

Unidad 1 **Programación estructurada**

Contenido

1	CONCEPTOS FUNDAMENTALES.....	3
1.1	¿QUÉ ES UN PROGRAMA DE ORDENADOR?.....	3
1.2	CODIFICACIÓN DE LA INFORMACIÓN	3
1.2.1	Códigos.....	3
1.2.2	Código binario.....	4
1.2.3	Código ASCII.....	5
1.2.4	Código hexadecimal.....	5
1.3	UNIDADES DE MEDIDA DE INFORMACIÓN.....	7
1.4	ESTRATEGIAS DE RESOLUCIÓN DE PROBLEMAS	7
1.4.1	Ingeniería del software	7
1.4.2	Ciclo de vida clásico	8
1.4.3	El papel del programador.....	10
1.5	ESTILOS DE PROGRAMACIÓN	10
1.5.1	Programación convencional.....	10
1.5.2	Programación estructurada.....	10
1.5.3	Programación modular	10
2	LOS DATOS.....	11
2.1	TIPOS DE DATOS	11
2.1.1	Tipos de datos simples	11
2.1.2	Tipos de datos complejos.....	12
2.2	OPERACIONES CON DATOS	13
2.2.1	Operaciones aritméticas.....	13
2.2.2	Operaciones lógicas (o booleanas).....	13
2.2.3	Prioridad de los operadores.....	14
2.2.4	Funciones.....	15
2.3	CONSTANTES Y VARIABLES	16
2.3.1	Identificadores.....	16
2.3.2	Declaración y asignación.....	16
2.4	EXPRESIONES.....	17
3	LOS ALGORITMOS	17
3.1	CONCEPTO DE ALGORITMO	17
3.2	NOTACIÓN DE ALGORITMOS.....	18
3.2.1	Escritura inicial del algoritmo	18
3.2.2	Diagramas de flujo	19
3.2.3	Pseudocódigo.....	20
4	LA PROGRAMACIÓN ESTRUCTURADA.....	23
4.1	TEOREMA DE LA PROGRAMACIÓN ESTRUCTURADA	23
4.2	ESTRUCTURA SECUENCIAL	23

4.3	ESTRUCTURAS SELECTIVAS (CONDICIONALES)	24
4.3.1	Condición simple	24
4.3.2	Condición doble	24
4.3.3	Condición múltiple	25
4.4	ESTRUCTURAS REPETITIVAS (BUCLES)	26
4.4.1	Bucle "mientras"	26
4.4.2	Bucle "repetir"	27
4.4.3	Bucle "para"	28
4.5	CONTADORES, ACUMULADORES, CONMUTADORES	29
4.6	REGLAS DE ESTILO	31
4.6.1	Partes de un algoritmo	31
4.6.2	Documentación	32
4.6.3	Estilo de escritura	32
5	PROGRAMACIÓN MODULAR.....	34
5.1	DESCOMPOSICIÓN MODULAR: ¡DIVIDE Y VENCERÁS!.....	34
5.1.1	Algoritmo principal y subalgoritmos.....	35
5.1.2	Técnicas de descomposición modular	36
5.2	FUNCIONES.....	37
5.2.1	Declaración de funciones.....	38
5.2.2	Invocación de funciones.....	38
5.3	PROCEDIMIENTOS	39
5.4	PASO DE PARÁMETROS	40
5.4.1	Paso de parámetros por valor	40
5.4.2	Paso de parámetros por referencia.....	40
5.4.3	Diferencias entre los métodos de paso de parámetros.....	41
5.5	EL PROBLEMA DEL ÁMBITO	41
5.5.1	Variables locales.....	41
5.5.2	Variables globales.....	42
5.5.3	Los efectos laterales.....	42
5.6	LA REUTILIZACIÓN DE MÓDULOS	42
6	ACTIVIDADES.....	43
	ACTIVIDADES INTRODUCTORIAS	43
	ACTIVIDADES SOBRE PROGRAMACIÓN ESTRUCTURADA	45
	ACTIVIDADES SOBRE PROGRAMACIÓN MODULAR	47

1 Conceptos fundamentales

1.1 ¿Qué es un programa de ordenador?

Empecemos por definir qué es un ordenador como lo entendemos hoy en día, teniendo en cuenta que ésta sólo es una de las muchas definiciones válidas:

“Un ordenador es una máquina digital y sincrónica, con cierta capacidad de cálculo numérico y lógico, controlada por un programa almacenado y con posibilidad de comunicación con el mundo exterior”¹

Veamos cada aspecto de la definición por separado para intentar comprenderla bien:

- * **Máquina digital:** el ordenador sólo maneja señales eléctricas que representan dos estados de información. Estos dos estados, en binario, son el 0 y el 1.
- * **Máquina sincrónica:** todas las operaciones se realizan coordinadas por un único reloj central que envía pulsos a todos los elementos del ordenador para que operen al mismo tiempo.
- * **Tienen cierta capacidad de cálculo:** los ordenadores, normalmente, sólo son capaces de realizar operaciones muy simples, ya sean aritméticas (sumas, restas, productos, etc) o lógicas (comparaciones de números)
- * **Está controlada por un programa almacenado:** significa que los ordenadores tienen guardado internamente un conjunto de instrucciones y las obedecen en el orden establecido por el programador, que es quien ha escrito esas instrucciones.
- * **Se comunica con el mundo exterior** a través de diferentes dispositivos periféricos de entrada (como el teclado, el ratón, el escáner...) o de salida (monitor, impresora...)

Según esta definición de ordenador, podemos deducir que **un programa de ordenador es un conjunto de instrucciones** ordenadas y comprensibles para un ordenador, además de un **conjunto de datos** manipulados por esas instrucciones, de manera que **el ordenador realice alguna tarea**.

Todos los programas deben tener una función específica, es decir, una **tarea** que realizar. Por ejemplo, gestionar las facturas de una empresa (si es un programa de facturación) o acabar con todo bicho viviente (si es un videojuego ultraviolento). Normalmente, el programa deberá alcanzar su objetivo en un tiempo finito, es decir, empieza en un momento dado y termina en otro momento posterior.

Los programas utilizan datos. Un **dato** es una **representación de algún objeto del mundo real** relacionado con la tarea que trata de realizar el programa. Representar los datos en un ordenador suele ser complicado porque, debido a su naturaleza digital, todos los datos deben tener forma binaria, cuando está claro que el mundo real no es binario en absoluto. Por lo tanto, para representar objetos reales en un programa es necesario transformarlos en objetos binarios. Éstos objetos binarios son los que llamamos *datos*.

Por ejemplo, en el programa que gestiona las facturas de una empresa, uno de los muchos objetos del mundo real que se han de manejar es el nombre de los clientes. ¿Cómo representar un nombre compuesto por letras en un ordenador que sólo admite código binario, es decir, ceros y unos? Este es uno de los problemas a los que se enfrenta el programador. Y la cosa se complica con objetos más complejos, como imágenes, sonidos, etc.

Resumiendo: los ordenadores son herramientas muy potentes que pueden resolver problemas muy diversos, pero es necesario programarlas, es decir, **proporcionarles las instrucciones y los datos adecuados**. Y eso es lo que vamos a aprender a hacer en este curso.

1.2 Codificación de la información

El ordenador es una máquina digital, es decir, **binaria**. Antes de proseguir, es conveniente repasar el **código binario** y sus implicaciones. Dedicaremos todo este apartado a hacerlo.

1.2.1 Códigos

Un **código** es un método de representación de la información. Se compone de un conjunto de símbolos, llamado **alfabeto**, y de un conjunto de **reglas** para combinar esos símbolos de forma correcta.

- * **Ejemplo 1: la lengua castellana** es un código. Su alfabeto es el abecedario (a, b, c, d, e ... z), pero los símbolos del alfabeto no se pueden combinar a lo loco, sino que existen unas reglas, y sólo siguiendo esas reglas se codifica correctamente la información, dando lugar a mensajes con sentido. Esas reglas las habéis estudiado en la asignatura de lengua castellana desde la enseñanza primaria.
- * **Ejemplo 2: el código morse** también es un código. Su alfabeto es muy reducido: sólo el punto (.) y la raya (–),

¹ DE GUISTI, Armando; *Algoritmos, datos y programas*, Prentice-Hall, 2001.

pero combinando los dos símbolos correctamente, se puede transmitir cualquier información.

- ★ **Ejemplo 3:** el **sistema de numeración decimal** es un código. Tiene un alfabeto de 10 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9). Combinándolos según ciertas reglas, puede usarse para transmitir información. Pero ojo, no cualquier información, solamente información numérica. Hemos dicho que los códigos sirven para representar información, pero no que tengan que servir para representar toda la información posible. Aunque sólo sirva para los números, el sistema de numeración también es un código.

1.2.2 Código binario

El **sistema de numeración binario** es muy parecido al sistema de numeración decimal; por lo tanto, también es un **código**. La única diferencia con el sistema decimal es la cantidad de símbolos del alfabeto. Si el decimal tiene diez, el binario sólo tiene **dos**: el **0** y el **1**. En todo lo demás son iguales, así que el sistema binario también sirve para **representar información numérica**.

Pero, ¿puede representarse cualquier número con sólo dos símbolos?

La respuesta es sí. El modo de hacerlo consiste en combinar los símbolos 0 y 1 adecuadamente, igual que hacemos con los números decimales. En el sistema decimal contamos así: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Cuando queremos pasar a la siguiente cantidad, empezamos a agrupar los dígitos de dos en dos: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19. Al volver a terminar las unidades, vamos incrementando las decenas: 20, 21, 22, etc.

(Esto se debe a que, en los sistemas de numeración, cada dígito tiene un **valor posicional**, es decir, tiene un valor diferente dependiendo del lugar que ocupe en el número general. Por ejemplo, en el número 283, el 3 tiene valor de *tres*, pero el 8 no tiene valor de *ocho*, sino de *ochenta*, y el 2 no tiene valor de *dos*, sino de *doscientos*)

En binario, el razonamiento es el mismo. Empezamos a contar por 0 y 1, pero entonces ya hemos agotado los símbolos, así que empezamos a agruparlos: 10, 11. Como hemos vuelto a agotarlos, seguimos combinándolos: 100, 101, 110, 111, 1000, 1001, 1010, y así sucesivamente.

Así, los 16 primeros números binarios comparados con sus equivalentes decimales son:

Decimal	Binario	Decimal	Binario	Decimal	Binario	Decimal	Binario
0	0	4	100	8	1000	12	1100
1	1	5	101	9	1001	13	1101
2	10	6	110	10	1010	14	1110
3	11	7	111	11	1011	15	1111

Los números escritos en código binario tienen el mismo valor que en decimal, y sólo cambia la representación. Es decir, “15” en decimal y “1111” en binario representan exactamente a la misma idea: *quince*.

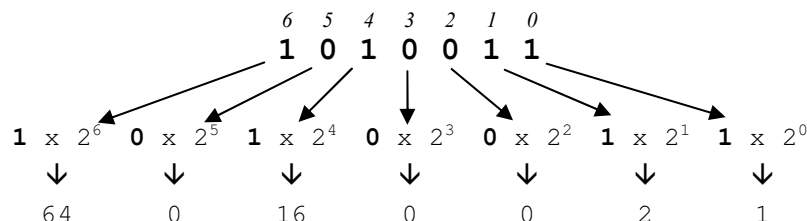
Convertir números binarios a decimales

Para obtener la representación decimal de un número binario hay que proceder según el Teorema Fundamental de la Normalización, del siguiente modo:

1) Numeramos la posición que ocupa cada dígito binario de derecha a izquierda, empezando por 0. Por ejemplo, en el número binario 1010011, numeraremos las posiciones así:

$$\begin{array}{cccccccc} 6 & 5 & 4 & 3 & 2 & 1 & 0 & \leftarrow \text{Posiciones de los dígitos} \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & \end{array}$$

2) Multiplicamos cada dígito binario por 2 elevado a la posición del dígito y sumamos todos los resultados. Con el número del ejemplo anterior:



Ahora sólo nos quedaría sumar los resultados de todas las multiplicaciones:

$$64 + 0 + 16 + 0 + 0 + 2 + 1 = 83$$

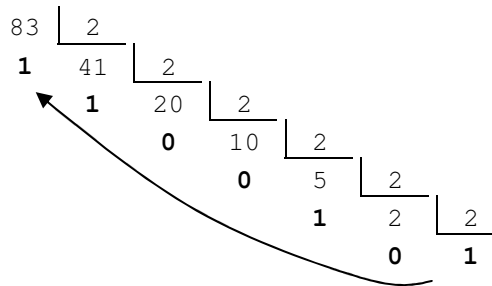
Por lo tanto, el número binario 1010011 es equivalente al número decimal 83. Es habitual indicar con un **subíndice** el sistema de numeración al que pertenece cada número, así:

$$1010011_2 = 83_{10}$$

Convertir números decimales a binarios

El proceso contrario se realiza **dividiendo sucesivamente** el número decimal **entre dos**, y cogiendo **el último cociente y todos los restos en el orden inverso al que los obtuvimos**.

Por ejemplo, vamos a hallar la representación binaria del número 83_{10} :



Tomando el último cociente (que siempre es 1) y todos los restos desde el último hacia el primero (es decir, 010011, siguiendo la dirección de la flecha), obtenemos el número binario 1010011. Por lo tanto, podemos decir que:

$$83_{10} = 1010011_2$$

Operaciones aritméticas binarias

Las operaciones aritméticas binarias se realizan **exactamente igual que las decimales**, aunque teniendo la precaución de usar sólo los dos símbolos permitidos (0 y 1), lo que puede parecernos un poco extraño al principio.

