RealInOut IMPORT ReadReal, WriteReal ;
    VAR
        n : INTEGER ;
        x, Resultado : REAL ;

    PROCEDURE Potencia( Base : REAL ; Exponente : INTEGER ) : REAL ;
        VAR
            i : INTEGER ;
            Acumulador : REAL ;
        BEGIN
            Acumulador := 1.0 ;
            FOR i := 1 TO Exponente DO
                Acumulador := Acumulador * Base ;
            END ;
            RETURN Acumulador ;
        END Potencia ;
    BEGIN
        WriteString( "--  Este programa sirve para calcular potencias " ) ;
        WriteString( "de exponente entero positivo --" );
        WriteLn ;
        WriteLn ;
        WriteString( "Deme la base: " );
        ReadReal( x ) ;
        WriteLn ;
        WriteString( "Deme el exponente: " );
        ReadInt( n ) ;
        Resultado := Potencia( x, n ) ;
        WriteLn ;
        WriteLn ;
        IF ( n >= 0) THEN
            WriteString( "El resultado es: " );
            WriteReal( Resultado, 5 ) ;
        END ;
        WriteLn ;
    END Potencias.
```

Puede haber varias sentencias RETURN en un mismo procedimiento, por ejemplo:

```
    PROCEDURE Potencia( Base : REAL ; Exponente : INTEGER ) : REAL ;
        VAR
            i : INTEGER ;
            Acumulador : REAL ;
        BEGIN
             IF Exponente = 0 THEN RETURN 1.0 END ; (* Así no se pierde tiempo entrando en el
                                                       bucle *)
            Acumulador := 1.0 ;
            FOR i := 1 TO Exponente DO
                Acumulador := Acumulador * Base ;
            END ;
            RETURN Acumulador ;
        END Potencia ;
```

En cuanto se encuentra la primera sentencia RETURN ejecutable, se deja sin realizar el resto del procedimiento.

Los procedimientos tipo función, aunque no utilicen argumentos, en la definición y en la llamada hay que escribir los paréntesis.

4.2.4.- Paso por Valor, Paso por Referencia

Hay dos formas de pasar un argumento a un procedimiento:

Paso por valor. Al hacer la llamada, se hace una copia del dato argumento. La variable formal es sustituida por esta copia. y dentro del procedimiento se trabaja con esa copia. Si es modificada, el dato original no sufre esa modificación. Este es el modo por defecto de pasar un argumento.
Paso por referencia. Al hacer la llamada, la información que recibe el procedimiento es el lugar donde está almacenado el dato. Esto tiene como consecuencia que desde dentro del procedimiento podemos modificar el dato. La forma de indicarle al compilador que un argumento lo queremos pasar por referencia es anteponer al nombre de la variable la palabra VAR.

Ejemplo:
Este programa tiene un procedimiento para intercambiar los valores de dos variables.

```
MODULE Cambiazo ;
    FROM InOut IMPORT  WriteLn, ReadInt, WriteInt, WriteString ;
    VAR
        x1, x2 : INTEGER ;

    PROCEDURE Intercambia( VAR Primero: INTEGER; VAR Segundo: INTEGER ) ;
        VAR
            Auxiliar : INTEGER ;
        BEGIN
            Auxiliar := Primero  ;
            Primero  := Segundo  ;
            Segundo  := Auxiliar ;

        END Intercambia ;

    BEGIN
        WriteString( " Este programa intercambia dos números enteros " ) ;
        WriteLn ;
        WriteLn ;
        WriteString( "Deme el primer número: " );
        ReadInt( x1 ) ;
        WriteLn ;
        WriteString( "Deme el segundo número: " );
        ReadInt( x2 ) ;
        WriteLn ;

        Intercambia( x1, x2 ) ;

        WriteString( "Después del intercambio la situación es: " );
        WriteLn ;
        WriteLn ;
        WriteString( "Primer número: " );
        WriteInt( x1, 2 ) ;
        WriteLn ;
        WriteString( "Segundo número: " );
        WriteInt( x2, 2 ) ;
        WriteLn ;

    END Cambiazo.
```

4.2.5.- Procedimientos Anidados. Visibilidad

Procedimiento anidado es el que está definido dentro de otro procedimiento. El anidamiento de bloques refleja el proceso de un análisis descendente, en el que hay ciertos bloques que sólo son utilizados por otros superiores. En Modula-2 cada procedimiento sólo puede acceder a los de su mismo nivel de anidamiento, y a los que tiene anidados dentro de sí mismo.

Las reglas de visibilidad de los identificadores en los procedimientos son:

1ª Regla Un identificador es reconocido por el procedimiento en que se declara; y por todos los procedimientos anidados en éste, salvo que se deba aplicar la 2ª regla.
2ª Regla Si un procedimiento anidado utiliza el mismo nombre de identificador que otro más externo, al hacer uso de ese nombre el identificador aludido es el más interno

Ejemplos:

```
MODULE Anidado1 ;
    FROM InOut IMPORT WriteLn, WriteString, WriteInt ;

    PROCEDURE Externo ;
        VAR
            IndiceBucleExterno : INTEGER ;

            PROCEDURE Interno ; (* Esta definición está dentro del externo *)
                VAR
                    IndiceBucleInterno : INTEGER ;
            BEGIN
                WriteString("El índice del bucle externo es: " ) ;
                WriteInt( IndiceBucleExterno, 1 ) ;
                WriteLn ;
                FOR IndiceBucleInterno := -5  TO 5  DO
                    WriteInt( IndiceBucleInterno, 3 ) ;
                END ;
                WriteLn ;
            END Interno ;
    BEGIN  (* Cuerpo del externo *)
        FOR Contador := 1 TO 3 DO
            WriteLn ;
            WriteLn ;
            WriteString("En este punto del externo se llama al interno.") ;
            Interno ;
        END ;
   END Externo ;

   BEGIN (* Cuerpo del Módulo principal *)
        Externo ;
   END Anidado1.
```

Salida generada:

```
En este punto del externo se llama al interno. El índice del bucle externo es: 1
-5  -4  -3  -2  -1  0  1  2  3  4  5

En este punto del externo se llama al interno. El índice del bucle externo es: 2
-5  -4  -3  -2  -1  0  1  2  3  4  5

En este punto del externo se llama al interno. El índice del bucle externo es: 3
-5  -4  -3  -2  -1  0  1  2  3  4  5
```

Si las dos variables (la del bucle externo y la del interno), tienen el mismo nombre:

```
MODULE Anidado2 ;
    FROM InOut IMPORT WriteLn, WriteString, WriteInt ;

    PROCEDURE Externo ;
        VAR
            IndiceBucle : INTEGER ;

            PROCEDURE Interno ; (* Esta definición está dentro del externo *)
                VAR
                    IndiceBucle : INTEGER ;   (* Tiene el mismo nombre *)
            BEGIN
                WriteLn ;
                FOR IndiceBucle := -5  TO 5  DO
                    WriteInt( IndiceBucle, 3 ) ;
                END ;
                WriteLn ;
            END Interno ;
    BEGIN  (* Cuerpo del externo *)
        FOR IndiceBucle := 1 TO 3 DO
            WriteLn ;
            WriteLn ;
            WriteString("En este punto del externo se llama al interno.") ;
            Interno ;
        END ;
   END Externo ;

   BEGIN (* Cuerpo del Módulo principal *)
        Externo ;
   END Anidado2.
```

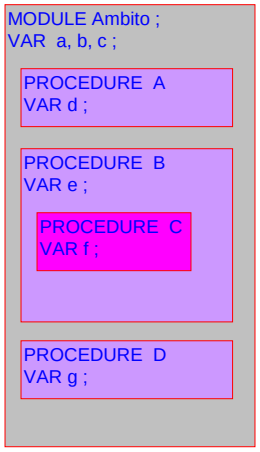
Salida generada:

```
En este punto del externo se llama al interno.
-5  -4  -3  -2  -1  0  1  2  3  4  5

En este punto del externo se llama al interno.
-5  -4  -3  -2  -1  0  1  2  3  4  5

En este punto del externo se llama al interno.
-5  -4  -3  -2  -1  0  1  2  3  4  5
```

Cuando se tienen dudas del ámbito de cada variable, puede ser muy útil realizar un diagrama del anidamiento de los procedimientos. Ejemplo:

```
MODULE Ambito
  VAR
     a, b, c : REAL ;

 PROCEDURE  A ;
     VAR
       d : INTEGER ;
 BEGIN
    . . .    (* Cuerpo del A *)
 END A ;

 PROCEDURE  B ;
    VAR
       e : INTEGER ;

    PROCEDURE  C ;
       VAR
         f : BOOLEAN ;
       BEGIN   (* C dentro de B *)
         . . .
      END  C ;

   BEGIN
     . . .     (* Cuerpo del  B *)
   END B ;

   PROCEDURE  D
      VAR g :INTEGER ;
   BEGIN
     . . .     (* Cuerpo del D *)
   END  D


   BEGIN
     . . .     (* Cuerpo del MODULE *)
   END Ambito.
```

DIAGRAMA DE ANIDAMIENTO



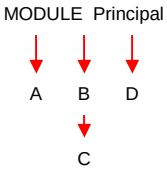


TABLA DE VISIBILIDAD DE MÓDULOS Y VARIABLES

	MODULE	A	B	C	D
Módulos accesibles	A, B, D	B, D	A, C, D		A, B
Variables reconocidas	a, b, c	a, b, c, d	a, b, c, e	a, b, c, e, f	a, b, c, g

4.2.6.- Problemas que origina una Situación Confusa en el Ámbito de los Identificadores

4.2.6.1.- Redefinición de identificadores

Si en un procedimiento anidado declaramos un objeto con un nombre que es igual que un identificador más externo, perdemos la posibilidad de acceder al más externo. Y aunque no pretendamos acceder al externo, el empleo de objetos diferentes con un mismo nombre aumenta la complejidad y legibilidad del programa.

4.2.6.2.- Efectos colaterales.

Ocurren cuando un procedimiento altera el valor de una variable global. Si esta modificación se hace sin que sea advertida fuera de ese procedimiento, puede ser una fuente de errores. El uso de efectos colaterales sólo está justificado cuando hay que pasar una lista muy larga de argumentos. En este caso, en lugar de trabajar con las variables formales se trabaja con las globales. Aún así, este método tiene una limitación, y es que en cada llamada siempre se procesan las mismas variables. Por ejemplo para intercambiar dos variables globales:

```
MODULE CambiazoUnico ;
    FROM InOut IMPORT  WriteLn, ReadInt, WriteInt, WriteString ;
    VAR
        x1, x2 : INTEGER ;

    PROCEDURE Intercambia ;
        VAR
            Auxiliar : INTEGER ;
        BEGIN
            Auxiliar := x1  ;
            x1  := x2  ;
            x2  := Auxiliar ;

        END Intercambia ;

    BEGIN
        WriteString( " Este programa intercambia dos números enteros " ) ;
        WriteLn ;
        WriteLn ;
        WriteString( "Deme el primer número: " );
        ReadInt( x1 ) ;
        WriteLn ;
        WriteString( "Deme el segundo número: " );
        ReadInt( x2 ) ;
        WriteLn ;

        Intercambia( x1, x2 ) ;

        WriteString( "Después del intercambio la situación es: " );
        WriteLn ;
        WriteLn ;
        WriteString( "Primer número: " );
        WriteInt( x1, 2 ) ;
        WriteLn ;
        WriteString( "Segundo número: " );
        WriteInt( x2, 2 ) ;
        WriteLn ;

    END CambiazoUnico.
```

4.2.6.3.- Doble referencia.

Un mismo elemento es referenciado con nombres diferentes. La causa puede ser:

a) Un procedimiento utiliza una variable global que también es pasada como argumento.
Ejemplo:

```
VAR   VariableGlobal : INTEGER ;
. . .
PROCEDURE   Procedimiento(  DatoEnviado : INTEGER ) ;
BEGIN
     VariableGlobal := 0 ;
     . . .
END  Procedimiento ;
. . .
VariableGlobal := 1 ;
Procedimiento( VariableGlobal ) ;
. . .
```

b) En una lista de argumentos se pasa una misma variable varias veces
Ejemplo:

```
PROCEDURE    CuadradoYCubo( VAR x, x2, x3 : INTEGER ) ;
BEGIN   (* Pasando por referencia, se pretende devolver:  *)
       x2 := x * x         ;   (* En x2 el cuadrado de x  *)
       x3 := x * x  * x  ;   (* En x3 el cubo de x  *)
END CuadradoYCubo ;
. . .
(* Si se hace esto:   *)
Numero := 4 ;
CuadradoYCubo( A, A, B) ;
(* en A se devuelve 16, pero en B se devuelve 4096  *)
```

4.2.7.- Recursividad

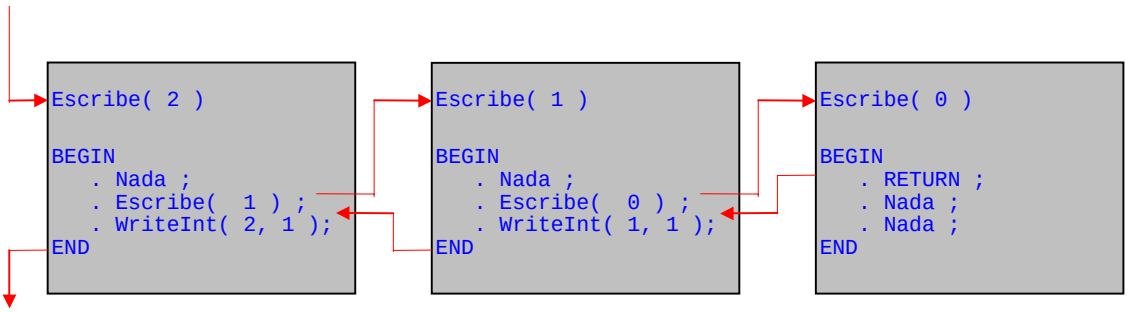
Una definición es recurrente cuando para hacerla se recurre al objeto que define. De la misma forma, un procedimiento es recursivo cuando en su definición se hace alguna llamada a sí mismo.
Ejemplo:
En este programa se define un procedimiento que escribe en pantalla los N primeros números enteros, desde 1 hasta N.

```
MODULE EscribeRecursivamente ;
    FROM InOut IMPORT  WriteInt ;

    PROCEDURE Escribe( N : INTEGER ) ;
    BEGIN
        IF N = 0 THEN RETURN END ;
        Escribe( N-1 ) ;
        WriteInt( N, 1)

    END Escribe ;

   BEGIN
        Escribe( 2) ;
   END EscribeRecursivamente.
```

Al entrar en el procedimiento, se hace sucesivas llamadas a sí mismo, con un argumento que cada vez es disminuido en una unidad. En el momento en que el argumento es cero, se continúa con la ejecución de todas las llamadas que habían quedado pendientes, desde la última hasta la primera.

Al hacer cada llamada recursiva, se mantienen los valores que tengan las variables en la llamada actual, para poder seguir procesándolos cuando vuelva a ella.

Todo procedimiento recursivo debe incluir una condición de salida de las llamadas, de lo contrario estaría eternamente autoinvocándose.



Muchos ejercicios de programación tienen dos soluciones alternativas. Una mediante un algoritmo iterativo y otra mediante un algoritmo recursivo. Por ejemplo, ya hemos visto un procedimiento para calcular el factorial de un número entero positivo. Su alternativa con un algoritmo recursivo es:

```
PROCEDURE  FactorialRecursivo( N : CARDINAL ) : CARDINAL ;
    BEGIN
        IF N = 1 THEN RETURN 1 END ;
        RETURN  N * FactorialRecursivo( N-1 ) ;
    END FactorialRecursivo ;
```

Para la mayoría de los programadores es más fácil entender y diseñar un algoritmo iterativo que su correspondiente recursivo.

En cada llamada a un procedimiento se guarda una serie de datos en la **pila** de memoria. Una limitación de los procedimientos recursivos es que si hacen demasiadas llamadas se puede desbordar esta zona de memoria, y ocurrir un error de pila.

4.2.8.- Bibliotecas de Procedimientos

El lenguaje Modula-2 incluye varios procedimientos, comunes a todos los compiladores. (Véase la tabla correspondiente). Además, cada compilador puede incorporar otras bibliotecas de procedimientos, los cuales es necesario importar con una sentencia IMPORT. Para conocer cuáles son y qué nombre tienen, se debe consultar el manual de referencia del compilador.

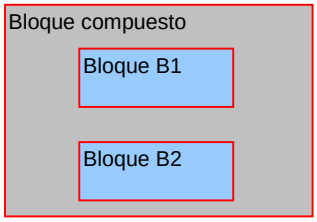
5 TÉCNICAS DE PROGRAMACIÓN

5.1 RECOMENDACIONES PARA CONSEGUIR UN BUEN ESTILO DE ESCRITURA

Si se siguen por costumbre, estas recomendaciones pueden contribuir a dar una mayor claridad a nuestros programas.

5.1.1.- Sangrado

Pueden colocarse varias sentencias en una misma línea. Pueden estar precedidas por un número cualquiera de espacios en blanco o tabuladores. Pero existe la costumbre de escalonar las sentencias de un mismo bloque (bucle, procedimiento, conjunto de sentencias interrelacionadas etc.) a una profundidad uniforme. Cada nuevo bloque se escalona en un nuevo nivel. Con cada END retrocedemos un nivel del escalonamiento. La finalidad del escalonamiento es hacer más legible el programa fuente para cualquier lector. El sangrado refleja el anidamiento entre bloques.



5.1.2.- Inclusión de Comentarios

Según el lugar donde los situemos, y su finalidad, los comentarios pueden ser de varios tipos:

Comentarios de cabecera de programa

Presentan una información general del programa. Los datos que suelen contener son:

. Nombre del programa
. Descripción a grandes rasgos.
. Finalidad
. Autor
. Fecha de creación y de la última revisión, etc.

Comentarios de cabeceras de sección

Encabezan los bloques principales, si el programa es muy largo, presentado información relativa a cada bloque.

Comentarios de funcionalidad

Hay grupos de sentencias que están destinadas a funcionar todas juntas como una acción no primitiva. Estos comentarios delimitan el comienzo y fin de cada grupo, indicando cuál es dicha acción.

Comentarios al margen

Están hechos para explicar una sola sentencia, cuando ésta no es fácil de entender a primera vista. Suelen situarse en la misma línea de dicha sentencia, a la derecha.

5.1.3.- Elección de los nombres

Los nombres de entidades definidas por el programador pueden cumplir una función nemotécnica si se eligen de forma que:

a) recuerden de forma inmediata el significado. Para ello se pueden formar palabras compuestas, en las que cada componente empiece con letra mayúscula y el resto de letras sean minúsculas. Ejemplo: `DatoEnviado`, `FechaActual`, etc

b) tengan la categoría gramatical concordante con el elemento referido:

- Las constantes y variables designadas con sustantivos. Ejemplo: `Area`, `Interes`, etc.
- Los procedimientos designados con verbos en infinitivo. Ejemplo : `CalcularArea`, etc.
- Los tipos mediante nombres genéricos. Ejemplo: `TipoLongitud`, `TipoAltura`, etc.

5.2.- CONSTANTES CON NOMBRE

Cuando un valor constante es usado con gran frecuencia en un programa, es conveniente tenerlo en una constante cuyo nombre sea descriptivo de su naturaleza. Ejemplo:

```
CONST cmPorPulgada = 2.54 ;
```

y luego en el programa, cada vez que se quiera usar ese valor:

```
LongitudCm := LongitudPulgada * cmPorPulgada ;
```

También suelen almacenarse en constantes ciertos valores que en el momento de hacer el programa son fijos, pero que en un momento posterior podrían variar. Por ejemplo, en un programa de contabilidad que defina:

```
CONST Intereses =  12 ;
```

si en un momento posterior la ley hace cambiar esa constante de 12 a otro valor, basta con modificar esta línea de código. Así no es necesario buscar el dato numérico 12 en todos los puntos del programa en que se utilice con esta misma finalidad

5.3.- PERFILAMIENTO MEDIANTE ABSTRACCIONES SUCESIVAS. REUTILIZACIÓN DE CÓDIGO

Los procedimientos, cuanto más genérico sea su comportamiento, más reutilizables son. Un procedimiento es reutilizable si podremos aplicarlo no sólo en el programa para el que está diseñado, sino también en otros programas en los que se requiera un procedimiento similar.

5.3.1.- Un ejemplo

Vamos a desarrollar un programa para dibujar en pantalla este velero:

Realizando un análisis descendente para resolver el problema, tenemos que la acción

- Dibujar el velero

no es una acción primitiva. Para descomponerla, podemos fijarnos en que la figura que queremos dibujar tiene varias partes diferenciadas: mástil, vela y casco. Dedicamos una acción para cada parte.

- Dibujar el mástil
- Dibujar la vela
- Dibujar el casco

Estas acciones aún no son primitivas. Más aún: la primera debemos descomponerla en otras dos una para cada parte visible del mástil, y entre ambas debe ir la de dibujar la vela, pues el orden en que deben ejecutarse es el que viene impuesto por las operaciones de escritura, en las que las líneas se van sucediendo de arriba a abajo.

- Dibujar la parte superior del mástil
- Dibujar la vela
- Dibujar el resto del mástil
- Dibujar el casco

Estas acciones, aunque no son primitivas, ya pueden realizarse cada una con un pequeño conjunto de operaciones sencillas de escritura, obteniendo este programa:

```
MODULE  Barco00;
        FROM    InOut IMPORT WriteString, WriteLn ;
        BEGIN
                (*  Dibujar el extremo superior del mástil *)
                WriteString("           **") ; WriteLn ;

                (*  Dibujar la vela  *)
                WriteString("    ******************") ; WriteLn ;
                WriteString("    ******************") ; WriteLn ;
                WriteString("    ******************") ; WriteLn ;

                (*  Dibujar el mástil *)
                WriteString("           **") ; WriteLn ;
                WriteString("           **") ; WriteLn ;
                (*  Dibujar el casco *)
                WriteString("**************************") ; WriteLn ;
                WriteString(" ************************") ; WriteLn ;

        END     Barco00.
```

Tenemos ya un programa que cumple el cometido que nos habíamos propuesto. Pero si no somos conformistas y queremos diseñarlo con procedimientos genéricos. Redactar familias de órdenes que realicen tares casi idénticas es un despilfarro de memoria, y delata un muy mal estilo de programación. Lo más práctico y elegante es encerrarlas en procedimientos genéricos.

Vamos a llevar a cabo un proceso de abstracciones sucesivas. En cada paso:

- Buscamos grupos de órdenes que sean similares entre sí.
- Las acciones comunes en cada grupo las encerramos en un procedimiento, el cual las realizará siempre.
- Las acciones que sean diferentes para cada orden del grupo las discernirá el procedimiento según los diferentes valores de los argumentos enviados.

1er Paso.

Hay dos grupos de órdenes que son muy parecidas entre sí:

1º Órdenes para escribir espacios en blanco.

Acciones comunes: Escribir el carácter `' '`.
Diferencias: Número de caracteres en blanco.

2º Órdenes para escribir asteriscos.

Acciones comunes: Escribir el carácter `'*'`.
Diferencias: Número de asteriscos.

Definiendo un procedimiento para cada familia de órdenes similares, el programa que resulta es:

```
MODULE  Barco01;
        FROM    InOut IMPORT WriteString, WriteLn, Write ;

        PROCEDURE  Avanza( n: INTEGER );
                VAR Posicion: INTEGER;
                BEGIN
                        FOR Posicion:=1 TO n DO
                                Write(" ");
                        END
                END Avanza;

        PROCEDURE  Rellena( n: INTEGER );
                VAR Posicion: INTEGER;
                BEGIN
                        FOR Posicion:=1 TO n DO
                                Write("*");
                        END
                END Rellena;

        BEGIN
                (*  Dibujar el extremo superior del mástil  *)
                Avanza(11) ; Rellena(2) ; WriteLn ;

                (*  Dibujar la vela  *)
                Avanza(3) ; Rellena(17) ; WriteLn ;
                Avanza(3) ; Rellena(17) ; WriteLn ;
                Avanza(3) ; Rellena(17) ; WriteLn ;

                (*  Dibujar el mástil *)
                Avanza(11) ; Rellena(2) ; WriteLn ;
                Avanza(11) ; Rellena(2) ; WriteLn ;

                (*  Dibujar el casco *)
                Rellena(23) ; WriteLn ;
                Avanza(1)   ; Rellena(21) ; WriteLn ;

        END     Barco01.
```

2º Paso

Tenemos por una parte un procedimiento para escribir una secuencia de espacios en blanco; y por otra parte, otro procedimiento para escribir asteriscos.

Acciones comunes: Escribir secuencias de un mismo carácter.
Diferencias: Carácter escrito; y número de caracteres existentes en la secuencia.

Podemos abstraer ambos procedimientos en uno sólo, más genérico. Los argumentos que debe recibir son los necesarios para discernir qué carácter escribir y conocer el número.

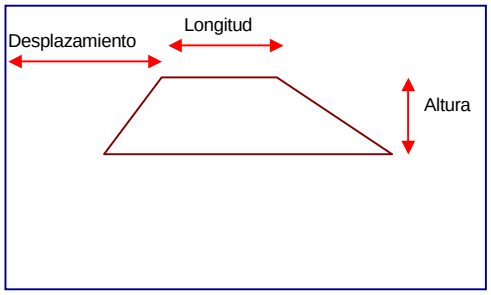
```
MODULE  Barco02;
        FROM    InOut IMPORT WriteString, WriteLn, Write ;

        PROCEDURE  Escribe(c:CHAR; n: INTEGER );
                VAR Posicion: INTEGER;
                BEGIN
                        FOR Posicion:=1 TO n DO
                                Write(c);
                        END
                END Escribe;

        BEGIN
                (*  Dibujar el extremo superior del mástil  *)
                Escribe(' ', 11) ; Escribe('*', 2) ; WriteLn ;

                (*  Dibujar la vela  *)
                Escribe(' ', 3) ; Escribe('*', 17) ; WriteLn ;
                Escribe(' ', 3) ; Escribe('*', 17) ; WriteLn ;
                Escribe(' ', 3) ; Escribe('*', 17) ; WriteLn ;

                (*  Dibujar el mástil *)
                Escribe(' ', 11) ; Escribe('*', 2) ; WriteLn ;
                Escribe(' ', 11) ; Escribe('*', 2) ; WriteLn ;

                (*  Dibujar el casco *)
                Escribe('*', 23) ; WriteLn ;
                Escribe(' ', 1)   ; Escribe('*', 21) ; WriteLn ;

        END     Barco02.
```

3er Paso

Cada zona la hemos estado dibujando siempre dentro del cuerpo del módulo principal con varias líneas. Todas las zonas son de forma trapezoidal, pues los cuadrados y rectángulos son casos particulares de trapecios. Vamos a definir aparte un procedimiento para dibujar un trapecio en general.

Acciones comunes: Dibujar un trapecio con una o varias líneas.
Diferencias: Forma concreta del trapecio.

Los argumentos del procedimiento deben ser los suficientes para poder conocer dentro de éste la forma del trapecio. Su significado es el que muestra esta figura:



RampaIzquierda: Número de columnas que se avanza hacia la derecha al pasar de una línea a la inferior, en el lado izquierdo del trapecio.
RampaDerecha: Número de columnas que se avanza hacia la derecha al pasar de una línea a la inferior, en el lado derecho del trapecio.

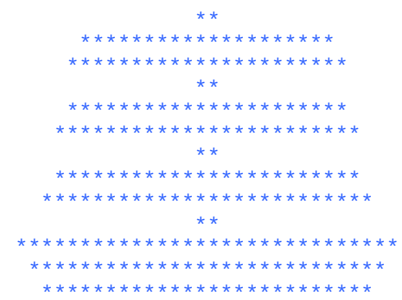
Cada uno de estos dos parámetros, RampaIzquierda y RampaDerecha, puede ser negativo, si su avance correspondiente es hacia la izquierda.

