Capítulo 2

ALGORITMOS Y ESTRUCTURAS DE DATOS

1. INTRODUCCIÓN

Uno de los objetivos de esta unidad es conocer las herramientas elementales para la construcción de algoritmos que den solución informática a los problemas planteados en cualquier entorno de trabajo. Son elementos básicos tales como variables, constantes, operadores, instrucciones, etc. que junto con otras estructuras típicas de programación servirán en la realización de algoritmos y, en definitiva, de programas.

2. LOS DATOS Y SU REPRESENTACIÓN

A veces, se confunden los conceptos de dato e información, o se usan indistintamente para referir a cualquiera de ellos. Los datos están constituidos por los registros físicos de los hechos, acontecimientos, transacciones, etc. Un **dato** es un símbolo físico que se utiliza para representar la información. Por dato se entiende un valor numérico, una serie de caracteres, un estado físico, etc., que pueda ser objeto de tratamiento informático. La **información**, sin embargo, implica que los datos estén procesados de manera que sean útiles y significativos para quien los recibe. Los programas de ordenador actúan sobre datos, los manipulan y producen otros datos de salida llamados resultados.

Los datos proceden de abstracciones de hechos reales. Por ejemplo, los datos pueden ser números, como horas trabajadas por un empleado, importe de una factura o años de un fósil; letras, como nombre de una persona, título de un libro, etc.; pueden también ser símbolos o combinaciones de todo ello. Los datos se producen de diferentes maneras y por distintas circunstancias.

Un **tipo de dato** es la forma general que tomará un conjunto de elementos que son datos de características similares. El tipo determina como se representarán internamente los datos y las operaciones que podrán realizarse sobre ellos.

Existen distintos métodos usados por los ordenadores para la representación interna de los datos. Todos estos métodos dependen de la cantidad de dígitos de que se dispongan para representar los números.

Se define el término **rango de representación** como el intervalo en el que se pueden mover los números representables mediante un método determinado. El rango vendrá especificado por el menor y el mayor valor permitido. Debe tenerse en cuenta que, al estar limitado el rango de valores de enteros en cualquiera de los métodos, al operar con números enteros pueden producirse errores por **desbordamiento de rango** de representación.

2.1 Tipos de datos numéricos

Numéricos son datos que representan cantidades y, con ellos, se pueden realizar todas las operaciones aritméticas y relacionales aplicables a los números, por ejemplo, sumar, hallar la raíz cuadrada, comprobar si un objeto es menor que otro, etc.

Los datos numéricos pueden dividirse en datos de tipo entero y de tipo real.

ENTEROS

Son los datos cuyos valores pertenecen al conjunto de los números enteros. Por tanto, pueden ser datos enteros positivos y datos enteros negativos. Cuando se omite el signo, se asume que el número es positivo.

Ejemplos: 18, 620, - 134, 0, - 10.

El conjunto de los números enteros tiene un rango de valores de $-\infty$ a $+\infty$, por lo que, en teoría, cualquier valor es válido. Sin embargo, las limitaciones de hardware y algunas consideraciones de tipo práctico dan lugar a que, en el ordenador, los valores tengan unos límites determinados.

Métodos de representación interna de los tipos de datos enteros

Algunos métodos de representación de enteros de los que se van a ver los siguientes:

- Binario puro
- Módulo y signo
- Otros

REALES

Son datos de tipo real los que pertenecen al conjunto de los números reales. Tienen una parte entera y una parte fraccionaria separadas por un punto decimal entre ellos y pueden ser positivos y negativos.

Generalmente se usa el sistema anglosajón para escribir estos números. El método consiste en usar el punto decimal en lugar de la coma decimal para separar las partes entera y fraccionaria de un número. Así por ejemplo escribiremos mil veintinueve con cincuenta 1029.50 en lugar de la manera habitual latina 1029,50.

Cuando se introduzca un dato real por teclado o deba escribirlo en un programa, utilice siempre el punto decimal con al menos un dígito a cada lado. Si el número no tiene parte decimal se escribirá cero tras el punto y si no tiene parte entera cero antes del punto.

Ejemplos: 18.02, 54.13, - 0.13, 0.0, - 10.15, 25.0

Para la representación, se utiliza lo que en matemáticas se llama **notación exponencial o científica** en la que una cantidad se expresa del siguiente modo:

$$\text{Número} = \text{mantisa} \times \text{base\_de\_exponenciación}^{\text{exponente}}$$

Recuerde que, con esta notación, el número tiene infinitas posibilidades de representación según el lugar donde se coloque el punto decimal (la coma en el sistema anglosajón), de todas ellas se toma como estándar la llamada **normalizada**. De acuerdo con esta norma, la mantisa se tomará sin parte entera y la primera cifra a la derecha del punto decimal debe ser significativa (distinta de cero), salvo en la representación de 0 que será 0.0. El exponente indica el lugar donde se coloca el punto decimal, conociéndose por este motivo como formato de punto (o coma) flotante. La cantidad de dígitos significativos del número es fija y es el punto decimal el que flota. A diferencia de los métodos de representación de enteros que se conocen como de punto (o coma) fijo, se les supone el punto decimal fijo e implícito a la derecha de los bits.

Ejemplos: Representar los números 181.3 y 72.345 utilizando como base de exponenciación 10.

$181.3 = 1813 \times 10^{-1} = 181.3 \times 10^{0} = 18.13 \times 10^{1} = 1.813 \times 10^{2} = 0.1813 \times 10^{3}$

$72.345 = 0.72345 \times 10^{2}$ notación exponencial normalizada.

Observe que el punto decimal puede moverse libremente de derecha a izquierda y viceversa, con tan sólo variar el exponente.

Las representaciones más usadas son los formatos de simple y doble precisión. La diferencia entre ellos afecta únicamente a la cantidad de dígitos significativos con que se almacena la mantisa, siguiendo en ambos casos las siguientes reglas para representación de reales en notación exponencial:

- El exponente se representa siguiendo alguno de los métodos de representación de enteros con signo: módulo y signo, complemento a uno, complemento a 2 o algún otro no comentado en este libro.

- De la mantisa se representan los dígitos significativos a la derecha del punto decimal, considerándose este implícito a su izquierda y se codifica también siguiendo alguno de los métodos de representación de enteros.

- Para representar la base de exponenciación, se toma una potencia de 2 que viene determinada por el fabricante de hardware.

Representación para simple precisión

Para ordenadores con microprocesadores que disponen de una palabra de 32 bits. Suele ocupar 4 bytes y la información se distribuye del siguiente modo:

- **Signo del número** (signo de la mantisa), generalmente codificado con lógica positiva y ocupando 1 bit.

- **Mantisa**, para la que se reservan 23 bits. Junto con el bit de signo del número ocupan 3 bytes. Puesto que se almacena en formato exponencial normalizado y la base de exponenciación es una potencia de 2 el *punto binario* se considera sobrentendido justamente delante de ella (0.mantisa).

- **Exponente**, se codifica ocupando los 8 bits restantes. En caso de tener un exponente negativo se almacena en complemento a dos.

2.2 Tipos de datos alfanuméricos

Los datos alfanuméricos almacenan los llamados **caracteres alfanuméricos**. Cualquier procesador trabaja internamente con un conjunto de caracteres que le permite manejar datos, instrucciones e información de cualquier tipo. Este conjunto de caracteres se puede clasificarlo de la siguiente forma:

- **Caracteres alfabéticos**. Son las letras de la A a la Z, excepto la Ñ, ya sean mayúsculas o minúsculas.

- **Caracteres numéricos**. Dígitos del 0 al 9. Con ellos no se pueden realizar operaciones aritméticas porque no son números.

- **Caracteres especiales**:

  - Guiones, paréntesis, asterisco, signos de puntuación, etc.

  - Caracteres de control: escape (Esc), tabulador, cursores, etc.

Los datos alfanuméricos pueden ser de dos clases:

- **Carácter**. Cuando almacenan un dato que es un sólo carácter.