Por ejemplo, para realizar una suma de dos números binarios, escribiremos ambos números uno encima de otro, alineados a la derecha, como hacíamos cuando éramos tiernos infantes y estábamos aprendiendo a sumar. Luego, iremos sumando los dígitos de derecha a izquierda, como haríamos con dos números decimales, con la precaución de sumar también el acarreo cuando se produzca.

Vamos a sumar los números 11001_2 y 1011_2 :

$$\begin{array}{r}
 \begin{array}{cccccc}
 & 1 & 1 & & 1 & \leftarrow \text{acarreo} \\
 & 1 & 1 & 0 & 0 & 1 \\
 + & & & 1 & 0 & 1 & 1 \\
 \hline
 1 & 0 & 0 & 1 & 0 & 0
 \end{array}
 \end{array}$$

Del mismo modo, pueden realizarse otras operaciones aritméticas como restas, productos o divisiones.

1.2.3 Código ASCII

Hasta ahora hemos visto que mediante el código binario se pueden representar números, pero no sabemos cómo se las apaña un ordenador para representar las letras, o, dicho en terminología informática, los **caracteres alfanuméricos** (que incluyen números, letras y otros símbolos habituales, como los signos de puntuación).

El **código ASCII** consiste en una correspondencia entre números binarios de 8 dígitos y caracteres alfanuméricos². Así, por ejemplo, al número 65_{10} (en binario, 01000001_2) se le hace corresponder la letra A, al 66_{10} la B, al 67_{10} la C, etc. De este modo, el ordenador puede también manejar letras, y lo hace del mismo modo en que maneja números: mediante combinaciones de ceros y unos.

Es importante resaltar que los códigos ASCII siempre tienen 8 dígitos binarios, rellenándose con ceros a la izquierda si fuera necesario. Así ocurre en el caso de la letra A, que, como hemos dicho, se representa con el código 01000001 .

El código ASCII no es el único que existe para representar letras en binario, pero sí el más extendido.

1.2.4 Código hexadecimal

Es importante conocer y saber manejar el **código binario** al ser el método de codificación que emplean los ordenadores digitales, pero este código tiene dos serios **inconvenientes**:

- * *Primero, resulta **difícil de manipular** para cerebros que, como los nuestros, están habituados a pensar en decimal (o habituados a no pensar en absoluto, que también se da el caso).*
- * *Segundo, los números binarios pueden llegar a tener **cantidades enormes de dígitos** (es habitual trabajar con*

² Realmente, el código ASCII original sólo emplea 7 dígitos binarios, utilizándose el octavo para distintas extensiones del ASCII. Para simplificar, supondremos que el ASCII utiliza 8 dígitos binarios.

números de 16, 32 ó 64 dígitos binarios), lo cual los convierte en inmanejables.

Por este motivo, suelen usarse, en programación, otros dos sistemas de numeración llamados **octal** y **hexadecimal**. El octal maneja 8 símbolos distintos y, el hexadecimal, 16. Sin duda, el más utilizado es el hexadecimal y por este motivo nos vamos a detener en él, aunque haciendo notar que el octal funciona de la misma manera, sólo que empleando los dígitos del 0 al 7.

Si el sistema binario utiliza dos símbolos (0 y 1) y el decimal utiliza 10 (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), el hexadecimal emplea **16 símbolos**, que son: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F**.

En hexadecimal, por tanto, es normal ver números cuyos dígitos son letras del alfabeto. Por ejemplo: 2AF5 es un número válido escrito en hexadecimal (exactamente, el 10997 en decimal). La forma de contar, por supuesto, es la misma: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, y después empezamos a agrupar los símbolos: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F. Seguiríamos con 20, 21, 22, etc.

Podemos construir una tabla para comparar los primeros números en los tres sistemas de numeración que conocemos. Hemos rellenado los primeros números binarios con ceros a la izquierda por razones que pronto se verán, pero en realidad los números no cambian (recuerda que un cero a la izquierda no tiene ningún valor, ni en binario ni en el resto de sistemas)

Decimal	Binario	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Si te fijas, **cada dígito hexadecimal se corresponde exactamente con una combinación de 4 dígitos binarios**. Así, por ejemplo, el número binario 1001 1101 se puede escribir más resumidamente como 9D en hexadecimal. Y esa es la gran utilidad del sistema hexadecimal: permite **manipular números binarios de forma más escueta y resumida**, de manera que nos sean más fáciles de manejar a nosotros, los humanos, que somos muy propensos a cometer errores.

Convertir números hexadecimales a decimales

El mecanismo es el mismo que ya utilizamos para convertir números binarios, sólo que cambiando la base del sistema de numeración de 2 a 16, ya que ahora vamos a manejar números hexadecimales. Por lo tanto, los pasos a seguir son:

- 1) **Numeramos las posiciones** que ocupa cada dígito hexadecimal de derecha a izquierda, empezando por 0. Por ejemplo, en el número hexadecimal 2AF, numeraremos las posiciones así:

$$\begin{array}{ccc} 2 & 1 & 0 \\ \mathbf{2} & \mathbf{A} & \mathbf{F} \end{array}$$

- 2) **Multiplicamos cada dígito hexadecimal por 16 elevado a la posición del dígito y sumamos todos los resultados**. Con el número 2AF lo haríamos así:

$$\mathbf{2} \times 16^2 + \mathbf{A} \times 16^1 + \mathbf{F} \times 16^0$$

Según la tabla de anterior, tenemos que el dígito hexadecimal A equivale a 10 en decimal, y que F equivale a 15. Por lo tanto la operación quedaría así:

$$\mathbf{2} \times 16^2 + \mathbf{10} \times 16^1 + \mathbf{15} \times 16^0$$

Ahora sólo nos falta resolver la operaciones y sumar:

$$2 \times 256 + 10 \times 16 + 15 \times 1 = 687$$

Por lo tanto, el número hexadecimal 2AF₁₆ es equivalente al número decimal 687₁₀. Indicándolo con subíndices, lo expresaríamos así:

$$2AF_{16} = 687_{10}$$

Convertir números decimales a hexadecimales

El proceso también es idéntico al realizado con números binarios, pero sustituyendo la división entre 2 por divisiones **entre 16**, que es la base del sistema hexadecimal.

Relación entre números hexadecimales y binarios

La verdadera utilidad del sistema hexadecimal es que se puede utilizar en lugar del binario, siendo más fácil de manejar. Para que ello sea posible, el paso de hexadecimal a binario y viceversa debe poder hacerse con mucha rapidez.

Para **convertir un número hexadecimal a binario**, basta con sustituir cada dígito hexadecimal por sus cuatro cifras binarias correspondientes, según la tabla de la página anterior. Por ejemplo:

$$2AF_{16} = 0010\ 1010\ 1111_2$$

Del mismo modo, para **convertir un número binario a hexadecimal**, lo agruparemos en bloques de 4 cifras binarias (empezando por la derecha) y buscaremos la correspondencia en la tabla. Por ejemplo, el número binario 100100 se convierte así:

$$0010\ 0100_2 = 24_{16}$$

Observa que hemos rellenado con ceros a la izquierda para obtener bloques de 4 dígitos binarios sin alterar la esencia del número. Por supuesto, no es obligatorio hacerlo, pero las primeras veces puede facilitar las cosas. Con un poco de práctica conseguirás convertir binarios a hexadecimales y viceversa de un sólo vistazo y sin necesidad de consultar la tabla.

1.3 Unidades de medida de información

Como hemos visto, el código binario es el fundamento del funcionamiento de los ordenadores: toda la información que el ordenador maneja, ya sea numérica o alfanumérica, se encuentra codificada en binario.

Del mismo modo que para medir distancias se utiliza el metro, o para medir masas se utiliza el gramo, para **medir la cantidad de información** almacenada o procesada en un ordenador existe otra unidad de medida. Como el ordenador representa toda la información en binario, la unidad fundamental es el dígito binario (es decir, 0 ó 1), también llamado **BIT** (de **BI**nary **digi**T)

Un bit es realmente muy poca cantidad de información. Recuerda que, por ejemplo, para almacenar un sólo carácter en código ASCII son necesarios 8 bits. ¡Para un único carácter! Del mismo modo que el metro dispone de múltiplos (el decámetro, el hectómetro, el kilómetro, etc), también los tiene el bit, y son los siguientes:

- ★ **Byte**: 1 byte equivale a 8 bits. Cuidado con el nombre, porque se parecen y es un error común confundir el bit con el byte.
- ★ **Kilobyte (KB)**³: 1 kilobyte son 1024 bytes. Fíjate que es parecido al kilómetro (1000 metros), pero no exactamente igual.
- ★ **Megabyte (MB)**: 1 megabyte equivale a 1024 kilobytes.
- ★ **Gigabyte (GB)**: 1 gigabyte equivale a 1024 megabytes.
- ★ **Terabyte (TB)**: 1 terabyte equivale a 1024 gigabytes

Podemos resumir las unidades de medida de información en la siguiente tabla:

1 Byte	=	8 bits
1 KB	=	1024 Bytes
1 MB	=	1024 KB
1 GB	=	1024 MB
1 TB	=	1024 GB

1.4 Estrategias de resolución de problemas

Después de este paréntesis dedicado al sistema binario, retomamos el hilo de nuestro discurso: la programación de ordenadores como método para **resolver problemas del mundo real**.

1.4.1 Ingeniería del software

Los programas de ordenador son **productos realmente complejos (y caros) de diseñar y construir**. Al principio, con

³ Es habitual escribir estas abreviaturas con la "B" de "Byte" en mayúscula, reservando la "b" minúscula para los múltiplos del bit: kilobit, megabit, etc

los primeros ordenadores de la historia, esto no era así. Aquellos ordenadores eran tan elementales que sus programas no podían ser demasiado complicados, y podían ser desarrollados por cualquiera con algunos conocimientos del funcionamiento del ordenador.

Pero, a lo largo de los años 70, el avance de la tecnología provocó que los ordenadores tuvieran cada vez más capacidad de cálculo y, por lo tanto, que los programas fueran cada vez más complejos. Llegó un momento en el que se hizo evidente que ningún ser humano era capaz de hacer un programa tan complejo que aprovechara todas las posibilidades de hardware de los ordenadores de esa época. A esto se le llamó la **crisis del software**, y estancó la industria informática durante varios años.

El problema era que, hasta entonces, **se programaba sin método ni planificación**. A nadie se le ocurriría, por ejemplo, construir un avión sin haber hecho antes, cuidadosamente, multitud de cálculos, estudios, planos, diseños, esquemas, etc. Pues bien, un programa de ordenador puede ser tan complejo, o más, que un avión o cualquier otro artefacto industrial, y, por lo tanto, es necesario construirlo con los mismos procesos de ingeniería.

Surgió así el concepto de **ingeniería del software**, que podemos definir como *el conjunto de procedimientos y técnicas encaminadas a diseñar y desarrollar – con economía, prontitud, elegancia y cumpliendo los estándares de calidad – programas informáticos, documentación y procedimientos operativos mediante los cuales los ordenadores puedan ser útiles al ser humano*.

Actualmente, los procesos de la ingeniería del software (que son muchos y variados) se aplican en todas las empresas y organismos en los que se desarrolla software de forma profesional y rigurosa, porque no hay otro modo de asegurar que el producto se va a terminar dentro de los plazos y costes previstos, y que éste va a funcionar correctamente y se va a ajustar a los niveles de calidad que el mercado exige.

1.4.2 Ciclo de vida clásico

Una de las primeras enseñanzas de la ingeniería del software fue que, al ser el proceso de producción de software tan complicado, debía **descomponerse en varias etapas** para poder abordarlo.

El conjunto de estas etapas, o fases, constituyen lo que se denomina el **ciclo de vida** del software.

Dependiendo de diversos factores (como el tipo de software que se va a desarrollar, el sistema en el que va a funcionar, o las propias preferencias de los ingenieros o de la empresa desarrolladora), se puede elegir entre **varios tipos de ciclos de vida** que han demostrado su eficacia a lo largo de los años. Pero la mayoría de ellos, con ligeras variaciones, constan de las siguiente **fases**:

- ★ *Análisis del problema*
- ★ *Diseño de una solución*
- ★ *Especificación de los módulos*
- ★ *Codificación*
- ★ *Pruebas*
- ★ *Mantenimiento*

A continuación se describen las fases del ciclo de vida, pero antes dejemos claro que **nuestro curso** se centrará, principalmente, en las **fases de especificación de módulos y codificación**. También nos adentraremos, aunque sea superficialmente y desde un punto de vista práctico, en las etapas anteriores y posteriores, pero existen otros módulos profesionales dedicados a ellas.

Análisis

La fase de análisis consiste en averiguar **QUÉ problema vamos a resolver**. Parece una obviedad, pero la experiencia demuestra que no sólo no es así, sino que el análisis suele ser la etapa que más problemas causa y a la que más tiempo se le debería dedicar.

Es imprescindible partir de una **especificación de requisitos** lo más exacta y detallada posible. El resultado debe ser un **modelo** preciso del entorno del problema, de los datos y del objetivo que se pretende alcanzar. Pero expliquémoslo todo con más detenimiento:

El mundo real, por definición, es muy complejo. Cuando pretendemos traspasar una parte de ese mundo a un ordenador es necesario extraer sólo los aspectos esenciales del problema, es decir, lo que realmente afecta a esa parte del mundo, desechando todo lo demás. El proceso de comprensión y simplificación del mundo real se denomina **análisis** del problema, y la simplificación obtenida como resultado del análisis se llama **modelo**.