```
(*
    Nombre:          Barco03.Mod
    Propósito:       Dibujar un velero
    Características: Utiliza un procedimiento de dibujo de trapecios
    Autor:           José Garzía
*)

MODULE  Barco03;
        FROM    InOut IMPORT WriteString, WriteLn, Write ;

        PROCEDURE  Escribe(c:CHAR; n: INTEGER );
                VAR Posicion: INTEGER;
                BEGIN
                        FOR Posicion:=1 TO n DO
                                Write(c);
                        END
                END Escribe;

        PROCEDURE       DibujaTrapecio( Desplazamiento, Longitud, Altura,
                                        RampaIzquierda, RampaDerecha: INTEGER ) ;

                VAR NumeroLinea, Origen, Fin: INTEGER ;

                BEGIN
                        FOR NumeroLinea:=0 TO (Altura-1) DO
                                Origen := Desplazamiento+NumeroLinea*RampaIzquierda ;
                                Fin    := Longitud+NumeroLinea*(RampaDerecha-RampaIzquierda) ;
                                Escribe(' ', Origen ) ;
                                Escribe('*', Fin ) ;
                                WriteLn ;
                        END ;
                END DibujaTrapecio ;

        BEGIN
                (*  Dibujar el extremo superior del mástil  *)
                DibujaTrapecio(11, 2, 1, 0, 0 ) ;

                (*  Dibujar la vela  *)
                DibujaTrapecio(3, 17, 3, 0, 0 ) ;

                (*  Dibujar el mástil *)
                DibujaTrapecio(11, 2, 2, 0, 0 ) ;

                (*  Dibujar el casco *)
                DibujaTrapecio(0, 23, 2, 1, -1 ) ;

        END     Barco03.
```

Comparando las diferentes versiones del programa, vemos que:
- la que usa procedimientos genéricos es mucho más larga.
- se ha sacado el código fuera del módulo principal, aislándolo en procedimientos.
- operaciones que antes se hacían por separado, se han refundido en una operación abstracta y parametrizada.

5.3.2.- Ventajas de la Parametrización

1ª Facilidad para modificar los resultados del programa, manteniendo la naturaleza de su comportamiento.
Por ejemplo, si quisiéramos dibujar un velero de diferente tamaño, sólo tendríamos que modificar los valores de los parámetros utilizados en las llamadas que se hacen a los procedimientos dentro del cuerpo del módulo principal. Además del tamaño podemos cambiar la forma. Para dibujar este otro:



basta con modificar el cuerpo del módulo principal, dejándolo así:

```
MODULE   Barco04
   . . .
BEGIN
                (*  Dibujar el extremo superior del mástil  *)
                DibujaTrapecio(14, 2, 1, 0, 0 ) ;

                (*  Dibujar las velas  *)
                FOR Contador := 1 TO 3 DO
                        DibujaTrapecio(6-Contador, 18+2*Contador, 2, -1, 1 ) ;
                DibujaTrapecio(14, 2, 1, 0, 0 ) ;
                END ;

                (*  Dibujar el mástil *)
                DibujaTrapecio(14, 2, 1, 0, 0 ) ;

                (*  Dibujar el casco *)
                DibujaTrapecio(0, 30, 3, 1, -1 ) ;

        END     Barco04.
```

2ª Reutilización
Si la operación final resultante de un proceso de abstraer operaciones similares tiene cierto sentido en sí misma, es muy probable que tenga utilidad en otros programas además de aquél para el cual fue diseñada. El diseño de otros programas que puedan utilizar esta operación será más sencillo. Como contrapartida, es más difícil elaborar unos subprogramas de uso general que un programa que tiene como meta resolver un problema muy concreto: al diseñar código reutilizable, no sólo hay que tener en cuenta las necesidades del programa que se esté desarrollando en ese momento. También hay que saber prever las posibles aplicaciones que una operación abstracta pueda tener en el futuro.

Como ejemplo, vamos a ver que los procedimientos parametrizados abstractos anteriores pueden aplicarse para hacer otros dibujos, pues cualquier dibujo elemental puede hacerse componiendo figuras trapezoidales, siempre que no haya en una misma línea más de un trapecio. Podemos dibujar esta casita de juguete:

cambiando tan sólo el cuerpo del módulo principal

```
MODULE  Casita ;
   . . .
BEGIN
                (*  Dibujar la chimenea *)
                DibujaTrapecio(5, 2, 2, 0, 0 ) ;

                (*  Dibujar el tejado  *)
                DibujaTrapecio(2, 20, 3, -1, 1 ) ;

                (*  Dibujar la pared *)
                DibujaTrapecio(2, 20, 3, 0, 0 ) ;

        END     Casita.
```

5.4.- PROGRAMACIÓN A LA DEFENSIVA

Cuando estamos metidos de lleno en el estudio de la mecánica normal de un programa, no solemos preocuparnos de lo que sucedería en determinadas circunstancias anómalas. Por ejemplo: Supongamos un procedimiento en que se deba leer por el teclado un número y lo que se introduce es cualquier otro carácter. Dicho procedimiento debe incluir una comprobación de que el dato leído es válido. De lo contrario, puede ser procesado y los resultados de ese procesamiento ser presentados sin aviso de que son erróneos.

Programa robusto es el que incluye una previsión de las posibles causas de error; y hace un tratamiento particular para cada tipo de error. Un error muy frecuente, que suele provocar el aborto del programa, es el intento de hacer una división por cero.

Otro ejemplo: Un procedimiento para calcular la media aritmética de varios números, si se intenta calcular la media sin haber introducido ningún dato, podría intentar hacer una división por cero. Sería conveniente incluir una comprobación del número de datos introducidos, y si fuera cero, programarlo para que en lugar de intentar hacer la división, enviara un mensaje de advertencia.

Si un fallo para el que no se ha previsto un tratamiento de errores se produce en un programa simple, el efecto no es muy grave. Pero si es en un proyecto muy grande, puede ser muy difícil encontrar el punto donde se ha producido el error.

La mejora en la robustez de un programa tiene como contrapartida un empeoramiento de la eficiencia, pues en realizar las comprobaciones siempre se pierde un tiempo adicional

5.5.- ANÁLISIS ASCENDENTE

Análisis ascendente es el proceso en el que se empieza creando los subprogramas, y se termina con el programa principal.

A veces es más fácil identificar operaciones aisladas que concebir el proyecto global en el cual tendrán aplicación. Es en estos casos cuando puede resultar muy fructífero un análisis ascendente.

Hay que distinguirlo del proceso de perfilamiento mediante abstracciones sucesivas. Allí los subprogramas procedían de la abstracción de operaciones similares que había en un programa. Aquí proceden del hecho de que antes de empezar el proyecto global ya están muy bien identificadas las operaciones que debe incluir.

Como ejemplo, tenemos muy claro que para implementar una calculadora de números complejos, en ésta debe haber operaciones de suma, resta, multiplicación y división. Lo primero que haremos será crear un procedimiento para cada operación, y algún otro auxiliar. En las listas de argumentos están las partes real y compleja de; por este orden: primer operando, segundo operando y una variable pasada por referencia donde se almacena el resultado de la operación correspondiente.

```
PROCEDURE Sumar( x1, y1, x2, y2 : REAL; VAR x3, y3 : REAL ) ;
    BEGIN
        x3 := x1 + x2 ;
        y3 := y1 + y2 ;
    END Sumar ;

PROCEDURE Restar( xMinuendo, yMinuendo, xSustraendo, ySustraendo : REAL; VAR xResultado, yResultado
: REAL                    ) ;
   BEGIN
        xResultado := xMinuendo - xSustraendo ;
        yResultado := yMinuendo - ySustraendo ;
   END Restar ;

PROCEDURE Multiplicar( x1, y1, x2, y2 : REAL; VAR x3, y3 : REAL ) ;
   BEGIN
        x3 := x1 * x2 - y1 * y2 ;
        y3 := x1 * y2 + y1 * x2 ;
   END Multiplicar ;

PROCEDURE CuadradoModulo( x, y : REAL ): REAL ;
   BEGIN
        RETURN ( x*x + y*y ) ;
   END CuadradoModulo ;

PROCEDURE Modulo( x, y : REAL ): REAL ;
    VAR m : REAL ;
   BEGIN
        m := sqrt( Modulo(x, y) ) ;
        RETURN m ;
   END Modulo ;


PROCEDURE Invertir( x, y : REAL; VAR xInverso, yInverso : REAL ) ;
   BEGIN
        xInverso := x / ( CuadradoModulo( x, y ) ) ;
        yInverso := y / ( CuadradoModulo( x, y ) )  * (-1.0) ;
   END Invertir ;

PROCEDURE Dividir( xDividendo, yDividendo, xDivisor, yDivisor : REAL; VAR xCociente, yCociente :
REAL ) ;
   VAR xAuxiliar, yAuxiliar : REAL ;
   BEGIN
        Invertir( xDivisor, yDivisor, xAuxiliar, yAuxiliar ) ;
        Multiplicar( xDividendo, yDividendo, xAuxiliar, yAuxiliar, xCociente, yCociente ) ;
   END Dividir ;
```

Después de haber implementado las operaciones con los números complejos, vemos que como paso intermedio entre estas operaciones, y el programa completo, hacen falta unos procedimientos para leer y escribir números complejos por el teclado.

```
PROCEDURE Leer( VAR x, y : REAL ) ;
   BEGIN
        WriteString("Parte real: ") ;
        ReadReal( x ) ; WriteLn ;
        WriteString("Parte compleja: ") ;
        ReadReal( y ) ; WriteLn ;
   END Leer ;

PROCEDURE Escribir( x, y: REAL ) ;
   BEGIN
       WriteString( "Resultado: (" ) ;
       WriteReal( x, 1 ) ;
       WriteString(", ") ;
       WriteReal( y, 1) ;
       Write( ')') ;
       WriteLn ;
   END Escribir ;
```

Las operaciones aritméticas se realizan entre un acumulador almacenado en una variable global y el dato leído. El resultado se almacena en el mismo acumulador. El programa completo tiene un bucle para leer las opciones +, - *, / , = (para presentar el resultado acumulado), Barra Espaciadora (para leer un nuevo número) y ESC (para salir).

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:          Complejo.mod                                      ║
  ║    Propósito:       Calculadora de números complejos                  ║
  ║    Autor:           José Garzía                                       ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE  NumerosComplejos;
FROM InOut     IMPORT WriteString, Write, WriteLn, Read ;
FROM RealInOut IMPORT WriteReal, ReadReal;
FROM MathLib0    IMPORT sqrt ;


PROCEDURE LeerOperacion( VAR CodigoOperacion : CHAR ) ;
    BEGIN
        Write('?') ; Read( CodigoOperacion ) ;
        CodigoOperacion := CAP( CodigoOperacion ) ;
        Write( CodigoOperacion ) ; Write( ' ') ;
    END LeerOperacion ;

VAR
    xAcumulador, yAcumulador : REAL ;
    xNuevoDato, yNuevoDato : REAL ;
    Operacion : CHAR ;

BEGIN
   xAcumulador := 0.0 ;
   yAcumulador := 0.0 ;

REPEAT
    LeerOperacion( Operacion) ;
    CASE Operacion OF
        '+':
            Leer( xNuevoDato, yNuevoDato ) ;
            Sumar( xAcumulador, yAcumulador, xNuevoDato, yNuevoDato, xAcumulador, yAcumulador ) |
        '-':
            Leer( xNuevoDato, yNuevoDato ) ;
            Restar( xAcumulador, yAcumulador, xNuevoDato, yNuevoDato, xAcumulador, yAcumulador ) |
        '*':
            Leer( xNuevoDato, yNuevoDato ) ;
            Multiplicar( xAcumulador, yAcumulador, xNuevoDato, yNuevoDato, xAcumulador, yAcumulador
) |
        '/':
            Leer( xNuevoDato, yNuevoDato ) ;
            Dividir( xAcumulador, yAcumulador, xNuevoDato, yNuevoDato, xAcumulador, yAcumulador ) |
        ' ':
            Leer( xAcumulador, yAcumulador ) ; |
        '=' :
            WriteString( '      ' );
            Escribir( xAcumulador, yAcumulador ) ; |

    ELSE
            WriteString( 'Pulse +. -, *, /, ESPACIO, = , ó ESC ' ) ;
            WriteLn ;
    END ;

    UNTIL Operacion = CHR( 27 ) ;

END NumerosComplejos.
```

6 FORMACIONES

6.1.- INTRODUCCIÓN

Formación. Es una colección ordenada de datos que pueden tener valores diferentes, pero que son todos del mismo tipo.

6.2.- CLASES DE FORMACIONES

6.2.1.- 1-Dimensional o lista, de tamaño *[n]*

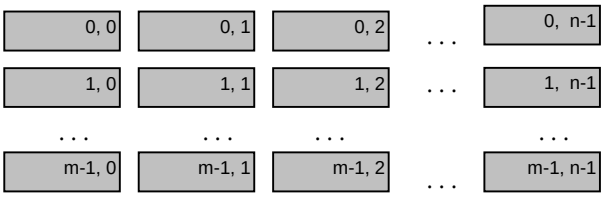
Es una lista de n elementos, ordenados de forma que a cada uno le corresponde una posición, designada con un número natural, desde 0 hasta n-1.
Índice Es cada un de estos números naturales. Se dice que los elementos están **indexados**

6.2.2.- N-Dimensional, de tamaño *[T(N), T(N-1), . . ., T(1)];* *(N ∈ N; N > 1), (T(N), T(N-1), . . ., T(1) ∈ N)*

Es un conjunto de *T(N)* formaciones, de tamaño *[T(N-1), . . ., T(1)]*
Las formaciones N-Dimensionales más utilizadas son:

6.2.2.1. Bidimensional o matriz de tamaño *[m, n]*

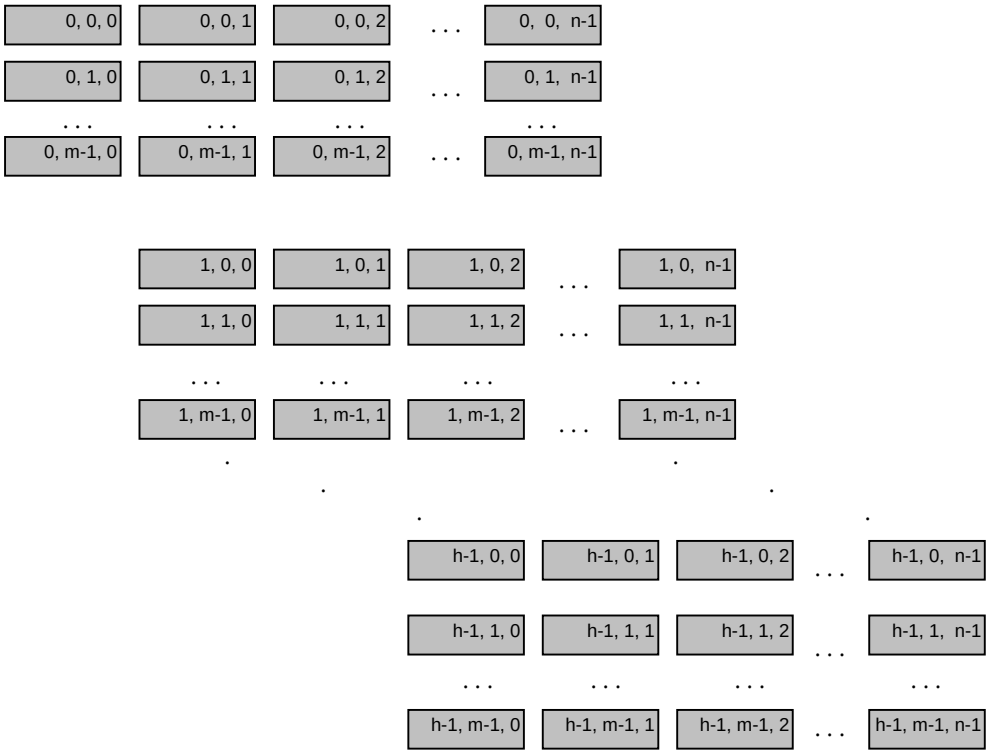
Es un conjunto de m elementos en el que cada elemento es una n-tupla.



Al aumentar las dimensiones [m,n], el tamaño de la formación crece cuadráticamente.

6.2.2.2. Tridimensional, de tamaño *[h, m, n]*

Es un conjunto de h elementos, en el que cada elemento es una matriz de tamaño [m, n]



Al aumentar las dimensiones *[h,m,n]*, el tamaño de la formación crece cúbicamente. Por este motivo, las formaciones de órdenes superiores son poco usadas.

6.3.- UTILIDAD DE LAS LISTAS

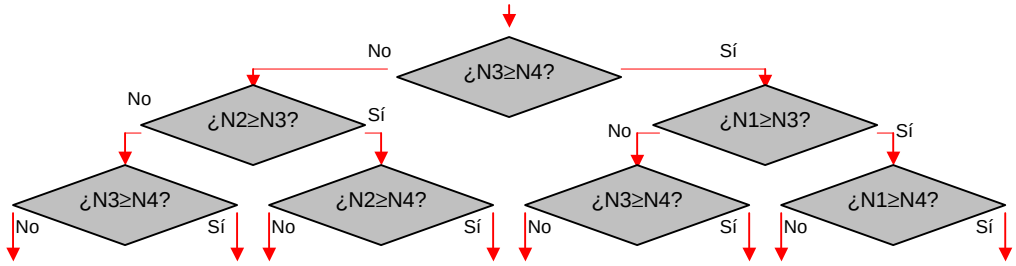
Supongamos que en una aplicación hay una serie de datos de características comunes que requieren un tratamiento similar. Se podría tenerlos dispersos utilizando una variable para cada uno. Pero es preferible tenerlos colocados todos en una formación, donde todos comparten el mismo nombre.
Desde los primeros tiempos de la Informática, el procesamiento de múltiples datos de características semejantes es una situación muy frecuente. Por este motivo los lenguajes de programación tienen métodos para:

- Recorrer secuencialmente una formación.
- Acceder de forma directa a un elemento cualquiera.

Ejemplo

Supongamos que se quiere encontrar el máximo de una lista de cuatro números;
N1, N2, N3 y N4. Utilizando cuatro variables, un posible algoritmo es:

- Comparar N1 y N2 ; y observar cuál es el mayor.
- Comparar el resultado con N3 ; y observar cuál es el mayor.
- Comparar el resultado con N4 ; y observar cuál es el mayor.



Hay *1 + 2 + 4 = 7* comparaciones.
Si en la lista hubiera **n** números, habría $2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n = \Sigma^n_{i=0}\, 2^i$ comparaciones. Este algoritmo es poco eficiente pues el número de comparaciones crece exponencialmente con **n**

Un algoritmo mejor es:

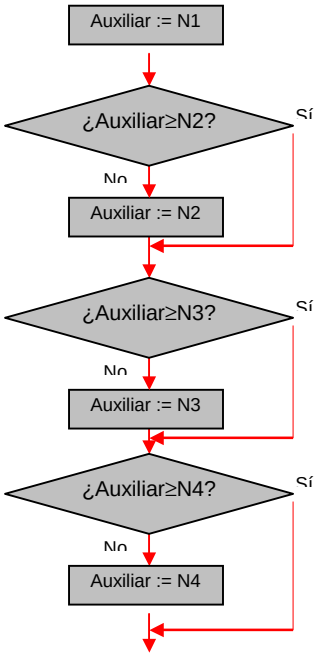
- Almacenar N1 en una variable llamada Auxiliar
- Comparar N2 Con Auxiliar, y almacenar el resultado en Auxiliar
- Comparar N3 Con Auxiliar, y almacenar el resultado en Auxiliar
- Comparar N4 Con Auxiliar, y almacenar el resultado en Auxiliar

Si están almacenados en una lista, pueden ser accedidos secuencialmente en un bucle, por lo que el algoritmo se puede expresar utilizando un método iterativo:

- Leer el primer número de la lista y almacenarlo en la variable Auxiliar.
- ★ PARA i = 2 HASTA i = n HAGA :
  - Leer el *siguiente* número de la lista
  - Compararlo con Auxiliar y si es mayor, almacenarlo en Auxiliar.

El concepto de *siguiente* tiene sentido debido a la relación de orden (indexamiento) que hay implícita en toda lista.

Según se puede ver por inducción en el diagrama de flujo, para una lista de **n** números, el número de comparaciones es (n-1). El tiempo requerido para ejecutarlo no aumenta con **n** exponencial, sino linealmente. En lo que respecta a la memoria, sólo utiliza una variable adicional. Por tanto. este algoritmo es mejor que el anterior.



6.4.- LISTAS EN MODULA-2

6.4.1.- Listas en General, de cualquier Tipo de Datos

Declaración:

```
VAR  < Nombre de la lista    > : ARRAY [ < Comienzo del índice > . . < Fin del índice > ] OF
     < Tipo de los elementos >
```

Cuando se quiere acceder a un elemento, se hace con el nombre de la N-tupla, y entre corchetes, el índice del elemento.

Ejemplo:
La codificación en Modula-2 del último algoritmo es:

```
. . .
CONST N = 4 ;
VAR
        Indice : CARDINAL ;
        Auxiliar : REAL ;
        Lista : ARRAY [ 1..N] OF REAL ;
. . .
Auxiliar := Lista[ 1 ] ;
FOR Indice := 2 TO N DO
      IF Auxiliar < Lista[ Indice ] THEN
                Auxiliar := Lista[ Indice ] ;
      END ;
END ;
. . .
```

6.4.2.- Listas de Caracteres

Un dato de tipo STRING (ristra de caracteres) es un caso especial de una N-tupla. Se trata de una N-tupla en la que los elementos son de tipo CHAR. Debido, entre otros motivos a que:

- el espacio necesario para almacenar cada carácter es la unidad de memoria, el byte
- la gran frecuencia con que se utilizan en Informática las ristras de caracteres

en casi todos los lenguajes de programación las ristras son tratadas con algunas peculiaridades.

En Modula-2 son:

1) El índice inferior debe ser siempre 0
2) Pueden hacerse inicializaciones de forma compacta:

```
VAR Saludo : ARRAY [0..7]  OF CHAR ;
. . .
Saludo := "¡ Hola !" ;
```

En lugar de tener que hacer:

```
Saludo[0] := '¡' :
Saludo[1] := ' ' ;
Saludo[2] := 'H' ;
    . . .
```

3) Salvo que ocupen todos los bytes declarados para utilizarlas, el último carácter es el nulo.
4) Hay gran cantidad de procedimientos de biblioteca destinados al tratamiento de ristras de caracteres.

Se recomienda consultar el manual de referencia del compilador para saber los procedimientos disponibles, sus nombres, cómo funcionan y en qué archivos están.

Ejemplo:

```
MODULE EncadenaRistras;
FROM InOut IMPORT ReadString, WriteString, WriteLn ;
FROM Strings IMPORT Concat ;

    VAR
        Frase1     : ARRAY [0..39] OF CHAR ;
        Frase2     : ARRAY [0..39] OF CHAR ;
        FraseUnida : ARRAY [0..79] OF CHAR ;
    BEGIN
        WriteLn ;
        WriteString("Escriba la primera ristra de caracteres: ");
        WriteLn ;
        ReadString( Frase1 ) ;

        WriteLn ;
        WriteString("Escriba la segunda ristra de caracteres: ");
        WriteLn ;
        ReadString( Frase2 ) ;

        Concat( Frase1, Frase2, FraseUnida ) ;
        WriteLn ;
        WriteString("Después de concatenarlas: ");
        WriteLn ;
        WriteString( FraseUnida ) ;

    END EncadenaRistras.
```

6.4.3.- Paso de Listas a un Procedimiento

Pueden ser pasadas tanto por valor como por referencia. En la definición del procedimiento no es necesario indicar el tamaño, pues puede conocerse mediante una llamda a HIGH. Dentro del procedimiento, el comienzo del índice debe ser siempre 0.

Ejemplo:
En la mayoría de compiladores `ReadString` termina de leer caracteres cuando se pulsa la barra espaciadora, tabulador, o INTRO. De esta forma no es posible leer una frase en la que haya espacios en blanco.

Podemos definir un procedimiento más completo para leer una frase con espacios en blanco y almacenarla en una ristra de caracteres.

Para que tenga la forma normal de las ristras de Modula-2, al final le añadimos un carácter nulo. (CHR(0)).

```
(*
    Nombre:          Leefrase.Mod
    Propósito:       Lee una frase desde el teclado, aceptando espacios
                     en blanco y tabuladores, hasta pulsar INTRO
    Autor:           José Garzía
*)

MODULE LeeFrase;
FROM InOut IMPORT Read, Write, WriteString, EOL;
    VAR
        Frase : ARRAY [ 0..79] OF CHAR ;

    PROCEDURE LeerFrase( VAR F : ARRAY OF CHAR ) ;
    VAR
        Letra : CHAR ;
        Contador : CARDINAL ;
    BEGIN
        Contador := 0 ;
        REPEAT
            Read( Letra ) ;
            Write( Letra ) ;
            IF ( Letra <> EOL ) AND ( Contador < HIGH( F) )THEN
               F[ Contador ] := Letra ;
               INC( Contador ) ;
            END ;
        UNTIL ( Letra = EOL ) OR ( Contador = HIGH( F ) ) ;
        F[ Contador ] := CHR( 0 ) ; (* Terminación de todas las ristras *)
    END LeerFrase ;
    BEGIN
        LeerFrase( Frase ) ;
        WriteString( Frase ) ;
    END LeeFrase.
```

6.5.- FORMACIONES MULTIDIMENSIONALES EN MODULA-2

6.5.1.- Formas de declararlas:

a) Anidada

```
VAR NombreFormacion : ARRAY [  < Comienzo del índice N-ésimo > . . < Fin del índice N-ésimo > ] OF
                                . . . . . . . . . . . . . . . . .  . . . . . . . . . . . . . . . .
                      ARRAY [  < Comienzo del índice segundo > . . < Fin del índice segundo > ] OF
                      ARRAY [  < Comienzo del índice primero > . . < Fin del índice primero > ]
                            OF < Tipo de los elementos > ;
```

b) Compacta. Es la más habitual.

```
VAR NombreFormacion : ARRAY [  < Comienzo del índice N-ésimo > . . < Fin del índice N-ésimo > ] ,

                                . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
                            [  < Comienzo del índice segundo > . . < Fin del índice segundo > ] ,
                            [  < Comienzo del índice primero > . . < Fin del índice primero > ]
                            OF < Tipo de los elementos > ;
```

Los procedimientos que tienen formaciones multidimensionales en su lista de argumentos deben especificar en su definición el tamaño de la formación que están preparados para procesar.

6.5.2.- Casos Particulares

6.5.2.1- Formaciones tridimensionales

Forma de declararlas:

```
VAR NombreFormacion : ARRAY [ < Comienzo del tercer índice  > . . < Fin del tercer índice  > ] ,
                            [ < Comienzo del segundo índice > . . < Fin del segundo índice > ] ,
                            [ < Comienzo del primer  índice > . . < Fin del primer índice  > ]
                      OF < Tipo de los elementos > ;
```

Sea la formación tridimensional

```
VAR  A : ARRAY  [1..2], [0..2], [-1..3] OF INTEGER ;
```

Para acceder a un elemento, se utiliza el nombre de la formación, y los índices separados por comas.
Para facilitar una concepción más intuitiva de la agrupación de datos, podemos imaginar que están dispuestos espacialmente así:

$$\begin{bmatrix} A[1,0,-1] & A[1,0,0] & A[1,0,1] & A[1,0,2] & A[1,0,3] \\ A[1,1,-1] & A[1,1,0] & A[1,1,1] & A[1,1,2] & A[1,1,3] \\ A[1,2,-1] & A[1,2,0] & A[1,2,1] & A[1,2,2] & A[1,2,3] \end{bmatrix}$$

$$\begin{bmatrix} A[2,0,-1] & A[2,0,0] & A[2,0,1] & A[2,0,2] & A[2,0,3] \\ A[2,1,-1] & A[2,1,0] & A[2,1,1] & A[2,1,2] & A[2,1,3] \\ A[2,2,-1] & A[2,2,0] & A[2,2,1] & A[2,2,2] & A[2,2,3] \end{bmatrix}$$

Aunque en realidad en memoria se almacenan de forma consecutiva según van aumentando los índices, desde el que está más a la derecha hasta el que está más a la izquierda:
Primero la fila A[1,0,i], después la A[1,1,i], . . . ; y así hasta la A[2,2,i]; siempre desde i=-1 hasta i=3

Se tiene:

A[k, j] ($1 \le k \le 2$; $0 \le j \le 2$) son listas de 5 elementos (indexados desde i = -1 hasta i = 3)
A[k] ($1 \le k \le 2$) son matrices ARRAY [0..2][-1..3]

6.5.2.2.- Matrices

Forma de declararlas

```
VAR NombreMatriz : ARRAY [ < Comienzo Indice Filas    > . . < Fin Indice Filas    > ] ,
                         [ < Comienzo Indice Columnas > . . < Fin Indice Columnas > ]
                   OF < Tipo de los Elementos > ;
```

Es posible hacer asignaciones entre formaciones multidimensionales siempre que sean del mismo tipo y tamaño, con una sentencia única, sin necesidad de tener que asignar elemento a elemento

Ejemplo:

```
VAR Matriz1, Matriz2 : ARRAY [1..10],[1..5] OF INTEGER ;
    Vector : ARRAY [1..5] OF INTEGER ;
```

La sentencia:

```
 Matriz2:= Matriz1 ;
```

asigna todos los 50 elementos de `Matriz1` a sus correspondientes de `Matriz2`. Esto también es correcto entre subformaciones:

```
 Vector := Matriz2[7] ;
```

copia:

```
       Matriz2[7, 1] en Vector[1]
       Matriz2[7, 2] en Vector[2]
       Matriz2[7, 3] en Vector[3]
       Matriz2[7, 4] en Vector[4]
       Matriz2[7, 5] en Vector[5]
```

6.6.- ALGORITMOS QUE UTILIZAN LAS FORMACIONES

Hay una serie de operaciones para las cuales las formaciones son el tipo de datos más adecuado.

6.6.1.- Recorrido

Es una operación repetitiva en la que se realiza la misma acción con todos y cada uno de los elementos de una formación. Dado que el número de elementos que hay que tratar es el número de elementos de la formación; y éste es conocido, el esquema iterativo que más se adecua al recorrido es el bucle FOR.

Si las formaciones son multidimensionales, se necesitan bucles anidados, uno para cada dimensión. El orden en que se pasa de un elemento al siguiente dentro de cada dimensión, es o bien siempre creciente, o bien siempre decreciente.

Ejemplo:

Dadas las matrices

```
VAR A : ARRAY [0..1], [0..2] OF REAL ;
    B : ARRAY [0..2], [0..1] OF REAL ;
```

se puede calcular su producto A * B = C

$$\begin{pmatrix} A[0,0] & A[0,1] & A[0,2] \\ A[1,0] & A[1,1] & A[1,2] \end{pmatrix} \bullet \begin{pmatrix} B[0,0] & B[0,1] \\ B[1,0] & B[1,1] \\ B[2,0] & B[2,1] \end{pmatrix} = \begin{pmatrix} C[0,0] & C[0,1] \\ C[1,0] & C[1,1] \end{pmatrix}$$

utilizando la fórmula iterativa: $c_{i,j} = \sum_{k=0}^{m} a_{i,k} \cdot b_{k,j}$

```
FOR  i := 0 TO 1 DO
     FOR j := 0 TO  2  DO

           C[i, j]  := 0 ;

            FOR k := 0  TO  1 DO
                  INC( C[i, j], A[i, k] * B[k, j ] ) ;
           END ;
    END ;
END ;
```

6.6.2.- Búsqueda

6.6.2.1.- Búsqueda Secuencial

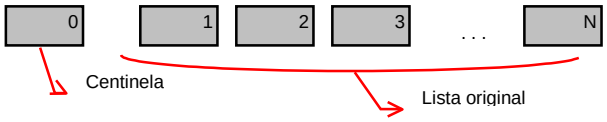
Se trata de hacer un recorrido por la formación, comprobando si alguno de los elementos contiene un valor dado. La búsqueda acaba cuando se encuentra dicho elemento, o bien cuando se han examinado todos.

Este fragmento de programa implementa una búsqueda secuencial:

```
Encontrado := FALSE ;  (* Encontrado es BOOLEAN *)
Posicion := 0 ;
WHILE ( NOT Encontrado ) AND ( Posicion < Ultimo ) DO
      INC( Posicion ) ;
      Encontrado  :=  DatoBuscado = Lista[ Posicion ] ;
END ;
```

Al salir del bucle hay que comprobar el valor de la variable booleana para saber si se ha encontrado el dato buscado, y el valor de la variable que nos da el lugar donde está. Si se sabe que la formación tiene al menos un elemento, se puede utilizar el bucle REPEAT.