  Ejemplos: ‘a’, ‘Z’, ‘1’

- **Cadena**. Está formada por una serie de caracteres. Una cadena puede ser una mezcla de letras, caracteres especiales y caracteres numéricos.

  Ejemplos*:* “ HOLA”, ”2_Coche”, “ ”. Este último ejemplo es una cadena de espacios en blanco.

El tipo de dato cadena no siempre está implementado en los distintos lenguajes de programación. En C, por ejemplo, no existe como tipo de dato simple incorporado al lenguaje, pero se suministran las herramientas necesarias para usarlo como si de un tipo de dato incorporado se tratara.

MÉTODOS DE REPRESENTACIÓN INTERNA DE LOS TIPOS DE DATOS ALFANUMÉRICOS

La forma de almacenar los caracteres es asignarle a cada uno de ellos un número entero y almacenar dicho número. A la tabla que relaciona cada carácter con su número correspondiente se la llama **código binario**.

Entre los códigos en uso actualmente se encuentran **EBCDIC** (**Extended Binary Coded Decimal Interchange Code**) y el **ASCII extendido** (**American Standard Code for Information Interchange**). Este último se llama así por ser una extensión del anterior código ASCII que usaba 7 bits para la representación de caracteres.

Como es lógico, mientras más bits emplee el código, mayor cantidad de caracteres diferentes se podrán almacenar. Los códigos de 8 bits como el ASCII son válidos para los idiomas basados en el alfabeto latino. Para utilizar alfabetos como el griego, el eslavo o los asiáticos (chino, coreano, japonés) se utiliza UNICODE, que emplea 16 bits.

3. IDENTIFICADORES

Los **identificadores** son nombres que se utilizan en los algoritmos y programas, que el programador asigna para referenciar a los distintos elementos que utilizan, ya sean objetos o procesos. Estos identificadores están formados por una serie de dígitos alfanuméricos, letras, números y otros caracteres, que generalmente no pueden contener espacios en blanco, ni caracteres acentuados, debiendo comenzar siempre por una letra. En principio, no está limitada la cantidad de caracteres que los forman y serán los propios lenguajes quienes determinen las reglas a seguir para su construcción y reconocimiento.

Como regla general, todos los identificadores deben ser declarados antes de ser usados, según los formatos establecidos para los distintos objetos que forman parte de un algoritmo o de un programa.

Los identificadores son construidos generalmente por los propios usuarios, aconsejándose el uso de nombres que den idea de su uso, esto hará fácil la lectura de los algoritmos para cualquier programador.

Ejemplos: precio, nombreAlumno, calcularNomina, sumar, mostrar, URNA1.

Algunos de estos identificadores están definidos para un uso específico en los algoritmos (leer, escribir, mientras, etc.), como se verá más adelante. También los lenguajes de programación tienen predefinidas palabras identificadoras de objetos y

procesos que se mantienen para usos específicos y que se conocen como palabras reservadas (en capítulos posteriores se estudiarán las palabras reservadas de Java).

CONVENCIONES

Razones para mantener convenciones:

- Las convenciones en los algoritmos y programas mejoran su lectura, permitiendo entender el código mucho más rápidamente y más a fondo.

- Si distribuyes tu código como un producto, necesitas asegurarte de que está bien hecho y **presentado** como cualquier otro producto.

Reglas para construir identificadores
(tareas (funciones, métodos), variables, constantes):

- Los nombres de los identificadores de datos deben ser **sustantivos**.

- Los nombres de los identificadores de tareas deben ser **verbos**.

- Cualquier nombre compuesto tendrá la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula.

- Los nombres no deben empezar por los caracteres "_" o "$", aunque ambos estén permitidos por el lenguaje.

- Los nombres deben ser **cortos pero con significado (código autodocumentado)**, de forma que sean un mnemónico, es decir indicará la función que realiza en el programa.

- Usar palabras **completas**, evitar acrónimos y abreviaturas (salvo que ésta sea mucho más conocida que el nombre completo, como dni, url, html…)

- Deben evitarse nombres de identificadores de un solo carácter salvo para índices temporales, como son i, j, k, m, y n para enteros; c, d, y e para caracteres.

- Las constantes se escriben con todas las letras en mayúsculas separando las palabras con un guión bajo ("_")

4. OPERADORES

Los operadores son símbolos que relacionan los argumentos de una **expresión**, **D**esde el punto de vista del programador, una expresión es un conjunto de argumentos u operandos relacionados mediante **operadores**.

En los algoritmos definidos con la técnica del pseudocódigo suelen utilizarse los símbolos descritos en las distintas tablas que aparecen en esta sección, pero tenga en cuenta que cuando los algoritmos tengan que escribirse con un lenguaje de

codificación específico, habrá que utilizar para la construcción de expresiones los operadores incorporados por dicho lenguaje.

Se puede realizar una clasificación de los operadores en función del tipo de expresión a la que afectan del siguiente modo:

4.1 Operadores aritméticos

Se utilizan para realizar operaciones aritméticas, por lo que los operandos sobre los que actúan deben ser de tipos de datos numéricos.

Los operadores pueden ser:

- **Binarios**. si actúan sobre dos operandos.

SÍMBOLO	OPERACIÓN
+	Suma
–	Resta
*	Multiplicación
/	División real
DIV	División entera
MOD	Resto de la división entera
↑ ∧ **	Potencia (los tres símbolos suelen usarse)

- **Unarios** o **monarios**. Actúan sobre un único operando

SÍMBOLO	OPERACIÓN
–	Cambio de signo

Ejemplos:

OPERACIÓN	RESULTADO
10 * 5	50
7 / 2	3.5
7 DIV 2	3
7 MOD 2	1
3 ∧ 4	81

4.2 Operadores relacionales

Se utilizan para realizar operaciones relacionales. Estas operaciones producen como resultado de su evaluación un valor booleano (verdadero o falso). Sus operandos pueden ser tipos de datos numéricos o alfanuméricos.

Los caracteres alfanuméricos se comparan en función del valor decimal correspondiente a su codificación en ASCII. En este caso, las comparaciones “mayor que” o “menor que” se entienden en el sentido “posterior a” o “anterior a”

respectivamente, es decir en el mismo orden en que se colocarían en el diccionario si son caracteres alfabéticos. Hay que tener en cuenta que el ASCII se basa en el alfabeto inglés, por lo que algunos caracteres propios de nuestro idioma (la ñ, vocales acentuadas) fueron añadidos en la extensión y tienen códigos superiores. Esto hará que si se ordenan palabras, el ordenador nos diga que "pasta" < "paño" es verdadero erróneamente. Se tratará este problema en capítulos posteriores.

Las cadenas o tiras de caracteres comparan carácter a carácter de izquierda a derecha. Sólo son iguales los operandos alfanuméricos si lo son en longitud y caracteres. Conviene saber que:

- Las letras minúsculas tienen mayor valor que las mayúsculas.
- Los caracteres numéricos son menores que las letras.

SÍMBOLO	OPERACIÓN
<	Menor que
==	Igual que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
!=	Distinto a

Ejemplos:

OPERACIÓN	RESULTADO
`10 < – 5`	Falso
`17 < = 21`	Verdadero
`3.5 > = 2`	Verdadero
`7 !=7`	Falso
`'A' < 'B'`	Verdadero ('A' es anterior a 'B')
`"MARIA" > "DOLORES"`	Verdadero ("MARIA" es posterior a "DOLORES")
`"hoja" > = "hojas"`	Falso ("hoja" NO es posterior o igual a "hojas")

4.3 Operadores lógicos o booleanos

Actúan sobre sus operandos de acuerdo con las normas que regulan el Álgebra de Boole. Son los siguientes:

`O` Operador binario que representa la "suma lógica" (*OR* en inglés). La operación que realiza es cierta cuando al menos uno de los dos operandos lo es y falsa si ambos son falsos. Se le conoce también como **operador de disyunción**.

`Y` Operador binario que representa el "producto lógico" (*AND* en inglés). La operación que realiza es cierta cuando los dos operandos lo son y falsa si

alguno de ellos es falso. Se le conoce también como **operador de conjunción**.