Por ejemplo, si lo que pretendemos es realizar un programa que calcule la trayectoria de un proyectil lanzado por un cañón de artillería (el clásico problema del tiro oblicuo, ¿recordáis vuestras clases de física?), lo lógico es que simplifiquemos el problema suponiendo que el proyectil es lanzado en el vacío (por lo que no hay resistencia del aire) y que la fuerza de la gravedad es constante. El resultado será muy aproximado al real, aunque no exacto. Esto es así porque nos hemos quedado con los aspectos esenciales del problema (la masa del proyectil, su velocidad, etc),

desechando los menos importantes (la resistencia del aire, la variación de la gravedad). Es decir, hemos realizado un **modelo** del mundo real.

En este ejemplo, el modelo del tiro oblicuo es muy fácil de construir ya que se basa en fórmulas matemáticas perfectamente conocidas. Necesitamos conocer algunos datos previos para que el modelo funcione: la velocidad del proyectil, su masa y su ángulo de salida. Con eso, nuestro programa podría calcular fácilmente la altura y la distancia que el proyectil alcanzará. Sin embargo, las áreas de aplicación de la Informática van más allá de la Física, por lo que la modelización suele ser bastante más difícil de hacer que en este problema.

Por ejemplo, en el programa de facturación de una empresa: ¿qué datos previos necesitamos conocer? ¿Qué fórmulas o cálculos matemáticos debemos realizar con ellos? ¿Qué resultado se espera del programa? Estas cuestiones deben quedar muy claras antes de la modelización porque, de lo contrario, el modelo no será adecuado para resolver el problema y todo el proceso de programación posterior dará como fruto un programa que no funciona o no hace lo que se esperaba de él.

Para que el modelo sea acertado, por lo tanto, es necesario tener muy clara la naturaleza del problema y de los datos que le afectan. A este respecto, es imprescindible establecer lo que se denomina una **especificación de requisitos**, que no es más que una definición lo más exacta posible del problema y su entorno. Sin una especificación detallada, es imposible comprender adecuadamente el problema y, por lo tanto, también es imposible hacer bien el análisis y construir un modelo que sea válido.

Diseño de soluciones

Una vez establecido el modelo del mundo real, y suponiendo que el problema sea computable, es necesario crear decidir **CÓMO se va a resolver el problema**, es decir, crear una estructura de hardware y software que lo resuelva (en este curso únicamente nos interesaremos por la parte del software)

Diseñar una solución para un modelo no es una tarea sencilla y sólo se aprende a hacerlo con la práctica. Típicamente, el diseño se resuelve mediante la técnica del **diseño descendente** (top-down), que consiste en **dividir el problema en subproblemas más simples**, y estos a su vez en otros más simples, y así sucesivamente hasta llegar a problemas lo bastante sencillos como para ser resueltos con facilidad.

Especificación de módulos y codificación

Para cada subproblema planteado en el diseño hay que inventarse una solución lo más eficiente posible, es decir, crear un **algoritmo**. Veremos qué son los algoritmos en la página 17, y dedicaremos el resto del curso a escribir algoritmos para todo tipo de problemas. Cada algoritmo que resuelve un subproblema se llama **módulo**.

Posteriormente, cada módulo debe ser traducido a un **lenguaje** comprensible por el ordenador, tecleado y almacenado. Estos lenguajes se llaman *lenguajes de programación*.

Los **lenguajes de programación** son conjuntos de símbolos y de reglas sintácticas especialmente diseñados para transmitir órdenes al ordenador. Existen multitud de lenguajes para hacer esto. Los veremos más adelante y aprenderemos a utilizar uno de ellos, llamado C.

Pruebas

Una vez que el programa está introducido en la memoria del ordenador, es necesario **depurar posibles errores**. La experiencia demuestra que hasta el programa más sencillo contiene errores y, por lo tanto, este es un paso de vital importancia.

Los errores más frecuentes son los **sintácticos** o de escritura, por habernos equivocado durante la codificación. Para corregirlos, basta con localizar el error (que generalmente nos marcará el propio ordenador) y subsanarlo.

Más peliagudos son los **errores de análisis o diseño**. Un error en fases tan tempranas dará lugar a un programa que, aunque corre en la máquina, no hace lo que se esperaba de él y, por lo tanto, no funciona. Estos errores obligan a revisar el análisis y el diseño y, en consecuencia, a rehacer todo el trabajo de especificación, codificación y pruebas. La mejor forma de evitarlos es realizar un análisis y un diseño concienzudos antes de lanzarnos a teclear código como posesos.

Existen varias técnicas, relacionadas con los **controles de calidad**, para generar software libre de errores y diseñar baterías de prueba que revisen los programas hasta el límite de lo posible, pero que quede claro: ningún programa complejo está libre de errores al 100% por más esfuerzos que se hayan invertido en ello.

Mantenimiento

Cuando el programa está en uso, y sobre todo si se trata de software comercial, suele ser preciso realizar un mantenimiento. El mantenimiento puede ser de varios tipos: **correctivo** (para enmendar errores que no se hubieran detectado en la fase de pruebas), **perfectivo** (para mejorar el rendimiento o añadir más funciones) o **adaptativo** (para adaptar el programa a otros entornos).

El coste de la fase de mantenimiento ha experimentado un fuerte incremento en los últimos años. Así, se estima que la mayoría de las empresas de software que dedican alrededor del 60% de sus recursos exclusivamente a mantener el software que ya tienen funcionando, empleando el 40% restante en otras tareas, entre las que se incluye el desarrollo de programas nuevos⁴. Esto es una consecuencia lógica del elevado coste de desarrollo del software.

1.4.3 El papel del programador

La figura del **programador artesanal** que, poseído por una *idea feliz* repentina se lanza a teclear como un poseso y, tras algunas horas de pura inspiración, consigue componer un programa para acceder, digamos, a las bases de datos de la CIA, es, digámoslo claro, pura fantasía romántica. El programador de ordenadores es una pieza más, junto con los analistas, diseñadores, jefes de proyecto, usuarios, etc., del complejo engranaje de la ingeniería del software.

Como es lógico, toda la maquinaria de esta ingeniería es excesiva si lo que pretendemos es realizar programas pequeños y sencillos, del mismo modo que no tomamos un avión para ir a comprar el pan a la esquina.

El programador, pues, debe estar capacitado para elaborar programas relativamente sencillos basándose en las especificaciones de los analistas y diseñadores. Esto no quiere decir que un programador no pueda ser, a la vez, analista y diseñador (en realidad, *a menudo* ejerce varias de estas funciones, dependiendo de su experiencia y capacidad y de la organización de la empresa en la que trabaje). Sin embargo, en este curso no nos ocuparemos de esas otras actividades y nos centraremos únicamente en las capacidades propias del programador puro. Nuestros programas serán forzosamente de tamaño modesto, aunque al final, cuando nos enfrentemos al proyecto de fin de curso, empezaremos a tropezar con las dificultades de los programas que *crecen demasiado* (tanto en tamaño como en complejidad) sin una adecuada planificación previa.

1.5 Estilos de programación

1.5.1 Programación convencional

Un programa de ordenador, como hemos dicho, es un conjunto de instrucciones que el ordenador puede entender y que ejecuta en un determinado orden. Generalmente, el orden de ejecución de las instrucciones es el mismo que el orden en el que el programador las escribió, pero en ocasiones, como veremos, es imprescindible repetir un conjunto de instrucciones varias veces (a esto se le llama técnicamente *bucle*), o saltar hacia delante o hacia atrás en la lista de instrucciones.

La **programación convencional o clásica** utiliza indistintamente bucles y saltos entremezclados hasta conseguir el correcto funcionamiento del programa. Debido a ésto, este tipo de programación es farragosa, confusa, e implica una alta probabilidad de errores. Estos defectos se hacen más patentes cuanto más grande es el programa, llegando a un punto en que el código se hace inmanejable (es lo que se suele denominar *código spaghetti*)

Este tipo de programación cayó en desuso tras la *crisis del software* de los años 70. Hoy se considera una mala práctica y debe ser evitada siempre.

1.5.2 Programación estructurada

E. W. Dijkstra, de la Universidad de Eindhoven, introdujo este concepto en los años 70 del siglo XX con el fin de eliminar las limitaciones de la programación convencional.

La **programación estructurada** es una técnica de programación que utiliza una serie de estructuras específicas que optimizan los recursos lógicos y físicos del ordenador. Estas estructuras (de ahí viene el nombre de *programación estructurada*) y las reglas de uso que implican las veremos en más adelante y las pondremos en práctica a lo largo de todo el curso.

1.5.3 Programación modular

Esta otra técnica de programación no es excluyente de la anterior, sino que se pueden utilizar conjuntamente. Es decir, un programa puede ser a la vez modular y estructurado.

La **programación modular** consiste en dividir un programa complejo en varios programas sencillos que interaccionan de algún modo. Cada programa sencillo se llama **módulo** y deben ser independientes entre sí, es decir, no deben interferir con otros módulos, aunque sí cooperar con ellos en la resolución del problema global. Las técnicas de programación modular las estudiaremos más adelante, en otro apartado, y las aplicaremos a lo largo de todo el curso.

⁴ Véase: KENDALL & KENDALL, *Análisis y Diseño de Sistemas 3ª ed*, Prentice-Hall 1997

2 Los datos

Como vimos al definir qué es un programa de ordenador, tan importantes son las instrucciones de que consta un programa como los datos que maneja.

Los **datos**, como definimos al principio del tema, son representaciones de los objetos del mundo real. Por ejemplo, en un programa de gestión de nóminas de una empresa, existen multitud de datos: los nombres de los empleados, el dinero que ganan, los impuestos que pagan, etc. Cada programa, pues, tiene su propia colección de datos.

2.1 Tipos de datos

Se llama **tipo de datos** a una clase concreta de objetos. Cada tipo de datos, además, tiene asociado un conjunto de **operaciones** para manipularlos.

Cada tipo de datos dispone de una representación interna diferente en el ordenador; por eso es importante distinguir entre tipos de datos a la hora de programar.

2.1.1 Tipos de datos simples

Existen tipos de datos simples y tipos complejos. Entre los **simples** tenemos:

- * *Números enteros*
- * *Números reales*
- * *Caracteres*
- * *Lógicos*

Así, por ejemplo, en el caso del programa de gestión de nóminas, la edad de los empleados será un dato de tipo *número entero*, mientras que el dinero que gana al mes será un dato de tipo *número real*.

Los tipos de datos complejos, también llamados **estructuras de datos**, los estudiaremos a partir del tema 3. Por ahora nos centraremos en los tipos simples.

Números enteros

Es probablemente el tipo más sencillo de entender. Los datos de tipo entero sólo pueden tomar como valores:

..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...

Como el ordenador tiene una memoria finita, la cantidad de valores enteros que puede manejar también es finita y depende del número de bits que emplee para ello (recuerda que el ordenador, internamente, representa todos los datos en binario).

Además, los enteros pueden ser **con signo** y **sin signo**. Si tienen signo, se admiten los números negativos; si no lo tienen, los números sólo pueden ser positivos (sería más correcto llamarlos *números naturales*).

(Los enteros con signo se almacenan en binario en **complemento a uno** o en **complemento a dos**. Estas representaciones internas las estudiarás en el módulo de Sistemas Operativos, por lo que no vamos a detenernos ahora en detallarlas)

Por lo tanto:

- * *Si se utilizan 8 bits para codificar los números enteros, el rango de valores permitido irá de 0 a 255 (sin signo) o de -128 a +127 (con signo).*
- * *Si se utilizan 16 bits para codificar los números enteros, el rango será de 0 a 65535 (sin signo) o de -32768 a 32767 (sin signo).*
- * *Si se utilizan 32, 64, 128 bits o más, se pueden manejar números enteros mayores.*

Números reales

El tipo de dato *número real* permite representar números con decimales. La cantidad de decimales de un número real puede ser infinita, pero al ser el ordenador una máquina finita es necesario establecer un número máximo de dígitos decimales significativos.

La representación interna de los números reales se denomina **coma flotante** (también existe la representación en coma fija, pero no es habitual). La coma flotante es una generalización de la notación científica convencional, consistente en definir cada número con una **mantisa** y un **exponente**.

La **notación científica** es muy útil para representar números muy grandes economizando esfuerzos. Por ejemplo, el número 12943900000000000000 tiene la siguiente representación científica:

$$1,29439 \times 10^{20}$$

Pero el ordenador representaría este número siempre **con un 0 a la izquierda de la coma**, así:

$$0,129439 \times 10^{21}$$

La mantisa es el número situado en la posición decimal (129439) y el exponente es 21.

La notación científica es igualmente útil para números decimales muy pequeños. Por ejemplo, el número 0,0000000000000000000259 tiene esta notación científica:

$$2,59 \times 10^{-23}$$

Pero el ordenador lo representará así:

$$0,259 \times 10^{-22}$$

Siendo 259 la mantisa y -22 el exponente.

Internamente, el ordenador reserva varios bits para la mantisa y otros más para el exponente. Como en el caso de los números reales, la magnitud de los números que el ordenador pueda manejar estará directamente relacionada con el número de bits reservados para su almacenamiento.

Overflow

Cuando se realizan operaciones con números (tanto enteros como reales), es posible que el resultado de una de ellas dé lugar a un número fuera del rango máximo permitido. Por ejemplo, si tenemos un dato de tipo entero sin signo de 8 bits cuyo valor sea 250 y le sumamos 10, el resultado es 260, que sobrepasa el valor máximo (255).

En estos casos, estamos ante un caso extremo denominado **overflow** o **desbordamiento**. Los ordenadores pueden reaccionar de forma diferente ante este problema, dependiendo del sistema operativo y del lenguaje utilizado. Algunos lo detectan como un error de ejecución del programa, mientras que otros lo ignoran, convirtiendo el número desbordado a un número dentro del rango permitido pero que, obviamente, no será el resultado correcto de la operación, por lo que el programa probablemente fallará.