Es posible disminuir el número de comparaciones en cada iteración si estamos seguros de que el dato buscado está en la lista. Esta certeza la podemos conseguir si colocamos una copia de dicho dato en el exterior del extremo de la lista. Obviamente, hay que definirla con espacio para contener un elemento más, aparte de los iniciales. Ese elemento suele denominarse **centinela**.




```
Tupla[o] = DatoBuscado ;
Posicion :=  Ultimo ;
WHILE   DatoBuscado <> Lista[ Posicion ]  DO
      DEC( Posicion ) ;
END ;
```

Esta técnica de inicializar a un valor conveniente los bordes también puede ser aplicada en las matrices multidimensionales. En estos casos no es un elemento, sino toda una subformación lo que hay que inicializar. Si son bidimensionales, las matrices obtenidas al añadir una línea por cada borde se llaman **matrices orladas.**

6.6.2.1.- Búsqueda Dicotómica

Si todos los datos están ordenados se puede aprovechar esta circunstancia para hacer una búsqueda más rápida.

Supongamos que está ordenada en orden creciente. El algoritmo es:

★ Mientras no se haya encontrado, y la lista considerada tenga más de un elemento:
- ☞ Se comprueba el elemento que hay en el centro de la lista.
  - ★ Si es el buscado, se acaba la búsqueda.
  - ★ Si es menor, se considera la lista que hay en la mitad derecha.
  - ★ Si es mayor, se considera la lista que hay en la mitad izquierda.

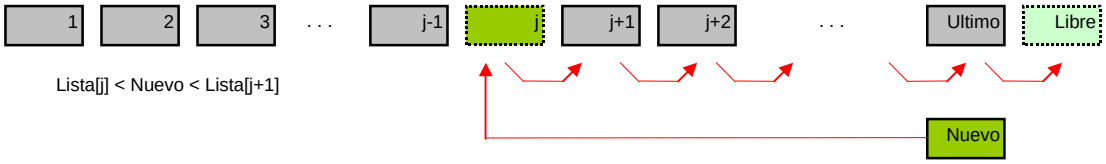
```
Posicion := 0 ;
Izquierda := 1 ;
Derecha := Ultimo ;
WHILE  ( Posicion = 0 ) AND ( Izquierda <= Derecha ) DO
      Mitad := ( Izquierda + Derecha ) DIV 2 ;
      IF DatoBuscado = Lista[ Mitad ] THEN
                  Posicion  :=  Mitad ;
      ELSIF DatoBuscado < Lista[ Mitad ]  THEN
                Derecha := Mitad - 1 ;
      ELSE
                Izquierda := Mitad + 1 ;
      END ;
 END ;
```

Si el orden es decreciente, las mitades que hay que considerar en cada caso son las contrarias.

6.6.3.- Inserción Ordenada

Si los valores se van almacenando en una formación uno por uno, se puede hacer de forma que queden ordenados.

Comenzando por un extremo, se van desplazando uno por uno, todos un lugar hacia ese extremo. Cuando se encuentra el primer elemento menor que el nuevo (mayor, si el orden queremos hacerlo en orden decreciente), se inserta el nuevo en el hueco libre.




```
j := N ;
WHILE ( NuevoElemento < Lista[j] ) AND (j >= 1 ) DO
   Lista[ j+1] := Lista[j]   (* Desplazamiento *)
   DEC( j ) ;
END ;
Lista[ j+1] := NuevoElemento ;
```

Este fragmento debe repetirse desde que `N=1` hasta que `N=Ultimo`.
La búsqueda se ha hecho secuencialmente. Como los ya incluidos están ordenados, también podría hacerse una búsqueda dicotómica.

6.6.4.- Ordenación

La cuestión es diferente. Tenemos ya inicializados todos los elementos de una n-tupla, y se trata de ordenarlos, en un orden creciente o decreciente. Es uno de los temas típicos de programación. Hay muchos algoritmos que lo tratan.

6.6.4.1.- Método de Ordenación por Inserción Directa.

- ★ Desde que N=2 hasta que N=Ultimo
  - ☞ Se extrae Tupla[N] y se almacena en una variable auxiliar.
  - ☞ Se inserta de forma ordenada en la subtupla que hay desde el primer elemento hasta el (N-1)-ésimo.

```
FOR  i := 2 TO Ultimo
   Auxiliar := Lista[i] ;
   j := i - 1 ;
  WHILE  ( Auxiliar < Lista[j] ) AND ( j >= 1 ) DO
         Lista[j+1] := Lista[j] ;
          DEC( j) ;
  END;
  Lista[ j+1] := Auxiliar ;
END ;
```

Utilizando la técnica del centinela:

```
FOR  i := 2 TO Ultimo
   Auxiliar := Lista[i] ;
   Lista[0] := Auxiliar ;
   j := i - 1 ;
  WHILE  ( Auxiliar < Lista[j] )  DO
          Lista[j+1] := Lista[j] ;
          DEC( j) ;
  END;
  Lista[ j+1] := Auxiliar ;
END ;
```

6.6.4.2.- Método de Ordenación de la Burbuja

Este método es famoso por ser uno de los peores algoritmos de ordenación que funcionan por ahí. Sin embargo, es un tópico mostrar su funcionamiento, pues ilustra muy bien las utilidades de las listas en programación.

Los elementos más grandes se van desplazando hacia el extremo `j=N`. El efecto producido es similar al de una burbuja que va subiendo desde el interior de un líquido a la superficie, o el de un trozo de plomo que se va hundiendo.

Debido a su popularidad, está muy estudiado, y tiene muchas variantes. Por ejemplo, se tiene la variante del **método de la sacudida**, en el cual las direcciones del empuje se van alternando de una iteración a la siguiente. Una vez hacia el extremo j=N y otra vez hacia el `j=1`.

El método sólo es eficiente en listas pequeñas y poco desordenadas. El número de iteraciones crece exponencialmente con el tamaño de la lista. Si está muy desordenada hay que hacer muchos intercambios.

```
PROCEDURE Intercambiar( VAR x, y : INTEGER ) ;
  VAR Auxiliar : INTEGER ;
BEGIN
          Auxiliar := x ;
          y := x ;
          x := Auxiliar ;
END Intercambiar ;

PROCEDURE MetodoBurbuja ;
    VAR  i, j, k, N : INTEGER ;
BEGIN
      N := HIGH( Lista ) ;
      FOR i := 1  TO  N  DO
              FOR  j := N  TO  i  BY  -1  DO
                    IF Lista[j-1] > Lista[j]  THEN
                              Intercambiar( Lista[j-1], Lista[j] ) ;
                   END;
             END ;
      END ;
END MetodoBurbuja ;
```

6.6.4.3.- Método de Ordenación Selectiva

0a.- Encontrar la posición k del mínimo de la tupla l[1], l[2], ..., l[n]
0b.- Intercambiar l[k] con l[1]
1a.- Encontrar la posición k del mínimo de la tupla l[2], l[3], ..., l[n]
1b.- Intercambiar l[k] con l[2]
2a.- Encontrar la posición k del mínimo de la tupla l[3], l[4], ..., l[n]
2b.- Intercambiar l[k] con l[3]
. . .
(n-1)a.- Encontrar la posición k del mínimo de la tupla l[n-1], l[n]

```
FOR  j := 1  TO  j = N  DO
     Lugar := j ;
     FOR  k := j  To  N  DO
               IF Lista[Lugar]  > Lista[k]  THEN
                              Lugar := k ;
               END ;
    END ;
    Intercambiar( Lista[j], Lista[Lugar] ) ;
```

6.6.4.4.- Método de Ordenación Quick-Sort

Se toma el elemento del centro de la lista; y se busca en la lista completa desde el extremo izquierdo hasta el centro un elemento mayor que él; y desde el centro hasta el extremo derecho uno menor. Luego se intercambian. Este proceso continúa hasta que se llega al elemento central de la lista. En ese momento hay dos sublistas, una con valores mayores y otra con valores menores. El elemento que estaba en el centro ahora ya está en su puesto definitivo. Después se procede de igual forma con cada sublista, y así sucesivamente hasta que las sublistas constan de un sólo elemento, lo que implica que ya están todos los elementos ordenados.

```
MODULE MetodoQuickSort ;
    FROM InOut IMPORT WriteString, WriteLine, WriteLn, WriteInt, ReadInt, Read ;
    CONST MaxNumeros = 5 ;
    VAR Numeros : ARRAY [0..MaxNumeros] OF INTEGER ;
        c : CHAR ;

    PROCEDURE Intercambia( VAR x: INTEGER ; VAR y : INTEGER ) ;
        VAR Auxiliar : INTEGER ;
    BEGIN
        Auxiliar := x ;
        x := y ;
        y := Auxiliar ;
    END Intercambia ;

    PROCEDURE LeeNumeros() ;
        VAR IndiceBucle : INTEGER ;
    BEGIN
        WriteLn ;
        WriteLine("Deme los números " );
        FOR IndiceBucle := 0 TO MaxNumeros DO
            WriteInt( IndiceBucle+1, 2 ) ;
            WriteString(" º  ") ;
            ReadInt( Numeros[ IndiceBucle ] ) ;
            WriteLn ;
        END ;
    END LeeNumeros ;

    PROCEDURE EscribeNumeros() ;
        VAR IndiceBucle : INTEGER ;
    BEGIN
        WriteLn ;
        WriteLine("Estos son los números ") ;
        FOR IndiceBucle := 0 TO MaxNumeros DO
            WriteInt( Numeros[ IndiceBucle ] , 3 ) ;
        END ;
    END EscribeNumeros ;

    PROCEDURE QuickSort( VAR Lista : ARRAY OF INTEGER; ExtremoIzquierdo,
                         ExtremoDerecho : INTEGER ) ;
        VAR HaciaDerecha, HaciaIzquierda, PuntoMedio : INTEGER ;
    BEGIN
        HaciaDerecha   := ExtremoIzquierdo ;
        HaciaIzquierda := ExtremoDerecho   ;
        PuntoMedio := ( ExtremoIzquierdo + ExtremoDerecho ) DIV 2 + 1 ;
        REPEAT
            WHILE Lista[ HaciaDerecha ] < Lista[ PuntoMedio]  DO
                INC( HaciaDerecha ) ;
            END ;
            WHILE Lista[ HaciaIzquierda ] > Lista[ PuntoMedio ]  DO
                DEC( HaciaIzquierda ) ;
            END ;

            IF HaciaDerecha <= HaciaIzquierda THEN
                Intercambia( Lista[ HaciaDerecha ], Lista[ HaciaIzquierda ] ) ;
                INC( HaciaDerecha ) ;
                DEC( HaciaIzquierda ) ;
            END ;
       UNTIL HaciaDerecha > HaciaIzquierda ;

       IF ExtremoIzquierdo < HaciaIzquierda THEN
            QuickSort( Lista, ExtremoIzquierdo, HaciaIzquierda ) ;
       END ;

       IF HaciaDerecha < ExtremoDerecho  THEN
            QuickSort( Lista, HaciaDerecha, ExtremoDerecho ) ;
       END ;

    END QuickSort ;

BEGIN
    LeeNumeros ;
    QuickSort( Numeros, 0, MaxNumeros ) ;
    WriteLn ;
    WriteLine("Estado Final ") ;
    EscribeNumeros ;
END MetodoQuickSort .
```

6.6.5.- Intercalación

Supongamos que tenemos dos tuplas, T1[N] y T2[M], ya ordenadas, y queremos intercalarlas, formando otra tupla T3[N+M], también ordenada. Un algoritmo muy burdo sería poner una tupla a continuación de la otra y reordenar la resultante, pero desaprovecharíamos el hecho de que ya estaban ordenadas.

Un algoritmo más adecuado es:

- ☞ Ir llenando la nueva lista cogiendo el menor de los que están al comienzo de ambas listas.
  - ★ Cuando una lista se termina, se llevan los restantes de la otra a continuación de la nueva

```
WHILE  ( n1 <> N ) AND ( n2 <> M )
       IF ( T1[n1] < T2[n2] )  THEN
            T3[i] := T1[n1] ;
            INC( i ) ;
            INC( n1 ) ;
      ELSE
            T3[i] := T2[n2] ;
            INC( i ) ;
            INC( n2 ) ;
      END ;

      IF ( n1 = N ) THEN
               FOR j = 0  TO  (N+M)-i   DO
                           T3[i+j]  :=  T2[n2 + j ] ;
               END ;
      END ;

      IF ( n2 = M ) THEN
               FOR j = 0  TO  (N+M)-i   DO
                           T3[i+j]  :=  T1[n1 + j ] ;
               END ;
      END ;
```

7 TIPOS DEFINIDOS POR EL USUARIO

7.1.- INTRODUCCIÓN

Los tipos de datos que proporciona cada lenguaje de alto nivel suelen bastar para manejar cualquier dato que se necesite en casi todo programa. Este manejo será de forma más o menos directa. Ejemplos:
Para procesar varios números enteros se almacena cada número en una variable tipo INTEGER. (forma muy directa). En un programa donde entren en juego como datos los días de la semana, se puede hacer corresponder a cada día un número entre 1 y 7, (forma indirecta).
Sin embargo, los lenguajes de alto nivel permiten la posibilidad al programador de definir y usar sus propios tipos de datos

Las definiciones de nuevos tipos en Modula-2 tienen lugar en la sección de declaraciones del programa; y se inician con la palabra TYPE. Hay varias formas de construir un nuevo tipo.

7.2.- REDEFINICIÓN DE TIPOS

Cualquier tipo de los existentes puede redefinirse con otro nombre. Forma de hacerlo:

```
TYPE < NuevoNombre > = < TipoAntiguo >
```

Ejemplos:

Declaración

```
TYPE
        Indice = CARDINAL ;
        Porcentaje = REAL ;
        IndiceBucle = Indice ; (* Puede redefinirse un tipo ya redefinido *)
```

Uso

```
VAR
        n : Indice ;
        i, j , k : IndiceBucle ;
```

Un tipo redefinido no es un nuevo tipo, sino un sinónimo de otro que existía previamente. Las ventajas que proporciona la redefinición de tipos son:
- Claridad en el programa. Hace que se necesiten menos comentarios explicativos, pues está autodocumentado. Ejemplo: al declarar una variable destinada a contener tantos por ciento, es más ilustrativo de su finalidad declararla de tipo `Porcentaje` que de tipo REAL .
- Comodidad en la reedición del programa. Ejemplo: Si en un programa se utilizan muchas variables de tipo `Indice` , y en una revisión se necesita que estas variables puedan ser negativas; en lugar de modificar la declaración de cada variable de tipo CARDINAL a INTEGER, basta con modificar la redefinición del tipo `Indice` , de CARDINAL a INTEGER.

7.3.- TIPOS ENUMERADOS

Una forma muy intuitiva de definir un nuevo tipo es enumerar todos los valores que puede tener.

Forma de hacerlo:

```
TYPE < NombreTipo > = (  < Valor1 >,  < Valor2 >, . . . ,  < ValorN > ) ;
```

Ejemplo:

Definición

```
TYPE    Ciencias = ( Fisicas, Matematicas, Quimicas ) ;
```

Declaración

```
VAR
        CarreraAlumno1, CarreraAlumno2 : Ciencias ;
```

Uso

```
        CarreraAlumno1 := Matematicas ;
        CarreraAlumno1 := Quimicas ;
```

7.3.1.- Operaciones con los Tipos Enumerados

7.3.1.1.- Operaciones de Relación

Al definir una enumeración se define implícitamente un ordenamiento creciente, desde el primer valor hasta el último. Por tanto:

```
CarreraAlumno1 <  CarreraAlumno2 es TRUE
CarreraAlumno1 >=  CarreraAlumno2 es FALSE
```

7.3.1.2.- Operaciones de Conversión CARDINAL ↔ ENUMERADO

`ORD( Valor )` Proporciona la posición (se comienza a contar desde 0) que ocupa el valor en la enumeración.
`VAL( <TipoDatoEnumerado>, Pos )` Proporciona el valor que en la enumeración `TipoDatoEnumerado` ocupa la posición `Pos`

Ejemplos:

`ORD( Matematicas )` devuelve un 1

`CarreraAlumno2 := VAL( Carreras, 0 )` asigna a la variable `CarreraAlumno2` el valor `Fisicas`

7.3.1.3.- Operaciones de Incremento y de Decremento

```
INC( Variable, Cantidad )
DEC( Variable, Cantidad )
```

Incrementan o decrementan (respectivamente) el valor de la variable en la cantidad dada.
Si se rebasa el extremo de la enumeración (por arriba o por abajo), se produce un error en tiempo de ejecución.
Si no se indica la cantidad, por defecto es 1.

Ejemplos:

```
CarreraAlumno1 := Fisicas ;
INC( CarreraAlumno1, 2 ) ;   (* Ahora CarreraAlumno1 vale Quimicas *)
INC( CarreraAlumno1 )    ;   (* Esto provoca un error en tiempo de ejecución *)
```

7.3.2.- Ventajas que proporcionan los tipos enumerados

- Claridad . Por el mismo motivo que el expuesto en las redefiniciones. El nombre de los valores puede ser ilustrativo de la finalidad de una variable.
- Seguridad. Facilitan la detección de errores. Supongamos que un dato de un cierto tipo sólo puede tomar un conjunto restringido de valores de ese tipo. Si por algún funcionamiento anómalo del programa tomara un valor no permitido, este error no sería detectado. Pero si definimos un nuevo tipo en el que estén sólo los valores permitidos, y la variable la declaramos de ese nuevo tipo, el error es detectado en tiempo de ejecución.

7.4.- TIPOS SUBRANGO

En algunas ocasiones un dato de un tipo dado, por la propia naturaleza del dato no puede tomar cualquier valor de ese tipo, sino sólo un rango restringido de valores. Se puede definir un tipo de datos llamado **subrango**, cuyos valores estarán comprendidos entre un valor inicial y otro final, y luego declarar una variable de ese nuevo tipo. La forma de definir un tipo subrango es:

```
TYPE  < Nombre > = [ ValorInicial .. ValorFinal ] ;
```

Ejemplo :

```
TYPE
  TipoHora = [1 .. 24 ] ;
  TipoLetra = [ 'A' .. 'Z' ]
  TipoDiaSemana = ( Domingo, Lunes, Martes, Miercoles, Jueves, Viernes, Sabado ) ;   (* Enumeración
*)
  TipoDiaDiario = [ Lunes .. Viernes ] ; (* Definido a partir de la enumeración *)

VAR
  Entrada, Salida : TipoHora ;
    . . .
  Entrada := 5 ;
  Salida := 9 ;
```

A veces es conveniente indicar el tipo base para evitar ambigüedades:

```
  TipoHora  = INTEGER[ 1 .. 24 ] ;
```

Esto evita posibles errores de compatibilidad de tipos al hacer operaciones en las que el compilador no tendría muy claro si utilizar el operador de INTEGER o el correspondiente de CARDINAL.

El tipo base puede ser INTEGER, CARDINAL, CHAR, enumerado u otro subrango, pero no REAL.

Una variable de tipo subrango puede utilizarse en la declaración de formaciones. Por ejemplo:

```
  Agenda : ARRAY TipoHora, [1 ..80] OF CHAR ;
```

Esto define una formación bidimensional de 24 elementos que son ristras de 80 caracteres.

7.5.- CONJUNTOS

7.5.1.-Creación de un Tipo Conjunto

Modula-2 es de los pocos lenguajes de alto nivel de propósito general que tiene instrucciones para definir conjuntos y operar con ellos. El método es el siguiente:

. Se parte de un conjunto completo, de un tipo base de datos.
. Se define un tipo de datos cuyos valores son subconjuntos de ese conjunto completo.

Forma de definir el nuevo tipo:

```
TYPE < NombreTipoConjunto > = SET OF < TipoBase >
```

Ejemplos:

```
TYPE
    (* Primero definimos los conjuntos completos, que serán los tipos base *)
    TipoFigura = ( Circulo, Elipse, Ovalo, Cuadrado, Rectangulo ) ;
    TipoDibujo = ( Monigote, Palote, Borron ) ;
    TipoRangoValores = [ 1 .. 31 ] ;

    (* Ahora los tipos Conjunto, que serán subconjuntos de los anteriores *)
    TipoConjuntoFigura = SET OF TipoFigura ;
    TipoConjuntoFiguraCurva = SET OF [ Circulo .. Ovalo ] ;
    TipoConjuntoImagen = SET OF [ Circulo .. Cuadrado, Monigote .. Borron ] ;
    TipoConjuntoLetras = SET OF CHAR ;

VAR
    Conjunto1Figuras : TipoConjuntoFiguras ;
    Conjunto1Letras : TipoConjuntoLetras ;
```

En las asignaciones, el conjunto que se asigna va entre llaves, los elementos separados por comas, y precedido por el nombre del tipo.

```
    Conjunto1Figuras := TipoConjuntoFiguras{ Circulo, Elipse, Rectangulo } ;
    Conjunto1Letras  := TipoConjuntoLetras{ 'B', 'D' .. 'G', 'S' ] ;
```

7.5.2.- Operaciones entre Datos de Tipo Conjunto

```
INCL(  < Variable de tipo Conjunto > , < Elemento > ) ;
EXCL( < Variable de tipo Conjunto > , < Elemento > ) ;
```
Incluyen o excluyen, respectivamente, un elemento en un conjunto.

`< Elemento > IN < Conjunto >` Devuelve TRUE o FALSE, según esté incluido o no el elemento en el conjunto.

Unión	(+)	C = A + B ;	Devuelve la unión de A y B
Intersección	(*)	C = A * B ;	Devuelve la intersección de A y B
Diferencia	(-)	C = A - B ;	Devuelve el conjunto de elementos que están en A pero no en B
Diferencia simétrica	(/)	C = A / B ;	Devuelve el conjunto de elementos que están en A o en B, pero no simultáneamente en los dos.

El tipo base debe ser uno cuyo conjunto completo sea finito, por lo que puede ser enumerado, subrango, CHAR, pero no INTEGER ni CARDINAL ni REAL

7.5.3. Tipo BITSET

BITSET es un conjunto de números comprendidos entre 0 y N-1, siendo N el número de bits de las palabras que utiliza el compilador. N suele ser 16.

Un dato tipo BITSET puede tomar valores entre 0 y N-1

Ejemplo:

Este programa lee caracteres desde el teclado, y los cuenta. Por una parte los alfabéticos, por otra los numéricos, por otra parte los espacios en blanco; y por otra los totales.

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:          Contar.Mod                                        ║
  ║    Propósito:       Cuenta las letras y dígitos de un texto           ║
  ║    Autor:           José Garzía                                       ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE Contar;
        FROM InOut IMPORT WriteString, WriteLn, WriteInt, Write, Read ;
        TYPE
             Caracteres = SET OF CHAR ;
             Letras     = SET OF ["A".."Z"] ;
             Digitos    = SET OF ["0".."9"] ;

        VAR
             ListaLetras : Letras ;
             ListaDigitos: Digitos ;
             Total, TotalLetras, TotalDigitos, TotalBlancos : INTEGER ;

       PROCEDURE Analizar ;
            VAR c : CHAR ;
        BEGIN
           Total := 0 ;
           TotalLetras := 0 ;
           TotalDigitos := 0 ;
           TotalBlancos := 0 ;
           ListaLetras := Letras{} ;
           ListaDigitos := Digitos{} ;
           c := ' ' ;
           WHILE c <> '.' DO
                Read(c) ;
                Write(c) ;
                INC( Total ) ;

                IF c IN Caracteres{ "a".."z", "A".."Z"} THEN
                    INC( TotalLetras );
                    c := CAP( c ) ;
                    INCL( ListaLetras, c ) ;
                ELSIF c IN Digitos{"0".."9"}          THEN
                    INC(TotalDigitos ) ;
                    INCL( ListaDigitos, c) ;
                ELSIF c = " "                         THEN
                        INC( TotalBlancos ) ;
               (*ELSE*) (* No hace nada *)

               END   (* Del IF *)
          END (* Del bucle WHILE *)
    END Analizar;
```

```
PROCEDURE DarResultados ;
    VAR c: CHAR ;
    BEGIN
        WriteLn ;
        WriteLn ;
        WriteString("   =================") ;
        WriteLn ;
        WriteLn ;
        WriteString("Caracteres totales:") ;
        WriteInt( Total, 4) ;
        WriteLn ;
        WriteString("Letras totales:") ;
        WriteInt( TotalLetras, 4) ;
        WriteLn ;
        WriteString("Digitos totales:") ;
        WriteInt( TotalDigitos, 4) ;
        WriteLn ;
        WriteString("Blancos totales:") ;
        WriteInt( TotalBlancos, 4) ;
        WriteLn ;
        WriteLn ;
        WriteString("LISTA DE LETRAS NO UTILIZADAS:") ;
        WriteLn ;

        FOR c := "A" TO "Z" DO
            IF NOT ( c IN ListaLetras ) THEN
                Write( c ) ;
                Write(" ") ;
            END
        END ;

        WriteLn ;
        WriteLn ;
        WriteString("LISTA DE DIGITOS UTILIZADOS:") ;
        WriteLn ;

        FOR c := "0" TO "9" DO
            IF     ( c IN ListaDigitos ) THEN
                Write( c ) ;
                Write(" ") ;
            END
        END

END DarResultados ;

BEGIN
    Analizar ;
    DarResultados
END Contar.
```

7.6.- REGISTROS

Con frecuencia existen agrupaciones de datos que aunque son de diferentes tipos, están muy relacionados entre sí, por lo que aunque son compuestos, pueden ser tratados como unidades.
La posibilidad de hacer referencia a toda una colección de elementos mediante un nombre único simplifica en muchos casos la redacción del programa

Ejemplo:

Consideremos una base da datos de una universidad, que tiene registrados los alumnos matriculados y los datos de cada alumno; y los clasifica por campos de esta manera:

ALUMNO				
Nombre	Apellido 1º	Apellido 2º	Año de nacimiento	Sexo

Registro. Es la agrupación de los datos, que aunque de tipos diferentes, están muy relacionados entre sí, y son tratados como una unidad.
Campo. Es cada uno de los datos que forman un registro.

El tipo de los datos de los campos puede ser cualquiera de los predefinidos por el lenguaje, o de los definidos por el programador (redefinición, enumeración, subrango, conjuntos). E incluso otros registros, por lo que puede haber registros anidados
Ejemplo:

DIRECCIÓN		
Ciudad	Calle	Número

ALUMNO					
Nombre	Apellido 1º	Apellido 2º	Año de nacimiento	Sexo	DIRECCIÓN
					Ciudad / Calle / Número

También puede haber listas de registros, formando una **tabla**

	ALUMNO							
Índice del alumno	Nombre	Apellido 1º	Apellido 2º	Año de nacimiento	Sexo	DIRECCIÓN		
						Ciudad	Calle	Número
1								
2								
3								
.....								
n								

De la misma forma que los procedimientos posibilitan la abstracción sucesiva de acciones, los registros posibilitan la abstracción sucesiva de datos. Ejemplo:

CUADRADO	
Lado	Color

Puede abstraerse en

RECTÁNGULO		
Base	Altura	Color

7.7.- REGISTROS EN MODULA-2

7.7.1.- Formas posibles de Definirlos

7.7.1.1.- Con nombre

```
TYPE < Nombre del tipo de registro > = RECORD
                                          < Nombre del Campo 1 > : < Tipo del Campo 1 > ;
                                          < Nombre del Campo 2 > : < Tipo del Campo 2 > ;
                                            .  .  .  .  .  .  .  .  .  .  .  .  .
                                          < Nombre del Campo N > : < Tipo del Campo N > ;
                                      END ;
```

7.7.7.2.- Anónima

```
VAR  < Nombre de la variable > = RECORD
                                      < Nombre del Campo 1 > : < Tipo del Campo 1 > ;
                                      < Nombre del Campo 2 > : < Tipo del Campo 2 > ;
                                       .  .  .  .  .  .  .  .  .  .  .  .  .
                                      < Nombre del Campo N > : < Tipo del Campo N > ;
                                  END ;
```

De cada tipo de registro definido de forma anónima sólo se puede declarar una variable. Los nombres de los campos son locales en cada registro. Puede haber tipos diferentes de registros en los que algún nombre de campo coincida. Los campos consecutivos del mismo tipo pueden ir separados por comas y sólo el último debe llevar la especificación del tipo.

Ejemplos:

```
TYPE
    TipoPunto = RECORD
                      x, y : REAL ;
                END ;

    Direccion = RECORD
                      Calle : ARRAY [1..20] OF CHAR ;
                      Numero : CARDINAL ;
                      Ciudad : ARRAY [1..10] OF CHAR ;
                END ;

    Alumno = RECORD
                      Nombre,
                      Apellido1,
                      Apellido2 : ARRAY [1..10] OF CHAR ;
                      Edad : CARDINAL ;
                      Domicilio : Direccion ;
             END ;

VAR
    NuevoAlumno : Alumno ;
    ListaAlumnos : ARRAY [1..1000] OF Alumno ;
    Punto1 : TipoPunto;
    Triangulo : ARRAY[0..2] OF TipoPunto ;
    RobinsonCrusoe : RECORD
                           Nombre : ARRAY [1..10] OF CHAR ;
                           Bandera : BOOLEAN ;
                     END ;
```

7.7.2.- Operaciones con Registros Enteros

7.7.2.1 Asignaciones

```
    ListaAlumnos[12] := NuevoAlumno ;
```

Tras la asignación, todos los campos de la variable `ListaAlumnos[12]` pasan a contener el mismo valor que sus correspondientes de `NuevoAlumno`
Los operandos a cada lado del operador de asignación deben ser del mismo tipo de registro. No basta con que sus campos sean compatibles una a uno.

Ejemplo:

```
TYPE OtroTipoPuntoIgual = RECORD  x, y : REAL ;  END ;

VAR Punto1 : TipoPunto ;
    Punto2 : OtroTipoPuntoIgual ;
     .  .  .  .  .  .
    Punto1 := Punto2 ;         (* Esto provoca un error de compilación *)
```

7.7.2.2.- Paso como parámetro a un procedimiento

Ejemplo:

```
PROCEDURE  Matricular( VAR  Novato : Alumno ) ;
```

Las variables definidas con un tipo de registro anónimo no pueden ser pasadas como parámetros. No tiene sentido:

```
PROCEDURE Procesar( VarReg : RECORD < Sección de definición de campos > END ) ;
```

pues aunque luego sea enviada una variable de campos compatibles, no sería reconocida.

7.7.3.- Modo de acceso a los campos de un registro

Se utiliza el nombre particular de la variable de tipo registro, y el nombre genérico del campo, separados por el operador punto. Si hay varios registros anidados, se utiliza un punto para acceder a cada uno de los sucesivos campos.

Ejemplos:

```
ReadString( NuevoAlumno.Nombre ) ;
ReadCard( NuevoAlumno.Direccion.Numero ) ;
Punto1.x := 5 * Punto2.x  ; (* Correcto, aunque Punto1 y Punto2 no sean del mismo tipo *)
Punto1.y := 4 * Punto2.x ;  (* Correcto, se opera con los valores, sin tener en cuenta en qué campo
están contenidos *)
```

El orden de evaluación de los operadores corchete y punto es de izquierda a derecha:

```
WriteString( ListaAlumnos[12].Apellido1 ) ; (* Primero se accede al elemento 12 de la tabla
                                               ListaAlumnos, y dentro de éste, al campo apellido1 *)
```

7.7.4.-El Operador WITH

Cuando dentro de un fragmento de programa se está accediendo con frecuencia a campos de un único registro, hay una forma de evitar la obligación de usar el operador punto. El operador WITH sirve para individualizar la variable tipo registro en ese fragmento de programa. De esta forma no hay posible confusión con campos de otras variables del mismo tipo si omitimos el operador punto. La forma de operar es:

```
WITH  < Nombre de la variable de tipo registro >  DO
            < Bloque de sentencias >
END ;
```

Ejemplo :

```
WITH Punto1 DO
      x := 5.0 ;
      y := 10.5 ;
END ;
```

Los operadores WITH pueden anidarse. Si hay campos de registros diferentes con igual nombre, para evitar que se cree una situación confusa hay que tener en cuenta esto:
Cada nombre de un campo se refiere al registro fijado por el operador WITH inmediatamente anterior que esté sin cerrar con su END correspondiente.

Ejemplo:

```
VAR
    VariableA : RECORD
                      uno : Tipo11;
                      dos : Tipo12 ;
                         .  .  .
                END ;

    VariableB : RECORD
                      dos : Tipo21;
                      tres : Tipo22 ;
                         .  .  .
                END ;
              .  .  .