`NO` Es un operador unario que representa la "negación" (*NOT* en inglés). La operación que realiza cambia el estado lógico de su operando. Si era verdadero, lo convierte en falso y viceversa. Se le conoce también como **operador de complemento**.

- **Binarios**:

SÍMBOLO	OPERACIÓN
`O`	Suma lógica (OR)
`Y`	Producto lógico (AND)

- **Unarios**:

SÍMBOLO	OPERACIÓN
`NO`	Negación (NOT)

Ejemplos:

OPERACIÓN	RESULTADO
Sevilla es un río O Madrid una montaña	Falso
Esto es un libro y estoy leyendo	Verdadero
Sevilla es una ciudad O Madrid una montaña	Verdadero
Esto es un libro y estoy nadando	Falso
NO estoy leyendo	Falso

4.4 Operador de asignación

Mediante la **asignación**, se almacena en el identificador que aparece a la izquierda del símbolo "=" o del símbolo "←", el valor que está a su derecha. El operador de asignación se lee como "toma": El objeto referenciado por el identificador de la izquierda "toma" el valor de la derecha del símbolo, o bien, el valor de la derecha "se asigna o almacena" en el objeto de la izquierda.

Aunque ambos operadores suelen usarse para representar la asignación, es costumbre utilizar directamente el operador del lenguaje de programación habitual. En este libro, se usará normalmente "=", por ser este el símbolo que posee el lenguaje C para representar la asignación.

SÍMBOLO	OPERACIÓN
= ←	ASIGNACIÓN

Ejemplos:

OPERACIÓN	RESULTADO
`Edad = 35`	**Se almacena** 35 en el objeto representado por el identificador Edad.
`Numero ← 16.56`	Numero **toma** el valor 16.56.
`LETRA = 'A'`	**Se asigna** 'A' al objeto representado por el identificador LETRA.

4.5 Operador de precedencia

Paréntesis. Al igual que en las expresiones matemáticas, los paréntesis se utilizan en programación para anidar expresiones marcando con ellos el orden en que deben evaluarse los operandos o, dicho de otro modo, pueden usarse para cambiar el orden de precedencia de los operadores que están incluidos en la expresión.

Ejemplos:

OPERACIÓN	RESULTADO
`(18 > −44) O ('a' = 'A')`	Verdadero
`Esto es un libro y ('a' != 'A')`	Verdadero
`(36 = (6*6)) y ("mamá" != "mamá")`	Falso
`((5+2.5)*(3−5))*(−1)+5`	20

4.6 Orden de evaluación de los operadores

Por regla general, el orden en que se evalúan los operadores que intervienen en una expresión es el siguiente en orden de mayor a menor prioridad:

- En primer lugar los paréntesis, comenzando siempre por los más internos.
- Potencia.
- Producto y división.
- Suma y resta.
- Operadores relacionales.
- Operadores lógicos.

Cuando se utilizan operadores de igual precedencia, se ejecutan de izquierda a derecha, por regla general.

Cada lenguaje implementa sus propios operadores, su orden de precedencia y el sentido de evaluación de los mismos (izquierda a derecha o viceversa). Se pueden consultar los manuales cuando se necesite información al respecto, aunque es una buena técnica usar el operador de precedencia en caso de duda. El uso de paréntesis no sólo no aumenta el código fuente de un programa, sino que tiene la ventaja de hacer las expresiones más fáciles de leer.

5. VARIABLES

Algunas entidades de datos que pueden cambiar el valor que almacenan. Se define **variable** como una posición de memoria que se referencia con un identificador conocido como **nombre de la variable** y donde se almacena el valor de un dato que puede cambiar durante la ejecución del programa.

Una variable no es un dato, sino una zona de memoria que contiene un dato que puede variar. Sin embargo, en la práctica el nombre de la variable se toma como la propia variable y su valor es lo que se dice que cambia.

5.1 Declaración de variables

Una **declaración** es una sentencia que asocia un identificador a un objeto o a un proceso para poder referirse a ese objeto o proceso por un nombre en particular. Declarar una variable supone especificar su nombre mediante un identificador y el tipo de dato que va a almacenar y que se denomina **tipo de la variable**. La declaración establece, además de las características del objeto declarado (identificador y tipo), una zona de memoria para almacenar los valores que tomará dicho objeto.

La declaración de variables se hará siempre antes de usarlas y de acuerdo con el siguiente formato:

```
Tipo Nombres_Variables
```

Donde *Tipo*, se refiere a alguno de los tipos de datos simples descritos con anterioridad: entero, real, carácter, cadena o booleano. Y con *Nombres_Variables*, se especifican los identificadores que se desea asignar a variables que son todas del mismo tipo de datos y que se separarán por comas. Los nombres de variables son elegidos por el programador y deben seguir las reglas establecidas para los identificadores. Recuerde que las *palabras reservadas*, identificadores propios de un lenguaje, no pueden usarse como identificadores, por tanto, tampoco como nombres de variables, si se va a codificar en ese lenguaje.

Ejemplos:

```
Entero numero, edad, cantidadMesas
Real precio
```

En este ejemplo, declaramos tres variables que podrán almacenar datos de tipo entero y cuyos identificadores son `numero`, `edad`, `cantidadMesas`, y una variable para almacenar datos de tipo real cuyo identificador es `precio`.

5.2 Tipos de datos de las variables

Según los tipos de datos que almacenan, las variables pueden ser de los siguientes tipos:

a) **Numéricas**: Si almacenan datos de tipo numéricos. De acuerdo con los tipos de datos numéricos las variables también podrán ser:

- **Enteras** Si almacenan un dato numérico de tipo entero.

  Ejemplos:

```
Enteras edad, j, numero
edad = 35
j = 7
numero = 136
```

- **Reales**. Si almacenan un dato numérico de tipo real.

  Ejemplos:

```
Real precio, total
precio = 25.25
total = 0
```

b) Alfanuméricas.Si almacenan datos de tipo alfanuméricos. Según esto las variables pueden ser de tipo:

- **Carácter**. Si almacenan un sólo carácter.

  Ejemplos:

```
Carácter estadoCivil, diaSemana
estadoCivil = 'S'
diaSemana = 'L'
```

- **Cadenas**. Si almacenan una secuencia de caracteres.

  Ejemplos:

```
Cadenas nombreAlumno.
nombreAlumno = "Angel Clemente"
```

c) **Booleanas**. Si almacenan tipos de datos lógicos.

Ejemplos:

```
Booleanas encontrado, final
encontrado = verdadero
final = falso
```

Con la asignación se almacena en una zona de memoria el valor que aparece a la derecha del símbolo "=". No solamente pueden asignarse valores literalmente expresados, sino que también se puede asignar a una variable el valor de otra variable o de una expresión, siempre que las variables y los valores asignados sean del mismo tipo de datos o de tipos de datos compatibles.

Ejemplos:

- Asignaciones correctas dadas las declaraciones de los ejemplos anteriores:

```
edad = 35
numero = 136
edad = numero
```

A partir de este momento, el valor almacenado en la variable Edad es el de la variable Numero, o sea, 136. Para dar otro valor a la variable bastará con realizar otra asignación.

```
edad = 12
encontrado = final
total = precio *4
nombreAlumno = "Carlos Miranda"
j = j +1
```

Observe en el último ejemplo que la misma variable puede aparecer a ambos lados del símbolo de asignación. Esto no debe leerse como una igualdad matemática, sino como la orden "almacénese en la dirección de memoria llamada **j** el valor que contiene en ese instante más una unidad". Como habíamos dado a **j** el valor 7, a partir de aquí el valor almacenado en **j** es 8.

- Asignaciones incorrectas, dadas las declaraciones de los ejemplos anteriores:

```
edad = 35.5
numero = encontrado
edad = letra
encontrado = precio
nombreAlumno = 'Z'
```

5.3 Inicialización de variables

Inicializar una variable significa asignarle un valor inicial. Esta operación suele hacerse después de su declaración y en las primeras líneas de código o a la vez que se declaran siguiendo los siguientes formatos:

Nombre_Variable = Valor_Inicial en el caso de hacerlo una vez declaradas.