Caracteres y cadenas

El tipo de dato **carácter** sirve para representar datos alfanuméricos. El conjunto de elementos que puede representar está estandarizado según el código ASCII, que, como ya vimos, consiste en una combinación de 8 bits asociada a un carácter alfanumérico concreto.

Las combinaciones de 8 bits dan lugar a un total de 255 valores distintos (desde 0000 0000 hasta 1111 1111), por lo que esa es la cantidad de caracteres diferentes que se pueden utilizar. Entre los datos de tipo carácter válidos están:

- * Las letras minúsculas: 'a', 'b', 'c' ... 'z'
- * Las letras mayúsculas: 'A', 'B', 'C' ... 'Z'
- * Los dígitos: '1', '2', '3' ...
- * Caracteres especiales: '\$', '%', '&', '!' ...

Nótese que no es lo mismo el valor entero 3 que el carácter '3'. Para distinguirlos, usaremos siempre **comillas para escribir los caracteres**.

Los datos tipo carácter sólo pueden contener UN carácter. Una generalización del tipo carácter es el tipo **cadena de caracteres**, utilizado para representar series de varios caracteres. Éste, sin embargo, es un **tipo de datos complejo** y será estudiado más adelante (en el tema 3). Sin embargo, las cadenas se utilizan tan a menudo que no podremos evitar usarlas en algunos ejercicios antes de estudiarlas a fondo.

Datos lógicos

El tipo *dato lógico*, también llamado *booleano*⁵, es un dato que sólo puede tomar un valor entre dos posibles. Esos dos valores son:

- * **Verdadero** (en inglés, true)
- * **Falso** (en inglés, false)

Este tipo de datos se utiliza para representar alternativas del tipo sí/no. En algunos lenguajes, el valor *true* se representa con el número 1 y el valor *false* con el número 0. Es decir, los datos lógicos contienen información **binaria**. Esto ya los hace bastante importantes, pero la mayor utilidad de los datos lógicos viene por otro lado: son el resultado de todas las operaciones lógicas y relacionales, como veremos en el siguiente epígrafe.

2.1.2 Tipos de datos complejos

Los tipos de datos complejos se componen a partir de *agrupaciones* de otros datos, ya sean simples o complejos. Por ejemplo, una lista ordenada de números enteros (datos simples) constituyen lo que se llama un **vector** de números

⁵ En honor a George Boole (1815-1864), matemático británico que desarrolló una rama del álgebra llamada *lógica* o *de Boole*

enteros (dato complejo)

Como los datos complejos son muy importantes, dedicaremos a ellos gran parte de este curso (a partir de la Unidad 3). Por ahora, sin embargo, utilizaremos sólo los datos simples hasta que tengamos un dominio suficiente sobre los mecanismos de la programación estructurada.

2.2 Operaciones con datos

Como dijimos más atrás, los tipos de datos se caracterizan por la clase de objeto que representan y por las operaciones que se pueden hacer con ellos. Los datos que participan en una operación se llaman **operandos**, y el símbolo de la operación se denomina **operador**. Por ejemplo, en la operación entera $5 + 3$, los datos 5 y 3 son los operandos y "+" es el operador.

Podemos clasificar las operaciones básicas con datos en dos grandes grupos: las operaciones **aritméticas** y las operaciones **lógicas**.

2.2.1 Operaciones aritméticas

Son análogas a las operaciones matemáticas convencionales, aunque cambian los símbolos. Sólo se emplean con datos de tipo entero o real (aunque puede haber alguna excepción):

Operación	Operador
suma	+
resta	-
multiplicación	*
división entera	div
división	/
módulo (resto)	%
exponenciación	^

No todos los operadores existen en todos los lenguajes de programación. Por ejemplo, en lenguaje Fortran no existe la división entera, en C no existe la exponenciación, y, en Pascal, el operador "%" se escribe "mod".

Señalemos que la **división entera (div)** se utiliza para dividir números enteros, proporcionando a su vez como resultado otro número entero, es decir, sin decimales. La operación **módulo (%)** sirve para calcular el resto de estas divisiones enteras.

El **tipo del resultado** de cada operación dependerá del tipo de los operandos. Por ejemplo, si sumamos dos números enteros, el resultado será otro número entero. En cambio, si sumamos dos números reales, el resultado será un número real. La suma de un número entero con otro real no está permitida en muchos lenguajes, así que intentaremos evitarla.

Por último, decir que **las operaciones "div" y "%" sólo se pueden hacer con números enteros**, no con reales, y que **la operación "/" sólo se puede realizar con reales**, no con enteros.

Aquí tenemos algunos ejemplos de operaciones aritméticas con números enteros y reales:

Operandos	Operador	Operación	Resultado
35 y 9 (enteros)	+	$35 + 9$	44 (entero)
35 y 9 (enteros)	-	$35 - 9$	26 (entero)
35 y 9 (enteros)	*	$35 * 9$	315 (entero)
35 y 9 (enteros)	div	$35 \text{ div } 9$	3 (entero)
35 y 9 (enteros)	%	$35 \% 9$	8 (entero)
35 y 9 (enteros)	^	$35 ^ 9$	overflow
8,5 y 6,75 (reales)	+	$8,5 + 6,75$	15,25 (real)
8,5 y 6,75 (reales)	-	$8,5 - 6,75$	1,75 (real)
8,5 y 6,75 (reales)	*	$8,5 * 6,75$	57,375 (real)
8,5 y 6,75 (reales)	/	$8,5 / 6,75$	1,259 (real)
8,5 y 6,75 (reales)	^	$8,5 ^ 6,75$	$1,877 \times 10^6$ (real)

Nótese que el operador "-" también se usa para preceder a los números negativos, como en el álgebra convencional.

2.2.2 Operaciones lógicas (o booleanas)

Estas operaciones sólo pueden dar como resultado **verdadero** o **falso**, es decir, **su resultado debe ser un valor lógico**.

Hay dos tipos de operadores que se utilizan en estas operaciones: los operadores de relación y los operadores lógicos.

- 1) **Operadores de relación**. Son los siguientes:

Operación	Operador
menor que	<
mayor que	>
igual que	==
menor o igual que	<=
mayor o igual que	>=
distinto de	!=

Muchos lenguajes prefieren el símbolo "<>" para "distinto de". En realidad, es un asunto de notación que no tiene mayor importancia.

Los operadores de relación se pueden usar con todos los tipos de datos simples: **entero, real, carácter o lógico**. El resultado será **verdadero** si la relación es cierta, o **falso** en caso contrario.

Aquí tienes algunos ejemplos:

Operandos	Operador	Operación	Resultado
35, 9 (enteros)	>	35 > 9	verdadero
35, 9 (enteros)	<	35 < 9	falso
35, 9 (enteros)	==	35 == 9	falso
35, 9 (enteros)	!=	35 != 9	verdadero
5, 5 (enteros)	<	5 < 5	falso
5, 5 (enteros)	<=	5 <= 5	verdadero
5, 5 (enteros)	!=	5 != 5	falso
"a", "c" (caracteres)	==	'a' == 'c'	falso
"a", "c" (caracteres)	>=	'a' >= 'c'	falso
"a", "c" (caracteres)	<=	'a' <= 'c'	verdadero

En cuanto a los datos lógicos, se considera que "falso" es menor que "verdadero". Por lo tanto:

Operandos	Operador	Operación	Resultado
verdadero, falso	>	verdadero > falso	verdadero
verdadero, falso	<	verdadero < falso	falso
verdadero, falso	==	verdadero == falso	falso

- 2) **Operadores lógicos.** Los operadores lógicos son **and** (y), **or** (o) y **not** (no). Sólo se pueden emplear con tipos de datos lógicos.

El operador **and**, que también podemos llamar **y**, da como resultado verdadero sólo si los dos operandos son verdaderos:

Operandos	Operador	Operación	Resultado
verdadero, falso	y	verdadero y falso	falso
falso, verdadero	y	falso y verdadero	falso
verdadero, verdadero	y	verdadero y verdadero	verdadero
falso, falso	y	falso y falso	falso

El operador **or** (también nos vale **o**) da como resultado verdadero cuando al menos uno de los dos operandos es verdadero:

Operandos	Operador	Operación	Resultado
verdadero, falso	o	verdadero o falso	verdadero
falso, verdadero	o	falso o verdadero	verdadero
verdadero, verdadero	o	verdadero o verdadero	verdadero
falso, falso	o	falso o falso	falso

El operador **not** (o **no**) es uno de los escasos operadores que sólo afectan a un operando (operador monario), no a dos (operador binario). El resultado es la negación del valor del operando, es decir, que le cambia el valor de verdadero a falso y viceversa:

Operando	Operador	Operación	Resultado
verdadero	no	no verdadero	falso
falso	no	no falso	verdadero

2.2.3 Prioridad de los operadores

Es habitual encontrar **varias operaciones** juntas en una misma línea. En estos casos es **imprescindible** conocer la **prioridad** de los operadores, porque las operaciones se calcularán en el orden de prioridad y el resultado puede ser muy distinto del esperado. Por ejemplo, en la operación $6 + 4 / 2$, no es lo mismo calcular primero la operación $6 + 4$ que calcular primero la operación $4 / 2$.

La prioridad de cálculo respeta las reglas generales del álgebra. Así, por ejemplo, la división y la multiplicación tienen más prioridad que la suma o la resta. Pero el resto de prioridades pueden diferir de manera importante de un lenguaje de programación a otro. Como nosotros vamos a usar C, emplearemos las prioridades de C, que son las siguientes:

Operador	Prioridad
\wedge	<div> <div>máxima</div> <div>↓</div> <div>mínima</div> </div>
$*, /, \text{div}, \%$	
no	
$+, -$	
$<, >, <=, >=$	
$==, !=$	
y	
o	

La prioridad del cálculo se puede alterar usando **paréntesis**, como en álgebra. Los paréntesis se pueden anidar tantos niveles como sean necesarios. Por supuesto, a igualdad de prioridad entre dos operadores, la operación se calcula **de izquierda a derecha**, en el sentido de la lectura de los operandos.

Aquí tenemos algunos ejemplos de operaciones conjuntas y su resultado según el orden de prioridad que hemos visto:

Operación	Resultado
$6 + 4 / 2$	8
$(6 + 4) / 2$	5
$(33 + 3 * 4) / 5$	9
$2 \wedge 2 * 3$	12
$3 + 2 * (18 - 4 \wedge 2)$	7
$5 + 3 < 2 + 9$	verdadero
$2 + 3 < 2 + 4 \text{ y } 7 > 5$	verdadero
$"A" > "Z" \text{ o } 4 / 2 + 4 > 6$	falso
$"A" > "Z" \text{ o } 4 / (2 + 2) \leq 6$	verdadero

2.2.4 Funciones

Además de todas estas operaciones aritméticas, lógicas y relacionales, los lenguajes de programación disponen de mecanismos para realizar operaciones más complejas con los datos, como, por ejemplo, calcular raíces cuadradas, logaritmos, senos, cosenos, redondeo de números reales, etc.

Todas estas operaciones (y muchas más) se realizan a través de operadores especiales llamados **funciones de biblioteca**. Cuando llegue el momento, ya explicaremos en detalle qué son las funciones de biblioteca, e incluso aprenderemos a hacer las nuestras. Por ahora nos basta saber que sirven para hacer cálculos más complejos y que varían mucho de unos lenguajes a otros, aunque hay cierto número de ellas que estarán disponibles en cualquier lenguaje y que, por lo tanto, podemos usar si las necesitamos.

Las funciones suelen tener al menos un **argumento** (pueden tener más de uno), que es el valor sobre el que realizan la operación. Los argumentos se indican entre paréntesis a continuación del nombre de la función.

Estas son algunas de las funciones que encontraremos en todos los lenguajes de programación:

Función	Descripción	Tipo de dato	Tipo de resultado
abs (x)	valor absoluto de x	Real o Entero	Real o Entero
sen (x)	seno de x	Real o Entero	Real
cos (x)	coseno de x	Real o Entero	Real
exp (x)	e^x	Real o Entero	Real
ln (x)	logaritmo neperiano de x	Real o Entero	Real
log10 (x)	logaritmo decimal de x	Real o Entero	Real
redondeo (x)	redondea el número x al valor entero más próximo	Real	Entero
trunc (x)	trunca el número x, es decir, le elimina la parte decimal	Real	Entero
raiz (x)	raíz cuadrada de x	Real o Entero	Real
cuadrado (x)	x^2	Real o Entero	Real o Entero
aleatorio (x)	genera un número al azar entre 0 y x	Entero	Entero

Aquí tienes algunos ejemplos de aplicación de estas funciones sobre datos reales:

Operación	Resultado
<code>abs(-5)</code>	5
<code>abs(6)</code>	6
<code>redondeo(5.7)</code>	6
<code>redondeo(5.2)</code>	5
<code>trunc(5.7)</code>	5
<code>trunc(5.2)</code>	5
<code>cuadrado(8)</code>	64
<code>raiz(64)</code>	8

2.3 Constantes y variables

Se define un **dato constante** (o, simplemente, "una constante") como un dato de un programa cuyo valor no cambia durante la ejecución. Por el contrario, un **dato variable** (o, simplemente, "una variable") es un dato cuyo valor sí cambia en el transcurso del programa.

2.3.1 Identificadores

A los datos variables se les asigna un identificador alfanumérico, es decir, un nombre. Por lo tanto, es necesario distinguir entre el **identificador** de una variable y su **valor**. Por ejemplo, una variable llamada X puede contener el valor 5. En este caso, X es el identificador y 5 el valor de la variable.