WITH  VariableA DO
      . . . uno . . .                    (* Se refiere a VariableA.uno *)
      . . . dos . . .                    (* Se refiere a VariableA.dos *)
     WITH VariableB  DO
        . . . uno . . .                  (* Se refiere a VariableA.uno *)
        . . . dos . . .                  (* Se refiere a VariableB.dos  no a  VariableA.dos  *)
        . . . tres . . .                 (* Se refiere a VariableB.tres *)
     END ;
END ;
```

7.8.- SOLAPAMIENTO DE VARIABLES

Hay aplicaciones en las que sería deseable que el tipo de un dato pudiera cambiar en el transcurso de la ejecución del programa, según que las circunstancias fueran de una manera o de otra. Los lenguajes de alto nivel permiten esto. La estrategia que utilizan es solapar varias variables de diferentes tipos en una misma localización de memoria. Después, según sean dichas circunstancias, el contenido de esa localización de memoria se interpreta de una manera o de otra.

Las situaciones típicas en las que tiene utilidad el solapamiento, entre otras son:

1ª Datos que pueden ser representados de maneras diferentes.

Ejemplo:

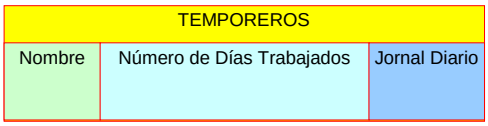
Un mismo número complejo, en unos momentos del programa puede ser conveniente expresarlo en forma cartesiana y en otros momentos en forma polar. Así pues, el programa en unos momentos debe trabajar con dos datos de tipo REAL (abscisa y ordenada); y en otros momentos con un REAL y un INTEGER (módulo y argumento, respectivamente).

2ª Registros con campos opcionales

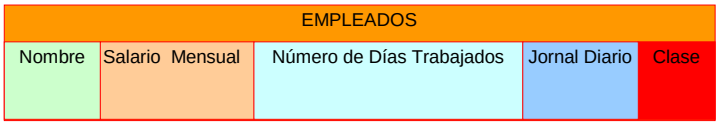
Puede suceder que de un registro unas veces se utilicen sólo ciertos campos, y otras veces sólo los restantes. El uso de unos u otros dependerá del estado de alguna variable. Si se sabe con seguridad que nunca se van a necesitar todos los campos a la vez, sería útil compactar el registro, y en los mismos campos guardar los diferentes datos, que unas veces serían interpretados de una forma y otras veces de otra, según sea el estado de la variable citada.

Veamos este ejemplo:

Supongamos que en una empresa hay dos clases de empleados: unos son fijos y otros trabajan sólo algunos días cada mes. Los registros para cada clase de empleados serían diferentes:



Un programa que procese los datos de todos los trabajadores será más eficiente si hace un procesamiento unificado, independientemente de cuál sea su clase. Debería haber un sólo tipo de registro para las dos clases:



Como vemos, hay espacios de memoria desaprovechados, pues dependiendo del valor que tenga el campo *Clase*, hay unos campos u otros cuyo valor no tiene ningún significado relevante.

Sería conveniente tener un campo el cual, dependiendo del valor del campo *Clase* contuviera el valor de *Salario Mensual*, o el subregistro anidado *Numero de Días Tabajados / Jornal Diario.*

7.9.- REGISTROS CON CAMPOS SOLAPADOS EN MODULA-2

La forma general de la parte solapada en un registro es:

```
CASE < Variable Selectora > : < tipo de la variable selectora > OF
           < Constante 1 > : < Lista 1 de campos y sus tipos correspondientes >  |
           < Constante 2 > : < Lista 2 de campos y sus tipos correspondientes >  |
                                 .  .  .
           < Constante N > : < Lista N de campos y sus tipos correspondientes >
           ELSE < Lista alternativa de campos por defecto y sus tipos correspondientes >
END ;
```

La variable selectora es uno más de los campos. Su declaración se hace sólo después de la palabra CASE. Por lo cual debe llevar su tipo entre su nombre y la palabra OF.

Ejemplo:

```
TYPE Categoria = ( Temporero, Fijo ) ;

TYPE Empleado = RECORD
           Nombre : ARRAY [0 .. 20 ] OF CHAR ;

          CASE Clase : Categoria OF           (* Parte solapada del registro *)
          Temporero  :
                      NumDiasTrabajados  : CARDINAL ;
                      JornalDiario       : CARDINAL ; |
          Fijo      :
                      SalarioMensual     : CARDINAL ;
          END ;      (* Del CASE  *)
END ;  (*  Del RECORD  *)
```

Estos registros pueden tener diferentes fomas en diferentes momentos del programa. Su forma se determina en tiempo de ejecución. Son una herramienta muy potente para ahorrar memoria. La contrapartida es que son proclives a provocar errores. Por ejemplo:

Supongamos que `UnEmpleado.Clase` tiene como valor `Temporero`; y hacemos

```
WriteCard( UnEmpleado.SalarioMensual, 10 ) ;
```

Como el compilador no hace ninguna comprobación automática, estaríamos imprimiendo unos datos sin nigún significado. Las comprobaciones de la variable que se utiliza como discriminante son responsabilidad del programador.

7.10.- CORRESPONDENCIA ENTRE TIPOS DE DATOS COMPUESTOS Y ESQUEMAS DE CONTROL

Se ve que existe un correspondencia entre estas parejas:

FORMACIÓN	⟷	ITERACIÓN	Colección de elementos del mismo tipo, procesados de forma similar
REGISTRO	⟷	SECUENCIA	Colección de elementos de tipos diferentes, combinados en un orden fijo.
SOLAPAMIENTO	⟷	SELECCIÓN	Colección de elementos de tipos diferentes, seleccionando cuál procesar

Los programas serán más claros si para cada tipo de datos compuesto usamos su esquema de control más adecuado. Ejemplo:

```
VAR
      Registro  : RECORD
                     Uno, Dos, Tres : CHAR ;
                  END ;

      Solapado : RECORD
                       CASE Discriminante : BOOLEAN  OF
                             TRUE   : Alfa : INTEGER |
                             FALSE  : Beta : REAL ;
                       END ;
                 END ;

      Formacion : ARRAY [1 .. 10]  OF  INTEGER ;
         .  .  .  .  .
        (*  Imprimir los campos de un registro *)
      Write( Registro.Uno  ) ;
      Write( Registro.Dos  ) ;
      Write( Registro.Tres ) ;

        (* Imprimir el campo significativo de varios solapados *)
      IF Solapado.Discriminante THEN
         WriteInt(  Solapado.Alfa,  5) ;
      ELSE
         WriteReal( Solapado.Beta, 5) ;
      END ;

        (* Imprimir los datos de una formación *)
      FOR  i := 1 TO 10 DO
          WriteInt(  Formacion[i],  5) ;
      END ;
```

Como ejemplo de los conceptos de este tema, tenemos el programa para cálculos con números complejos mejorado. Los nombres de los procedimientos y las variables son lo suficientemente explicativos de sus respectivas funciones, por lo que explican casi por sí solos el funcionamiento del programa.

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:          Complej1.Mod                                      ║
  ║    Propósito:       Calculadora de números complejos                  ║
  ║    Autor:           José Garzía                                       ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)


MODULE  NumerosComplejos;
FROM InOut     IMPORT WriteString, Write, WriteLn, Read ;
FROM RealInOut IMPORT WriteReal, ReadReal;
FROM MathLib0    IMPORT sqrt, sin, cos, arctan ;

TYPE Letras = SET OF CHAR ;

TYPE TipoCoordenadas = ( Cartesianas, Polares ) ;
TYPE
     NumeroComplejo = RECORD
                          CASE Coordenadas : TipoCoordenadas  OF
                               Cartesianas : ParteReal       : REAL ;
                                             ParteImaginaria : REAL ; |
                               Polares     : Modulo          : REAL ;
                                             Argumento       : REAL ;
                          END ;
                      END ;

PROCEDURE ArcoTangente( Abscisa, Ordenada : REAL ) : REAL ;
    VAR Angulo, Tangente : REAL ;
    BEGIN
           (* Este IF evita una división por cero *)
        IF ( ( Abscisa >= -0.00001 ) AND (Abscisa <= 0.00001 ) ) THEN
            IF Ordenada < 0.0 THEN Angulo := -3.1416 / 2.0 ;
            ELSE Angulo := 3.1416 / 2.0 ;
            END ;

        ELSE  (* No hay peligro de dividir por cero *)
            Tangente := Ordenada / Abscisa ;

            Angulo := arctan( Tangente ) ;

            IF ( ( Abscisa < 0.0)  AND ( Ordenada > 0.0) ) THEN
                 Angulo := Angulo + 3.1416  ;
            END ;

            IF ( (Abscisa < 0.0)  AND ( Ordenada < 0.0)  ) THEN
                 Angulo := Angulo - 3.1416 ;
            END ;

      END ;
    RETURN Angulo ;
    END ArcoTangente ;


PROCEDURE CartesianasAPolares( VAR z : NumeroComplejo ) ;
    VAR
        Mod : REAL ;
        Arg : REAL ;

    BEGIN
        IF z.Coordenadas = Cartesianas THEN
            WITH z DO
                Mod := CalculaModulo( z ) ;
                Arg := ArcoTangente( ParteReal, ParteImaginaria ) ;

                Coordenadas := Polares ;
                Modulo    := Mod ;
                Argumento := Arg ;
            END ;
        END ;
    END   CartesianasAPolares ;
```

```
PROCEDURE PolaresACartesianas( VAR z : NumeroComplejo ) ;
    VAR
        x, y : REAL ;

    BEGIN
        IF z.Coordenadas = Polares THEN
            WITH z DO
                x := Modulo * cos( Argumento ) ;
                y := Modulo * sin( Argumento ) ;

                Coordenadas := Cartesianas ;
                ParteReal       := x ;
                ParteImaginaria := y ;
            END ;
        END ;
    END   PolaresACartesianas ;


PROCEDURE Sumar( z1, z2 : NumeroComplejo; VAR z3 : NumeroComplejo ) ;
    VAR EstadoInicial : TipoCoordenadas ;
    BEGIN
        EstadoInicial := z3.Coordenadas ;
        PolaresACartesianas( z1 ) ;
        PolaresACartesianas( z2 ) ;
        PolaresACartesianas( z3 ) ;

        z3.ParteReal       := z1.ParteReal       + z2.ParteReal ;
        z3.ParteImaginaria := z1.ParteImaginaria + z2.ParteImaginaria ;

        (* Sólo es necesario restaurar el sistema inicial de coordenadas
           en el argumento pasado por referencia                        *)
        IF EstadoInicial = Polares THEN CartesianasAPolares( z3 ) ; END ;

    END Sumar ;

PROCEDURE Restar( Minuendo, Sustraendo : NumeroComplejo;
                  VAR Resultado : NumeroComplejo ) ;
   VAR EstadoInicial : TipoCoordenadas ;
   BEGIN
        EstadoInicial := Resultado.Coordenadas ;
        PolaresACartesianas( Minuendo ) ;
        PolaresACartesianas( Sustraendo ) ;
        PolaresACartesianas( Resultado ) ;

        WITH Resultado DO
            ParteReal       := Minuendo.ParteReal       - Sustraendo.ParteReal  ;
            ParteImaginaria := Minuendo.ParteImaginaria - Sustraendo.ParteImaginaria ;
        END ;

        IF EstadoInicial = Polares THEN CartesianasAPolares( Resultado ) ; END ;

   END Restar ;

PROCEDURE Multiplicar( z1, z2 : NumeroComplejo; VAR z3 : NumeroComplejo ) ;
   VAR EstadoInicial : TipoCoordenadas ;
   BEGIN
        EstadoInicial := z3.Coordenadas ;
        CartesianasAPolares( z1 ) ;
        CartesianasAPolares( z2 ) ;
        CartesianasAPolares( z3 ) ;
        WITH z3 DO
            Modulo    :=  z1.Modulo     *  z2.Modulo ;
            Argumento :=  z1.Argumento  +  z2.Argumento ;
        END ;

        IF EstadoInicial = Cartesianas THEN PolaresACartesianas( z3 ) END ;

   END Multiplicar ;

PROCEDURE Dividir( Dividendo, Divisor : NumeroComplejo; VAR Cociente : NumeroComplejo ) ;
   VAR EstadoInicial : TipoCoordenadas ;
   BEGIN
        EstadoInicial := Cociente.Coordenadas ;
        CartesianasAPolares( Dividendo ) ;
        CartesianasAPolares( Divisor   ) ;
        CartesianasAPolares( Cociente  ) ;
        WITH Cociente DO
            Modulo    :=  Dividendo.Modulo     /  Divisor.Modulo ;
            Argumento :=  Dividendo.Argumento  -  Divisor.Argumento ;
        END ;

        IF EstadoInicial = Cartesianas THEN PolaresACartesianas( Cociente ) END ;

   END Dividir ;
```

```
PROCEDURE CalculaModulo( z : NumeroComplejo ): REAL ;
   BEGIN
        PolaresACartesianas( z ) ;
        RETURN sqrt( z.ParteReal       * z.ParteReal +
                     z.ParteImaginaria * z.ParteImaginaria   ) ;
   END CalculaModulo ;



PROCEDURE Leer( VAR z : NumeroComplejo ) ;
   VAR Letra : CHAR ;

   BEGIN
        WriteLn ;
        WriteString("¿ En qué forma me va a dar el número complejo ?  ( C ó P ) " );
        REPEAT
             Read( Letra ) ;
        UNTIL ( Letra IN Opciones ) ;

        WriteLn ;

        CASE Letra OF
            'c', 'C' :
                       WriteString("Parte real: ") ;
                       ReadReal( z.ParteReal ) ; WriteLn ;
                       WriteString("Parte imaginaria: ") ;
                       ReadReal( z.ParteImaginaria ) ; WriteLn ;
                       z.Coordenadas := Cartesianas ;  |
            'p', 'P':
                       WriteString("Módulo: ") ;
                       ReadReal( z.Modulo ) ; WriteLn ;
                       WriteString("Argumento: ") ;
                       ReadReal( z.Argumento ) ; WriteLn ;
                       z.Coordenadas := Polares ;

        END ;


   END Leer ;

PROCEDURE Escribir( z : NumeroComplejo ) ;
   VAR Letra : CHAR ;
   BEGIN
        WriteLn ;
        WriteString("¿ En qué forma quiere ver el número complejo ?  ( C ó P ) " );
        REPEAT
             Read( Letra ) ;
        UNTIL ( Letra IN Opciones ) ;

        WriteLn ;

        CASE Letra OF
            'c', 'C' :
                         PolaresACartesianas( z ) ;
                         WriteString( "Resultado: (" ) ;
                         WriteReal( z.ParteReal, 1 ) ;
                         WriteString(", ") ;
                         WriteReal( z.ParteImaginaria, 1) ;
                         Write( ')') ;  |
            'p', 'P':
                         CartesianasAPolares( z ) ;
                         WriteString( "Resultado: " ) ;
                         WriteReal( z.Modulo, 1 ) ;
                         WriteString("  *  exp( i * ( ") ;
                         WriteReal( z.Argumento, 1 ) ;
                         WriteString(" ) ) ") ;
        END ;

   END Escribir ;



VAR
        Acumulador, NuevoNumero : NumeroComplejo ;
        Operacion : CHAR ;
        Opciones : Letras ;

PROCEDURE LeerOperacion( VAR CodigoOperacion : CHAR ) ;
    BEGIN
        WriteLn ;
        Write('?') ; Read( CodigoOperacion ) ;
        CodigoOperacion := CAP( CodigoOperacion ) ;
        Write( CodigoOperacion ) ; Write( ' ') ;
    END LeerOperacion ;
BEGIN
```

```
   Acumulador.Coordenadas := Cartesianas ;
   Acumulador.ParteReal       := 0.0 ;
   Acumulador.ParteImaginaria := 0.0 ;
   Opciones := Letras{ "c", "C", "p", "P" } ;

REPEAT
    LeerOperacion( Operacion) ;
    CASE Operacion OF
        '+':
            Leer( NuevoNumero ) ;
            Sumar( Acumulador, NuevoNumero, Acumulador ) |
        '-':
            Leer( NuevoNumero ) ;
            Restar( Acumulador, NuevoNumero, Acumulador ) |
        '*', 'x', 'X':
            Leer( NuevoNumero ) ;
            Multiplicar( Acumulador, NuevoNumero, Acumulador ) |
        '/':
            Leer( NuevoNumero ) ;
            Dividir( Acumulador, NuevoNumero, Acumulador ) |
        ' ':
            Leer( Acumulador ) ; |
        '=' :
            WriteString( '      ' );
            Escribir( Acumulador ) ; |

    ELSE
            WriteString( 'Pulse +. -, *, /, ESPACIO, = , ó ESC ' ) ;
            WriteLn ;
    END ;

    UNTIL Operacion = CHR( 27 ) ;

END NumerosComplejos.
```

8 MÓDULOS

8.1.- INTRODUCCIÓN

En todo programa relativamente complejo pueden identificarse fragmentos que tienen entidad propia. Pueden considerarse como unidades lógicas; por ende, pueden ser desarrollados de forma independiente.

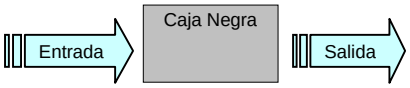
Módulo. Fragmento de programa desarrollado de forma independiente del resto del programa.

Los lenguajes de alto nivel modernos permiten la posibilidad de compilar por separado varios ficheros; y enlazarlos en un programa.

8.1.1.- Ventajas de la Modularidad

Facilidad de desarrollo

Es la más manifiesta de todas ellas. Consiste en la simplificación que se deriva de toda división del trabajo. Un gran proyecto puede dividirse en módulos y asignar cada uno de éstos a un grupo de programadores. Cada grupo está encargado de un subprograma cuya complejidad es muy inferior a la del programa completo. El grupo puede concentrarse en él, dejando como secundarios los detalles de cómo su módulo interactuará con los demás. A su vez, los demás módulos lo utilizarán como una **caja negra**, de la cuál se conoce cómo es la salida que se presenta con cada entrada que se aplica, pero no es necesario conocer cómo está hecha por dentro.



Por esto, el concepto de módulo está muy ligado al de abstracción. Desde fuera se sabe *qué* hace, pero no *cómo*.

Aunque la coordinación de todos los grupos de programadores es un trabajo extra, lo más probable es que la suma de las complejidades de todos los módulos sea menor que la complejidad del programa completo.

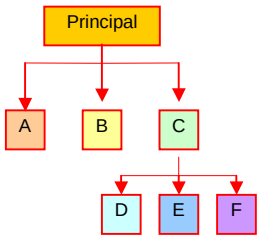
Facilidad de mantenimiento

Después de estar hecho el programa, para mejorarlo, quizá se quiera modificar de forma que una determinada tarea la realice de otra manera más eficiente. Entonces basta con modificar el módulo donde estaba definida dicha tarea.

8.2.- CRITERIOS PARA UNA ACERTADA DESCOMPOSICIÓN MODULAR.

Normalmente, cada módulo usa sólo alguno de los demás, y el programa principal utiliza directamente sólo unos pocos de los secundarios.

Ejemplo:



El principal sólo utiliza directamente A, B y C.
D es utilizado por B y C
C utiliza D, E y F.

8.2.1.- Cohesión

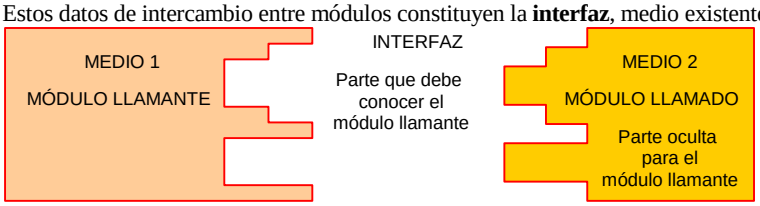
Al hacer la división hay que procurar que las tareas de cada módulo sean lo más coherentes posible entre sí.

Supongamos que las tareas de E son muy diferentes de las de F, pero que siempre que se utiliza E también se utiliza F. Por una parte no parece una mala medida unificar E y F en un único módulo, pero si realmente las diferencias llegaran a ser muy grandes, quizá fuera más lógico tener un módulo para cada grupo de tareas.

Es conveniente que el grado de relación entre los elementos de un módulo sea lo más grande posible.

8.2.2.- Conexión

Siempre que un módulo es llamado por otro, el llamante debe conocer cuál es la información necesaria que debe proporcionar al llamado. Estos datos de intercambio entre módulos constituyen la **interfaz**, medio existente entre dos medios que están en contacto.



Es conveniente que la interfaz sea lo más reducida posible.

8.3.- MÓDULOS EN MODULA-2

El nombre de este lenguaje indica que uno de los propósitos al ser creado fue proporcionar una herramienta para el desarrollo modular de los programas.

Además de las ventajas generales de la modularidad, en Modula-2 existen otras:

Compilación segura

En las llamadas entre módulos, el compilador hace comprobaciones de que los datos proporcionados son consistentes con la interfaz de ese módulo.

Encapsulamiento

No es necesario el conocimiento por parte del llamador de los detalles de la realización interna del llamado. Puede desconocerlos. Pueden ser considerados como si estuvieran encerrados en una caja negra.

Mayor flexibilidad en las reglas de ámbito de las variables

Podemos optar por que los identificadores de un módulo sean o no conocidos en los demás. Esto sirve para evitar efectos colaterales y otros conflictos en los nombres de los identificadores. Además, permiten la síntesis de variables estáticas, que mantienen su valor desde que el flujo del programa sale de su ámbito, hasta que vuelve a entrar.

En Modula-2 hay cuatro tipos de módulos:

Módulo Principal
Módulo de Definición
Módulo de Implementación
Módulo Local

Cada biblioteca está formada el módulo de definición y su correspondiente de implementación.

8.3.1.- Módulo Principal

Contiene el código donde se hacen llamadas a procedimientos construidos en otros módulos. Este módulo no es llamado por ningún otro. Todo programa tiene como mínimo un módulo: el principal.
Forma general:

```
MODULE NombreModuloPrincipal ;
      < Lista de importaciones de los módulos de biblioteca > ;
      < Lista de  definiciones de tipos >
      < Lista de declaraciones de variables >
BEGIN
      < Cuerpo del módulo > ;
END  NombreModuloPrincipal .
```

8.3.2.- Módulo de Definición

Describe **qué** hace la bibioteca. Es decir, es la interfaz de la biblioteca. Contiene las declaraciones de los procedimientos y datos.
Forma general:

```
DEFINITION MODULE NombreBiblioteca ;
      < Lista de importaciones de otros módulos  > ;
      EXPORT QUALIFIED  < Lista de indentificadores exportados > (* Lista de exportaciones *)
      <  Definiciones  y declaraciones > ;
END  NombreBiblioteca .
```

La lista de importaciones indica los identificadores que utiliza de otros módulos. Su forma y finalidad son iguales que en la lista de importaciones del módulo principal.
La lista de exportaciones expresa los identificadores que pueden ser conocidos por el resto del programa. Sólo puede haber una sentencia EXPORT en cada módulo de definición.

En la mayoría de compiladores este módulo está en un fichero cuyo nombre es el de la biblioteca, y la extensión es .DEF

8.3.3.- Módulo de Implementación

Describe **cómo** actúa la biblioteca. Contiene el código de las acciones.

Forma general:

```
IMPLEMENTATION MODULE NombreBiblioteca ;
          < Lista de importaciones >
          < Lista de  definiciones de tipos >
          < Lista de declaraciones de variables >
BEGIN
          < Cuerpo del módulo. ( Opcional, si no lo tiene, no necesita la sentencia BEGIN ) >
END  NombreBiblioteca .
```

En la mayoría de compiladores este módulo está en un fichero que tiene el nombre de la biblioteca y la extensión .MOD

8.3.4.- Creación de una bilioteca

Vamos a ver un ejemplo muy simple, pero ilustrativo de cómo se crea una biblioteca en Modula-2.

Proyecto completo:
Crear un programa que sirva para calcular el factorial de un número dado.

Biblioteca:
Contiene el procedimiento `Factorial`, su código está en el módulo de implementación, y la declaración de su interfaz está en el módulo de definición.

```
(*   MiBiblio.Def  *)

DEFINITION MODULE MiBiblio ;
    EXPORT QUALIFIED Factorial ;
    PROCEDURE Factorial( C : CARDINAL ) : CARDINAL ;
END MiBiblio .


(*    MiBiblio.Mod    *)

IMPLEMENTATION MODULE MiBiblio ;
    PROCEDURE Factorial( Numero : CARDINAL ) : CARDINAL ;
    VAR
        Acumulador, Contador, NumeroLeido : CARDINAL ;
    BEGIN
        Acumulador := 1 ;

        FOR Contador := 2 TO Numero BY 1  DO
            Acumulador := Acumulador * Contador ;
        END ;
        RETURN Acumulador ;
    END Factorial ;
END MiBiblio .


(*      Factor3.Mod     *)

MODULE Factor3 ;
    FROM InOut IMPORT WriteString, ReadCard, WriteCard, WriteLn ;
    FROM MiBiblio IMPORT Factorial ;
    VAR NumeroLeido : CARDINAL ;

    BEGIN
        WriteString( "Dígame el número cuyo factorial quiere calcular: " );
        ReadCard( NumeroLeido ) ;
        WriteLn ;
        WriteString( "El factorial de " );
        WriteCard( NumeroLeido, 2 ) ;
        WriteString( " es " );
        WriteCard( Factorial( NumeroLeido ), 3 ) ;
        WriteLn ;

END Factor3 .
```

8.3.5.-Inicialización de los módulos de implementación

En los módulos de implementación se puede incluir un conjunto de sentencias, encerradas entre BEGIN y END.

La finalidad normal de estas sentencias es dar valores iniciales a las variables locales del módulo, o poner en marcha algunos procedimientos.

Esta inicialización se hace sólo una vez, al importar el módulo, y antes de empezar a ejecutarse el código del módulo importador.

Si hay anidamiento en las importaciones (A importa desde B y B importa desde C) se comienza a ejecutar desde el último módulo importado: C, B y finalmente A.

Como ejemplo, vamos a reelaborar el programa para leer un párrafo de texto, no mayor de 80 caracteres, acabado en un punto. Los procedimientos los pondremos en un módulo de implementación. Por supuesto, se necesita un módulo de definición donde se describe la interfaz de la biblioteca.

```
DEFINITION MODULE LibFrase ;
    EXPORT QUALIFIED Frase, Lleno, Almacena, Remata ;

    VAR Frase : ARRAY [0..79] OF CHAR ;
    PROCEDURE Almacena( c : CHAR ) ;
    PROCEDURE Lleno() : BOOLEAN ;
    PROCEDURE Remata() ;
END LibFrase.

IMPLEMENTATION MODULE LibFrase ;
    FROM InOut IMPORT WriteString ;
    VAR
        Contador : CARDINAL ;

    PROCEDURE Almacena( c : CHAR ) ;
        BEGIN
            Frase[ Contador ] := c ;
            INC( Contador ) ;
        END Almacena ;

    PROCEDURE Lleno() : BOOLEAN ;
        BEGIN
            RETURN  Contador = HIGH( Frase ) ;
        END Lleno ;

    PROCEDURE Remata() ;
        BEGIN
            Frase [ Contador]  := CHR(0) ;
        END Remata ;

BEGIN
    Contador := 0 ; (* Inicialización del módulo de implementación *)
END LibFrase .

MODULE LeeUnaFrase ;
    FROM InOut IMPORT Read, Write, WriteLine, WriteLn ;
    FROM LibFrase IMPORT Frase, Almacena, Lleno, Remata ;
    VAR Letra : CHAR ;

BEGIN
    LOOP
        Read( Letra ) ;
        Write( Letra ) ;

        IF ( Letra = '.' ) OR ( Lleno() ) THEN
            EXIT
        ELSE
            Almacena( Letra ) ;
        END ;
    END ;

    Remata ;
    WriteLn;
    WriteLine("Esta es la frase leída: ") ;
    WriteLine( Frase ) ;
END LeeUnaFrase .
```

8.3.6.- Módulos Locales

Un módulo local es un módulo definido dentro de otro módulo. Tiene las mismas características que el módulo principal, con dos diferencias:

- Debe terminar en punto y coma.
- Sólo puede importar y exportar identificadores conocidos desde y hacia el módulo en que está anidado.

Los únicos identificadores exteriores que conoce el módulo son los importados. Los únicos identificadores interiores que son conocidos fuera son los exportados.

La utilidad de los módulos locales es dar mayor versatilidad en las reglas de ámbito de las variables. Posibilitan:

- que el ámbito de una variable sea menor que el módulo principal.
- la síntesis de variables estáticas.