Tipo Nombres_Variables [= Valor_Inicial] si se inicializan a la vez que se declaran.

Donde *Tipo* se refiere a cualquiera de los tipos de datos básicos; *Valor_Inicial* representa aquellos valores que toman las variables al comienzo del algoritmo. Los corchetes se usan para simbolizar que la operación que se encuentra entre ellos es opcional. En este caso, significa que se pueden dar (o no dar) valores de comienzo a las variables que se van a utilizar.

Ejemplos:

```
enteras edad = 35, numero
reales precio, iva = 11.5
carácter estado_civil = ' ', dia_semana = ' '
cadenas nombreAlumno
booleanas encontrado = verdadero
```

```
numero = 0
nombreAlumno = "                "
```

Existe la costumbre de inicializar las variables numéricas a 0 y las alfanuméricas a espacio o espacios en blanco, ya que con esto nos aseguramos que esa zona de memoria no contenga otra información que pudiera influir posteriormente en los valores tomados por esas variables. Esto se conoce como **limpiar** esa zona de memoria. Al valor indeterminado que tiene una variable cuando se declara y no se inicializa, se le llama **basura**.

Por ejemplo, suponga que *numero* no ha sido inicializada y que en un punto de nuestro programa se hace la operación *numero = numero +1*, con la intención de que la variable aumente de uno en uno para contar objetos. Ahora habrá almacenado en la dirección de memoria referenciada por *numero* lo que contenía antes más 1 y ¿qué contenía antes?, pues no se sabe, pero se puede asegurar que basura desde el punto de vista de nuestro ejercicio o lo que es igual, un dato no significativo. Suponga que es el ASCII correspondiente a la letra B, cuyo decimal es 66, a partir de la asignación *numero* tendrá almacenado el entero 67.

6. CONSTANTES

Se llama **constante** a las entidades de datos que no cambian su valor durante la ejecución del programa.

Las constantes pueden utilizarse en los algoritmos y en los programas de dos formas diferentes:

a) Como **constantes literales**, en cuyo caso su valor es lo que expresa su representación. Por ejemplo, si queremos hallar el área de un círculo de radio R, escribimos *area* = 3.14 * (radio $\wedge$ 2). En este caso, decimos que 3.14 es una constante con valor literal o, simplemente, un literal. Las constantes literales no necesitamos declararlas, sólo usarlas en el momento deseado.

b) **Constantes simbólicas**. En vez de escribir las constantes literalmente cada vez que aparecen en nuestros programas y dado que tenemos la posibilidad de asignar identificadores a los objetos que usamos, también es posible dar a cada constante un nombre y usar dicho nombre para referirnos a ellas a lo largo del programa. Las constantes así utilizadas se llaman constantes simbólicas o **constantes con nombre**. Esta manera no es más que otra forma de representar los literales y presenta la ventaja de hacer los algoritmos y programas mas fáciles de leer y de modificar.

Las constantes simbólicas se declaran antes de usarlas, asociando el nombre al literal correspondiente mediante una asignación, según el siguiente formato:

```
Nombre_Constante = valor_literal_asociado
```

Ejemplos:

```
PI = 3.14
GRAVEDAD = 9.8
```

A partir de aquí, en aquel lugar del programa donde debieran aparecer estos literales, escribiremos los nombres que los identifican.

```
area =  PI * (radio ∧ 2)
peso = masa * GRAVEDAD
```

No hay que definir el tipo de dato de una constante, puesto que se les asocia de manera automática el tipo del literal asignado. En el caso del ejemplo anterior, las constantes simbólicas *PI* y *GRAVEDAD* son constantes reales puesto que 3.14 y 9.8 lo son.

La declaración de una constante simbólica no puede confundirse con una asignación de un valor a una variable.

6.1 Tipos de datos de las constantes

Según los tipos de datos que almacenan, las constantes pueden ser de los siguientes tipos:

a) **Numéricas**. Si almacenan datos de tipo numéricos, por lo que podrán ser de los tipos:

- **Enteras**. Si almacenan un dato numérico de tipo entero.

  Ejemplos:

  ```
  DIASANNO = 365
  MAXIMO = 100
  TIPOIVA = 17
  ```

- **Reales**. Si almacenan un dato numérico de tipo real.

  Ejemplos:

  ```
  TIPOINTERES = 6.5
  9.8
  ```

b) **Alfanuméricas**. Si almacenan datos de tipo alfanuméricos. Podrán ser:

- **Carácter**. Si almacenan un sólo carácter.

  Ejemplos:

  ```
  NACIONALIDAD = 'E',
  CODIGOAUTONOMICO = 'A'
  'S'
  ```

- **Cadenas**. Si almacenan una secuencia de caracteres.

  Ejemplos:

```
NOMBREPAIS = "España"
MENSAJE = " error de división"
"FIN"
```

Observe que las constantes literales de tipo carácter y las de tipo cadena van entre comillas simples y dobles respectivamente. La razón es para diferenciarlas de los identificadores. Esto no ocurre con los literales numéricos, puesto que un identificador no puede comenzar por un dígito numérico y un número no puede contener caracteres.

Se recomienda encarecidamente el uso de constantes simbólicas en los programas, en vez de utilizar literales, siempre que las constantes tengan un significado útil, se vayan a usar con cierta frecuencia y sean susceptibles de cambio, con el fin de aumentar la legibilidad y facilitar la modificabilidad.

7. EXPRESIONES

Una **expresión** está formada por constantes, variables, operadores o un conjunto de ellos. Los operadores permitidos en una expresión dependen de los tipos de datos de los objetos que forman la expresión. Las normas concretas para escribir expresiones correctas las incorporan los distintos lenguajes de programación pero, habitualmente, siguen las reglas generales del álgebra.

Dadas las siguientes declaraciones:

```
CONSTANTES:
   PI =3.14
VARIABLES:
   enteros edad=0
   real peso=12, area, radio, precio
   carácter nacionalidad
```

Las siguientes son todas expresiones válidas:

```
nacionalidad ='e'
area =  PI * (radio ∧ 2)
(36 = (6*6)) y ("mamá" < > "mamá")
edad >= 30;
(2+peso)/3;
precio = 4.5*peso + 34
```

Una **instrucción** o **sentencia** es una combinación de constantes, variables, operadores e, incluso, palabras reservadas, que siguiendo la sintaxis propia de un lenguaje de programación, permite que el procesador realice una determinada acción. Cuando se escriben los algoritmos, en nuestro caso en español, también las sentencias tienen una sintaxis propia que facilitará su traducción al lenguaje de compilación elegido.

Por ejemplo, la siguiente expresión está perfectamente expresada como instrucción en un algoritmo escrito en Español: `area =  PI * (radio ∧ 2).`

Pero, si se desea que sea una instrucción codificada en PASCAL estándar, habrá que escribir `area := PI * sqr(radio)`

Observe que el símbolo del operador de asignación es distinto.

En JAVA `area = PI * (radio * radio);`

Observe que en JAVA tan sólo se ha añadido un punto y coma a la expresión para convertirla en instrucción. JAVA no tienen operador potencia, pero si la función **pow** que realiza esta operación.

Debe tenerse en cuenta que son los propios lenguajes de programación los que tienen incorporadas las normas que rigen la formación de expresiones e instrucciones y que es a los manuales donde hay que dirigirse cuando se decida codificar algoritmos en un lenguaje en particular. En capítulos posteriores, se dirá que son expresiones válidas para el lenguaje Java.

8. ALGORITMOS

8.1 Diseño de algoritmos

Una vez analizado y comprendido el problema que se desea informatizar y escritas detalladamente todas las especificaciones, se pasa a diseñar un **algoritmo** que lo resuelva. La forma de solucionar la mayoría de los problemas no es única. Las especificaciones detalladas de los requerimientos que debe cumplir un programa dice *qué* debe hacer pero no *cómo* debe hacerse.