Los identificadores o nombres de variable deben cumplir ciertas reglas que, aunque varían de un lenguaje a otro, podemos resumir en que:

- Deben empezar por una letra y, en general, no contener símbolos especiales excepto el subrayado (" _ ")
- No deben coincidir con alguna palabra reservada del lenguaje

Identificador	¿Es válido?
x	Sí
5x	No, porque no empieza por una letra
x5	Sí
pepe	Sí
_pepe	No, porque no empieza por una letra
pepe_luis	Sí
pepe!luis	No, porque contiene caracteres especiales (!)
raiz	No, porque coincide con la función <code>raiz(x)</code>

Las **constantes** también pueden tener un identificador, aunque no es estrictamente obligatorio. En caso de tenerlo, ha de cumplir las mismas reglas que los identificadores de variable.

2.3.2 Declaración y asignación

Las variables tienen que ser de un **tipo de datos determinado**, es decir, debemos indicar explícitamente qué tipo de datos va a almacenar a lo largo del programa. Esto implica que, en algún punto del programa (luego veremos dónde) hay que señalar cual va a ser el identificador de la variable, y qué tipo de datos va a almacenar. A esto se le llama **declarar la variable**.

Una declaración de variables será algo así:

```
X es entero
Y es real
letra es carácter
```

X, Y y letra son los identificadores de variable. Es necesario declararlas porque, como vimos, el ordenador maneja internamente cada variable de una forma diferente: en efecto, no es lo mismo una variable entera de 8 bits sin signo que otra real en coma flotante. El ordenador debe saber de antemano qué variables va a usar el programa y de qué tipo son para poder asignarles la memoria necesaria.

Para adjudicar un valor a una variable, se emplea una sentencia de **asignación**, que tienen esta forma:

```
X = 5
Y = 7.445
LETRA = 'J'
```

A partir de la asignación, pueden hacerse operaciones con las variables exactamente igual que se harían con datos. Por ejemplo, la operación `X + X` daría como resultado 10. A lo largo del programa, la misma variable X puede contener otros valores (siempre de tipo entero) y utilizarse para otras operaciones. Por ejemplo:


```

X es entero
Y es entero
Z es entero
X = 8
Y = 2
Z = X div Y
X = 5
Y = X + Z

```

Después de esta serie de operaciones, realizadas de arriba a abajo, la variable X contendrá el valor 5, la variable Y contendrá el valor 9 y, la variable Z, el valor 4.

En cambio, las **constantes** no necesitan identificador, ya que son valores que nunca cambian. Esto no significa que no se les pueda asociar un identificador para hacer el programa más legible. En ese caso, sólo se les puede **asignar valor una vez**, ya que, por su propia naturaleza, son invariables a lo largo del programa.

2.4 Expresiones

Una **expresión** es una combinación de constantes, variables, operadores y funciones. Es decir, se trata de operaciones aritméticas o lógicas como las que vimos en el apartado 2.2 (página 13), pero en las que, además, pueden aparecer variables.

Por ejemplo:

$$(5 + X) \text{ div } 2$$

En esta expresión, aparecen dos constantes (5 y 2), una variable (X) y dos operadores (+ y div), además de los paréntesis, que sirven para alterar la prioridad de las operaciones. Lógicamente, para resolver la expresión, es decir, para averiguar su resultado, debemos conocer cuál es el valor de la variable X. Supongamos que la variable X tuviera el valor 7. Entonces, el resultado de la expresión es 6. El cálculo del resultado de una expresión se suele denominar **evaluación** de la expresión.

Otro ejemplo:

$$(-b + \text{raiz}(b^2 - 4 * a * c)) / (2 * a)$$

Esta expresión, más compleja, tiene tres variables (a, b y c), 4 operadores (−, +, ^ y *, aunque algunos aparecen varias veces), 2 constantes (2 y 4, apareciendo el 2 dos veces) y una función (*raiz*, que calcula la raíz cuadrada). Si el valor de las variables fuera a = 2, c = 3 y b = 4, al evaluar la expresión el resultado sería −0.5

La forma más habitual de encontrar una expresión es **combinada con una sentencia de asignación** a una variable.

Por ejemplo:

$$Y = (5 + X) \text{ div } 2$$

En estos casos, la expresión (lo que hay a la derecha del signo "=") se evalúa y su resultado es asignado a la variable situada a la izquierda del "=". En el ejemplo anterior, suponiendo que la variable X valiera 7, la expresión (5 + X) div 2 tendría el valor 6, y, por lo tanto, ese es el valor que se asignaría a la variable Y.

3 Los algoritmos

3.1 Concepto de algoritmo

Para realizar un programa es necesario idear previamente un algoritmo. Esto es importante hasta el extremo de que, sin algoritmo, no existiría el programa (ver "*Ciclo de vida del software*", pág. 8).

Un **algoritmo** es **una secuencia ordenada de pasos que conducen a la solución de un problema**. Los algoritmos tienen tres características fundamentales:

- 1) Son precisos, es decir, deben indicar el orden de realización de los pasos.
- 2) Están bien definidos, es decir, si se sigue el algoritmo dos veces usando los mismos datos, debe proporcionar la misma solución.
- 3) Son finitos, esto es, deben completarse en un número determinado de pasos.

Por ejemplo, vamos a diseñar un algoritmo simple que determine si un número N es par o impar:

```

1. Inicio
2. Si N es divisible entre 2, entonces ES PAR
3. Si N no es divisible entre 2, entonces NO ES PAR
4. Fin

```

Si te fijas bien, este algoritmo cumple las tres condiciones enumeradas anteriormente (precisión, definición y finitud) y resuelve el problema planteado. Lógicamente, al ordenador no le podemos dar estas instrucciones tal y como las hemos escrito, sino que habrá que expresarlo en un lenguaje de programación, pero esto es algo que trataremos más adelante.

3.2 Notación de algoritmos

Los algoritmos deben representarse con algún método que permita **independizarlos del lenguaje de programación** que luego se vaya a utilizar. Así se podrán traducir más tarde a cualquier lenguaje. En el ejemplo que acabamos de ver hemos especificado el algoritmo en lenguaje español, pero existen otras formas de representar los algoritmos. Entre todas ellas, destacaremos las siguientes:

- 1) Lenguaje español
- 2) Diagramas de flujo
- 3) Diagramas de Nassi-Schneiderman (NS)
- 4) Pseudocódigo

Nosotros utilizaremos, principalmente, el **pseudocódigo** y los **diagramas de flujo**. El pseudocódigo es un lenguaje de especificación de algoritmos basado en la lengua española que tiene dos propiedades que nos interesarán: facilita considerablemente el aprendizaje de las técnicas de programación y logra que la traducción a un lenguaje de programación real sea casi instantánea. Los diagramas de flujo son representaciones gráficas de los algoritmos que ayudan a comprender su funcionamiento.

Dedicaremos todo el apartado siguiente a aprender las técnicas básicas de programación usando pseudocódigo y diagramas de flujo, pero, como adelanto, ahí va el algoritmo que determina si un número N es par o impar, escrito en pseudocódigo. Es recomendable que le eches un vistazo para intentar entenderlo y para familiarizarte con la notación en pseudocódigo:

```

algoritmo par_impar
variables
  N es entero
  solución es cadena
inicio
  leer (N)
  si (N div 2 == 0) entonces solución = "N es par"
  si_no solución = "N es impar"
  escribir (solución)
fin

```

3.2.1 Escritura inicial del algoritmo

Una vez superadas las fases de análisis y diseño, es decir, entendido bien el problema y sus datos y descompuesto en problemas más sencillos, llega el momento de resolver **cada problema sencillo** mediante un **algoritmo**.

Muchos autores recomiendan escribir una **primera versión** del algoritmo en **lenguaje natural** (en nuestro caso, en castellano), siempre que dicha primera versión cumpla **dos condiciones**:

*que la solución se exprese como una **serie de instrucciones** o pasos a seguir para obtener una solución al problema*

*que las instrucciones haya que ejecutarlas de una en una, es decir, **una instrucción cada vez***

Ejemplo Consideremos un problema sencillo: el cálculo del área y del perímetro de un rectángulo. Evidentemente, tenemos que conocer su base y su altura, que designaremos con dos variables de tipo real. Una primera aproximación, en lenguaje natural, podría ser:

1. Inicio
2. Preguntar al usuario los valores de **base** y **altura**
3. Calcular el **área** como $\text{área} = \text{base} * \text{altura}$
4. Calcular el **perímetro** como $\text{perímetro} = 2 * \text{base} + 2 * \text{altura}$
5. Fin

Describir un algoritmo de esta forma puede ser útil si el problema es complicado, ya que puede ayudarnos a entenderlo mejor y a diseñar una solución adecuada. Pero esto sólo es una primera versión que puede **refinarse** añadiendo cosas. Por ejemplo, ¿qué pasa si la base o la altura son negativas o cero? En tal caso, no tiene sentido averiguar el área o el perímetro. Podríamos considerar esta posibilidad en nuestro algoritmo para hacerlo más completo:

1. Inicio
2. Preguntar al usuario los valores de **base** y **altura**
3. **Si** **base** es mayor que cero y **altura** también, **entonces**:
 - 3.1. Calcular el **área** como $\text{área} = \text{base} * \text{altura}$
 - 3.2. Calcular el **perímetro** como $\text{perímetro} = 2 * \text{base} + 2 * \text{altura}$
4. **Si no**:
 - 4.1. No tiene sentido calcular el área ni el perímetro
5. Fin




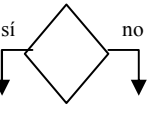
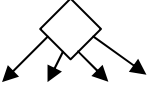


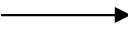
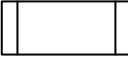
Estos refinamientos son habituales en todos los algoritmos y tienen la finalidad de conseguir una solución lo más general posible, es decir, que pueda funcionar con cualquier valor de "base" y "altura".

3.2.2 Diagramas de flujo

El diagrama de flujo es una de las técnicas de representación de algoritmos más antigua y también más utilizada, al menos entre principiantes y para **algoritmos sencillos**. Con la práctica comprobaremos que, cuando se trata de problemas complejos, los diagramas de flujo se hacen demasiado grandes y complicados.

Un **diagrama de flujo** o *flowchart* es un gráfico en el que se utilizan símbolos (o *cajas*) para representar los pasos del algoritmo. Las cajas están unidas entre sí mediante flechas, llamadas *líneas de flujo*, que indican el orden en el que se deben ejecutar para alcanzar la solución.

Los símbolos de las cajas están estandarizados y son muy variados. En la tabla siguiente tienes los más habituales, aunque existen algunos otros que no vamos a utilizar.

Símbolo	Función
	Terminal. Representa el comienzo o el fin de un programa.
	Entrada / Salida. Indica una introducción de datos desde un dispositivo externo (por defecto, el teclado) o una salida de datos hacia algún dispositivo externo (por defecto, la pantalla)
	Proceso. Representa cualquier operación que se lleve a cabo con los datos del problema.
	Condición. Señala una bifurcación del flujo de instrucciones. La bifurcación está siempre controlada por una operación relacional llamada <i>condición</i> , cuyo resultado puede ser "verdadero" o "falso" (o también "sí" o "no"), dependiendo del valor de los datos de la expresión condicional. En función del resultado de dicha expresión, el flujo de ejecución continúa por una u otra rama (pero nunca por las dos a la vez)
	Condición múltiple. Sirve para indicar una bifurcación del flujo en varias ramas, no sólo en una. En este caso, la condición no puede ser booleana, sino entera.
	Conector. Para enlazar un fragmento del diagrama de flujo con otro fragmento situado en la misma página. Se usa cuando el diagrama es muy grande y no puede dibujarse entero de arriba a abajo.
	Conector. Como el anterior, pero para conectar un fragmento del diagrama con otro fragmento situado en una página diferente.
	Dirección del flujo. Indica el orden de ejecución de los pasos del algoritmo.
	Subrutina. Llamada a un subproceso o módulo independiente (ver apartado de "Programación Modular")

Ejemplo Representación del algoritmo que calcula el área y el perímetro de un rectángulo mediante un diagrama de flujo. Antes, tengamos en cuenta que:

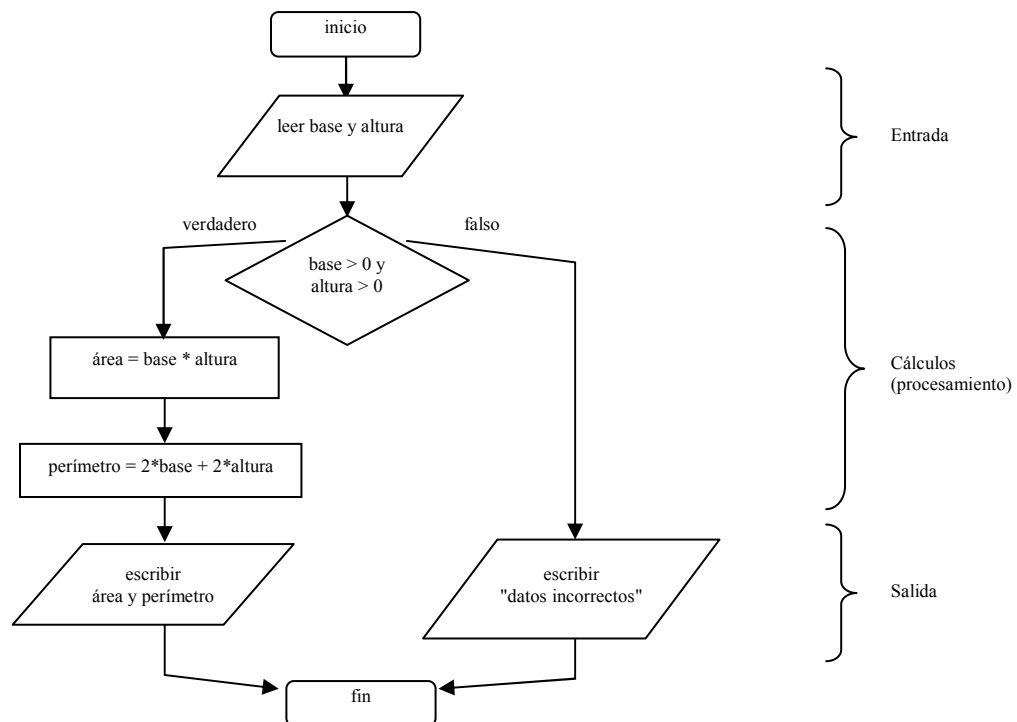
los valores de "base" y "altura" los introducirá el usuario del programa a través del teclado; así, el programa servirá para cualquier rectángulo

después se realizarán los cálculos necesarios

los resultados, "área" y "perímetro", deben mostrarse en un dispositivo de salida (por defecto, la pantalla) para que el usuario del programa vea cuál es la solución

Esta **estructura en 3 pasos** es muy **típica** de todos los algoritmos: primero hay una entrada de datos, luego se hacen cálculos con esos datos, y por último se sacan los resultados.