Ejemplo:

```
MODULE Externo ;
    FROM InOut IMPORT WriteString, WriteLn, ReadCard, WriteCard ;
    MODULE Interno ;
        (* No puede importar de InOut, sólo del externo *)
        IMPORT WriteString, WriteLn, WriteCard ;
        EXPORT Acumulador, Suma ;
        VAR Acumulador : CARDINAL ; (* Variable estática *)
        PROCEDURE Suma( Incremento : CARDINAL )  ;
        BEGIN
            INC( Acumulador, Incremento ) ;
            WriteLn ;
            WriteString( "La suma parcial es: ") ;
            WriteCard( Acumulador, 3 ) ;
        END Suma ;
   BEGIN
        (* Esta sentencia inicializa el módulo interno.
           Se ejecuta una sóla vez, al hacer la importación *)
        Acumulador := 1 ;
   END Interno ;   (* Este módulo termina en punto y coma *)

   VAR c : CARDINAL ; (* Variable global *)
BEGIN
    REPEAT
         WriteLn ;
         WriteString( " Deme un incremento ") ;
         ReadCard( c ) ;
         WriteLn ;
         Suma( c ) ; (* Al hacer esta llamada, Acumulador conserva su valor anterior *)
    UNTIL c = 0 ;
    WriteLn ;
    WriteString( "La suma final es: ") ;
    WriteCard( Acumulador, 5 ) ;
END Externo .
```

Cuando en el módulo local no hay ningún identificador cuyo nombre coincida con el de otro del módulo externo, no es necesario QUALIFIED en la sentencia de importación. En caso de existir esa coincidencia, sí es necesario.

Ejemplo:

```
(*
    Nombre:    Local1.Mod
    Propósito: mostrar la necesidad de cualificar la exportación cuando dos
               módulos, uno local al otro, tienen variables con el mismo nombre
    Autor:     José Garzía
*)
MODULE MExterno ;
    FROM InOut IMPORT WriteString, WriteLn,  WriteInt ;
    VAR Variable : INTEGER ; (* Variable global *)

    MODULE MInterno;
        EXPORT QUALIFIED Variable ;
        (* Esta variable interna tiene el mismo nombre que la variable global *)
        VAR Variable : INTEGER ;
    BEGIN
       Variable := 1 ;
    END MInterno ;
BEGIN
    Variable := 2 ; (* La global *)
    WriteLn ;
    WriteString("La variable del módulo interno es: ") ;
    WriteInt( MInterno.Variable, 2 ) ;
    WriteLn ;
    WriteLn ;
    WriteString("La variable del módulo externo es: ") ;
    WriteInt( Variable, 2 ) ;
    WriteLn ;
END MExterno.
```

Si se omite QUALIFIED en la sentencia de importación, se produce un error de compilación: Se intenta definir un identificador que ya existe.

Al incluir QUALIFIED, el compilador automáticamente cualifica los identificadores del módulo importado. La variable del módulo importado pasa a llamarse `MInterno.Variable`, y así debe referenciarla el programador.

Hay que enfatizar que los identificadores que cualifica son los del módulo importado. Los que pertenecen al módulo importador no los cualifica. Si por ejemplo, en el módulo externo el programador escribe `MExterno.Variable`, el compilador interpreta que se está intentando acceder al campo de un registro.

8.3.7.- Importaciones cualificadas

La forma usual de importación

```
FROM  < Nombre de la Biblioteca > IMPORT < Lista de identificadores importados > ;
```

junto con

```
EXPORT QUALIFIED < Lista de identificadores exportados > ;
```

se llama **importación-exportación cualificada**. El enlazador cualifica automáticamente cada identificador asociándole el nombre de la biblioteca de donde procede.

También es posible importar todos los identificadores de una biblioteca con una única sentencia:

```
IMPORT < Nombre de la Biblioteca > ;
```

Ejemplo:

```
IMPORT InOut ;
```

Esta se llama **importación no cualificada**. Si el programador elige esta forma, siempre que utilice un identificador importado, debe cualificarlo él mismo, con el nombre de la biblioteca y el operador punto:

```
InOut.WriteLn ;
```

Esta forma hace más "parlanchín" y quizá más ilegible el programa. Una posible ventaja es que se pueden rescribir los procedimientos de las bibliotecas del compilador, sin renombrarlos, manteniendo la versión antigua y la nueva; sin que surjan conflictos de nombre.

8.3.6.- Exportación transparente versus exportación opaca

Los tipos enumerados y los registros de biblioteca pueden definirse ora en el módulo de definición, ora en el módulo de implementación.

Ejemplo:

```
DEFINITION MODULE MiBiblio ;
   EXPORT QUALIFIED EnumeracionOpaca, EnumeracionTransparente, RegistroOpaco, RegistroTransparente ;
    TYPE
         EnumeracionOpaca, RegistroOpaco,
         EnumeracionTransparente = ( Transparente0, Transparente1, Transparente2 ) ;
         RegistroTransparente = RECORD
                                   LetraTransparente  : CHAR ;
                                   NumeroTransparente : INTEGER ;
                                END ;
END MiBiblio .

IMPLEMENTATION MODULE MiBiblio ;
    TYPE
         EnumeracionOpaca =  ( Opaco0, Opaco1, Opaco2 ) ;
         RegistroOpaco = RECORD
                           LetraOpaca  : CHAR ;
                           NumeroOpaco : INTEGER ;
                         END ;
END MiBiblio .
```

Exportación opaca.

Los tipos se definen en los módulos de implementación y se declaran en los módulos de definición. El módulo principal que importa la biblioteca tiene acceso a los tipos (puede declarar variables de esos tipos, hacer asignaciones entre ellas, pasarlas como parámetros a los procedimientos , etc.) pero no tiene acceso a los valores de la enumeración ni a los campos del registro.

Exportación transparente.

Los tipos se definen en los módulos de implementación. El módulo principal tiene acceso a los tipos, a los valores de las enumeraciones y a los campos de los registros.

En proyectos complejos, en los que participen varios grupos de programadores y tengan repartidos los módulos es preferible la exportación opaca. Así se impide que un programador interfiera inadvertidamente en el trabajo de otros grupos.

8.4.- TIPOS ABSTRACTOS DE DATOS

Tradicionalmente, un **tipo de datos** se entiende como una **clase de valores**. Por ejemplo:

Tipo	INTEGER	→	Valores de la clase Numero Entero
Tipo	CHAR	→	Valores de la clase Carácter

Pero esta interpretación liga al **tipo de datos** más a **su representación** que a **sus valores** posibles.

Por ejemplo, pensemos en la clase de valores *Días de la Semana*. Podemos representar estos valores de maneras diferentes:
- Con siete INTEGER, de 1 a 7.
- Con ristras de caracteres: `"Domingo"`, `"Lunes"`, `"Martes"`, etc.

Por supuesto, según cada representación, las operaciones posibles serían diferentes. No son iguales las operaciones con INTEGER que con ristras de caracteres.

Una estrategia más acertada es definir un tipo enumerado:

```
TYPE DiaSemana = ( Domingo, Lunes, Martes, Miercoles, Jueves, Viernes, Sabado ) ;
```

Las ventajas ya las conocemos:
- Claridad. El programa está autodocumentado.
- Comprobación automática de errores. Con las anteriores representaciones no serían detectados errores de estos:
  . valores inválidos como `37`, `"Dopingo"`,
  . operaciones como `3*7` ó `Delete( "Domingo", 2, 4 )`

Podemos ir un paso más allá en la abstracción de tipos de datos:
Tipo Abstracto de Datos. Es una asociación de:
- un conjunto de valores;
- y un conjunto de operaciones que trabajan con esos valores.

Si por ejemplo hacemos:
```
VAR Dia : DiaSemana ;
    .   .   .
Dia := Sabado ;
INC( Dia ) ;
```
se produce (y es detectado) un error en tiempo de ejecución, pues se sobrepasa el rango de valores de la enumeración.

Este error no se produce nunca si el programador define un conjunto de operaciones asociado al conjunto de valores.

Además, la mayor potencia de los tipos abstractos de datos se consigue cuando su implementación (El conjunto de valores y el conjunto de operaciones) está **encapsulada**, es decir, se define en una biblioteca y se hace una exportación opaca.

Desde el módulo principal no se pueden utilizar por separado valores del tipo enumerado, ni campos en caso de tratarse de un registro. Pero sí se puede:
- Declarar variables del tipo abstracto.
- Hacer asignaciones entre estas variables.
- Hacer comparaciones de igualdad entre estas variables.
- Pasarlas como parámetros a un procedimiento.
- Devolverlas desde un procedimiento.
- Operar con ellas utilizando sólo el conjunto de operaciones asociado.

Ejemplo:

```
DEFINITION MODULE LibDias ;
   EXPORT QUALIFIED DiaSemana, Avanza, Retrocede ;
   TYPE DiaSemana ;
   PROCEDURE Avanza(    VAR D : DiaSemana ; Avance    : INTEGER ) ;
   PROCEDURE Retrocede( VAR D : DiaSemana ; Retroceso : INTEGER ) ;
END LibDias .


IMPLEMENTATION MODULE LibDias ;
  TYPE DiaSemana = ( Domingo, Lunes, Martes, Miercoles, Jueves, Viernes, Sabado ) ;
     PROCEDURE Avanza(  VAR  D : DiaSemana ; Avance : INTEGER ) ;
           VAR Auxiliar : INTEGER ;
           BEGIN
                  Auxiliar := ORD( D ) ;
                  INC( Auxiliar, Avance ) ;
                  WHILE  Auxiliar > 7  DO
                            Auxiliar := Auxiliar - 7 ;
                  END ;
                  D := VAL( DiaSemana, Auxiliar ) ;
            END Avanza ;

     PROCEDURE Retrocede(  VAR  D : DiaSemana ; Retroceso : INTEGER ) ;
           VAR Auxiliar : INTEGER ;
           BEGIN
                  Auxiliar := ORD( D ) ;
                  DEC( Auxiliar, Retorceso ) ;
                  WHILE  Auxiliar < 1  DO
                            Auxiliar := Auxiliar + 7 ;
                  END ;
                  D := VAL( DiaSemana, Auxiliar ) ;
            END Avanza ;
END LibDias.
```

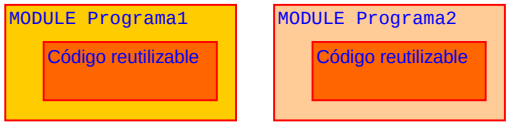
8.5.- DATOS ENCAPSULADOS

Cuando se sabe que sólo se va a utilizar un única variable de un tipo dado, puede declararse esta variable en el módulo de implementación, junto con un conjunto asociado de operaciones para trabajar con dicha variable. Estas operaciones son necesarias pues la variable no puede ser accedida desde fuera de dicho módulo. Ejemplo:

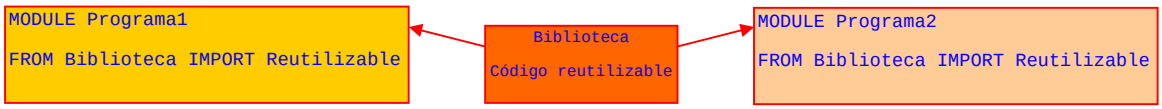
```
DEFINITION MODULE DatOcult ;
    EXPORT QUALIFIED Leer, Escribir ;
    PROCEDURE Leer ;
    PROCEDURE Escribir ;
END DatOcult .

IMPLEMENTATION MODULE DatOcult ;
   FROM InOut IMPORT ReadString, WriteString ;
   VAR  DatoOculto : ARRAY [ 0..29] OF CHAR ;
   PROCEDURE Leer ;
       ReadString( DatoOculto ) ;
   END Leer ;
   PROCEDURE Escribir ;
      WriteString( DatoOculto ) ;
   END Escribir ;
END DatoOculto .
```

8.6.- MÓDULOS Y ABSTRACCIONES SUCESIVAS

El perfilamiento mediante abstracciones sucesivas consiste en implementar una acción abstracta con un procedimiento parametrizado, o un dato abstracto con un registro. La finalidad es obtener código reutilizable. Si los programas constan de un único módulo, cada programa debe copiar el código reutilizable en su módulo.



Pero si el código reutilizable lo separamos en una biblioteca, no necesitamos copiarlo entero en cada uno de los demás programas que lo usen. Basta con una sentencia IMPORT en cada uno de ellos.



9 ENTRADA-SALIDA GENERALIZADA

9.1.- INTRODUCCIÓN

Las operaciones de entrada y salida son siempre un flujo de datos desde una fuente hasta un destino; a través de un canal.
Canal. Entidad cuya misión es permitir y controlar la recepción o emisión de una corriente de datos.

Según el sentido de la corriente, puede ser:
- Canal de entrada
- Canal de salida
- Canal de entrada y salida

Según su forma de acceso, puede ser:
- Secuencial. El flujo de datos está ordenado. Para emitir o recibir uno cualquiera, hay que hacerlo antes con todos los anteriores.
- Selectivo. Cualquier dato puede ser emitido o recibido sin tener que pasar por todos los anteriores.

Según su materialización, puede ser:
- Un dispositivo de E/S, (teclado, monitor, digitalizador, impresora etc.)
- Un área de memoria principal destinada a almacenamiento temporal
- Un área de memoria secundaria destinada a almacenamiento permanente. (**Fichero**)

9.2.- CANALES DE ACCESO SECUENCIAL. REDIRECCIÓN DE LA ENTRADA-SALIDA

Algunos de los procedimientos que hay en **InOut** y en **RealInOut** son de Entrada/Salida generalizada. Con esto queremos decir que son válidos para cualquier clase de canal secuencial, independientemente de cuál sea su materialización.
Para utilizar un procedimiento con un canal dado, previamente hay que indicar al procesador cuál es ese canal. Si no se indica nada, se hará con los canales por defecto: teclado para la entrada, y monitor para la salida.

La redirección de la Entrada/Salida se hace con estos recursos que hay en InOut:
OpenInput(< Extensión > **)** Al ejecutarse este procedimiento aparece la petición **in>** y el usuario debe escribir el nuevo canal al que se redirige la entrada.
OpenOutput(< Extensión >) Al ejecutarse este procedimiento aparece la petición **output>** y el usuario debe escribir el nuevo canal al que se redirige la salida.
< Extensión > es una ristra de caracteres que se añade con un punto al nombre proporcionado por el usuario. en caso de que éste no indique la extensión.
CloseInput. Devuelve al teclado la condición de canal de entrada.
CloseOutput. Devuelve al monitor la condición de canal de salida.
Done. Variable booleana que indica si la operación de redirección se hizo con éxito (TRUE) o no (FALSE).

Ejemplo:
Este programa sirve para copiar de un canal a otro.

```
MODULE RedirigeES ;
       FROM InOut IMPORT  WriteString,  OpenInput, CloseInput, Write,
                          OpenOutput, CloseOutput, Done, Read, WriteLine ;
     VAR
      Letra : CHAR ;
      BEGIN
        WriteLine("Está pidiendo el nombre del archivo de entrada" ) ;
        REPEAT
             OpenInput("") ;
       UNTIL Done ;
       WriteLine("Está pidiendo el nombre del archivo de salida" ) ;
       REPEAT
              OpenOutput("") ;
       UNTIL Done ;
        REPEAT
             Read( Letra ) ;
             Write( Letra ) ;
       UNTIL NOT Done ;
       CloseInput ;
       CloseOutput ;
      END RedirigeES.
```

Según sea el conjunto de datos proporcionados, así será su operación:
. Si *in> c:\autoexec.bat*
output> prueba
Crea el fichero *prueba*, y en un bucle lee caracteres uno por uno del fichero *autoexec.bat*; y los escribe en *prueba*

. Si *in> c:\ redirig.mod*
output> PRN
Lee los caracteres uno por uno del fichero *redirig.mod*; y los escribe uno por uno en el *fichero impresora* (PRN). Es decir, los saca por impresora. En la mayoría de versiones de MS-DOS y Modula-2 la impresora es tratada como un archivo secuencial cuyo nombre es PRN.

9.3.- CANALES DE ACCESO SELECTIVO. FICHEROS

La mayoría de aplicaciones necesitan la memoria secundaria, por dos razones:
1ª Los datos con que trabajan son permanentes, deben durar desde una sesión hasta la siguiente.
2ª La cantidad de datos puede ser tan grande que no sea posible tenerlos todos juntos en la memoria principal.

En cuanto a su contenido, los ficheros pueden ser:
Fichero de texto. Sólo contiene caracteres ASCII imprimibles.
Fichero binario. Contiene todo tipo de caracteres.

9.3.1.- Biblioteca FileSystem

Evidentemente, un canal de acceso selectivo es más rápido que uno de acceso secuencial. Los ficheros, además de ser accedidos secuencialmente, lo pueden ser selectivamente. Las herramientas dedicadas a operar de este último modo están en la biblioteca **FileSystem** (en algunas compiladores puede tener otro nombre).

9.3.1.1.- Definiciones de tipos

```
TYPE
   Response = ( done, notdone );
(* Los valores de esta enumeración indican los posibles resultados tras intentar realizar cada
operación *)

       IOMode   = ( read, write, io );

   File = RECORD
      id :INTEGER;
      res   :Response;
      eof   :BOOLEAN;(* Se utiliza para detectar si se alcanzó el fin del fichero, (carácter CHR(0))*)
      mode  :IOMode;
      fdptr :FDPtr;     (* Sólo para uso interno  *)
   END;
```

Siempre que se pretenda utilizar un fichero, previamente hay que declarar una variable de tipo **File**.
Después, en el intento de abrir o crear, se hace una asociación entre el fichero y esta variable. Los campos de esta variable registro contienen información acerca del estado del fichero asociado. Se actualizan tras cada llamada a un procedimiento de FileSystem.

Cursor de fichero. Es una variable interna que indica en qué posición del fichero se va a hacer la proxima operación de escritura o de lectura. Tras cada operación de éstas, avanza tantas posiciones como el trozo escrito o leído. El programador dispone de algunos procedimientos para modificarlo.
Si se intenta llevar más allá del final del archivo (señalado con el carácter nulo), se produce un error.
La posición del comienzo del fichero es 0.

9.3.1.2.- Procedimientos

```
PROCEDURE Lookup( VAR F :File; NombreFichero :ARRAY OF CHAR; Nuevo :BOOLEAN );
```
Abre el Fichero `NombreFichero`. En caso de que no exista:

Si `Nuevo = TRUE` lo crea

Si `Nuevo = FALSE` hace `F.res := notdone` ;

Siempre intenta primero abrir el fichero para lectura y escritura, si falla, lo intenta abrir para lectura.
El modo en que queda abierto se graba en `F.mode`

```
PROCEDURE Create( VAR F :File; NombreMedio :ARRAY OF CHAR );
```
Crea un nuevo fichero temporal. Si `NombreMedio` coincide con alguna variable de entorno, se asume que será la ruta donde será creado.

```
PROCEDURE Close( VAR F:File );
```
Cierra el fichero. Si fue renombrado, la entrada del directorio se modifica al mismo tiempo. Si el fichero es renombrado con una ruta en la que cambia la unidad de disco, es copiado.

Normalmente se utiliza un área temporal de transferencia en memoria principal, con el contenido de las modificaciones del archivo. En la operación de clausura se realizan las actualizaciones pendientes. Si se termina un programa sin cerrar un fichero, se pierde la información contenida en estas actualizaciones pendientes.

```
PROCEDURE Reset( VAR F :File );
```
Coloca el cursor de archivo al comienzo del fichero

```
PROCEDURE Rewrite( VAR F:File );
```
Borra el contenido del fichero y coloca el puntero de fichero al comienzo.

```
PROCEDURE Rename( VAR F :File; NomfreFichero :ARRAY OF CHAR );
```
Cambia el nombre del fichero al que se refiere `F` por `NombreFichero`. Puede cambiarse (moverse) a una unidad de disco diferente. La modificación no se actualiza hasta que el fichero es cerrado.

```
PROCEDURE ReadChar( VAR F :File; VAR Letra :CHAR );
```
Lee un carácter del fichero referido por `F`. El carácter `ASCII.CR` es convertido a `ASCII.EOL` y `ASCII.LF` es ignorado.

```
PROCEDURE WriteChar( VAR F:File; Letra :CHAR );
```
Escribe un carácter en el fichero referido por `F`. `ASCII.EOL` es convertido a la secuencia `ASCII.CR ASCII.LF`.

```
PROCEDURE GetPos( VAR F :File; VAR PalabraAlta, PalabraBaja :CARDINAL );
```
Indica la posición actual del puntero de fichero.
Los ficheros pueden tener más de 65535 bytes (máximo valor que cabe en un CARDINAL). Para contener el valor del cursor de fichero hay que utilizar dos palabras: `PalabraAlta` y `PalabraBaja`. Dicho valor se calcula con la fórmula
```
Posición = PalabraAlta*65536+PalabraBaja
```

```
PROCEDURE SetPos( VAR F :File; VAR PalabraAlta, PalabraBaja :CARDINAL );
```
Coloca el puntero de fichero en la posición `PalabraAlta*65536+PalabraBaja`.

```
PROCEDURE GetLPos( VAR F :File; VAR PalabraDoble :LONGCARD );
```
Devuelve la posición actual del puntero de fichero.

```
PROCEDURE SetLPos( VAR F :File; VAR PalabraDoble :LONGCARD );
```
Coloca el puntero de fichero en la posición que indica `PalabraDoble`.

```
PROCEDURE Length( VAR F:File; VAR PalabraAlta, PalabraBaja :CARDINAL );
```
Indica el tamaño del fichero.

```
PROCEDURE LLength( VAR F :File; VAR PalabraDoble :LONGCARD );
```
Indica el tamaño del fichero.

```
PROCEDURE ReadWord( VAR F :File; VAR Palabra :WORD );
```
Lee 16 bits (Esta cantidad puede depender del ordenador) del fichero referido por `F`.

```
PROCEDURE WriteWord( VAR F :File; Palabra :WORD );
```
Escribe 16 bits (Esta cantidad puede depender del ordenador) en el fichero referido por `F`.

```
PROCEDURE ReadNBytes( VAR F :File; DirAlmacen :ADDRESS; n :CARDINAL; VAR nLeidos :CARDINAL );
```
Intenta leer el número `n` de bytes . El número de bytes leídos correctamente lo devuelve en `nLeidos`. Los almacena a partir de la dirección `DirAlmacen`.

```
PROCEDURE WriteNBytes( VAR F :File; DirAlmacen :ADDRESS; n :CARDINAL; VAR nLeidos :CARDINAL );
```
Intenta escribir el número n de bytes que hay almacenados a partir de la dirección `DirAlmacen`. El número de bytes escritos correctamente lo devuelve en `nLeidos`

Los cuatro últimos procedimientos son sólo para ficheros binarios. `ReadChar` y `WriteChar` son sólo para ficheros de texto. Los demás procedimientos son para ficheros tanto de texto como binarios.

9.3.2.- Ejemplos:

Nueva versión del programa de copiar de un canal en otro. Utiliza procedimientos de acceso selectivo

```
MODULE AccesoDirecto ;
    FROM InOut IMPORT  ReadString, WriteString, Write, WriteLn,
                       Done, Read, WriteLine, EOL ;

    FROM FileSystem IMPORT File, Response, Lookup, Close, ReadChar, WriteChar ;

    VAR
        Letra : CHAR ;
        Fichero1, Fichero2 : File ;
        NombreFichero : ARRAY [0..30] OF CHAR ;
BEGIN
    REPEAT
        WriteLine("Deme el nombre del archivo de entrada" ) ;
        ReadString( NombreFichero ) ;
        WriteLn ;
        Lookup( Fichero1, NombreFichero, FALSE ) ;(* Debe ser uno existente *)
    UNTIL Fichero1.res = done ;

    REPEAT
        WriteLine("Deme el nombre del archivo de salida" ) ;
        ReadString( NombreFichero ) ;
        WriteLn ;
        Lookup( Fichero2, NombreFichero, TRUE ) ;(* Si no existe se crea *)
    UNTIL Fichero2.res = done ;

    REPEAT
        ReadChar( Fichero1, Letra ) ;
        IF Letra <> CHR(0) THEN WriteChar( Fichero2, Letra ) ; END ;
    UNTIL Fichero1.eof ;

    Close( Fichero1 ) ;
    Close( Fichero2 ) ;
END AccesoDirecto .
```

El siguiente programa lee un carácter del fichero y la posición especificados por el usuario. La salida del programa se produce cuando se indica una posición negativa.

```
MODULE EjemploSetPos ;
    FROM InOut IMPORT  ReadString, WriteString, Write, WriteLn,
                       ReadInt, WriteLine ;

    FROM FileSystem IMPORT File, Response, Lookup, Close, ReadChar,
                           SetPos ;

    VAR
        Letra : CHAR ;
        Fichero : File ;
        NombreFichero : ARRAY [0..30] OF CHAR ;
        Posicion : INTEGER ;
BEGIN
    REPEAT
        WriteString("Deme el nombre del fichero de entrada:     " ) ;
        ReadString( NombreFichero ) ;
        WriteLn ;
        Lookup( Fichero, NombreFichero, FALSE ) ;(* Debe ser uno existente *)
    UNTIL Fichero.res = done ;

    LOOP
        WriteLine("Deme la posición donde quiere leer el carácter." ) ;
        WriteLine("( Si quiere terminar dé un número negativo )" ) ;
        ReadInt( Posicion ) ;
        WriteLn ;
        IF  Posicion < 0 THEN EXIT END ;
        SetPos( Fichero, 0, Posicion ) ;
        IF Fichero.res <> done THEN
            WriteLine("Fuera de rango");
        ELSE
            ReadChar( Fichero, Letra ) ;
            Write( Letra ) ;
            WriteLn ;
       END ;

    END ;

    Close( Fichero ) ;
END EjemploSetPos.
```

No hay procedimientos para leer o escribir directamente números reales ni registros. Los números reales podemos:
Leerlos como ristras de caracteres y convertirlos a REAL
Convertirlos a ristras de caracteres y escribirlos carácter a carácter.
Las rutinas de conversión están en la biblioteca **RealConversions**

Pero es preferible usar los procedimientos de bajo nivel para ficheros binarios.
Ejemplo:

```
MODULE BinarioConReales ;
    FROM InOut IMPORT ReadString, WriteString, WriteLn ;
    FROM RealInOut IMPORT WriteReal ;
    FROM FileSystem IMPORT File, Response, Lookup, Close,
                           WriteNBytes, ReadNBytes, Reset ;

    FROM SYSTEM IMPORT TSIZE, ADR ;

    VAR
        Correcto : BOOLEAN ;
        Fichero : File ;
        NombreFichero : ARRAY[0..30] OF CHAR ;
        Escritos, Leidos : CARDINAL ;
        NumeroEscrito, NumeroLeido : REAL ;

BEGIN
    REPEAT
        WriteString("Deme el nombre del fichero:  ") ;
        ReadString( NombreFichero ) ;
        WriteLn ;
        Lookup( Fichero, NombreFichero, TRUE ) ; (* Si no existe, lo crea *)
    UNTIL Fichero.res = done ;

    NumeroEscrito := 3.1416 ;
    NumeroLeido := 1.0 ;

    (* Escribe el número real en el archivo *)
    WriteNBytes( Fichero, ADR( NumeroEscrito ), TSIZE( REAL ), Escritos ) ;
    IF Escritos <> TSIZE( REAL ) THEN WriteString("Error de escritura") ; END ;
    Close( Fichero ) ;

    REPEAT
        Lookup( Fichero, NombreFichero, FALSE ) ; (* Lo vuelve a abrir *)
    UNTIL Fichero.res = done ;

    ReadNBytes( Fichero, ADR( NumeroLeido ), TSIZE( REAL ), Leidos ) ;
    IF Leidos <> TSIZE( REAL ) THEN WriteString("Error de lectura") ; END ;
    Close( Fichero ) ;

    WriteLn ;
    WriteString("El número escrito es ") ;
    WriteReal( NumeroEscrito, 5 ) ;
    WriteLn ;
    WriteString("El número leido es ") ;
    WriteReal( NumeroLeido, 5 ) ;

END  BinarioConReales.
```

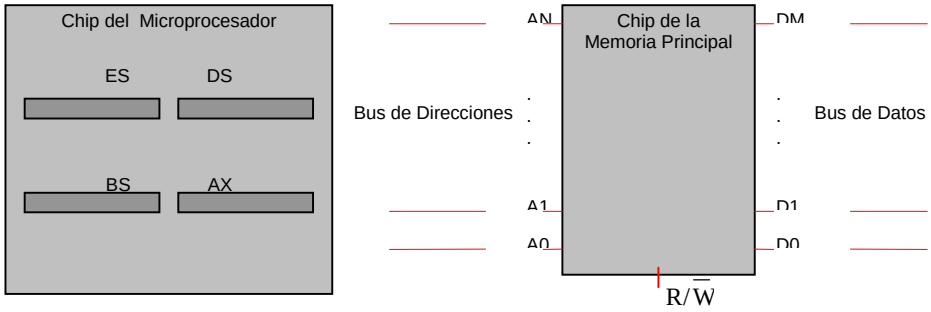
Este otro ejemplo escribe y lee registros en ficheros. La lectura la hace con acceso selectivo. Lee el segundo registro sin pasar por el primero.

```
MODULE BinarioConRegistros ;
    FROM InOut IMPORT ReadString, WriteString, WriteLine, WriteLn ;
    FROM RealInOut IMPORT WriteReal ;
    FROM FileSystem IMPORT File, Response, Lookup, Close, SetPos,  WriteNBytes, ReadNBytes, Reset ;

    FROM SYSTEM IMPORT TSIZE, ADR ;

    TYPE
        TRegistro = RECORD
                        Nombre : ARRAY [ 0..20] OF CHAR ;
                        Numero : REAL ;
                    END ;

    VAR
        IndiceBucle : CARDINAL ;
        Fichero : File ;
        NombreFichero : ARRAY[0..30] OF CHAR ;
        Escritos, Leidos : CARDINAL ;
        RegistrosEscritos : ARRAY [1..2] OF TRegistro ;
        RegistroLeido : TRegistro ;

BEGIN
    RegistrosEscritos[1].Nombre := "Nombre1" ;
    RegistrosEscritos[1].Numero := 0.1 ;
    RegistrosEscritos[2].Nombre := "Nombre2" ;
    RegistrosEscritos[2].Numero := 0.2 ;

    REPEAT
        WriteString("Deme el nombre del fichero:  ") ;
        ReadString( NombreFichero ) ;
        WriteLn ;
        Lookup( Fichero, NombreFichero, TRUE ) ; (* Si no existe, lo crea *)
    UNTIL Fichero.res = done ;

    (* Escribe los registros en el fichero *)
    FOR IndiceBucle := 1 TO 2 DO
       WriteNBytes( Fichero, ADR( RegistrosEscritos[IndiceBucle] ), TSIZE( TRegistro ), Escritos ) ;
       IF Escritos <> TSIZE( TRegistro ) THEN WriteString("Error de escritura") ; END ;
    END ;

    Close( Fichero ) ;

    REPEAT
        Lookup( Fichero, NombreFichero, FALSE ) ; (* Lo vuelve a abrir *)
    UNTIL Fichero.res = done ;

    (* Queremos leer el segundo registro, pasando por alto uno, el primero *)
    SetPos( Fichero, 0, 1 * TSIZE( TRegistro ) ) ;

    ReadNBytes( Fichero, ADR( RegistroLeido ), TSIZE( TRegistro ), Leidos ) ;
    IF Leidos <> TSIZE( TRegistro ) THEN WriteString("Error de lectura") ; END ;
    Close( Fichero ) ;

    WriteLn ;
    WriteLine("El registro leído es ") ;
    WriteString("Nombre:  ") ;
    WriteLine( RegistroLeido.Nombre ) ;
    WriteString("Número:  ") ;
    WriteReal( RegistroLeido.Numero, 8 ) ;

END  BinarioConRegistros.
```

10 GESTIÓN DINÁMICA DE LA MEMORIA PRINCIPAL

10.1.- MEMORIA PRINCIPAL

En un contexto de Electrónica Digital, materialmente, la memoria principal es un chip. En su interior hay una colección de biestables, agrupados en conjuntos de 8 (bytes). En su exterior hay varios conjuntos de patillas. Uno de éstos tiene N+1 patillas, es el A0 ... AN; y otro, con M+1 patillas es el D0 ... DM. Los números N y M dependen del modelo del microprocesador.