Para diseñar un algoritmo se pueden utilizar estrategias diferentes. Por ejemplo las siguientes:

Diseño Top-Down

Se puede utilizar la estrategia de **diseño descendente** o **top-down** (de arriba hacia abajo). Este sistema basado en el lema *divide y vencerás*, parte el problema en un primer paso en otras tareas menores que la primera y que se denominan **módulos**. De esta forma, se obtiene un **módulo principal** (llamado también **programa principal**) que se divide a su vez en unidades menores o módulos menores llamados **subprogramas** y así sucesivamente..

La intención del método es la de *retrasar los detalles tanto como sea posible* a medida que se pasa de una solución general a una solución específica. Este método constituye una filosofía de diseño orientada a los procesos y es la base de la **programación estructurada**.

Sin embargo, podemos utilizar otra estrategia acorde con la metodología de **programación orientada a objetos** que propone una nueva filosofía de programación orientada a la estructura de los datos.

Diseño Bottom-Up

En este diseño se trata de identificar como es la estructura de datos del programa y que interacciones aparecen entre los distintos datos. A partir de los datos y sus interacciones se desarrollan las **funcionalidades** que producirán las salidas adecuadas.

El programa se compone de una serie de objetos con un estado interno propio y que interactúan intercambiando mensajes.

Un ejemplo seria un juego de cartas:

- Para realizar un juego de cartas empezamos diseñando los objetos carta, mazo, tapete, jugador, etc.
- Los objetos se prueban de forma independiente y posteriormente se integran para formar el juego.
- Los objetos son fácilmente reutilizables para cualquier otro juego de cartas.

La complejidad de un algoritmo aumenta exponencialmente con su tamaño, medido usualmente en líneas de código. Por ello, nuestro objetivo es trabajar siempre con algoritmos cortos, que podamos comprender, construir y depurar más cómodamente.

Como en los primeros temas trabajaremos con entidades de datos simples no será necesario aplicar el diseño Bottom-Up puesto que no diseñaremos objetos como tales, por lo que se utilizará la metodología Top-Down. Cuando se hayan conseguido las destrezas suficientes con las estructuras de control de flujo tomaremos el enfoque orientado a objetos.

Conviene recordar que la técnica Top-Down es utilizable en la resolución de **funcionalidades** en cualquier paradigma de programación (estructurada u orientada a objetos en nuestro caso).

Independientemente de la filosofía utilizada los algoritmos, al igual que los programas, constarán fundamentalmente de dos partes:

1. **Descripción de entidades de datos**. Donde se dará una relación de las que van a utilizarse. Si los datos son simples se especificarán sus tipos y se inicializarán si fuera necesario. Se añadirán comentarios que expliquen el cometido de esos objetos, siguiendo, por ejemplo, el siguiente orden:

```
Constantes:

Variables:

Otros objetos:
```

2. **Descripción de instrucciones**. Se detallarán una a una y en orden, todas las sentencias que deben ser ejecutadas por el ordenador para la resolución

del problema. De momento, se hará en lenguaje natural y, más adelante, lo se codificarán.

```
Instrucción 1
Instrucción 2
Instrucción 3
.
.
.
Instrucción n
```

8.2 Características de los algoritmos

Si bien es cierto que existen diferentes formas de dar solución a un problema, es decir, de construir algoritmos, cada uno con sus ventajas e inconvenientes, no todos deben considerarse buenos. Existen una serie de características que los algoritmos deben cumplir para ser considerados correctos y que nos ayudarán a elegir el más adecuado:

- **Eficientes**. Deben ocupar la mínima memoria y minimizar el tiempo de ejecución.

- **Legibles**. El texto que lo describe debe ser claro, de forma que permita entenderlo y leerlo fácilmente.

- **Modificables**. Estarán diseñados de modo que sus posteriores modificaciones sean fáciles de realizar, incluso por programadores diferentes a sus propios autores.

- **Reutilizables.** Los objetos o las funcionalidades bien diseñados pueden utilizarse como base de otros sistemas, que se constituyen como una combinación de los objetos o funcionalidades ya existentes.

- **Modulares.** La filosofía utilizada para su diseño debe favorecer la división del problema en entes más sencillos.

- **Único punto de entrada, único punto de salida**. A los algoritmos y a los módulos que lo integran, se entra por un sólo punto (inicio), y se sale por un sólo punto también (fin).

Cualquiera de estas propiedades no siempre pueden llevarse a su máximo grado de cumplimiento si no es en detrimento de alguna otra. Quiere esto decir que será el programador el que marque el grado de compromiso entre todas ellas para conseguir buenos programas.

8.3 Estructuras de programación

En el intento de dar solución informática a un problema mediante un algoritmo, las instrucciones pueden relacionarse de alguna de las formas siguientes:

- **Estructura lineal o secuencial**. Las instrucciones se ejecutarán una a una según el orden en que han sido escritas.

  Ejemplo:

  ```
  PI = 3.14
  RADIO = 5
  AREA = PI*(R∧2)
  ```

  Ejecutadas estas tres sentencias o instrucciones, el valor de AREA será 78.5

  En el ejemplo de la tortilla de patatas de la unidad anterior, tenemos la siguiente secuencia lineal de instrucciones:

  1. Pelar las patatas
  2. Cortarlas a cuadraditos
  3. Freírlas, no demasiado, en abundante aceite (de oliva)
  4. Batir una cantidad de huevos suficiente
  5. Añadir las patatas fritas y escurridas
  6. Poner sal al gusto
  7. Echar la mezcla en una sartén untada de aceite

- **Estructura repetitiva o cíclica**. Un grupo de instrucciones se ejecutará varias veces.

  Ejemplo: En el ejemplo de la tortilla de patatas del tema anterior, podíamos haber añadido antes del punto 8:

  ```
  Repetir
      8. Dejar que se cuaje unos minutos
      9. Dar la vuelta a la mezcla con ayuda de un plato
  Hasta que la tortilla haya cuajado a nuestro gusto
  ```

  Las instrucciones 8 y 9 se repetirán el número de veces necesario para que la tortilla esté a nuestro gusto.

- **Estructura alternativa**. Situaciones que provocan la ejecución de instrucciones diferentes en función de que se cumplan o no ciertas condiciones establecidas.

  Siguiendo con el ejemplo de la tortilla de patatas, podíamos haber añadido al principio:

  ```
  Inicio
  Si tenemos patatas, aceite y huevos y todos los
       utensilios
  ```

```
        Pelar las patatas
        Cortarlas a cuadraditos
        Freírlas, no demasiado, en abundante
        aceite (de oliva)
        Batir una cantidad de huevos suficiente
        Añadir las patatas fritas y escurridas
        Poner sal al gusto
        Echar la mezcla en una sartén untada de
        aceite
        Repetir
            Dejar que se cuaje unos minutos
            Dar la vuelta a la mezcla con ayuda de
            un plato
        Hasta que la tortilla haya cuajado a
        nuestro gusto
    En caso contrario
        Mensaje: "Debemos comprar los
        ingredientes"
    FinSi
    FIN
```

8.4 Formas de expresar los algoritmos: Pseudocódigo

Existen distintas maneras de representar los algoritmos, entre las que se encuentran las siguientes:

- **Diagramas de flujo**. Esta técnica utiliza símbolos gráficos para la representación del algoritmo. Se encuentra en desuso en el ámbito de la programación por acarrear graves inconvenientes, pero suele utilizarse en las fases de análisis. Entre estos inconvenientes, se puede contar la obligación de reorganizar todo el dibujo en el caso de que se deba efectuar la mínima modificación, o las dificultades para leer y seguir la traza del algoritmo.

- **Pseudocódigo**. Es una técnica de representación de algoritmos que utilizando palabras claves en lenguaje natural (el español en nuestro caso), constantes, variables, otros objetos, instrucciones y las estructuras de programación, expresan de forma escrita la solución del problema. Es la técnica más usada actualmente y la que se utilizará en este libro.

Una premisa muy importante: un algoritmo bien escrito en pseudocódigo es lo suficientemente claro como para que cualquier programador, poco experimentado, pueda comprender cuál es la función que realiza.