El diagrama de flujo será más o menos así (ver pág. siguiente):



3.2.3 Pseudocódigo

El pseudocódigo es un **lenguaje de descripción de algoritmos**. El paso desde el pseudocódigo hasta el lenguaje de programación real (por ejemplo, C), es relativamente fácil. Además, la descripción de algoritmos en pseudocódigo ocupa mucho menos espacio que su equivalente con un diagrama de flujo, por lo que lo preferiremos a la hora de diseñar algoritmos complejos.

El pseudocódigo es bastante parecido a la mayoría de los lenguajes de programación reales, pero no tiene unas reglas tan estrictas, por lo que el programador puede trabajar en la estructura del algoritmo sin preocuparse de las limitaciones del lenguaje final que, como veremos al estudiar C, son muchas y variopintas.

El pseudocódigo utiliza ciertas **palabras reservadas** para representar las acciones del programa. Estas palabras originalmente están en inglés (y se parecen mucho a las que luego emplean los lenguajes de programación), pero por suerte para nosotros su traducción española está muy extendida entre la comunidad hispanohablante.

Relación de instrucciones del pseudocódigo

Las **palabras reservadas** del pseudocódigo son relativamente pocas, pero, como iremos aprendiendo en el transcurso del curso, con un conjunto bastante reducido de instrucciones, correctamente combinadas, podemos construir programas muy complejos.

A continuación presentamos una **tabla-resumen** con todas las palabras reservadas del pseudocódigo, y en los siguientes apartados iremos viéndolas una a una.

Instrucción	Significado
algoritmo nombre	Marca el comienzo de un algoritmo y le adjudica un nombre
inicio	Marca el comienzo de un bloque de instrucciones
fin	Marca el final de un bloque de instrucciones
variables nombre_var es tipo_de_datos	Declaración de variables. Indica el identificador y el tipo de las variables que se van a usar en el algoritmo
constantes nombre_const = expresión	Declaración de constantes. La expresión se evalúa y su resultado se asigna a la constante. Este valor no puede modificarse a lo largo del programa.

leer (variable)	Entrada de datos. El programa lee un dato desde un dispositivo de entrada (si no se indica otra cosa, el teclado), asignando ese dato a la variable
escribir (variable)	Salida de datos. Sirve para que el programa escriba un dato en un dispositivo de salida (si no se indica otra cosa, la pantalla).
variable = expresión	Asignación. La expresión se evalúa y su resultado es asignado a la variable
si (condición) entonces inicio acciones-1 fin si_no inicio acciones-2 fin	Instrucción condicional doble. El ordenador evaluará la condición, que debe ser una expresión lógica. Si es verdadera, realiza las acciones-1, y, si es falsa, las acciones-2. Instrucción condicional simple. Es igual pero carece de la rama "si_no", de modo que, si la expresión de falsa, no se realiza ninguna acción y la ejecución continúa por la siguiente instrucción
según (expresión) hacer inicio valor1: acciones-1 valor2: acciones-2 ... valorN: acciones-N si_no: acciones-si_no fin	Instrucción condicional múltiple. Se utiliza cuando hay más de dos condiciones posibles (verdadero o falso) . Se evalúa la expresión, que suele ser de tipo entero, y se busca un valor en la lista valor1, valor2,... valorN que coincida con ella, realizándose las acciones asociadas al valor coincidente. Si ningún valor de la lista coincide con la expresión del "según", se realizan las acciones de la parte "si_no".
mientras (condición) hacer inicio acciones fin	Bucle mientras. Las acciones se repiten en tanto la condición, que debe ser una expresión lógica, sea verdadera. La condición se evalúa antes de entrar al bloque de acciones, de modo que pueden no ejecutarse ninguna vez.
repetir inicio acciones fin mientras que (condición)	Bucle repetir. Las acciones se repiten en tanto que la condición, que debe ser una expresión lógica, sea verdadera. Se parece mucho al anterior, pero la condición se evalúa al final del bucle, por lo que éste se ejecuta, como mínimo, una vez.
para variable desde expr-ini hasta expr-fin hacer inicio acciones fin	Bucle para. Se evalúa la expresión expr-ini, que debe ser de tipo entero, y se asigna ese valor a la variable. Dicha variable se incrementa en una unidad en cada repetición de las acciones. Las acciones se repiten hasta que la variable alcanza el valor expr-fin.

Las instrucciones básicas del pseudocódigo

Hay varias instrucciones de pseudocódigo que son muy simples, así que las vamos a explicar ahora mismo, junto con un ejemplo para ir acostumbrándonos al aspecto de los algoritmos:

algoritmo: sirve para ponerle un nombre significativo al algoritmo

inicio: marca el principio de un proceso, de un módulo o, en general, de un conjunto de instrucciones

fin: marca el fin de un proceso, módulo o conjunto de instrucciones. "Inicio" y "fin" siempre van por parejas, es decir, cuando aparezca un "Inicio", debe existir un "fin" en algún sitio más abajo. Y al revés: todo "fin" se corresponde con algún "Inicio" que aparecerá más arriba.

= (asignación): se utiliza para asociar un valor a una variable, como vimos en el apartado anterior.

leer: sirve para leer un dato de un dispositivo de entrada (típicamente, el teclado)

escribir: sirve para enviar un dato a un dispositivo de salida (si no se indica otra cosa, la pantalla)

Ejemplo Volvemos al algoritmo del área y el perímetro de un rectángulo:

```

algoritmo rectángulo
inicio
    leer (base)
    leer (altura)
    área = base * altura
    perímetro = 2 * base + 2 * altura
    escribir (área)
    escribir (perímetro)
fin

```

Recuerda que los programas se ejecutan de arriba a abajo, una instrucción cada vez.

Cuando este programa se haya introducido en un ordenador y le pidamos que lo ejecute, la máquina irá mirando las instrucciones en el orden en que el programador las introdujo y las irá ejecutando. Veamos, instrucción por instrucción,

qué acciones provocan en el ordenador:

algoritmo rectángulo → simplemente, le pone título al algoritmo y marca su principio (esta instrucción no hace nada "útil")

Inicio → marca el comienzo de las instrucciones (por lo tanto, ni esta instrucción ni la anterior realizan ninguna tarea: sólo son marcas)

leer(base) → el ordenador se queda a la espera de que el usuario del programa introduzca algún dato a través del teclado. Cuando el usuario lo hace, ese dato queda almacenado en la variable "base". Supongamos que el usuario teclea un 7: será como haber hecho la asignación $\text{base} = 7$.

leer(altura) → vuelve a ocurrir lo mismo, pero ahora el dato tecleado se guarda en la variable "altura". Supongamos que se teclea un 2. Por lo tanto, $\text{altura} = 2$.

área = base * altura → según vimos en el apartado anterior, se evalúa la expresión situada a la derecha del símbolo "=". El resultado de la misma será $7 * 2$, es decir, 14. Ese valor se asigna a la variable situada a la izquierda del "=". Por lo tanto, $\text{área} = 14$.

perímetro = 2 * base + 2 * altura → en esta ocasión, la evaluación de la expresión da como resultado 18, que se asigna a la variable perímetro, o sea, $\text{perímetro} = 18$.

escribir(área) → el ordenador muestra en la pantalla el valor de la variable área, que es 14.

escribir(perímetro) → el ordenador muestra en la pantalla el valor de la variable perímetro, es decir, 18.

Fin → marca el punto final del algoritmo

Podemos concluir que el algoritmo presentado resuelve el problema de calcular el área y el perímetro de un rectángulo y posee las tres cualidades básicas de todo algoritmo: precisión, definición y finitud.

Declaración de variables y constantes

Como regla general, diremos que todas las **variables** deben ser **declaradas ANTES de usarse por primera vez**. Recuerda que la declaración se usa para comunicar al ordenador el tipo y el identificador de cada variable.

La sintaxis de estas declaraciones es como sigue:

```
variables
nombre_de_variable es tipo_de_datos
```

Ejemplo Si te fijas en el ejemplo anterior, no hemos declarado ninguna de las variables del algoritmo y, por lo tanto, éste no es del todo correcto. Vamos a completarlo:

```
algoritmo rectángulo
variables
base es real
altura es real
área es real
perímetro es real
inicio
leer (base)
leer (altura)
área = base * altura
perímetro = 2 * base + 2 * altura
escribir (área)
escribir (perímetro)
fin
```

Fíjate que hemos definido las variables antes del inicio de las instrucciones del algoritmo.

A veces, también es útil declarar ciertas **constantes** para usar valores que no van a cambiar en todo el transcurso del programa. Las constantes se deben declarar también antes del inicio de las instrucciones del programa.

Ejemplo de declaración de constantes:

```
algoritmo ejemplo
constantes
pi = 3.141592
g = 9.8
txt = "En un lugar de La Mancha"
inicio
...instrucciones...
fin
```

4 La programación estructurada

4.1 Teorema de la programación estructurada

El término *programación estructurada* se refiere a un conjunto de técnicas que han ido evolucionando desde los primeros trabajos del holandés E. Dijkstra⁶. Estas técnicas aumentan la productividad del programador, reduciendo el tiempo requerido para escribir, verificar, depurar y mantener los programas.

Allá por mayo de 1966, Böhm y Jacopini⁷ demostraron que se puede escribir cualquier *programa propio* utilizando solo **tres tipos de estructuras de control**: la secuencial, la selectiva (o condicional) y la repetitiva. A esto se le llama *Teorema de la programación estructurada*, y define un *programa propio* como un programa que cumple tres características:

Posee un sólo punto de inicio y un sólo punto de fin

Existe al menos un camino que parte del inicio y llega hasta el fin pasando por todas las partes del programa

No existen bucles infinitos

Realmente, el trabajo de Dijkstra basado en este teorema fue revolucionario, porque lo que venía a decir es que, para construir programas más potentes y en menos tiempo, lo que había que hacer era simplificar las herramientas que se utilizaban para hacerlos, en lugar de complicarlas más. Este regreso a la simplicidad, unido a las técnicas de ingeniería del software, acabó con la crisis del software de los años 70.

Por lo tanto, **los programas estructurados deben limitarse a usar tres estructuras**:

Secuencial

Selectiva (o condicional)

Repetitiva

Vamos a estudiar cada estructura detenidamente y veremos cómo se representan mediante diagramas de flujo y pseudocódigo.

4.2 Estructura secuencial

La estructura secuencial es aquella en la que **una acción sigue a otra** (en secuencia). Esta es la estructura algorítmica básica, en la que las instrucciones se ejecutan una tras otra, en el mismo orden en el que fueron escritas.

La estructura secuencial, por lo tanto, es la más simple de las tres estructuras permitidas. A continuación vemos su representación mediante diagrama de flujo y pseudocódigo:

```

inicio
  acción 1
  acción 2
  ...
  acción N
fin

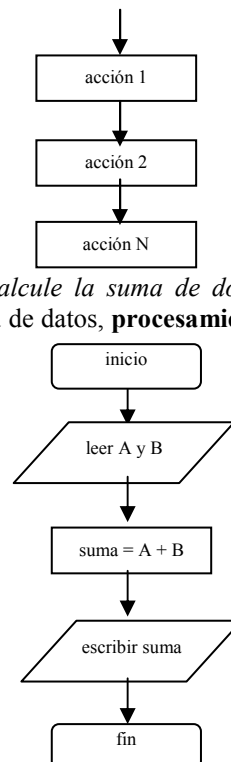
```

Ejemplo *Vamos a escribir un algoritmo completamente secuencial que calcule la suma de dos números, A y B. Recuerda que, generalmente, los algoritmos se dividen en tres partes: **entrada** de datos, **procesamiento** de esos datos y **salida** de resultados.*

```

algoritmo suma
variables
  A, B, suma son enteros
inicio
  leer (A)
  leer (B)
  suma = A + B
  escribir (suma)
fin

```



⁶ Dijkstra, pese a ser físico, se convirtió en uno de los más importantes científicos de la computación hasta su muerte en 2002. Una de sus frases más famosas es: "la pregunta de si un computador puede pensar no es más interesante que la pregunta de si un submarino puede nadar"

⁷ BÖHM, C. y JACOPINI, G.; *Flow diagrams, turing machines and languages only with two formation rules*, Communications of the ACM, vol.9, nº 5, pg. 366-371, 1966

4.3 Estructuras selectivas (condicionales)

Los algoritmos que usan únicamente estructuras secuenciales están muy limitados y no tienen ninguna utilidad real. Esa utilidad aparece cuando existe la posibilidad de **ejecutar una de entre varias** secuencias de instrucciones dependiendo de alguna condición asociada a los datos del programa.