Con cada combinación diferente de valores en A0 ... AN se activa un byte diferente. El microprocesador especifica cuál es la combinación que debe haber en A0 ... AN grabándola previamente en alguno de sus registros internos. (Por ejemplo, en los microprocesadores de Intel suelen ser el ES y el DS). Desde los registros va hasta el chip de memoria a través de una serie de líneas llamada **bus de direcciones**.

Cada biestable del byte está conectado con una patilla del conjunto D0 ... DM. Y cada patilla de éstas está conectada a una línea de una serie de líneas llamada **bus de datos**.

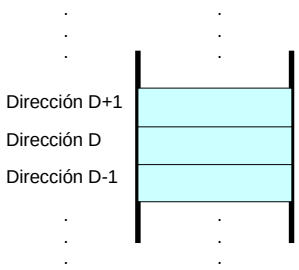
Las operaciones con la memoria son de dos clases:

Operaciones de Lectura. El contenido del byte activado es volcado (sin ser alterado) en el bus de datos.

Operaciones de Escritura. Los valores presentes en el bus de datos son grabados en los biestables del byte activado.

Para discernir el tipo de operación existe una patilla $R / \overline{W}$, la cual está a 1 cuando hay que leer, y se pone a 0 para escribir.

En un contexto de Programación, esquemáticamente, la memoria podemos imaginarla como una hilera de bytes. Cada byte está en una posición; le corresponde una dirección. Dicha dirección se especifica con la combinación de valores presentada en A0 ... AN.



10.2.- MODOS DE GESTIÓN DE LA MEMORIA

Todo programa necesita memoria para su ejecución. La necesita para almacenar temporalmente entre otras cosas, el código ejecutable del programa, las variables con las que trabaja, etc.

Hay dos formas de reservar memoria:

Estática. En tiempo de compilación se reserva la memoria necesaria sólo una vez, al declarar cada variable, simple o compuesta. Esta cantidad reservada es conocida y constante, antes de empezar la ejecución.

Dinámica En tiempo de ejecución, siempre que sea preciso, puede reservarse la cantidad necesaria de memoria. Esta cantidad puede ser diferente en cada ejecución de un mismo programa.

La gestión estática tiene como ventaja la simplicidad. Al declarar cada variable se hace automáticamente la reserva. Pero tiene una desventaja muy grande. Hay aplicaciones en las que a priori no se sabe cuánta memoria se va a necesitar.

Ejemplo:

Hemos visto un programa para leer caracteres desde el teclado. Eran almacenados en una lista de 80 elementos tipo CHAR que había sido declarada previamente. El número de caracteres tecleados podía ser:

1.- casualmente igual a 80. Esta es una situación tan afortunada como improbable.
2.- menor que 80. Se estaría desaprovechando memoria.
3.- mayor que 80. El programa no serviría.

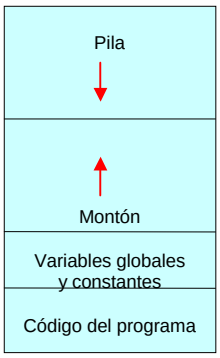
Una mala solución del programador sería aumentar el tamaño de la lista. Pero, ¿hasta cuánto de grande?. No es posible saberlo. Por muy grande que la declarara , siempre podría haber un usuario que tecleara más caracteres de los previstos por el programador.

Además, si la declarara muy grande, en algunos ordenadores que tuvieran poca memoria principal, ésta no tendría capacidad suficiente para almacenar la lista, y el programa fallaría antes de que el usuario empezara a introducir caracteres.

La solución más aceptable es hacer una reserva dinámica de memoria cada vez que se espera la introducción de un nuevo dato.

10.3.- ADMINISTRACIÓN DE LA MEMORIA PARA CADA APLICACIÓN

Cada vez que empieza a ejecutarse una aplicación, el sistema operativo le proporciona una zona de memoria. En los modelos más simples de memoria se la distribuye así:



Pila Es la porción que empieza en la parte superior de la zona asignada. Está destinada a almacenar, entre otras cosas, las direcciones donde se debe retornar al terminar un procedimiento. Según se va llenando, crece hacia las direcciones inferiores de memoria.

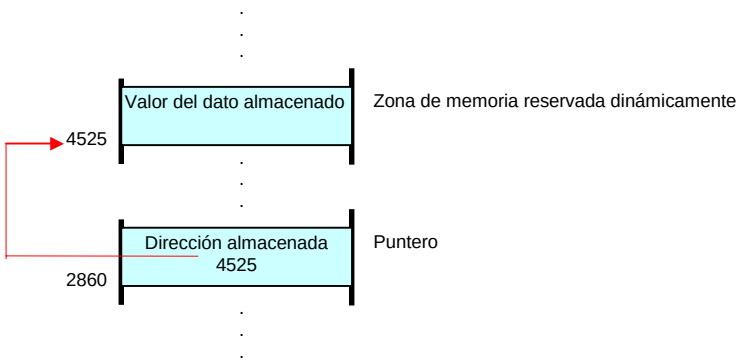
Montón. Es la porción utilizada para hacer las reservas dinámicas de memoria. Según se va llenando, crece hacia las direcciones superiores de memoria.

Dado que la pila crece hacia donde está el montón; y el montón crece hacia donde está la pila, puede suceder que una zona llegue hasta la otra. Se agotaría la memoria asignada por el sistema operativo. Esta situación es poco probable.

Las otras dos porciones tienen un tamaño constante durante la ejecución del programa.

10.4 MEDIOS Y PROCEDIMIENTOS PARA LA GESTIÓN DEL MONTÓN

Al reservar una zona de memoria, esta zona no tiene un nombre, como cualquier variable convencional. Para referirnos a ella, hemos de utilizar el valor de su dirección. O más cómodamente, almacenar esa dirección en otra variable, y utilizar el nombre de esta nueva variable.
Puntero. Variable que tiene como finalidad contener una dirección de memoria.



Los lenguajes de alto nivel suelen incluir procedimientos para:
- Comprobar si hay memoria disponible en el montón.
- Reservar dinámicamente una zona de memoria, y asignar la dirección donde comienza esa zona a un puntero. Se dice que *el puntero apunta a dicha zona*
- Procesar el valor del dato almacenado en la zona apuntada por el puntero.
- Encontrar la dirección donde está almacenada una variable convencional, y asignar dicha dirección a un puntero.
- Liberar la zona de memoria reservada dinámicamente cuando se sepa que el dato almacenado allí ya no se va a necesitar. Esto es para evitar el choque entre las zonas de pila y del montón.

Las tareas de reserva y liberación son responsabilidad del programador, pues es quien sabe cuándo se necesita una variable dinámica y cuándo no va a ser usada nunca más. En la pila sólo puede extraerse el último dato en ser introducido. De esto se encarga usualmente el compilador.

10.5.- PUNTEROS EN MODULA-2

10.5.1.- Definición y declaración

Puede definirse un nuevo tipo de datos que serán punteros, y declarar una variable de ese nuevo tipo:

```
TYPE  < TipoPuntero > = POINTER TO  < Tipo de los Datos > ;
VAR   < Nombre del Puntero > : < TipoPuntero > ;
```

O declararse directamente una variable puntero destinada a apuntar datos de un tipo dado:

```
VAR   < Nombre del Puntero >  :  POINTER  TO  < Tipo de los Datos > ;
```

10.5.2.- Comprobación de si hay disponible una determinada cantidad de memoria.

No es imprescindible, pero sí conveniente.

```
Available(  < Nimero de Bytes necesitados >  ) ;
```

10.5.3.- Reserva y liberación dinámicas de memoria

```
NEW(  < Nombre del Puntero >  ) ;
DISPOSE  < Nombre del Puntero >  ) ;
```

Estos procedimientos son parte del lenguaje. No es necesario importarlos. Pero utilizan los procedimientos de biblioteca ALLOCATE y DEALLOCATE, respectivamente, los cuales sí hay que importar desde el módulo **Storage.**

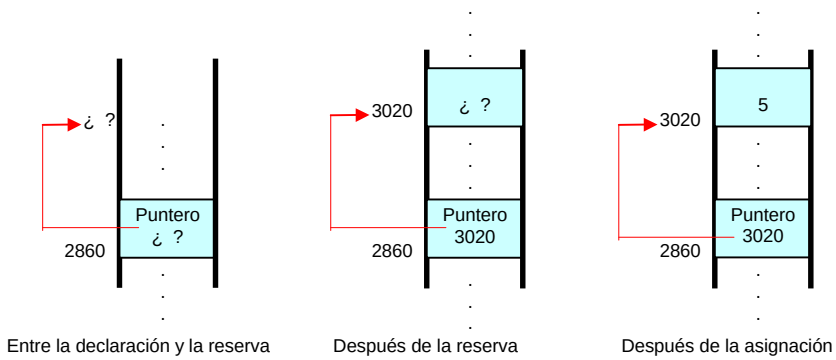
10.5.4.- Acceso al dato almacenado en la zona de memoria apuntada por el puntero

Se realiza con el **operador flecha:** ^

Ejemplo:

```
VAR Puntero : POINTER TO INTEGER ;

NEW( PUNTERO ) ;
Puntero^ := 5 ;
```

Fases por las que pasa la memoria:



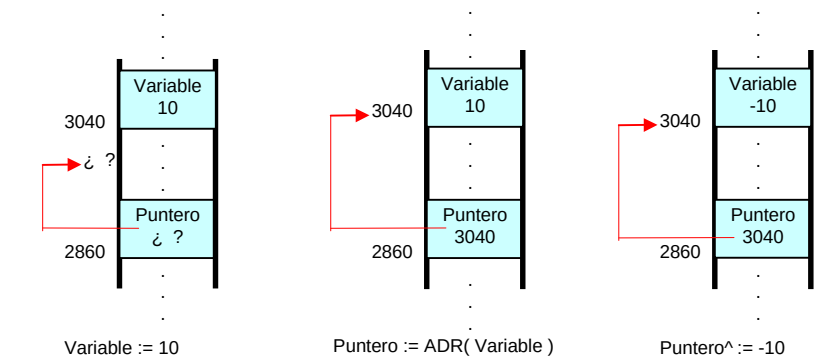
Entre la declaración y la reserva | Después de la reserva | Después de la asignación

10.5.5.- Encontrar la dirección donde está almacenada una variable convencional.

Esta búsqueda se hace con el procedimiento ADR. Su argumento es el nombre de la variable. Si a un puntero se le va a asignar la dirección de una variable, no es necesario reservar memoria para él.
Ejemplo:

```
VAR
        Variable : INTEGER ;
        Puntero : POINTER TO INTEGER ;

        Variable := 10 ;
        Puntero := ADR( Variable ) ;
        Puntero^ :=  -10 ;  (* Variamos la variable mediante el puntero *)
```

Fases por las que pasa la memoria:



Variable := 10 | Puntero := ADR(Variable) | Puntero^ := -10

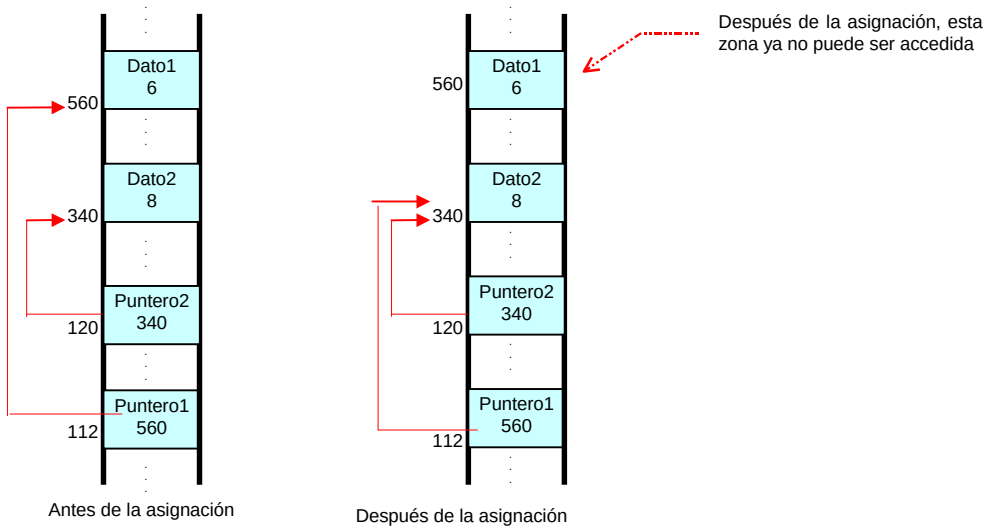
10.5.6.- Asignaciones de punteros

Los punteros deben ser del mismo tipo. Después de la asignación, ambos apuntan a la zona donde estaba apuntando el segundo operando

Ejemplo:

```
MODULE EjemploAsignacionPunteros ;
    FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available ;
    FROM SYSTEM IMPORT TSIZE, ADR ;

    VAR
        Puntero1, Puntero2 : POINTER TO INTEGER ;
BEGIN
    IF Available( 2 * TSIZE( INTEGER ) ) THEN
        NEW( Puntero1 ) ; (* Se obtiene memoria *)
        NEW( Puntero2 ) ;
    END ;
    Puntero1^ := 6 ;
    Puntero2^ := 8 ;

    Puntero1 := Puntero2 ;
    DISPOSE( Puntero1 ) ;
END EjemploAsignacionPunteros.
```

Estado de la memoria antes y después de la asignación `Puntero1 := Puntero2;`



10.5.7.- Precauciones que hay que tener con los punteros

- Si un puntero contiene la dirección de una variable convencional, ésta no puede liberarse.
- Si un puntero contiene la dirección de una variable convencional, podemos estar variando ésta sin darnos cuenta.
- Una misma zona no puede ser liberada más de una vez. En el anterior programa hay dos punteros (`Puntero1` y `Puntero2`) que apuntan a la misma zona. Sólo se puede liberar uno cualquiera de los dos. No se puede hacer:

```
DISPOSE( Puntero1 ) ;
DISPOSE( Puntero2 ) ;   (* Erróneo *)
```

Una vez liberada la zona destruyendo uno de los dos punteros, no es posible acceder a ella con el otro.

- Al hacer una asignación entre punteros, puede quedar inaccesible una zona con algún dato significativo, como se ha visto en el anterior ejemplo. Una solución podría ser almacenar la dirección de esta zona en otro tercer puntero antes de hacer la asignación.
- No se debe utilizar el operador flecha sin antes haber inicializado el puntero con NEW o con una asignación de una dirección válida.

Ejemplo :

```
VAR Puntero : POINTER TO CHAR ;
BEGIN
  Puntero^ := 's' ;  (* ¿En qué zona de memoria se está almacenando este carácter? *)
```

Podría estar apuntando a alguna una zona en la que hubiera otro dato de interés, alguna variable, constante, código del programa etc.; e incluso el sistema operativo, alterando el contenido de esa zona.
De suceder esto, no sólo no funcionaría el programa, sino que haría "cascar" todo lo que se estuviera ejecutando.

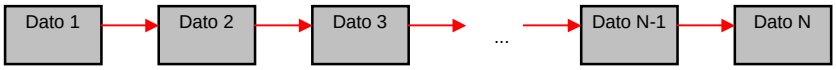
Para indicar que la zona apuntada por un puntero no se debe utilizar, se le puede dar el valor NIL.
El valor exacto de esta constante depende de cada compilador y de cada ordenador. Pero siempre simboliza que el puntero apunta a algún valor no conocido de la memoria principal.

11 CONSTRUCCIONES DINÁMICAS DE DATOS

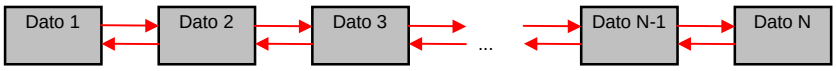
11.1.- LISTAS ENLAZADAS DE DATOS

El uso de punteros aislados no aporta muchas ventajas a la hora de obtener memoria dinámica, pues no hay forma de hacer un tratamiento conjunto de todos los punteros declarados. Para tener la posibilidad de este tratamiento conjunto sería conveniente que una colección de datos almacenados en la memoria dinámica tuvieran entre sí alguna forma de enlace. Así, las mismas ventajas que tenían las formaciones respecto de las variables simples, las tendrían estas colecciones de datos enlazados respecto de los punteros simples.

Lista simplemente encadenada. Es una secuencia de datos con significado propio, junto a cada uno de los cuales hay un enlace con el siguiente, excepto el último dato, que no tiene siguiente.



Lista doblemente enlazada. Es una secuencia de datos con significado propio, junto a cada uno de los cuales hay dos enlaces; uno con el siguiente y otro con el anterior, excepto el último dato, que no tiene siguiente; y el primero, que no tiene anterior.



A primera vista puede parecer que una forma natural de implementar estas listas podría ser con registros en los que un campo fuera el dato significativo, y en otro campo estuviera el siguiente registro. (con otro campo para el anterior, si es doblemente enlazada). Algo así:

REGISTRO	
Campo 1º: Dato significativo	Campo 2º: Un registro de este mismo tipo

Pero si lo pensamos detenidamente, es lógico que un registro no se pueda contener a sí mismo. Una parte no puede materialmente contener al todo. La solución es que el enlace sea un puntero para contener la dirección del siguiente elemento. (y otro puntero para el anterior, si es doblemente enlazada).

REGISTRO	
Campo 1º: Dato significativo	Campo 2º:Un puntero a registros de este mismo tipo

11.2.- LISTAS CON ENLACE SENCILLO EN MODULA-2

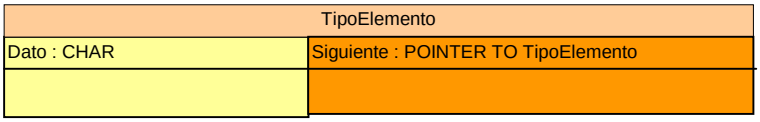
11.2.1.- Implementación

Los elementos de la lista son registros de un tipo dado. Uno de los campos es un puntero a ese tipo de registro. Se presenta el dilema de saber qué hay que definir primero:

- El puntero. En este caso apuntará a un tipo de datos aún no definido.
- El registro. En este caso uno de los campos será un objeto aún no definido.

Es como la pregunta " ¿Qué fue primero, el huevo, o la gallina?".
En esta situación, Modula-2 permite la definición de un puntero a un tipo de datos aún no definido.

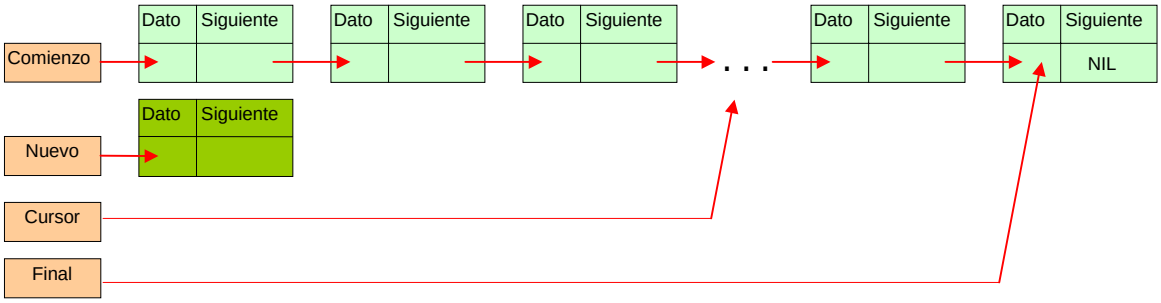
```
TYPE
    Apuntador = POINTER TO TipoElemento ;
    TipoElemento = RECORD
                     Dato : CHAR ;
                     Siguiente : Apuntador ;
                     END ;
```



Además de los elementos que forman la lista, es conveniente tener una serie de objetos auxiliares y un conjunto de operaciones.

11.2.2.- Objetos Auxiliares

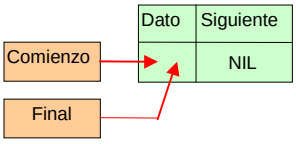
Comienzo. Puntero que tiene la dirección del primer registro
Final. Puntero que contiene la dirección del último registro
Cursor. Puntero que sirve para apuntar a cualquier elemento al recorrer la lista
Nuevo. Puntero que sirve para apuntar a un nuevo registro



11.2.3.- Operaciones

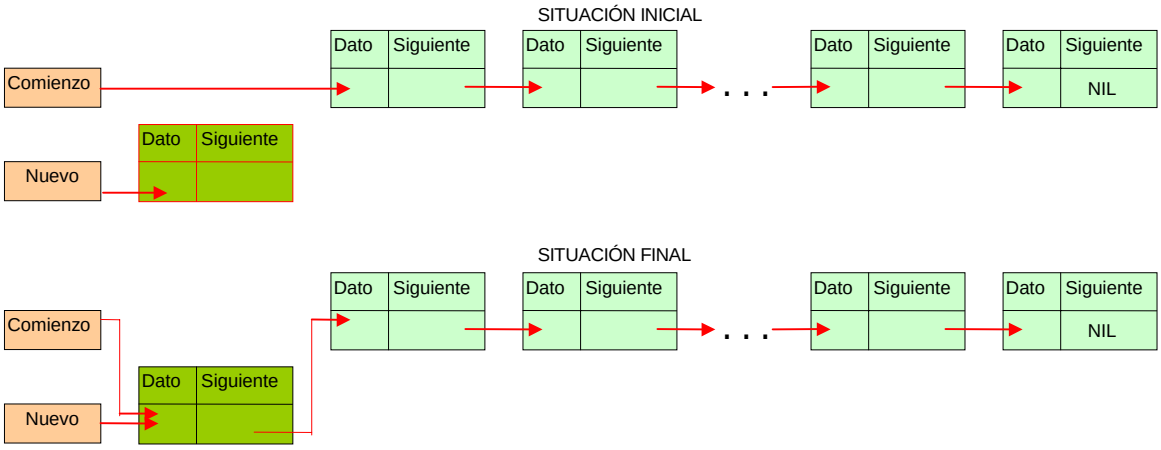
11.2.3.1.- Creación de la lista

```
NEW( Comienzo ) ;
Comienzo^.Dato  :=  < Primer dato obtenido >  ;
Comienzo^.Siguiente := NIL ;
Final := Comienzo  ;
```



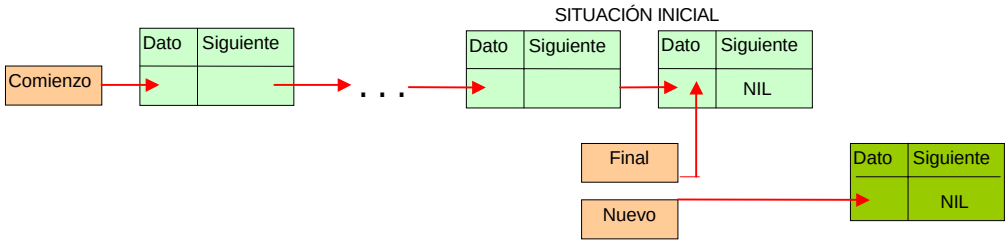
11.2.3.2.- Inserción de un nuevo elemento al comienzo de la lista

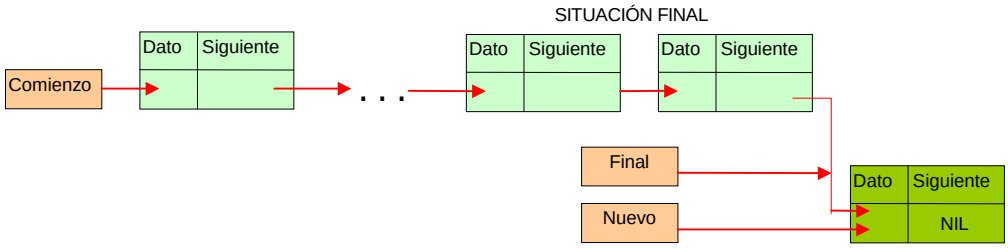
```
NEW( Nuevo ) ;
Nuevo^.Dato  :=  < Nuevo dato obtenido  > ;
Nuevo^.Siguiente := Comienzo ;
Comienzo := Nuevo
```



11.2.3.3.- Inserción de un nuevo elemento al final de la lista

```
NEW( Nuevo ) ;
Nuevo^.Dato  :=  < Nuevo dato obtenido  > ;
Nuevo^.Siguiente := NIL ;
Final^.Siguiente :=  Nuevo ;
Final := Nuevo ;
```



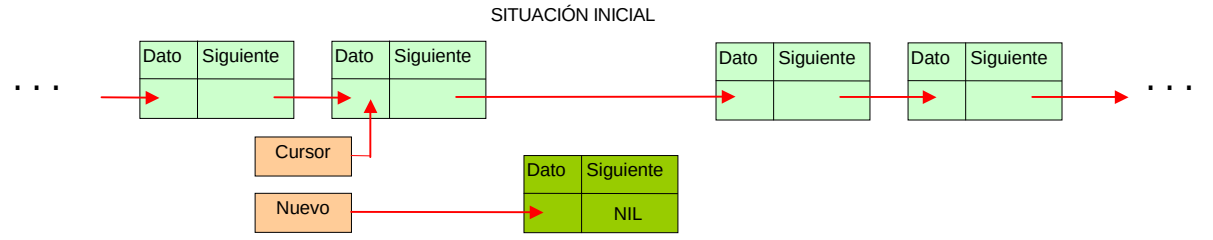


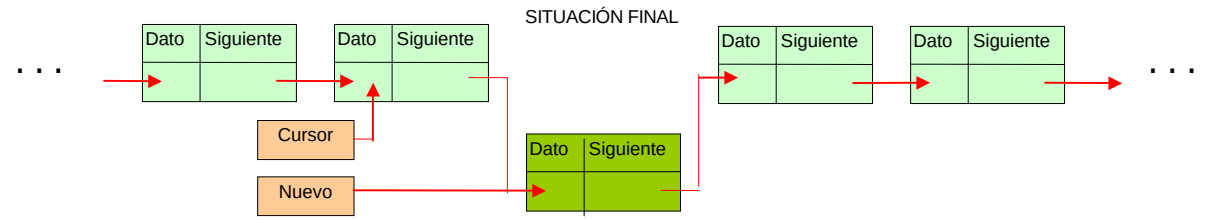
11.2.3.4.- Inserción de un nuevo elemento en el interior de la lista

El nuevo elemento se inserta tras el apuntado por el cursor.

```
Nuevo^.Siguiente  := Cursor^.Siguiente ;
Cursor^.Siguiente := Nuevo ;
```

El cursor no ha variado. Sigue apuntando donde antes.