La estructura general para la resolución informática de un problema será la siguiente:

1. Análisis del problema

El análisis indica la especificación de requisitos del cliente. Se realizará un estudio previo de la tarea a realizar. Si la tarea es descrita verbalmente, se harán preguntas adecuadas hasta tener claro lo que se ha de diseñar, es decir, hasta obtener los **requisitos**, es decir, **escritura detallada de todas las especificaciones**. Si la tarea se obtiene por escrito, se aconseja comenzar subrayando palabras o frases decisivas, escribiendo al margen ideas y dudas, o cualquier otra cosa que ayude a concretar especificaciones, además de realizar verbalmente todas las preguntas que den respuesta a nuestras dudas.

Algunas preguntas elementales serán las siguientes:

- ¿Cuáles son los datos de entrada? ¿cómo se leerán?
- ¿Qué salida debe producir el programa? ¿dónde se van a escribir?
- ¿Qué apariencia (formato) tendrán los datos, tanto de entrada como de salida?
- ¿Qué pasos son necesarios para poder procesar la entrada hasta llegar a la salida?
- ¿Cuántas veces se deben repetir los procesos?
- ¿Hay que hacer suposiciones?. En caso afirmativo se hará un listado de ellas.
- ¿Pueden aparecer condiciones especiales de error?
- ¿Se ha resuelto algún problema parecido o se sabe que existe una solución para una tarea análoga?. Si la respuesta es afirmativa, utilícese esa solución, aunque hubiera que modificarla. Por ejemplo, hallar la máxima y mínima temperatura es un proceso idéntico al cálculo de la mayor y menor nota de un examen.
- ¿Hay juegos de ensayo adecuados?. Si la respuesta es afirmativa, búsquense para usarlos.

2. Diseño del programa. Escribir en pseudocódigo el programa principal

En la fase de diseño, la especificación de requisitos se convierte en un diseño detallado del algoritmo.

Para un diseño orientado a objetos, la salida de la fase de diseño será un conjunto de clases/objetos que satisfacen los requisitos. Las clases/objetos deben estar completamente definidas, mostrando cómo deben comportarse y cómo se comunican entre sí. Y además habrá que detallar la estructura física del Programa principal.

Para un diseño estructurado, la salida de esta fase será la descripción de todos los módulos/subprogramas que formarán la estructura física del programa y las relaciones entre ellos, de manera que satisfacen los requisitos establecidos.

Independientemente a la filosofía de diseño el programa principal será una secuencia ordenada de instrucciones.

En los primeros ejercicios tan sólo tendremos una clase principal, por tanto el diseño será similar al estructurado. Se empezará dividiendo el problema en módulos más sencillos, hasta llegar a un nivel de detalle razonable, utilizando para ellos identificadores con significado. Si de momento no se sabe resolver alguno de estos módulos, no debe preocuparse, piense que alguien lo resolverá o que mañana tendrá una idea genial para solucionarlo y siga adelante con el diseño. Tal vez incluso, tenga la suerte de encontrarlo ya hecho.

3. Escribir en pseudocódigo los módulos restantes

A continuación y en caso de que los hubiera, se escribirá cada uno de los módulos, detallando la secuencia de instrucciones que deben realizar. Se hablará más sobre este particular en capítulos posteriores.

4. **Prueba**

- **Lectura del pseudocódigo**. Recorrer el algoritmo escrito en pseudocódigo inspeccionando, reordenando, detectando y corrigiendo errores, revisando todo lo que sea necesario. Puede incluso que, tras la inspección, haya que planificar algún cambio. No hay que tener miedo a empezar de nuevo si fuera necesario.

- **Seguir la traza**. Se hará la prueba del algoritmo siguiendo la traza con los datos elegidos como juego de ensayo. Observe los resultados minuciosamente, tratando de detectar y corregir los errores si estos resultados no son los esperados.

5. Optimización del algoritmo

Después de disponer del algoritmo formalmente validado y acorde con las especificaciones, es conveniente estudiar si puede obtenerse una mejora de la solución adoptada para que sea más eficiente, más fácil de leer y su interfaz sea más amigable.

La inclusión de mejoras puede remitir de nuevo a alguna de las fases anteriores, por lo que aumentará el coste del producto. Por tanto, habrá de estudiar concienzudamente si vale la pena la inversión en estas mejoras.

6. Documentación

Una vez diseñado un algoritmo, la siguiente actividad a desarrollar, sumamente importante en programación, es producir programas bien

documentados, para que se cumplan los objetivos básicos: que los programas sean legibles, comprensibles y fácilmente modificables.

La documentación incluye descripciones, comentarios, especificaciones, incluso un breve manual de uso si el algoritmo es suficientemente extenso.

Hay dos tipos de documentación en el desarrollo de algoritmos bien diseñados:

a) **Documentación externa**. Incluye la escritura de información que es exterior al cuerpo del algoritmo. Puede incluir, además de todas las especificaciones detalladas, una explicación extensa del desarrollo del algoritmo y de las modificaciones sufridas. Deben añadirse igualmente descripciones detalladas del problema, de los módulos que lo componen, adjuntando un diagrama jerárquico (ver capítulo 6, 6.2) del diseño descendente y, por último, un manual de usuario donde se explicará al cliente (o al profesor en su defecto) como debe usarse el programa. Esta documentación debe adjuntarse sólo en el caso de que los algoritmos sean suficientemente grandes, lo que se consideran *aplicaciones informáticas* de cierta envergadura, divididas en varios programas. Existen métodos, que no son objetivos de este libro, que enseñan técnicas de análisis adecuadas para la construcción de grandes aplicaciones, donde se incluyen, además, sistemas para desarrollar una buena documentación.

b) **Documentación interna.** Incluye la escritura de información que se suministra con el cuerpo del algoritmo. Estará formada por comentarios, código autodocumentado e impresión agradable del texto del pseudocódigo. Se va a ver cada uno de ellos por separado:

- **Comentarios en línea**

  Los comentarios son frases cortas en línea con expresiones del pseudocódigo, que ayudan al lector a comprender lo que el programador pretende en ese punto del programa. No es necesario comentar cada sentencia, de algunas resulta evidente su cometido y comentarlas, además de inútil por no añadir claridad al texto lo sobrecargan. Por ejemplo, no tiene sentido comentar las siguientes sentencias:

```
carácter estadoCivil = ' '    /* se declara e
inicializa la variable estadoCivil */

contadorAlumnos = contadorAlumnos +1    /* Se
incrementa en 1 el contador de alumnos */
```

  En general sería bueno comentar los siguientes puntos:

  - En las declaraciones. Explicar para qué sirve cada objeto declarado.

Ejemplo:

```
Entero contadorAlumnos  /* Cantidad de
alumnos del Centro*/
Real precio/* Precio unitario del
artículo*/
```

- En las estructuras de programa. Poner un comentario en la misma línea o antes de entrar en un ciclo o en una estructura alternativa para explicar su cometido.

Ejemplo:

```
/*Estructura alternativa para evitar que
se intente hacer la tortilla sin
ingredientes*/
Si tenemos patatas, aceite y huevos
.........................
.............
En caso contrario
Mensaje: "Debemos comprar los
ingredientes"
FinSi
/* ciclo para evitar que la tortilla se
pegue y asegurar que saldrá a nuestro
gusto*/
Repetir
Dejar que se cuaje unos minutos.
Dar la vuelta a la mezcla con ayuda de
un plato.
Hasta que la tortilla haya cuajado a
nuestro gusto
```

- En las llamadas a los módulos del programa.

- Antes de comenzar a escribir el código de cada módulo, es decir, grupo de instrucciones que lo forman.

- Estos dos últimos casos se comentará en capítulos posteriores.

- Cuando el código no es evidente, conviene comentar cualquier punto del programa cuya lectura y comprensión resulte dificultosa.

En un primer momento y hasta que el futuro programador tenga la suficiente experiencia en programación se aconseja que los comentarios se añadan al pseudocódigo, más adelante bastará con que los incluya en el código fuente del programa.