Las estructuras selectivas pueden ser de tres tipos:

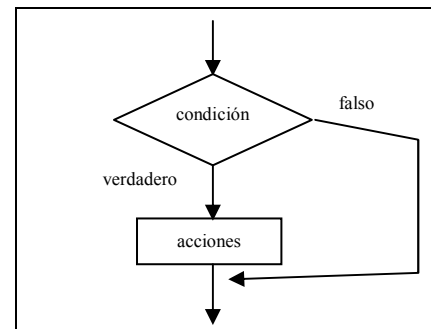
simples
dobles
múltiples

4.3.1 Condicional simple

La estructura condicional **simple** tiene esta representación:

```

si condición entonces
inicio
    acciones
fin
  
```



La condición que aparece entre "si" y "entonces" es siempre una **expresión lógica**, es decir, una expresión cuyo resultado es "verdadero" o "falso". Si el resultado es verdadero, entonces se ejecutan las acciones situadas entre "inicio" y "fin". Si es falso, se saltan las acciones y se prosigue por la siguiente instrucción (lo que haya debajo de "fin")

Ejemplo Recuperemos algoritmo del área y el perímetro del rectángulo para mostrar la condicional simple en pseudocódigo (el diagrama de flujo lo tienes en la página 20)

```

algoritmo rectángulo
variables
    base, altura, área, perímetro son reales
inicio
    leer (base)
    leer (altura)
    si (área > 0) y (altura > 0) entonces
        inicio
            área = base * altura
            perímetro = 2 * base + 2 * altura
            escribir (área)
            escribir (perímetro)
        fin
    si (área <= 0) o (altura <= 0) entonces
        inicio
            escribir ('Los datos son incorrectos')
        fin
    fin
  
```

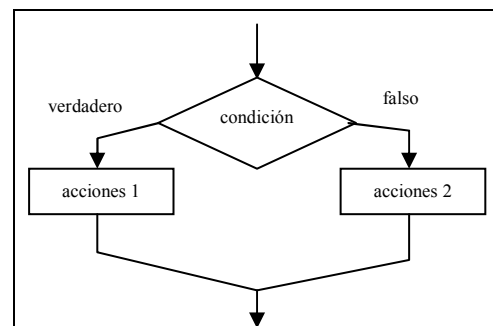
Observa que, en la primera instrucción condicional (**si** (área > 0) **y** (altura > 0) **entonces**) se comprueba que los dos datos sean positivos; en caso de serlo, se procede al cálculo del área y el perímetro mediante las acciones situadas entre **inicio** y **fin**. Más abajo hay otra condicional (**si** (área <= 0) **o** (altura <= 0) **entonces**) para el caso de que alguno de los datos sea negativo o cero: en esta ocasión, se imprime en la pantalla un mensaje de error.

4.3.2 Condicional doble

La **forma doble** de la instrucción condicional es:

```

si condición entonces
    inicio
        acciones-1
    fin
si_no
    inicio
        acciones-2
    fin
  
```



En esta forma, la instrucción funciona del siguiente modo: si el resultado de la condición es verdadero, entonces se ejecutan las acciones de la primera parte, es decir, las acciones-1. Si

es falso, se ejecutan las acciones de la parte "si_no", es decir, las acciones-2.

Ejemplo Podemos reescribir nuestro algoritmo del rectángulo usando una alternativa doble:

```

algoritmo rectángulo
variables
  base, altura, área, perímetro son reales
inicio
  leer (base)
  leer (altura)
  si (área > 0) y (altura > 0) entonces
    inicio
      área = base * altura
      perímetro = 2 * base + 2 * altura
      escribir (área)
      escribir (perímetro)
    fin
  si_no
    inicio
      escribir ('Los datos de entrada son incorrectos')
    fin
fin

```

Lo más interesante de este algoritmo es compararlo con el anterior, ya que hace exactamente lo mismo. ¡Siempre hay varias maneras de resolver el mismo problema! Pero esta solución es un poco más sencilla, al ahorrarse la segunda condición, que va implícita en el **si_no**. Observa también que este algoritmo en pseudocódigo es equivalente al diagrama de flujo de la página 20.

4.3.3 Condicional múltiple

En algunas ocasiones nos encontraremos con **selecciones en las que hay más de dos alternativas** (es decir, en las que no basta con los valores "verdadero" y "falso"). Siempre es posible plasmar estas selecciones complejas usando varias estructuras **si-entonces-si_no** anidadas, es decir, unas dentro de otras, pero, cuando el número de alternativas es grande, esta solución puede plantear grandes problemas de escritura y legibilidad del algoritmo.

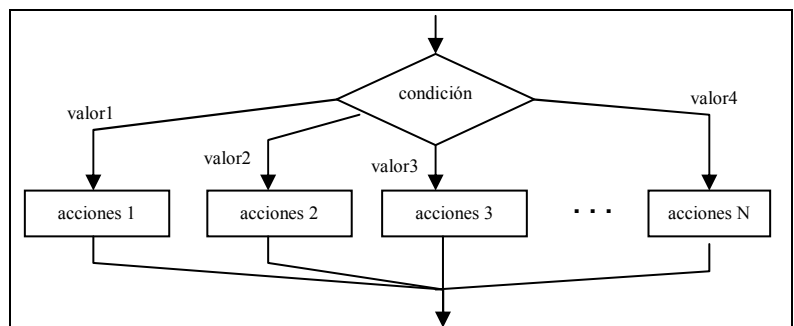
Sin embargo, hay que dejar clara una cosa: cualquier instrucción condicional múltiple puede ser sustituida por un conjunto de instrucciones condicionales simples y dobles totalmente equivalente.

La **estructura condicional múltiple** sirve, por tanto, para simplificar estos casos de condiciones con muchas alternativas. Su sintaxis general es:

```

según expresión hacer
inicio
  valor1: acciones-1
  valor2: acciones-2
  valor3: acciones-3
  ...
  valorN: acciones-N
si_no: acciones-si_no
fin

```



Su funcionamiento es el siguiente: se evalúa **expresión**, que en esta ocasión no puede ser de tipo lógico, sino entero, carácter, etc⁸. El resultado de **expresión** se compara con cada uno de los valores **valor1, valor2... valorN**. Si coincide con alguno de ellas, se ejecutan únicamente las acciones situadas a la derecha del valor coincidente (**acciones-1, acciones-2... acciones-N**). Si se diera el caso de que ningún valor fuera coincidente, entonces se ejecutan las **acciones-si_no** ubicadas al final de la estructura. Esta última parte de la estructura no es obligatorio que aparezca.

Ejemplo Construyamos un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable entera llamada "día". Su valor se introducirá por teclado. Los valores posibles de la variable "día" serán del 1 al 7: cualquier otro valor debe producir un error.

```

algoritmo día_semana
variables
  día es entero
inicio
  leer (día)
  según (día) hacer

```

⁸ Sin embargo, no suele admitirse una expresión de tipo real por motivos en los que ahora no nos vamos a detener. Lo más habitual es que sea de tipo entero.

```

inicio
1: escribir('lunes')
2: escribir('martes')
3: escribir('miércoles')
4: escribir('jueves')
5: escribir('viernes')
6: escribir('sábado')
7: escribir('domingo')
si_no: escribir('Error: el día introducido no existe')
fin

```

Hay dos cosas interesantes en este algoritmo. Primera, el uso de la instrucción selectiva múltiple: la variable **día**, una vez leída, se compara con los siete valores posibles. Si vale 1, se realizará la acción **escribir**('lunes'); si vale 2, se realiza **escribir**('martes'); y así sucesivamente. Por último, si no coincide con ninguno de los siete valores, se ejecuta la parte **si_no**. Es conveniente que pienses cómo se podría resolver el mismo problema sin recurrir a la alternativa múltiple, es decir, utilizando sólo alternativas simples y dobles.

El otro aspecto digno de destacarse no tiene nada que ver con la alternativa múltiple, sino con la sintaxis general de pseudocódigo: no hemos empleado **inicio** y **fin** para marcar cada bloque de instrucciones. Lo más correcto hubiera sido escribirlo así:

```

según día hacer
inicio
1: inicio
   escribir('lunes')
   fin
2: inicio
   escribir('martes')
   fin
..etc..

```

Sin embargo, cuando el bloque de instrucciones consta sólo de UNA instrucción, podemos prescindir de las marcas de **inicio** y **fin** y escribir directamente la instrucción.

4.4 Estructuras repetitivas (bucles)

Los ordenadores se diseñaron inicialmente para **realizar tareas sencillas y repetitivas**. El ser humano es de lo más torpe acometiendo tareas repetitivas: pronto le falla la concentración y comienza a tener descuidos. Los ordenadores programables, en cambio, pueden realizar la misma tarea muchas veces por segundo durante años y nunca se aburren (o, al menos, hasta hoy no se ha tenido constancia de ello)

La **estructura repetitiva**, por tanto, reside en la naturaleza misma de los ordenadores y consiste, simplemente, en **repetir varias veces un conjunto de instrucciones**. Las estructuras repetitivas también se llaman **bucles, lazos o iteraciones**. Nosotros preferiremos la denominación "bucle".

Los bucles tienen que repetir un conjunto de instrucciones **un número finito de veces**. Si no, nos encontraremos con un **bucle infinito** y el algoritmo no funcionará. En rigor, ni siquiera será un algoritmo, ya que no cumplirá la condición de finitud.

El **bucle infinito** es un peligro que acecha constantemente a los programadores y nos toparemos con él muchas veces a lo largo de este curso. Para conseguir que el bucle se repita sólo un número finito de veces, tiene que existir una **condición de salida** del mismo, es decir, una situación en la que ya no sea necesario seguir repitiendo las instrucciones.

Por tanto, los bucles se componen, básicamente, de dos elementos:

*un **cuerpo del bucle** o conjunto de instrucciones que se ejecutan repetidamente*

*una **condición de salida** para dejar de repetir las instrucciones y continuar con el resto del algoritmo*

Dependiendo de dónde se coloque la condición de salida (al principio o al final del conjunto de instrucciones repetidas), y de la forma de realizarla, existen tres tipos de bucles, aunque hay que resaltar que, con el primer tipo, se puede programar cualquier estructura iterativa. Pero con los otros dos, a veces el programa resulta más claro y legible. Los tres tipos de bucle se denominan:

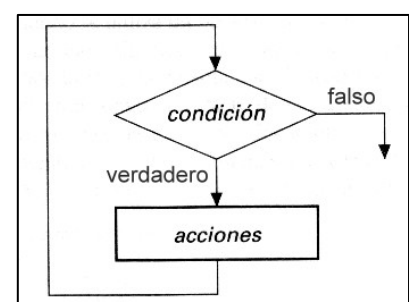
***Bucle "mientras"**: la condición de salida está al principio del bucle.*

***Bucle "repetir"**: la condición de salida está al final del bucle.*

***Bucle "para"**: la condición de salida está al principio y se realiza con un contador automático.*

4.4.1 Bucle "mientras"

El bucle "mientras" es una estructura que se repite **mientras una condición sea verdadera**. La condición, en forma de expresión lógica, se escribe en la cabecera del bucle, y a continuación aparecen las acciones que se repiten



(cuerpo del bucle):

```

mientras (condición) hacer
inicio
    acciones (cuerpo del bucle)
fin

```

Cuando se llega a una instrucción **mientras**, se evalúa la condición. Si es verdadera, se realizan las acciones y, al terminar el bloque de acciones, se regresa a la instrucción **mientras** (he aquí el bucle o lazo). Se vuelve a evaluar la condición y, si sigue siendo verdadera, vuelve a repetirse el bloque de acciones. Y así, sin parar, hasta que la condición se haga falsa.

Ejemplo Escribir un algoritmo que muestre en la pantalla todos los números enteros entre 1 y 100

```

algoritmo contar
variables
    cont es entero
inicio
    cont = 0
    mientras (cont <= 100) hacer
        inicio
            cont = cont + 1
            escribir (cont)
        fin
    fin

```

Aquí observamos el uso de **un contador en la condición de salida** de un bucle, un elemento muy común en estas estructuras. Observa la evolución del algoritmo:

cont = 0. Se le asigna el valor 0 a la variable **cont** (contador)

mientras (cont <= 100) hacer. Condición de salida del bucle. Es verdadera porque **cont** vale 0, y por lo tanto es menor o igual que 100.

cont = cont + 1. Se incrementa el valor de **cont** en una unidad. Como valía 0, ahora vale 1.

escribir(cont). Se escribe el valor de **cont**, que será 1.

Después, el flujo del programa regresa a la instrucción **mientras**, ya que estamos en un bucle, y se vuelve a evaluar la condición. Ahora **cont** vale 1, luego sigue siendo verdadera. Se repiten las instrucciones del bucle, y **cont** se incrementa de nuevo, pasando a valer 2. Luego valdrá 3, luego 4, y así sucesivamente.

La condición de salida del bucle hace que éste se repita mientras **cont** valga menos de 101. De este modo nos aseguramos de escribir todos los números hasta el 100.

Lo más problemático a la hora de diseñar un bucle es, por lo tanto, **pensar bien su condición de salida**, porque si la condición de salida nunca se hiciera falsa, caeríamos en un bucle infinito. Por lo tanto, **la variable implicada en la condición de salida debe sufrir alguna modificación en el interior del bucle**; si no, la condición siempre sería verdadera. En nuestro ejemplo, la variable **cont** se modifica en el interior del bucle: por eso llega un momento, después de 100 repeticiones, en el que la condición se hace falsa y el bucle termina.