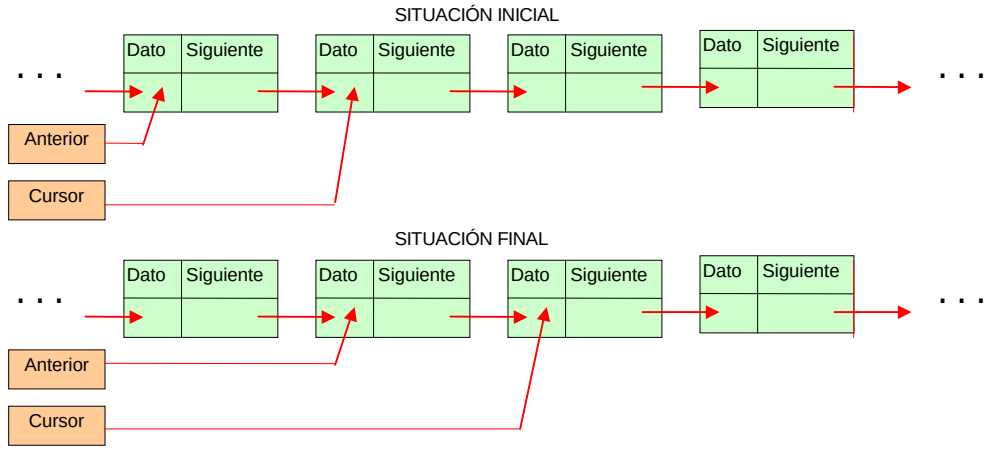


11.2.3.5.- Recorrido y Búsqueda de un Elemento

```
Cursor := Comienzo ;
WHILE  ( Cursor <> NIL ) AND ( NOT < Condición de búsqueda  > ) DO
     Cursor := Cursor^.Siguiente ;
END ;
```

Al salir del bucle el cursor apunta al primer elemento que cumple la condición de búsqueda. Pero en algunas operaciones se necesita también el inmediatamente anterior. Estas listas sólo se pueden recorrer en un sentido. Por tanto, para tener disponible el anterior, debemos tenerlo en otro registro:

```
Anterior := NIL ;
Cursor := Comienzo ;
WHILE  ( Cursor <> NIL ) AND ( NOT  < Condición de búsqueda  > ) DO
     Anterior := Cursor ;
     Cursor := Cursor^.Siguiente ;
END ;
```



Ejemplos:

```
(*
   ╔═══════════════════════════════════════════════════════════════════════╗
   ║ Nombre:          Leetext0.Mod                                         ║
   ║ Propósito:       Lee un párrafo de texto de cualquier longitud,       ║
   ║                  terminado en un punto.                               ║
   ║ Características: Utiliza una lista de datos con enlace simple        ║
   ║ Autor:           José Garzía                                          ║
   ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE LeeParrafoTexto ;
    FROM InOut IMPORT Read, WriteString, WriteLine, WriteLn, Write, WriteInt ;
    FROM Storage IMPORT ALLOCATE, DEALLOCATE ;

    TYPE TPuntero = POINTER TO TCaracterEnlazado ;
    TYPE TCaracterEnlazado = RECORD
                                 Dato      : CHAR     ;
                                 Siguiente : TPuntero ;
                             END ;

    VAR
        Contador : INTEGER ;
        Comienzo ,
        Final ,
        Cursor,
        Anterior,
        Nuevo : TPuntero ;

    BEGIN
        (* Creación de la lista enlazada de caracteres *)
        NEW( Comienzo ) ;
        Read( Comienzo^.Dato ) ;
        Write( Comienzo^.Dato ) ;
        Comienzo^.Siguiente := NIL ;
        Final := Comienzo ;
        Nuevo := Comienzo ;

        (* Los caracteres leídos se van añadiendo al final de la lista *)
        WHILE ( Nuevo^.Dato <> '.' ) DO
            NEW( Nuevo ) ;
            Read( Nuevo^.Dato ) ;
            Write( Nuevo^.Dato ) ;
            Nuevo^.Siguiente := NIL ;
            Final^.Siguiente := Nuevo ;
            Final := Nuevo ;
            INC( Contador ) ;
        END ;
        Final^.Siguiente := NIL ;
        WriteLn ;

        WriteString("Número de caracteres: ") ;
        WriteInt( Contador, 2) ;
        WriteLn ;

        (* Los caracteres leídos se van escribiendo *)
        WriteLine("Este es el párrafo tecleado:") ;
        Cursor := Comienzo ;
        WHILE ( Cursor <> NIL ) DO
            Write( Cursor^.Dato ) ;
            Cursor := Cursor^.Siguiente ;
        END ;
        WriteLn ;

END LeeParrafoTexto .
```

```
(*
   ╔═══════════════════════════════════════════════════════════════════════╗
   ║ Nombre:          ListaOrd.Mod                                         ║
   ║ Propósito:       Lee una lista de números, terminada en uno negativo. ║
   ║                  La presenta ordenada de menor a mayor.               ║
   ║ Características: Utiliza una lista con enlace simple, y la inserción ║
   ║                  se hace en el lugar necesario para que la lista quede║
   ║                  ordenada                                             ║
   ║ Autor:           José Garzía                                          ║
   ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE ListaOrdenada ;
    FROM InOut IMPORT ReadInt, WriteInt, WriteLine, WriteLn, Write ;
    FROM Storage IMPORT ALLOCATE, DEALLOCATE ;

    TYPE TPuntero = POINTER TO TElementoLista ;
    TYPE TElementoLista = RECORD
                              Dato : INTEGER ;
                              Siguiente : TPuntero ;
                          END ;

    VAR
        Comienzo,
        Cursor,
        Anterior,
        Nuevo,
        Final       : TPuntero ;

    BEGIN
        WriteLine("Introduzca números enteros.") ;
        WriteLine("Cuando quiera terminar introduzca uno negativo ") ;
        NEW( Comienzo ) ;
        ReadInt( Comienzo^.Dato ) ;
        Write(' ') ;
        Comienzo^.Siguiente := NIL ;
        Nuevo       := Comienzo ;

        WHILE ( Nuevo^.Dato >= 0 ) DO
            NEW( Nuevo ) ;
            ReadInt( Nuevo^.Dato ) ;
            Write(' ') ;

            (* Recorrido con búsqueda en la lista. Si la condición de búsqueda
               fuera (Cursor^.Dato >= Nuevo^.Dato)
               la lista quedaría ordenada en orden decreciente
            *)

            Cursor := Comienzo ;
            Anterior := NIL ;
            WHILE  ( Cursor <> NIL ) AND (Cursor^.Dato <= Nuevo^.Dato) DO
                 Anterior := Cursor ;
                 Cursor := Cursor^.Siguiente ;
            END ;
            Cursor := Anterior ; (* La inserción se hará tras este elemento *)

           IF Cursor <> NIL THEN (* Inserta detrás del cursor *)
                Nuevo^.Siguiente := Cursor^.Siguiente ;
                Cursor^.Siguiente := Nuevo ;
           ELSE                  (* Inserta al comienzo *)
                Nuevo^.Siguiente := Comienzo ;
                Comienzo := Nuevo ;
           END ;
        END ;

        WriteLn ;
        WriteLine("Estos son los números tecleados, ordenados:") ;
        (* Recorrido sin búsqueda *)
        Cursor := Comienzo^.Siguiente ; (* El primero no lo imprimimos *)
        WHILE Cursor <> NIL DO
            WriteLn ;
            WriteInt( Cursor^.Dato, 10 ) ;
            Cursor := Cursor^.Siguiente ;
        END ;
END ListaOrdenada.
```

12 MENÚS EN MODULA-2

12.1 MENÚS

Los datos de entrada siempre pueden ser introducidos usando las funciones de lectura desde el teclado de la biblioteca InOut. Cuando el número de valores posibles es finito, es posible elaborar un **menú** con todos y cada uno de los valores posibles, para que el usuario del programa opte por alguno de ellos.

Una sencilla estrategia para construir un menú es presentar una lista de opciones numeradas; y en una línea aparte, mediante una operación de lectura de números CARDINAL o INTEGER, leer el número asociado a la opción elegida. Esta estrategia es tan rudimentaria que no la consideraremos en mayor profundidad.

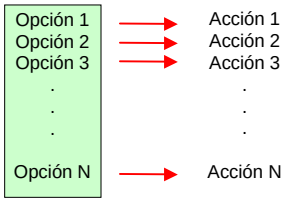
Los menús más elaborados son los que dividen la pantalla en regiones. A cada región le corresponde una opción; y cada opción esta asociada con una acción (que puede ser la ejecución de un menú anidado o **submenú**). El cursor puede viajar por las diferentes regiones, usando las teclas (normalmente de flechas) o el ratón. Al pulsar INTRO, se ejecuta la acción asociada a la región donde estaba el cursor.

Básicamente hay dos tipos de menús:

- Menú flotante. El programador puede hacer que aparezca en cualquier parte de la pantalla.
- Barra de menú con submenús. De una barra horizontal “cuelgan” submenús y o acciones directas.

El entorno de programación en Modula-2 de Roger Carvalho incluye dos procedimientos orientados a menús, uno para menús flotantes y otro para barras de menús con submenús.

12.2.- MENÚ FLOTANTE



12.2.1.- Descripción del procedimiento

```
PROCEDURE     PopMenu( NumeroLinea, NumeroColumna  :  CARDINAL ; Menu : ARRAY OF CHAR ;
                       Anchura  :  CARDINAL ;  Borrado  BOOLEAN ;  VAR  Elección  :  CARDINAL  ) ;
```

Localización : Módulo *Menu*

Significado de los argumentos :

NumeroLinea y *NumeroColumna* especifican las coordenadas de la esquina superior izquierda de la ventana.
Menu es una ristra de caracteres, que encadena estas subrristras:

Título del menú
Opción 1
.
.
.
Opción N

Las separa con el símbolo ' | ' así: "Título del menú|Opción 1|Opción 2| . . . |Opción N"

Anchura es la anchura mínima de la ventana

Borrado =
TRUE si queremos que la ventana sea borrada al salir del procedimiento.
FALSE si no queremos que sea borrada. De todas formas, quedará inactiva.

Elección sirve para conocer la opción elegida, una vez que se haya salido del procedimiento.

12.2.2.- Modo de operación

. Al hacer la llamada, aparece el menú.
. Se pueden recorrer las diferentes opciones con las teclas de flechas
. Se sale del procedimiento con ESC; o pulsando INTRO sobre alguna opción.
. Después de salir, la variable *Eleccion* contiene:

0 si se salió con ESC

k si se eligió la Opción k (1 ≤ k ≤ N)

Para mantener el menú, la llamada al procedimiento debe estar dentro de un bucle. De esta forma es posible estar eligiendo sucesivas opciones (y realizando sucesivas acciones) y al mismo tiempo tener siempre a la vista el menú.

Tras pulsar INTRO, se sale de una llamada, se ejecuta la acción que corresponda, y se hace la siguiente llamada

Tras pulsar ESC, se sale del bucle.

12.2.3.- Esquemas de programación que mejor se adecuan a la implementación de un menú permanente

.- De ramificación Pueden servir todos (IF anidados, ELSIF y CASE). Por su simplicidad, la más recomendable es el CASE : una rama para cada opción

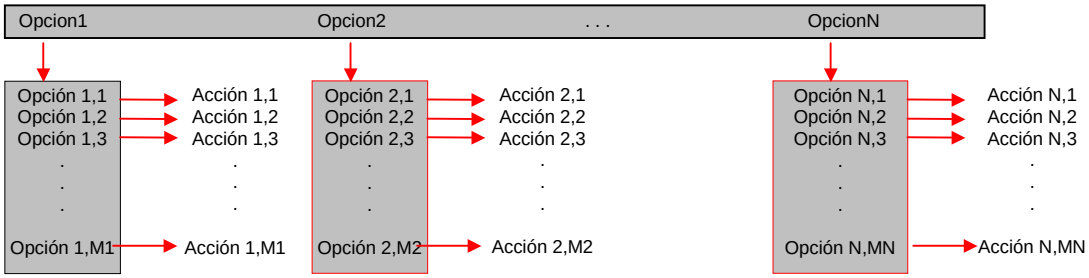
.- De iteración En principio, todos los bucles pueden servir (WHILE, REPEAT, LOOP incluso FOR). La comprobación de salida (pulsación de ESC) puede hacerse tanto al principio como al final. Pero posiblemente se necesite incluir una opción adicional de salida entre las opciones del menú. Sólo el bucle LOOP permite tener varios puntos de salida.

```
  LOOP
           PopMenu( NLinea, NColumna,  "Menú Principal | Opción 1 | Opción 2 | . . . | Opción N ",
                    Anchura, TRUE, Eleccion ) ;

           CASE Eleccion OF
               0 :  EXIT        ;  |
               1 : < Acción 1 > ;  |
               2 : < Acción 2 > ;  |
                 . . .
               N : < Acción N > ;
   END ;
```

Ejemplo :

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre: PopMenus.Mod                                               ║
  ║    Autor:  José Garzía                                                ║
  ║    Propósito: Mostrar el funcionamiento del procedimiento PopMenu     ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)
MODULE PopMenus ;
FROM Menu IMPORT PopMenu ;
FROM InOut IMPORT Read, WriteInt, WriteString ;
FROM Display IMPORT ClrEOL, SetCursorPosition, SetDisplayMode ;

    VAR
        Eleccion : CARDINAL ;
        c : CHAR ;
    BEGIN
        SetDisplayMode(2) ;
        LOOP
            PopMenu( 0,0,  "Menú Principal|¡ Hola !|¡ Salud !|Salir",20, TRUE, Eleccion ) ;
            SetCursorPosition(20, 2) ;
            WriteString("has elegido ") ;

            CASE Eleccion OF
                0 : ClrEOL ;
                    SetDisplayMode( 2 ) ;
                    WriteString('Has pulsado ESC ') ;
                    Read( c ) ; (* Espera la pulsación de una tecla cualquiera *)
                    EXIT ; |

                1 : ClrEOL ;
                    WriteString('la opción "¡ Hola !" ') ; |

                2 : ClrEOL ;
                    WriteString('la opción "¡ Salud !" ') ; |

                3 : ClrEOL ;
                    WriteString('la opción "¡ Salir !" ') ;
                    Read( c ) ; (* Espera la pulsación de una tecla cualquiera *)
                    EXIT ;
            END ;
        END ;
        SetDisplayMode(2) ;
    END PopMenus.
```

12.3.- BARRA DE MENÚ CON SUBMENÚS



12.3.1.- Descripción del procedimiento

```
PROCEDURE     PullDownMenu(   NumeroLinea                 :      CARDINAL       ;
                              Borde                       :      BOOLEAN        ;
                              MenuPrincipal               :      ARRAY OF CHAR  ;
                              Submenus                    :      ARRAY OF CHAR  ;
                              Borrado                     :      BOOLEAN        ;
                              VAR  Elección1, Eleccion2   :      CARDINAL

                            ) ;
```

Localización : Módulo *Menu*

Significado de los argumentos :

NumeroLinea especifica la línea donde estará situada la barra.

Borde especifica si la barra se dibujará con o sin borde.

MenuPrincipal es una ristra de caracteres que encadena las subrristras de cada opción de la barra, separadas por el símbolo '|'.

Submenus es una ristra de caracteres que encadena los submenús, separados por el símbolo '&'.
A su vez, los submenús son ristras de caracteres que encadenan las subrristras de cada opción del submenú, separadas por el símbolo '|'.

Borrado =
TRUE si queremos que el menú sea borrado al salir del procedimiento.
FALSE si no queremos que sea borrado. De todas formas, quedará inactivo.

Eleccion1. Su valor al salir del procedimiento es:
0 si se salió mediante la pulsación de la tecla ESC.
1 si se pulsó INTRO habiendo elegido la opción **i** de la barra. (1 ≤ i ≤ N)

Eleccion2. Su valor al salir del procedimiento es:
j si se pulsó INTRO sobre la opción **i, j** del submenú **i** (1≤ i ≤ N , 1≤ j ≤Ni)

La metodología de programación recomendada es la misma que con el procedimiento *PopMenu*. Pero como ahora hay dos variables indicadoras de la elección, se necesitan dos esquemas CASE anidados.

```
LOOP
          PullDownMenu( NLinea,
                                Borde,
                                "Opción 1 | Opción 2 | . . . | Opción N ",

                               " Opción 1,1  | Opción 1,2 | . . . | Opción 1,M1  &
                                 Opción 2,1  | Opción 2,2 | . . . | Opción 2,M2  &
                                                                   . . .
                                 Opción N,1  | Opción N,2 | . . . | Opción N,MN
                               " ,
                               Borrado,
                               Eleccion1,
                               Eleccion2,
                             ) ;

        CASE Eleccion1 OF

             0 : EXIT ; |

             1 : CASE Eleccion2 OF

                     1   : < Acción 1, 1    > ; |
                     2   : < Acción 1, 2    > ; |
                                . . .
                     N1  : < Acción 1, N1   > ; |
                 END ;

             2 : CASE Eleccion2 OF

                     1   : < Acción 2, 1    > ; |
                     2   : < Acción 2, 2    > ; |
                                . . .
                     N2  : < Acción 1, N2   > ; |
                 END ;

                           . . .

             N : CASE Eleccion2 OF

                     1    : < Acción N, 1    > ; |
                     2    : < Acción N, 2    > ; |
                                . . .
                     NN   : < Acción N, NN   > ; |

                 END ;  (* del CASE interno *)
        END ;   (* del CASE externo *)
    END ;   (* del LOOP *)
```

Ejemplo:

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    BarrMenu.Mod                                            ║
  ║    Autor:     José Garzía                                             ║
  ║    Propósito: Mostrar el funcionamiento del procedimiento PullDownMenu ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE BarraDeMenu ;
FROM Menu IMPORT PullDownMenu ;
FROM InOut IMPORT Read, WriteInt, WriteString, WriteLine, WriteLn ;
FROM Display IMPORT ClrEOL, SetCursorPosition, SetDisplayMode ;

    VAR
        Eleccion1, Eleccion2 : CARDINAL ;
        c : CHAR ;

(* Limpia la zona donde escribimos los mensajes *)
    PROCEDURE Limpia ;
    BEGIN
         SetCursorPosition(20, 0) ;
         ClrEOL ;
         WriteLn ;
         ClrEOL ;
         WriteLn ;
         ClrEOL ;
         SetCursorPosition(20, 0) ;
    END Limpia ;
    BEGIN

        SetDisplayMode(2) ;

        LOOP

            PullDownMenu( 0, FALSE, "Archivo|Edición|Ayuda",
                        "Nuevo|Abrir|Guardar|Guardar como|Salir&Pegar|Cortar|Copiar & Contenido
                         |Acerca de ",
                        TRUE, Eleccion1, Eleccion2 );

            CASE Eleccion1 OF

                0 : Limpia ;
                    WriteString('Has pulsado ESC ') ;
                    Read( c ) ; (* Espera la pulsación de una tecla cualquiera *)
                    EXIT ; |

                1 : CASE Eleccion2 OF

                        1 : Limpia ;
                            WriteString("Crear un archivo nuevo") ; |

                        2 : Limpia ;
                            WriteString("Abrir un archivo existente") ; |

                        3 : Limpia ;
                            WriteString("Guardar el archivo actual") ; |

                        4 : Limpia ;
                            WriteString("Cambiar de nombre el archivo actual") ; |

                        5 : Limpia ;
                            WriteString('Has elegido la opción "¡ Salir !" ') ;
                            Read( c ) ; (* Espera la pulsación de una tecla cualquiera *)
                            EXIT ;
                    END |

                2 : CASE Eleccion2 OF

                        1 : Limpia ;
                            WriteString("Pegar desde el portapapeles") ; |

                        2 : Limpia ;
                            WriteString("Cortar hasta el portapapeles") ; |

                        3 : Limpia ;
                            WriteString("Copiar en el portapepeles") ;
                     END |


                3 : CASE Eleccion2 OF

                        1 : Limpia ;
                            WriteString("Ver el contenido de la ayuda") ; |
```

```
                    2 : Limpia ;
                        WriteLine("Autor: José Garzía") ;
                        WriteLine("Propósito: mostrar el funcionamiento del procedimiento
PullDownMenu") ;
                        WriteString("Enero de 1997") ;
                END ;

            END ;

        END ;

        SetDisplayMode(2) ;

 END BarraDeMenu.
```

13 PROGRAMACIÓN ORIENTADA A OBJETOS CON MODULA-2

13.0 INTRODUCCCIÓN

El Modula-2 estándar no recoge la Programación Orientada a Objetos (POO). Pero el entorno de programación FST tiene una extensión que proporciona soporte para programar en este estilo. Por supuesto, al usar esta extensión se pierde portabilidad. Además, en el mismo FST, la sintaxis puede variar de una versión a otra. De todas formas, puede ser útil para quienes, conociendo Modula-2, quieran aprender los rudimentos de la POO.

Los procedimientos y definiciones de tipos residen en el módulo `Objects`. Las palabras reservadas son CLASS, INHERIT, INIT, DESTROY, SELF y MEMBER.

```
DEFINITION MODULE Objects;

(* (C) Copyright 1992 Fitted Software Tools. All rights reserved. *)

(*
    Please read the chapter "Classes" in the user documentation
    for information about this module.
*)

FROM SYSTEM IMPORT ADDRESS;

TYPE
    Class = POINTER TO ClassPtr;
    ClassPtr = POINTER TO ClassHeader;
    ClassHeader = RECORD
        parent      :Class;
        size        :CARDINAL;
        filler      :CARDINAL;
        InitProc    :PROCEDURE( ADDRESS );
        DestroyProc :PROCEDURE( ADDRESS );
    END;

    ObjPtr = POINTER TO RECORD
        class :ClassPtr
        (* data *)
    END;

PROCEDURE ALLOCATEOBJECT( VAR op :ObjPtr; cp :ClassPtr );
(*
    Allocates storage (calling Storage.ALLOCATE) for an object
    of class cp, places the object's address (handle) in op
    and invokes the INIT methods defined for class cp.
*)

PROCEDURE DEALLOCATEOBJECT( VAR op :ObjPtr );
(*
    Invokes the DESTROY methods in op's class and deallocates the
    storage used by op. op is set to NIL.
*)

PROCEDURE MEMBEROBJECT( cp :ClassPtr; class :ClassPtr ) :BOOLEAN;
(*
    TRUE if cp is of class class or one of its descendants.
*)

END Objects.
```

13.1 CARENCIAS DE LA PROGRAMACIÓN NO ORIENTADA A OBJETOS

Gran parte de los programas de ordenador tienen como propósito simular modelos de la vida real. Estos modelos son muy complejos. Existen muchos componentes y muchas interacciones entre los componentes de estos modelos.

Es algo comprobado que el ser humano puede tener en mente aproximadamente 7 datos, no muchos más. Ejemplo: Al diseñar una vivienda, no se diseñan *de golpe* todos sus componentes. No se diseñan los rodamientos de la silla giratoria del despacho al mismo tiempo que los quemadores de la cocina de butano. El diseño descendente procede de esta manera:

Primero se piensa en la entidad vivienda.

Después se piensa que la vivienda consta de varios componentes, varias estancias.

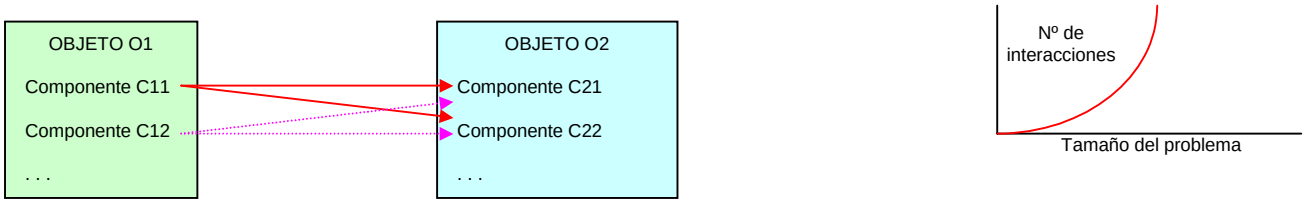
Después, para cada estancia, se piensa en los aparatos, muebles y dispositivos que hay en ella.

A continuación se piensa que estos aparatos y muebles, cada uno de ellos está formado por varias piezas.

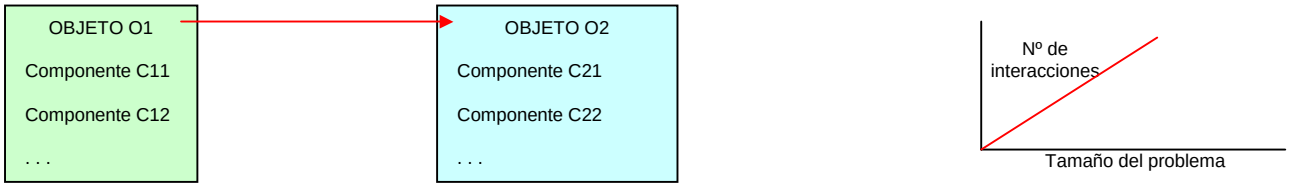
Por supuesto, en algún momento hay que dejar de descender. Esto ocurre cuando el comportamiento de una entidad ya puede modelarse con primitivas del lenguaje.

Desde el exterior de un aparato, por ejemplo la lavadora, se la ve y se la trata como una unidad, sin importarnos la piezas de que está compuesta. Pero si se entra dentro de ella, sí que hay que distinguir una piezas de otras, por ejemplo para poder repararla.

Este proceso de diseño descendente lo puede hacer muy bien la programación no orientada a objetos. Pero el problema es que no deja claro que los componentes de unos objetos no necesitan interactuar con los componentes de otros objetos. Si hubiera que considerar estas interacciones, su número crecería exponencialmente con el tamaño del problema.



Afortunadamente, en el mundo real las interacciones tienen lugar entre objetos, no entre sus componentes internos. Así, el número de interacciones crece linealmente con el tamaño del problema. Por ejemplo, las piezas de la lavadora no interactúan con las piezas del microondas.



En la programación tradicional no existe un sillón aquí y una lavadora allí, sino piezas de un sillón por aquí y piezas de una lavadora por allí. El programa funciona igual de bien, pero no hay entidades unitarias, que gestionen sus propias piezas internas, descargando de esta tarea al programador en el momento de diseñar a nivel de objetos. Cuando se está planeando dónde colocar la lavadora, no es necesario pensar en sus circuitos internos.

En la POO, el programa está basado en un modelo que se inspira en el mundo real. Una correspondencia más evidente entre el programa y el mundo real hace más fácil de entender el programa, más fácil de mantener; y así se reducen los costes de desarrollo.

También sucede que no es necesario manipular los componentes de un objeto desde fuera del entorno de ese objeto. Otro ejemplo, desde el despacho se conoce que en la cocina hay una lavadora y se sabe para qué sirve. Pero quien está en el despacho no necesita saber cómo se maneja. Puede enviar un mensaje hasta la cocina (o ir él mismo, pero ya no sería un componente del objeto despacho, sino del objeto cocina) para ponerla en funcionamiento.

La POO va más allá del establecimiento de la jerarquía descendente:

Proporciona un encapsulado de componentes internos. Este encapsulado hace que el usuario de un objeto no necesite conocer todos los detalles de su funcionamiento interno, sino sólo su interfaz. Por ejemplo, el funcionamiento de un sistema de aire acondicionado depende de muchas variable, tales como temperatura, humedad ambiental, etc. Pero la gestión en función de todas ellas es automática. El usuario sólo debe conocer las instrucciones de manejo de los mandos externos, su interfaz.

Define las interacciones entre objetos a través del intercambio de mensajes. El programador dota a los objetos de una personalidad y un comportamiento definido. De esta forma confiamos que al pulsar del botón de inicio de la lavadora, no se enciendan los fuegos de la cocina.

Otra carencia en la programación tradicional es que no existe una relación de ligadura entre las sentencias o procedimientos y los datos que pueden ser tratados con dichas sentencias o procedimientos.

13.2 DEFINICIONES DE LOS CONCEPTOS CLAVE

13.2.1.- Objeto

El objeto es una entidad con unos atributos y comportamientos que lo caracterizan. Los atributos son lo que en la programación tradicional se conoce como variables. En la terminología de la Programación Orientada a Objetos (POO) se conocen como miembros. Los comportamientos son sintetizados con lo que en la programación tradicional se conoce como procedimientos, subrutinas o funciones. En la terminología de la POO se conocen como métodos.

Los objetos son creados en algún momento del programa, interactúan con otros objetos, sufren modificaciones; y son destruidos en algún otro momento del programa.

13.2.2.- Clase

Una clase es la *abstracción* de objetos similares. Contiene las declaraciones de los miembros y métodos de los objetos que pertenecen a dicha clase. Es decir, si se observa que hay un conjunto de objetos que poseen rasgos comunes, se considera que pertenecen a una misma clase. La clase se describe mediante esos rasgos comunes. Este es un proceso que va de lo particular a lo general.

Pero en POO también se puede dar el punto de vista contrario. Cada objeto es la *materialización* de una clase (ente abstracto) en un individuo, un ejemplar concreto. Así, en la terminología POO, se dice que un determinado objeto es una instancia de su clase.

Al contener la clase dos tipos de componentes, miembros y métodos (es decir, datos y código), ya se puede vislumbrar una relación de ligadura entre datos y procedimientos, pues lo habitual será que los métodos de un objeto estarán diseñados para procesar los miembros de ese objeto.

La POO refleja la realidad mucho mejor que la programación no orientada a objetos, pues trabaja con modelos que están inspirados en los objetos reales.

Recomendaciones a la hora de crear una nueva clase.

1ª.- Debe incluir una interfaz de métodos adecuada para poder manejar el objeto.

2ª.- Debe representar un concepto presente en el problema (concepto físico o mental). Si pronunciando en voz alta el nombre de la clase, no se evoca inmediatamente un objeto, es probable que haya que cuestionar la creación de dicha clase.

3ª.- Debe ser lo más completa posible. Si delega tareas extra en otra clase, debe documentarse cuáles son esas tareas y cómo intercambian datos y resultados.

4ª.- Debe ser segura. Si se le aplica una entrada no documentada, debe rechazarla. Por ejemplo, si el televisor tiene cincuenta canales, sería inadmisible que al seleccionar el canal cincuenta y uno, empezara a arder.

Ejemplo:

Podríamos idear una clase para representar la figura geométrica rectángulo. Estos serían sus atributos:

Miembros:

`px`	Coordenada *x* de la esquina superior izquierda.
`py`	Coordenada *y* de la esquina superior izquierda.
`base`	Longitud de la base
`altura`	Longitud de altura

Métodos:

`Dibujar()`	Dibuja un rectángulo de dimensiones `base` y `altura`, en el punto *(px, py)* de la pantalla.
`Borrar()`	Hace desaparecer el rectángulo de la pantalla. No entraremos en los detalles de cómo realiza el borrado.
`CalcularArea()`	Calcula el área del rectángulo, y devuelve su valor como un entero.
`Desplazar()`	Mueve el rectángulo hasta una nueva posición.