- **Codigo Autodocumentado**

  Sobre este particular ya se ha hablado con anterioridad. Si se utilizan nombres de identificadores con significado para dar idea de la función que realizan, se dispondrá de más información sobre las tareas desempeñadas por los algoritmos. En este caso se dice que el código del programa está autodocumentado.

  Ejemplo: Objetos con los nombres que aparecen en las siguientes declaraciones no dicen nada sobre su cometido.

```
CONSTANTES:
     C = 3.14
     N = 9.8
VARIABLES:
     Reales X = 0.0, Y = 0.0
```

  Sin embargo, los que aparecen a continuación, dan mucha mayor información:

```
CONSTANTES:
     PI = 3.14
     gravedad = 9.8
VARIABLES:
     Reales PESO = 0.0, RADIO = 0.0
```

- **Impresión Agradable**

  Esta característica es conocida también como *formateado de programas*. Se trata de *dar forma*, escribir el pseudocódigo y, más adelante, los programas, de manera tal que aumente su grado de legibilidad y comprensión. El objetivo es que los programas resulten fáciles de entender, añadiendo al texto los espacios o líneas en blanco que nos parezcan adecuados o usando una indentación apropiada. Tenga en cuenta que cuando se traduzca el pseudocódigo a C, al compilador no le afecta en absoluto ni espacios, ni líneas en blanco, ni tabulaciones, sólo le preocupa la corrección sintáctica de los programas.

  Ejemplo: El siguiente pseudocódigo es funcionalmente correcto, pero difícil de leer.

```
Suponemos que: Disponemos de todos los
ingredientes: patatas, huevos, aceite y
sal Disponemos de todos los utensilios:
sartén, plato, espumadera y tenedor.
Sabemos: pelar, cortar, freír, batir,
Añadir,...Realización: Pelar las
patatas. Cortarlas a cuadraditos.
Freírlas, no demasiado, en abundante
aceite (de oliva)...
```

Por el contrario, escrito de la forma en que aparece en el capítulo 1, la lectura es más cómoda.

Se observará a lo largo de los ejemplos el estilo adoptado en este libro para escribir pseudocódigos y programas, y se anima al lector a adoptarlo como propio.

ESTRUCTURA GENERAL DE UN ALGORITMO ESCRITO EN PSEUDOCÓDIGO

Se verá a continuación una forma de detallar las distintas partes de las que consta un algoritmo escrito en pseudocódigo. A este respecto, cabe decir que no son éstas unas reglas estándares, sino que pretenden ser una guía para los alumnos que comienzan su aprendizaje en programación. Ésta no es más que una manera, más o menos extendida, que hará que cualquier programador sin demasiada experiencia pueda comprender las funciones que realizan los algoritmos descritos.

El algoritmo escrito en pseudocódigo tendrá la siguiente estructura:

PROGRAMA: Nombre_del_programa.

Comentario: *Cualquiera que nos resulte significativo para nuestro algoritmo.*

ANÁLISIS:

Discusión y comprensión del problema.
Propósito: *Breve descripción de la tarea realizada por el algoritmo.*
Entrada: *Cuáles serán los datos de entrada al programa.*
Salida: *Cuál será la información de salida del programa.*
Suposiciones: *Todas aquellas que vayan a configurar el entorno en el que se desarrolla el algoritmo.*
ENTORNO:
CONSTANTES:
Declaraciones e inicializaciones de constantes. Comentario
VARIABLES:
Declaraciones e inicializaciones de variables. Comentario
OTROS OBJETOS:
Declaraciones de los módulos, si los hay u otros objetos que vayamos a usar (los veremos en capítulos posteriores). Comentario

Programa principal

```
Inicio
.
.
<instrucciones que forman el programa>
.
.
<llamadas a los módulos, si los hubiera>
.
```

```
.
Fin del Programa Principal
.....................................

NOMBRE_MODULO_1
Inicio
Entorno del módulo 1
.
.
Fin NOMBRE _MODULO_1
.
.
.
NOMBRE_MODULO_N
Inicio
Entorno del módulo n
.
.
Fin NOMBRE _MODULO_N
```

El nombre del programa lo elige el programador y se aconseja que sea alusivo a la tarea realizada por el algoritmo. Se hará a continuación una breve descripción del proceso principal resuelto en el programa.

Junto a los nombres de constantes, variables y otros objetos, se hará también un breve comentario para informar de su cometido.

Debe tenerse en cuenta que los algoritmos (y programas) escritos en lenguajes imperativos suelen realizar tres tareas de forma secuencial:

1.- Entrada de datos

2.- Procesamiento de los datos

3.- Salida de resultados

Los dos últimos puntos están recogidos por `<instrucciones que forman el programa>` y por `<llamadas a los módulos, si los hubiera>`

Podemos representar las distintas partes de un programa con el siguiente gráfico:

Figura 2.2. Proceso general de un programa

Se detallan a continuación, las instrucciones para la entrada de datos desde el teclado (dispositivo de entrada estándar) y las instrucciones para la salida de resultados hacia el monitor (dispositivo de salida estándar). Las instrucciones correspondientes a la parte de *Proceso*, se estudiarán en el capítulo siguiente.

9. INSTRUCCIONES DE ENTRADA

Las instrucciones o sentencias de entrada se utilizan en los algoritmos para tomar datos del exterior y depositarlos en memoria principal, es decir, almacenarlos en las variables para su posterior proceso. Para la entrada de datos, se seguirá la siguiente sintaxis o formato:

```
Leer (Nombres_Variables)
```

Se entiende que el ordenador "está leyendo" lo que se escribe a través del teclado y lo almacena en las variables que aparecen entre paréntesis.

Leer es una palabra que se reserva para la instrucción de entrada. *Nombres_Variables*, son los nombres de las variables que se van a leer separadas por comas. Los nombres de estas variables especifican donde han de colocarse los valores leídos. Estos valores sólo podrán ser cambiados por alguna asignación a lo largo del algoritmo o por otra lectura posterior.

Los pasos correspondientes a la sentencia *Leer* son los siguientes:

1. Obtener dato del terminal.
2. Almacenar el dato en la dirección de memoria referenciada por el nombre de variable correspondiente de la lista.
3. Repetir 1 y 2 mientras queden variables en la lista.

Gráficamente:

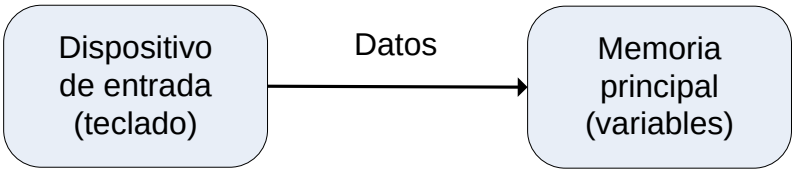

Figura 2.3: Entrada de datos

Ejemplo 1. Resolver las lecturas que aparecen a la izquierda de la tabla, dadas las siguientes declaraciones:

```
enteras edad, numero
real precio, IVA, totalPagar = 0
carácter estadoCivil, diaSemana
cadena nombreAlumno
```

Instrucción	Datos de entrada	Contenido tras la lectura
`Leer (edad, IVA)`	35 11.5	edad 35 IVA 11.5
`Leer (precio)`	25.25	precio 25.25
`Leer (estadoCivil, Numero)`	S 3	estadoCivil S numero 3
`Leer (nombreAlumno)`	Juan Vila	nombreAlumno Juan Vila
`Leer (diaSemana)`	L	diaSemana L

Observe que cuando se introducen por teclado las variables de tipo carácter o cadena, no se escriben entre comillas. Esto sólo se utiliza para las constantes de esos tipos de datos.

No existe ninguna diferencia entre escribir una sola instrucción *Leer* con varias variables en su lista y varias instrucciones *Leer* con una sola variable cada una, salvo la conveniencia para el programador si usa de la primera opción.