Aquí va nuestro primer programa POO en Modula-2. Por empezar de la forma más simple posible, sólo definimos un método en la clase, el método `Dibujar()`.

```
(*
   ╔═══════════════════════════════════════════════════════════════════╗
   ║    Nombre:    Poo01.Mod                                           ║
   ║    Propósito: Programa ejemplo Nº 1 de la                         ║
   ║               Programación Orientada a Objetos en Modula-2        ║
   ║     Acciones: Crea una clase llamada Rectangulo                   ║
   ║               y un objeto de esta clase                           ║
   ║    Autor:     José Garzía                                         ║
   ╚═══════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaRectangulo;

    FROM InOut IMPORT WriteString, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;

    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición (sin declaración) de la clase *)
    CLASS Rectangulo;
        Px:     CARDINAL;
        Py:     CARDINAL;
        Base:   CARDINAL;
        Altura: CARDINAL;
        PROCEDURE Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;

        BEGIN
            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Py, Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Py+nFila, Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write('*');
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)

    END Rectangulo;

    VAR
        MiRectangulo:   Rectangulo;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Px := 2;
        MiRectangulo.Py := 5;
        MiRectangulo.Base := 10;
        MiRectangulo.Altura := 6;
        MiRectangulo.Dibujar;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

Es posible que uno de los atributos de una clase sea un objeto de otra clase:
Ejemplo:

```
(*
    Nombre:    Poo02.Mod
    Propósito: Programa ejemplo Nº 2 de la
               Programación Orientada a Objetos en Modula-2
     Acciones: Crea una clase llamada Punto,
               una clase Rectangulo, que contiene un objeto Punto
               y un objeto de la clase Rectangulo
    Autor:     José Garzía
*)

MODULE PruebaRectangulo;

    FROM InOut IMPORT WriteString, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición de la clase Punto *)
    CLASS Punto;
        Px:     CARDINAL;
        Py:     CARDINAL;
    END Punto;

    (* Definición de la clase Rectangulo *)
    CLASS Rectangulo;
        Vertice:    Punto;
        Base:       CARDINAL;
        Altura:     CARDINAL;
        PROCEDURE  Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;

        BEGIN
            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Vertice.Py, Vertice.Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Vertice.Py+nFila, Vertice.Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write('*');
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)

    END Rectangulo;

    VAR
        MiRectangulo:   Rectangulo;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Vertice.Px := 2;
        MiRectangulo.Vertice.Py := 5;
        MiRectangulo.Base := 10;
        MiRectangulo.Altura := 6;
        MiRectangulo.Dibujar;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

13.3 OTRAS DEFINICIONES

13.3.1.- Herencia

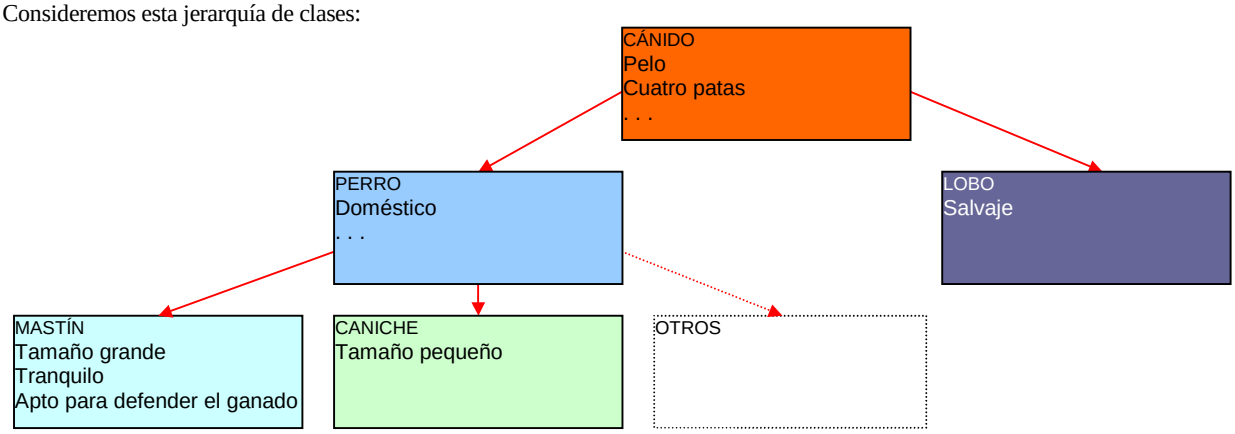
Se puede definir una clase que contenga todas las características de otra (llamada clase base) y quizá otras características adicionales. Este concepto posibilita la reutilización del código.

Un objeto de la clase derivada contiene igual o más datos que la clase base. Por esto, un objeto de la clase derivada se puede asignar a un objeto de la clase base, pero no al revés.

Algunos lenguajes permiten la herencia múltiple. Una clase hereda todas las características de varias clases base diferentes.

Ejemplo:

Consideremos esta jerarquía de clases:



Si alguien nos pregunta qué es un mastín, podríamos responder: *es un perro de tamaño grande, tranquilo, especializado en la defensa de rebaños*. Al heredar todas las características de la *clase perro* (doméstico, pelo, cuadrúpedo, etc), si nuestro oyente conoce dicha clase, para definirle la *clase mastín*, nos las podemos ahorrar en la descripción pedida.

Ejemplo:

Supongamos que ya tenemos compilada la clase `Rectangulo` y hemos comprobado que funciona bien. Y que ahora necesitamos una clase que represente un cuadrado rellenándolo con un carácter que se pueda elegir. Podríamos modificar la clase `Rectangulo`, añadiéndole como miembro una variable que almacene el carácter; y como método, uno para dibujar un cuadrado llenado con este carácter. Pero nos resistimos a modificar algo que tenemos comprobado que funciona bien. La solución más elegante, económica, cómoda y segura es definir una nueva clase que herede todo lo de la anterior e incorpore estos dos nuevos atributos.

```
(*
    Nombre:     Poo03.Mod
    Propósito:  Programa ejemplo Nº 3 de la
                Programación Orientada a Objetos en Modula-2
     Acciones:  Crea una clase base llamada Rectangulo,
                una clase derivada llamada RectanguloMejorado
                y un objeto de la clase derivada
    Autor:      José Garzía
*)

MODULE PruebaRectangulo;

    FROM InOut IMPORT WriteString, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición de la clase Punto *)
    CLASS Punto;
        Px:     CARDINAL;
        Py:     CARDINAL;
    END Punto;
```

```
    (* Definición de la clase base *)
    CLASS Rectangulo;
        Vertice:    Punto;
        Base:       CARDINAL;
        Altura:     CARDINAL;
        PROCEDURE  Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;

        BEGIN
            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Vertice.Py, Vertice.Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Vertice.Py+nFila, Vertice.Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write('*');
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)

    END Rectangulo;

    (* Definición de la clase derivada*)
    CLASS RectanguloMejorado;
        INHERIT Rectangulo;

        (* Atributos que hereda:
            Vertice
            Base
            Altura
            Dibujar()
        *)

        PROCEDURE DibujarMejor();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;
                Letra:      CHAR;
                x:          CARDINAL;
                y:          CARDINAL;

        BEGIN
            WriteString("Elija un carácter para rellenar el cuadrado ");
            Read(Letra);

            (* Estas variables no se utilizan para nada.
               Pero aunque parezca una tontería, dentro del bucle
               Vertice.Px  y Vertice.Py "es como si" valieran 0
            *)
            x := Vertice.Px;
            y := Vertice.Py;

            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Vertice.Py, Vertice.Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Vertice.Py+nFila, Vertice.Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write(Letra);
                END; (* FOR interno*)
            END; (* FOR externo *)

        END DibujarMejor; (* del procedimiento *)

    END RectanguloMejorado;

    VAR
        MiRectangulo:   RectanguloMejorado;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Vertice.Px := 8;
        MiRectangulo.Vertice.Py := 5;
        MiRectangulo.Base := 10;
        MiRectangulo.Altura := 6;
        MiRectangulo.DibujarMejor;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

13.3.2.- *Es_un* versus *tiene_un*

Es_un.
Si la clase B deriva de la clase A, se dice que cada objeto de la clase B es un objeto de la clase A. Por ejemplo sean A la clase de los animales; y B la clase de los perros. Los perros poseen todas las características de los animales. Se dice que los perros son animales.

Tiene_un.
La clase B puede no derivar de A, sino contener un objeto de A.

Ya hemos visto un ejemplo en el que es preferible la relación *es_un*. Queríamos un rectángulo mejorado, es decir, una figura que siguiera siendo un rectángulo, no que contuviera un rectángulo.

Un ejemplo en el que es preferible la relación *tiene_un* a la *es_un* es el siguiente:
Algunos lenguajes tienen predefinida la clase `Lista` para contener una lista de objetos. Esta clase usualmente proporciona métodos para insertar nuevos elementos en cualquier parte de la lista Si el programador desea definir una clase `Pila` para contener estructuras de datos tipo stack podría derivarla de la clase `Lista`. Pero así, los métodos de `Lista` los heredaría `Pila`. Un usuario poco respetuoso con las normas podría utilizarlos para insertar elementos en cualquier parte dela pila, lo cual se sale de las normas. Lo que procede es que la clase `Pila` tenga un miembro privado de tipo `Lista` (para que alguien ajeno al diseñador de `Pila` tenga acceso a los métodos de `Lista`) y dotar a la clase `Pila` de un único método de inserción y otro único de extracción.

13.3.3.- Omisión de métodos

Omitir un método de una clase base es crear en la clase derivada otro con mismo nombre, mismo tipo devuelto y misma lista de argumentos, aunque quizá con diferente cuerpo. Cuando dentro de la clase derivada se haga una llamada a dicho método, sólo se tendrá en cuenta el de la clase derivada. El de la clase base será ignorado. Aunque este detalle es importante en la omisión hay otro aún más:
Si dentro de la clase base se hace una llamada al método, no hay que pasar de largo el sutil detalle de en qué momento se realiza la decisión de si llamar al de esta base o al de la derivada.

Enlace temprano (*early binding*). El procedimiento que se va a utilizar en un punto del programa se conoce en tiempo de compilación.

Enlace tardío (*late binding*). La decisión de qué procedimiento se utiliza se posterga hasta el tiempo de ejecución. El compilador gestiona el enlace tardío de forma transparente al programador.

Métodos virtuales. Son los métodos que procesan objetos polimórficos con enlace tardío. Deben declarase como virtuales en la clase base. Las implementaciones del método son diferentes en cada clase derivada. Las llamadas al método virtual se mantienen sin resolver hasta el momento de la ejecución, que es cuando se decide si se llama a una implementación o a otra. Son los que se declaran en el módulo de definición.

Métodos estáticos. Son métodos que sólo procesan objetos no polimórficos, con enlace temprano. Son los que no se declaran en el módulo de definición.

Ejemplo: El siguiente programa no puede ser compilado, pues los procedimientos estáticos no pueden ser sobrecargados mi omitidos.

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:     Poo04.Mod                                              ║
  ║    Propósito: Programa ejemplo Nº 4 de la                             ║
  ║               Programación Orientada a Objetos en Modula-2            ║
  ║               Muestra que no es posible omitir métodos estáticos      ║
  ║     Acciones: Crea una clase base llamada Rectangulo,                 ║
  ║               una clase derivada llamada RectanguloMejorado           ║
  ║               y un objeto de la clase derivada                        ║
  ║               Pero no puede ser compilado                             ║
  ║    Autor:      José Garzía                                            ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaRectangulo;

    FROM InOut IMPORT WriteString, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición de la clase base *)
    CLASS Rectangulo;
        Px:     CARDINAL;
        Py:     CARDINAL;
        Base:   CARDINAL;
        Altura: CARDINAL;
        PROCEDURE Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;
```

```
        BEGIN
            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Py, Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Py+nFila, Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write('*');
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)

    END Rectangulo;

    (* Definición de la clase derivada*)
    CLASS RectanguloMejorado;
        INHERIT Rectangulo;

        (* Atributos que hereda:
            Px
            Py
            Base
            Altura
            Dibujar()
        *)

        (* Omisión del método *)
        PROCEDURE Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;
                Letra:      CHAR;

        BEGIN
            WriteString("Elija un carácter para rellenar el cuadrado ");
            Read(Letra);

            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Py, Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Py+nFila, Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write(Letra);
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)

    END RectanguloMejorado;


    VAR
        MiRectangulo:   RectanguloMejorado;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Px := 2;
        MiRectangulo.Py := 5;
        MiRectangulo.Base := 10;
        MiRectangulo.Altura := 6;
        MiRectangulo.Dibujar;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

La forma correcta para poder omitir un procedimiento es la compilación de módulos por separado.
Ejemplo:

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Poo05.Mod                                               ║
  ║    Propósito: Programa ejemplo Nº 5 de la                             ║
  ║               Programación Orientada a Objetos en Modula-2            ║
  ║     Acciones: Crea un objeto de una clase derivada para probar        ║
  ║               la omisión de métodos                                   ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaRectangulo;
FROM InOut IMPORT Read;

FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

FROM Rectango IMPORT Rectangulo, RectanguloMejorado;

    VAR
        MiRectangulo:   RectanguloMejorado;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Px := 2;
        MiRectangulo.Py := 5;
        MiRectangulo.Base := 10;
        MiRectangulo.Altura := 6;
        MiRectangulo.Dibujar;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Rectango.Def                                            ║
  ║    Propósito: Módulo de definición de la clase Rectangulo             ║
  ║    Acciones:  Define una clase base llamada Rectangulo,               ║
  ║               una clase derivada llamada RectanguloMejorado           ║
  ║               y declara en ambas el método Dibujar                    ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)


DEFINITION MODULE Rectango;

    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;
    EXPORT QUALIFIED Rectangulo, RectanguloMejorado;

    (* Definición de la clase base *)
    CLASS Rectangulo;
        Px:     CARDINAL;
        Py:     CARDINAL;
        Base:   CARDINAL;
        Altura: CARDINAL;

        PROCEDURE Dibujar();
    END Rectangulo;

    (* Definición de la clase derivada*)
    CLASS RectanguloMejorado;
        INHERIT Rectangulo;
        (* Atributos que hereda:
            Px
            Py
            Base
            Altura
            Dibujar()
        *)
        PROCEDURE Dibujar();
    END RectanguloMejorado;

END Rectango.
```

```
(*
    Nombre:    Rectango.Mod
    Propósito: Módulo de Implementación de la clase Rectangulo
    Acciones:  Implementa la clase base Rectangulo, la clase derivada
               RectanguloMejorado y omite en la derivada un método.
    Autor:     José Garzía
*)

IMPLEMENTATION MODULE Rectango;
    FROM InOut IMPORT WriteString, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;

    (* Definición de la clase base *)
    CLASS Rectangulo;

        PROCEDURE Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;
        BEGIN
            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Py, Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Py+nFila, Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write('*');
                END; (* FOR interno*)
            END; (* FOR externo *)

        END Dibujar; (* del procedimiento *)
    END Rectangulo;

    (* Definición de la clase derivada*)
    CLASS RectanguloMejorado;
        (* Omisión del m,todo *)
        PROCEDURE Dibujar();
            VAR
                nFila:      CARDINAL;
                nColumna:   CARDINAL;
                Letra:      CHAR;

        BEGIN
            WriteString("Elija un carácter para rellenar el cuadrado ");
            Read(Letra);

            SetCursorPosition(0, 0);
            ClrEOS;

            SetCursorPosition(Py, Px);
            FOR nFila := 0 TO (Altura-1) DO
                SetCursorPosition(Py+nFila, Px) ;
                FOR nColumna := 0 TO (Base-1) DO
                    Write(Letra);
                END; (* FOR interno*)
            END; (* FOR externo *)
        END Dibujar; (* del procedimiento *)
    END RectanguloMejorado;

END Rectango.
```

13.3.4.- Polimorfismo

Es uno de los conceptos más potentes de la POO. Sobre él hay una cierta confusión entre los estudiantes y entre algunos autores. Es confundido con la sobrecarga de métodos.

Consiste en que un mismo objeto, en unas ejecuciones del programa puede pertenecer a una clase; y en otras ejecuciones puede pertenecer a otra clase diferente (ambas clases proceden de una misma clase base). No es posible saber a priori, en tiempo de compilación a qué clase va a pertenecer. Cuando se necesite invocar a un método de un objeto polimórfico, no se sabrá si se llamará al de una clase o al de la otra. Por ello, el método debe tener igual nombre, igual tipo devuelto e igual lista de argumentos. Quizá venga de aquí la confusión, pero nótese que en la sobrecarga, las listas de argumentos deben ser diferentes.

La utilidad del polimorfismo está en que se pueden tratar de forma genérica objetos de clases diferentes.

Ejemplo:

Supongamos que tenemos dos clases diferentes, pero relacionadas. Si desde un principio (desde el momento de la compilación) conocemos de qué tipo va a ser cada figura, cuando codifiquemos un algoritmo en el que haya que calcular el área, el compilador sabrá a qué método `Mensaje` debe llamar.

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Poo07.Mod                                               ║
  ║    Propósito: Programa ejemplo Nº 7 de la                             ║
  ║               Programación Orientada a Objetos en Modula-2            ║
  ║    Acciones:  Crea dos clases, similares, sin polimorfismo.           ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaFiguras;

    FROM InOut IMPORT WriteLine, WriteLn, Read ;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición de la primera clase *)
    CLASS Rectangulo;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine("Soy un rectángulo");
        END Mensaje;
    END Rectangulo;

    (* Definición de la segunda clase *)
    CLASS Circulo;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine("Soy un círculo");
        END Mensaje;
    END Circulo;

    VAR
        MiRectangulo:   Rectangulo;
        MiCirculo:      Circulo;
        Letra:          CHAR;
        AreaRectangulo: REAL;
        AreaCirculo:    REAL;

    BEGIN
        NEW(MiRectangulo);
        MiRectangulo.Mensaje;

        NEW(MiCirculo);
        MiCirculo.Mensaje;

        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaFiguras.
```

Pero puede suceder que cuando estamos codificando el algoritmo, no sepamos si vamos a tener un cuadrado o un círculo. Por ejemplo:

```
(*
   ╔═══════════════════════════════════════════════════════════════════╗
   ║    Nombre:    Poo08.Mod                                           ║
   ║    Propósito: Programa ejemplo Nº 8 de la                         ║
   ║               Programación Orientada a Objetos en Modula-2        ║
   ║    Acciones:  Crea dos clases similares, con intento de polimorfismo.
   ║               Ocurre un error de compilación.                     ║
   ║    Autor:     José Garzía                                         ║
   ╚═══════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaFiguras;

    FROM InOut IMPORT WriteLine, WriteString, WriteLn, Read ;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;

    (* Definición de la clase base *)
    CLASS Figura;
    END Figura;

    (* Definición de la primera clase derivada *)
    CLASS Rectangulo;
    INHERIT Figura;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine("Soy un rectángulo");
        END Mensaje;
    END Rectangulo;

    (* Definición de la segunda clase derivada *)
    CLASS Circulo;
    INHERIT Figura;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine("Soy un círculo");
        END Mensaje;
    END Circulo;

    VAR
        MiFigura:         Figura;
        MiRectangulo:    Rectangulo;
        MiCirculo:       Circulo;
        Letra:            CHAR;
        AreaRectangulo:  REAL;
        AreaCirculo:     REAL;

    BEGIN

        (* La forma de la figura no se conoce en tiempo de compilación *)
        REPEAT
            WriteLine("Elija una figura");
            WriteString("R: rectángulo, C: círculo; ");
            Read(Letra);
        UNTIL ( (Letra='c') OR (Letra='C') OR (Letra='r') OR (Letra='R') );

        CASE Letra OF
            'r', 'R':   NEW(MiRectangulo);
                        MiFigura := MiRectangulo; |

            'c', 'C':   NEW(MiCirculo);
                        MiFigura := MiCirculo;
        END; (* del CASE *)

        MiFigura.Mensaje; (* ¿Cuál es el método invocado? *)

        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaFiguras.
```

En la programación tradicional, una solución pasaría por:

1º.- Tener una enumeración: `TYPE TipoFigura( FCuadrado, FCirculo);`

2º.- Tener un miembro de tipo `TipoFigura` en ambas clases.

3º.- Tener dos métodos con nombre diferente `MensajeCuadrado()` y `MensajeCirculo()`, uno en cada clase.

4º.- Tener un esquema *CASE*:

```
CASE MiFigura.Tipo OF
   FCuadrado: MiFigura.MensajeCuadrado();
   FCirculo:  MiFigura.MensajeCirculo ();
END;
```

La solución que proporciona el polimorfismo consiste en:

1º.- Hacer que las clases sean subclases de una clase base.
2º.- Hacer que el método esté presente en la clase base (aunque su cuerpo esté vacío).
3.- Omitir el método en las clases derivadas.

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Poo09.Mod                                               ║
  ║    Propósito: Programa ejemplo Nº 9 de la                             ║
  ║               Programación Orientada a Objetos en Modula-2            ║
  ║    Acciones:  Crea dos clases similares, con polimorfismo.            ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)


MODULE PruebaFiguras;

    FROM InOut IMPORT WriteLine, WriteString, WriteLn, Read ;
    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;
    FROM Figuras IMPORT Figura, Rectangulo, Circulo;

    VAR
        MiFigura:       Figura;
        MiRectangulo:   Rectangulo;
        MiCirculo:      Circulo;
        Letra:          CHAR;
        AreaRectangulo: REAL;
        AreaCirculo:    REAL;

    BEGIN

        (* La forma de la figura no se conoce en tiempo de compilación *)
        REPEAT
            WriteLine("Elija una figura");
            WriteString("R: rectángulo, C: círculo; ");
            Read(Letra);
        UNTIL ( (Letra='c') OR (Letra='C') OR (Letra='r') OR (Letra='R') );

        CASE Letra OF
            'r', 'R':   NEW(MiRectangulo);
                        MiFigura := MiRectangulo; |

            'c', 'C':   NEW(MiCirculo);
                        MiFigura := MiCirculo;
        END; (* del CASE *)

        MiFigura.Mensaje; (* ¿Cuál es el método invocado? *)

        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaFiguras.
```

```
(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Figuras.Def                                             ║
  ║    Propósito: Módulo de definición de las clases de figuras.          ║
  ║    Acciones:  Define una clase base llamada Figura                    ║
  ║               y dos clases derivadas llamadas Rectangulo y Circulo.   ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)


DEFINITION MODULE Figuras;

    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;
    EXPORT QUALIFIED Figura, Rectangulo, Circulo;

    (* Definición de la clase base *)
    CLASS Figura;
        PROCEDURE Mensaje();
    END Figura;

    (* Definición de la primera clase derivada *)
    CLASS Rectangulo;
        INHERIT Figura;
        PROCEDURE Mensaje();
    END Rectangulo;

    (* Definición de la segunda clase derivada *)
    CLASS Circulo;
        INHERIT Figura;
        PROCEDURE Mensaje();
    END Circulo;

END Figuras.


(*╔═══════════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Figuras.Mod                                             ║
  ║    Propósito: Módulo de implementación de las clases de figuras.      ║
  ║    Acciones:  Implementa una clase base llamada Figura                ║
  ║               y dos clases derivadas llamadas Rectangulo y Circulo.   ║
  ║    Autor:     José Garzía                                             ║
  ╚═══════════════════════════════════════════════════════════════════════╝
*)

IMPLEMENTATION MODULE Figuras;
    FROM InOut IMPORT WriteLine, WriteLn, Write, Read ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;

    (* Implementación de la clase base. En realidad, no se hace nada *)
    CLASS Figura;
        PROCEDURE Mensaje();
        BEGIN
            (* No se hace nada *)
        END Mensaje;
    END Figura;

    (* Implementación de la primera clase derivada *)
    CLASS Rectangulo;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine(" Soy un rectángulo ");
        END Mensaje; (* del procedimiento *)
    END Rectangulo;

    (* Implementación de la segunda clase derivada *)
    CLASS Circulo;
        PROCEDURE Mensaje;
        BEGIN
            WriteLine(" Soy un círculo ");
        END Mensaje; (* del procedimiento *)
    END Circulo;

END Figuras.
```

13.3.5.- Encapsulación

Al unir en una misma entidad datos y código, se puede restringir el acceso a esos datos desde otras partes del código. Los miembros y métodos pueden ser:

Públicos. Son accesibles desde cualquier punto en que sea conocida la clase. Suelen declararse como públicos los métodos destinados a la comunicación entre el objeto y el programador. La forma de hacerlo es declararlos en el módulo de definición.

Privados. Sólo son accesibles desde los métodos de la propia clase. Suelen declararse como privados los miembros y métodos que no necesita usar directamente el programador. Por supuesto, los manipulará mediante otros métodos que sí serán accesibles. La forma de hacerlo es declararlos en el módulo de implementación.

Como norma general, una clase debe ocultar toda la información posible. Es conveniente ocultar todo aquello que los intrusos no necesitan saber. Se recomienda que los miembros sean privados; y para acceder a ellos, construir métodos públicos. De esta forma, el acceso a los miembros de una clase está controlado. Además, para usar una clase no es necesario conocer cómo está construida por dentro. Sólo es necesario conoce la interfaz de la clase, es decir, la forma en que hay que usar los métodos. Es como una caja negra en la que se conoce cuál es la respuesta ante cada entrada aplicada, pero no se sabe cómo está construida por dentro.

Ventajas de la encapsulación:

1ª.- Seguridad. El intruso sólo puede cambiar lo que le permite el diseñador de la clase.

2ª.- Facilidad para que el programador la mejore. Si la implementación de una clase está oculta, el programador puede mejorar su funcionamiento interno. Y si no modifica la interfaz, el usuario de la clase no necesita aprender una nueva forma de interactuar con ella. Por ejemplo, en una primera versión de un programa se puede tener un fichero para almacenar los datos; y en una segunda versión se puede tener una base de datos para dicho almacenamiento. Si el programador no modifica la forma de presentar los datos, no necesita buscar todos los usuarios para comunicarles las modificaciones. Y los usuarios no necesitan aprender una nueva forma de ver los datos.

Ejemplo:

```
(*╔═══════════════════════════════════════════════════════════════════╗
  ║    Nombre:    Poo10.Mod                                           ║
  ║    Propósito: Programa ejemplo Nº 10 de la                        ║
  ║               Programación Orientada a Objetos en Modula-2        ║
  ║    Acciones:  Crea un objeto de una clase que tiene miembros ocultos ║
  ║               y métodos públicos.                                 ║
  ║    Autor:     José Garzía                                         ║
  ╚═══════════════════════════════════════════════════════════════════╝
*)

MODULE PruebaRectangulo;
FROM InOut IMPORT Read;
FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;
FROM Rectango IMPORT Rectangulo;

    VAR
        MiRectangulo:   Rectangulo;
        Letra:          CHAR;

    BEGIN
        NEW(MiRectangulo);

    (*  Esto es imposible

        MiRectangulo.Base := 12;
        MiRectangulo.Altura := 8;
        WriteCard(MiRectangulo.Base, 3);
        WriteCard(MiRectangulo.Altura, 3);
    *)

        MiRectangulo.EscribeBase;
        MiRectangulo.EscribeAltura;
        Read(Letra);    (* espera la pulsación de una tecla *)

        MiRectangulo.LeeBase;
        MiRectangulo.LeeAltura;
        Read(Letra);    (* espera la pulsación de una tecla *)

END PruebaRectangulo.
```

```
(*
   ╔═══════════════════════════════════════════════════════════════════════╗
   ║    Nombre:    Rectango.Def                                            ║
   ║    Propósito: Módulo de definición de la clase Rectangulo.            ║
   ║    Acciones:  Define una clase llamada Rectangulo.                    ║
   ║               Sólo tiene visibles los métodos.                        ║
   ║    Autor:     José Garzía                                             ║
   ╚═══════════════════════════════════════════════════════════════════════╝
*)


DEFINITION MODULE Rectango;

    FROM Objects IMPORT ALLOCATEOBJECT, DEALLOCATEOBJECT;
    EXPORT QUALIFIED Rectangulo;

    CLASS Rectangulo;

        PROCEDURE LeeBase();
        PROCEDURE LeeAltura();

        PROCEDURE EscribeBase();
        PROCEDURE EscribeAltura();

    END Rectangulo;

END Rectango.


(*
   ╔═══════════════════════════════════════════════════════════════════════╗
   ║    Nombre:    Rectango.Mod                                            ║
   ║    Propósito: Módulo de implementación de la clase Rectangulo.        ║
   ║    Acciones:  Implementa la clase Rectangulo                          ║
   ║               implementa los métodos y añade unos miembros ocultos.   ║
   ║    Autor:     José Garzía                                             ║
   ╚═══════════════════════════════════════════════════════════════════════╝
*)

IMPLEMENTATION MODULE Rectango;
    FROM InOut IMPORT WriteString, WriteLn, WriteCard, Read, ReadCard ;
    FROM Display IMPORT SetCursorPosition, ClrEOS;

    (* Implementación de la clase *)
    CLASS Rectangulo;

    (* Miembros ocultos *)
        Px:     CARDINAL;
        Py:     CARDINAL;
        Base:   CARDINAL;
        Altura: CARDINAL;

    (* Métodos implementados *)
        PROCEDURE LeeBase();
        BEGIN
                WriteString("La base es: ");
                WriteCard(Base, 3);
                WriteLn;
        END LeeBase;

        PROCEDURE LeeAltura();
        BEGIN
                WriteString("La altura es: ");
                WriteCard(Altura, 3);
                WriteLn;
        END LeeAltura;

        PROCEDURE EscribeBase();
        BEGIN
                WriteString("Dígame la base: ");
                ReadCard(Base);
                WriteLn;
        END EscribeBase;

        PROCEDURE EscribeAltura();
        BEGIN
                WriteString("Dígame la altura: ");
                ReadCard(Altura);
                WriteLn;
        END EscribeAltura;

    END Rectangulo;

END Rectango.
```

El miembro SELF. Hace referencia no a un miembro en particular del objeto en curso, sino a todo el objeto completo. Es útil cuando, dentro de un procedimiento, necesitamos devolver la dirección del objeto en el que está dicho procedimiento. Un uso típico es para construir una lista enlazada:

```
. . .(* Definiciones de las clases *)

CLASS Item;
 . . .
PROCEDURE IncorporaaLista( Lista: ListaEnlazada);
BEGÍN
   Lista.Incorpora(SELF);
END IncorporaaLista;
END Item;

CLASS Listaenlazada;
. . .
PROCEDURE Incorpora(Elemento: Item);
END ListaEnlazada;

. . .(* Uso de las clases *)
VAR
   MiElemento: Item;
   MiLista:    ListaEnlazada;
. . .
NEW(MiElemento);
MiElemento.IncorporaLista(MiLista);
```

13.3.6.- Constructores y destructores

Todos los objetos son creados dinámicamente (como punteros). Por tanto, es necesario crearlos con NEW. Con esto ya se reserva memoria en el momento de la creación. Si además se precisa inicializar algún miembro, el lugar para hacerlo es el procedimiento constructor INIT.

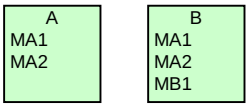
No hay recolección automática de memoria. Por ello, cuando un objeto no vaya a ser usado nunca más (o cuando se salga de su ámbito) hay que liberar la memoria que ocupaba. Esto se realiza con el procedimiento destructor DESTROY.

El programador nunca llama directamente ni al constructor ni al destructor. Las llamadas a INIT y DESTROY las hace automáticamente el compilador cuando el programador llama a NEW y DISPOSE.

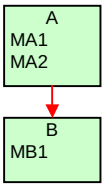
13.4 COMPARACIÓN ENTRE LAS FORMAS DE REUTILIZAR EL CÓDIGO

13.4.1.- Reutilización mediante herencia y adición

Supongamos que tenemos la clase A con los métodos MA1 y MA2; y que necesitamos una clase B, con los métodos MA1, MA2 y MB1. Se podría copiar todo el código de A, cambiar de nombre (A por B) y añadir MB1.

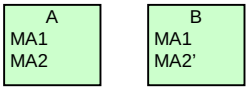


Pero es más fácil heredar B de A y añadir MB1.



13.4.2.- Reutilización mediante herencia y omisión

Supongamos que tenemos la clase A con los métodos MA1 y MA2; y que necesitamos una clase B, con los métodos MA1, MA2'. Se podría copiar todo el código de A, cambiar de nombre (A por B) y cambiar el cuerpo de MA2 por el de MA2'.



Pero es más fácil heredar B de A y omitir MA2.