Ejemplo 2 En pseudocódigo

Es igual escribir:

```
Leer (edad, IVA)
```

Que escribir:

```
Leer (edad)
Leer (IVA)
```

10. INSTRUCCIONES DE SALIDA

Las instrucciones o sentencias de salida se utilizan en los algoritmos para sacar los resultados de los procesos realizados y que han sido depositados en memoria principal en las variables correspondientes, hacia exterior y escribir mensajes. Para la salida de información, se seguirá la siguiente sintaxis o formato:

```
Escribir (mensajes , Nombres_Variables)
```

Escribir también es una palabra reservada para la instrucción de salida. Los *mensajes* son constantes de cadena que se desean que se impriman tal cual en el monitor, por tanto, en la lista de *Escribir* aparecerán escritos entre dobles comillas. Cuando esta sentencia se codifique en C, observará que las comillas no aparecerán en pantalla. *Nombres_Variables*, son los nombres de las variables cuyos valores se desea escribir separadas por comas.

Gráficamente:

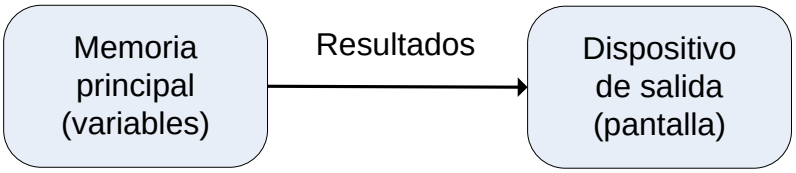

Figura 2.4: Salida de datos

Es muy importante recordar que LOS RESULTADOS son datos producto de alguna operación, es decir, son calculados por el procesador, por tanto, no pueden entrar por el teclado, o lo que es igual **NO SE LEEN**.

Ejemplo 3. Dadas las declaraciones y lecturas del ejemplo anterior:

Instrucciones de proceso	Resultado
`totalPagar= precio +(precio * IVA)/100`	totalPagar 28.15375

Instrucciones de escritura	Salida (se imprime en pantalla)
`Escribir ("Cálculos realizados")`	Cálculos realizados
`Escribir ("Pagará por el artículo", totalPagar)`	Pagará por el artículo 28.15375
`Escribir ("El IVA aplicado es:",IVA)`	El IVA aplicado es: 11.5

11. EJERCICIO PRÁCTICO DE APLICACIÓN

En este ejercicio se observarán todos los ítems estudiados en esta unidad.

Realizar un algoritmo para hallar la media ponderada de dos puntuaciones de exámenes. Los pesos asociados a cada uno de los exámenes serán fijos y las notas serán de tipo entero.

Análisis o discusión del problema: Si tuviéramos los siguientes datos:

Pesos fijos de cada ejercicio
30%
70%

Notas del alumno A 6 y 9 respectivamente

La nota media para el alumno A será : $6 * 0.3 + 9 * 0.7$

Entrada de datos de nuestro algoritmo: dos números enteros correspondientes a las notas de un alumno.

Salida de datos: Saldrán en pantalla los datos de entrada, la media y un texto de cabecera.

Suposiciones: Vamos a suponer que todos los datos de entrada serán correctos, es decir, entrarán números enteros positivos entre 1 y 10.

Pseudocódigo generalizado del algoritmo principal // aplicar la técnica modular o //Top-Down

```
Inicio
 obtenerdatos
 calcularMedia
 mostrarResultados
FinPP
```

Módulos // desarrollar todos los módulos del

```
obtenerdatos
 Escribir ("Introduzca las dos notas")
 Leer (nota1, nota2)
Fin obtenerdatos

calcularMedia
 media = nota1 * PESO1/ 100 + nota2* PESO2/ 100
Fin calcularMedia

mostrarResultados
 imprimirCabecera
 imprimirDatosEntrada
 imprimirMedia
Fin mostrarResultados

imprimirCabecera
 Escribir ("Notas del test           Peso (%)")
 Escribir ("salto de línea") /*Con esto saltará el cursor
                              a la línea siguiente */
Fin imprimirCabecera

imprimirDatosEntrada
 Escribir (nota1, PESO1)
 Escribir ("salto de línea")
 Escribir (nota2, PESO2)
```

```
Fin imprimirDatosEntrada

imprimirMedia
 Escribir ("salto de línea")
 Escribir ("Media ponderada: ---", MEDIA)
Fin imprimirMedia
```

Se realiza a continuación la inspección ocular del pseudocódigo generalizado y de los módulos diseñados. Se puede comprobar en este nivel de diseño que se cumplen los requisitos establecidos en el enunciado, por tanto se pasa a la fase siguiente.

Los nombres de los módulos aparecen como documentación interna en el pseudocódigo detallado, donde se han incluido como comentarios. También aparecen autodocumentados loa nombres de las variables.

Pseudocódigo

```
/*Nombre del programa: MediaTest*/

ENTORNO:
    CONSTANTES:
      PESO1= 50
      PESO2= 20

    VARIABLES:
      entero nota1, nota2
      real media
```

Pseudocódigo detallado

```
<Inicio>
  //obtenerdatos
    Escribir ("Introduzca las tres notas")
    Leer (nota1, nota2)
  //Fin obtenerdatos

  //calcularMedia
    media = nota1*PESO1/100+ nota2*PESO2/100
  //Fin calcularMedia

  //mostrarResultados

    //imprimirCabecera
      Escribir ("Notas del test     Peso (%)")
      Escribir (" saltar línea") /* saltará el cursor
                                 a la línea siguiente.*/
    //Fin imprimirCabecera

    //imprimirDatosEntrada
      Escribir (nota1, PESO1)
      Escribir ("salto de línea")
```

```
        Escribir (nota2, PESO2)
        Escribir ("salto de línea")
        //Fin imprimirDatosEntrada

      //imprimirMedia
        Escribir ("salto de línea")
        Escribir ("Media ponderada: ---", MEDIA)
      //Fin imprimirMedia

    //Fin mostrarResultados
  Fin_PP
```

Se realiza, con papel y lápiz, la traza del algoritmo para observar si la salida se atiene a las expectativas previstas en el análisis, es decir, para asegurar que se cumplen los requisitos especificados.

Aparece el mensaje en pantalla: *Introduzca las dos notas*

Damos a nota1, nota2, los valores 6 y 9 respectivamente.

Se halla la media, media = $6 * 0.3 + 9 * 0.7 = 8.1$

La salida en pantalla es:

Notas del test	Peso (%)
6	30%
9	70%
Media del test	8.1

Se añade como documentación externa el diagrama jerárquico de módulos. Esta es otra de las técnicas para documentar programas.

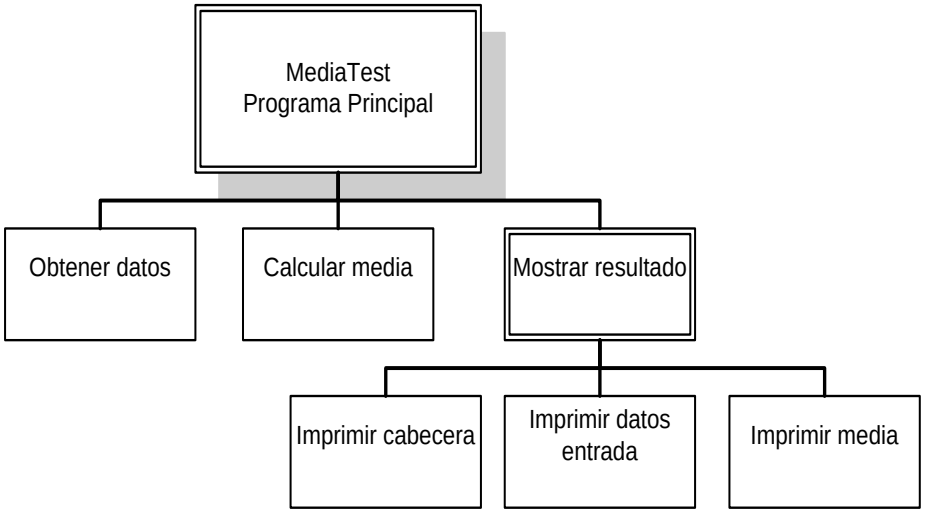

Figura 2.5: Diagrama jerárquico MediaTest