4.4.2 Bucle "repetir"

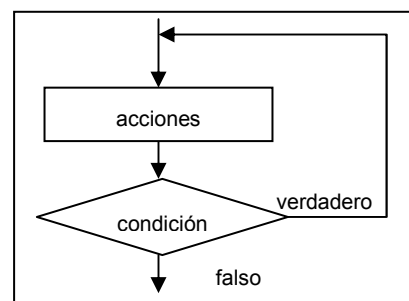
El bucle de tipo "repetir" es muy similar al bucle "mientras", con la salvedad de que **la condición de salida se evalúa al final del bucle**, y no al principio, como a continuación veremos. Todo bucle "repetir" puede escribirse como un bucle "mientras", pero al revés no siempre sucede.

La forma de la estructura "repetir" es la que sigue:

```

repetir
inicio
    acciones
fin
mientras que (condición)

```



Cuando el ordenador encuentra un bucle de este tipo, ejecuta las acciones escritas entre **inicio** y **fin** y, después, evalúa la **condición**, que debe ser de tipo lógico. Si el resultado es falso, se vuelven a repetir las acciones. Si el resultado es verdadero, el bucle se repite. Si es falso, se sale del bucle y se continúa ejecutando la siguiente instrucción.

Existe, pues, una diferencia fundamental con respecto al bucle "mientras": **la condición se evalúa al final**. Por lo tanto, las acciones del cuerpo de un bucle "repetir" se ejecutan **al menos una vez**, cuando en un bucle "mientras" es posible que no se ejecuten ninguna (si la condición de salida es falsa desde el principio)

Ejemplo Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, pero esta vez utilizando un bucle "repetir" en lugar de un bucle "mientras"

```

algoritmo contar
variables
  cont es entero
inicio
  cont = 0
  repetir
    inicio
      cont = cont + 1
      escribir (cont)
    fin
  mientras que (cont <= 100)
fin

```

Observa que el algoritmo es básicamente el mismo que en el ejemplo anterior, pero hemos cambiado el lugar de la condición de salida.

4.4.3 Bucle "para"

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones del cuerpo del bucle. Cuando el número de repeticiones es fijo, lo más cómodo es usar un bucle "para", aunque sería perfectamente posible sustituirlo por uno "mientras".

La estructura "para" **repite las acciones del bucle un número prefijado de veces e incrementa automáticamente una variable contador** en cada repetición. Su forma general es:

```

para cont desde valor_inicial hasta valor_final hacer
inicio
  acciones
fin

```

cont es la variable contador. La primera vez que se ejecutan las acciones situadas entre **inicio** y **fin**, la variable **cont** tiene el valor especificado en la expresión **valor_inicial**. En la siguiente repetición, **cont** se incrementa en una unidad, y así sucesivamente, hasta alcanzar el **valor_final**. Cuando esto ocurre, el bucle se ejecuta por última vez y después el programa continúa por la instrucción que haya a continuación.

El **incremento** de la variable **cont** siempre es de 1 en cada repetición del bucle, salvo que se indique otra cosa. Por esta razón, la estructura "para" tiene una **sintaxis alternativa**:

```

para cont desde valor_inicial hasta valor_final inc | dec paso hacer
inicio
  acciones
fin

```

De esta forma, se puede especificar si la variable **cont** debe incrementarse (**inc**) o decrementarse (**dec**) en cada repetición, y en qué cantidad (**paso**).

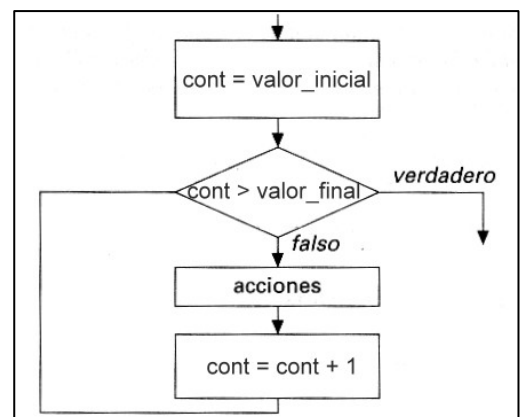
Ejemplo 1 Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, utilizando un bucle "para"

```

algoritmo contar
variables
  cont es entero
inicio
  para cont desde 1 hasta 100 hacer
    inicio
      escribir (cont)
    fin
  fin

```

De nuevo, lo más interesante es observar las diferencias de este algoritmo con los dos ejemplos anteriores. Advierte que ahora no es necesario asignar un valor inicial de 0 a **cont**, ya que se hace implícitamente en el mismo bucle; y tampoco es necesario incrementar el valor de **cont** en el cuerpo del bucle (**cont = cont + 1**), ya que de eso se encarga el propio bucle "para". Por último, no hay que escribir condición de salida, ya que el bucle "para" se repite hasta que **cont**



vale 100 (inclusive)

Ejemplo 2 Diseñar un algoritmo que escriba todos los números enteros **impares** entre 1 y 100, utilizando un bucle "para"

```
algoritmo contar
variables
  cont es entero
inicio
  para cont desde 1 hasta 100 inc 2 hacer
  inicio
    escribir (cont)
  fin
fin
```

Este ejemplo, similar al anterior, sirve para ver el uso de la sintaxis alternativa del bucle "para". La variable **cont** se incrementará en 2 unidades en cada repetición del bucle.

4.5 Contadores, acumuladores, conmutadores

Asociadas a los bucles se encuentran a menudo algunas variables auxiliares. Como siempre se utilizan de la misma manera, las llamamos con un nombre propio (contador, acumulador, etc.), pero hay que dejar claro que no son más que **variables comunes, aunque se usan de un modo especial**.

Contadores

Un contador es una variable (casi siempre de **tipo entero**) cuyo valor **se incrementa o decrementa en cada repetición** de un bucle. Es habitual llamar a esta variable "cont" (de contador) o "i" (de índice).

El **contador** suele usarse de este modo:

- 1) **Se inicializa antes de que comience el bucle.** Es decir, se le da un valor inicial. Por ejemplo:

```
cont = 5
```

- 2) **Se modifica dentro del cuerpo del bucle.** Lo más habitual es que se **incremente** su valor en una unidad. Por ejemplo:

```
cont = cont + 1
```

Esto quiere decir que el valor de la variable "cont" se incrementa en una unidad y es asignado de nuevo a la variable contador. Es decir, si cont valía 5 antes de esta instrucción, cont valdrá 6 después.

Otra forma típica del contador es:

```
cont = cont - 1
```

En este caso, la variable se **decrementa** en una unidad; si cont valía 5 antes de la instrucción, tendremos que cont valdrá 4 después de su ejecución.

El incremento o decremento no tiene por qué ser de una unidad. La cantidad que haya que incrementar o decrementar vendrá dada por la naturaleza del problema.

- 3) **Se utiliza en la condición de salida del bucle.** Normalmente, se compara con el valor máximo (o mínimo) que debe alcanzar el contador para dejar de repetir las instrucciones del bucle.

Ejemplo Escribir un algoritmo que escriba la tabla de multiplicar hasta el 100 de un número *N* introducido por el usuario

```
algoritmo tabla_multiplicar
variables
  cont es entero
  N es entero
inicio
  leer (N)
  cont = 1
  mientras (cont <= 100) hacer
  inicio
    escribir (N * cont)
    cont = cont + 1
  fin
fin
```

El uso de contadores es casi obligado en bucles "mientras" y "repetir" que deben ejecutarse un determinado número de veces. Recuerda que siempre hay que asignar al contador un **valor inicial** para la primera ejecución del bucle (**cont = 1** en nuestro ejemplo) e ir **incrementándolo** (o decrementándolo, según el algoritmo) en cada repetición con una instrucción del tipo **cont = cont + 1** en el cuerpo del bucle. De lo contrario habremos escrito un bucle infinito.

Por último, hay que prestar atención a la **condición de salida**, que debe estar asociada al valor del contador en la última

repetición del bucle (en nuestro caso, 100). Mucho cuidado con el operador relacional (<, >, <=, >=, etc) que usemos, porque el bucle se puede ejecutar más o menos veces de lo previsto⁹.

Acumuladores

Las variables acumuladoras tienen la misión de **almacenar resultados sucesivos**, es decir, de *acumular* resultados, de ahí su nombre.

Las variables acumuladores también debe ser **inicializadas**. Si llamamos "acum" a un acumulador, escribiremos antes de iniciar el bucle algo como esto:

```
acum = 0
```

Por supuesto, el valor inicial puede cambiar, dependiendo de la naturaleza del problema. Más tarde, en el **cuerpo del bucle**, la forma en la que nos la solemos encontrar es:

```
acum = acum + N
```

...siendo N otra variable. Si esta instrucción va seguida de otras:

```
acum = acum + M
acum = acum + P
```

... estaremos acumulando en la variable "acum" los valores de las variables M, N, P, etc, lo cual resulta a veces muy útil para resolver ciertos problemas repetitivos.

Ejemplo *Escribir un algoritmo que pida 10 números por el teclado y los sume, escribiendo el resultado*

```
algoritmo sumar10
variables
  cont es entero
  suma es entero
  N es entero
inicio
  suma = 0
  para cont desde 1 hasta 10 hacer
    inicio
      leer (N)
      suma = suma + N
    fin
  escribir (suma)
fin
```

En este algoritmo, **cont** es una variable contador típica de bucle. Se ha usado un bucle "para", que es lo más sencillo cuando conocemos previamente el número de repeticiones (10 en este caso). La variable **N** se usa para cada uno de los números introducidos por el teclado, y la variable **suma** es el **acumulador**, donde se van sumando los diferentes valores que toma **N** en cada repetición.

Observa como, al principio del algoritmo, se le asigna al acumulador el valor 0. Esta es una precaución importante que se debe tomar siempre porque el valor que tenga una variable que no haya sido usada antes es **desconocido** (no tiene por qué ser 0)

Conmutadores

Un **conmutador** (o **interruptor**) es una variable que sólo puede tomar **dos valores**. Pueden ser, por tanto, de tipo booleano, aunque también pueden usarse variables enteras o de tipo carácter.

La variable conmutador recibirá uno de los dos valores posibles **antes de entrar en el bucle**. Dentro del **cuerpo** del bucle, debe **cambiarse ese valor** bajo ciertas condiciones. Utilizando el conmutador en la **condición de salida** del bucle, puede controlarse el número de repeticiones.

Ejemplo *Escribir un algoritmo que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo.*

```
algoritmo sumar
variables
  suma es entero
  N es entero
  terminar es lógico
inicio
  suma = 0
  terminar = falso
  mientras (terminar == falso)
```

⁹ Hay que evitar el operador "==" en las condiciones de salida de los bucles, sobre todo si estamos trabajando con números reales.

```

inicio
  escribir ('Introduce un número (negativo para terminar)')
  leer (N)
  si (N >= 0) entonces
    suma = suma + N
  si_no
    terminar = verdadero
  fin
fin
escribir (suma)
fin

```

Este algoritmo es una variación del ejemplo con acumuladores (página 30). Entonces el usuario introducía 10 números, y ahora puede ir introduciendo números indefinidamente, hasta que se canse. ¿Cómo indica al ordenador que ha terminado de introducir números? Simplemente, tecleando un número negativo.

El bucle se controla por medio de la variable "terminar": es el **conmutador**. Sólo puede tomar dos valores: "verdadero", cuando el bucle debe terminar, y "falso", cuando el bucle debe repetirse una vez más. Por lo tanto, "terminar" valdrá "falso" al principio, y sólo cambiará a "verdadero" cuando el usuario introduzca un número negativo.

A veces, el conmutador puede tomar más de dos valores. Entonces ya no se le debe llamar, estrictamente hablando, conmutador. Cuando la variable toma un determinado valor especial, el bucle termina. A ese "valor especial" se le suele denominar valor **centinela**.

Ejemplo *Escribir un algoritmo que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo.*

```

algoritmo sumar
variables
  suma es entero
  N es entero
inicio
  suma = 0
  repetir
    inicio
      escribir ('Introduce un número (negativo para terminar)')
      leer (N)
      si (N >= 0) entonces
        suma = suma + N
      fin
    mientras que (N >= 0)
      escribir (suma)
  fin

```

Tenemos aquí un ejemplo de cómo no siempre es necesario usar contadores para terminar un bucle "mientras" (o "repetir"). Las repeticiones se controlan con la variable N, de modo que el bucle termina cuando $N < 0$. Ese es el valor **centinela**.

4.6 Reglas de estilo

La escritura de un algoritmo debe ser siempre lo más clara posible, ya se esté escribiendo en pseudocódigo o en un lenguaje de programación real. La razón es evidente: los algoritmos pueden llegar a ser muy complejos, y, si a su complejidad le añadimos una escritura sucia y desordenada, se volverán ininteligibles.

Esto es un aviso para navegantes. Todos los programadores han experimentado la frustración que se siente al ir a **revisar un algoritmo redactado pocos días antes y no entender ni una palabra** de lo que uno mismo escribió. Multiplíquese esto por mil en el caso de revisión de algoritmos escritos por otras personas.

Por esta razón, y ya desde el principio, debemos acostumbrarnos a **respetar ciertas reglas básicas de estilo**. Ciertamente que cada programador puede luego desarrollar su estilo propio, pero siempre dentro de un marco aceptado por la mayoría.

4.6.1 Partes de un algoritmo

Los algoritmos deberían tener siempre una **estructura en tres partes**:

```

1 - Cabecera
2 - Declaraciones
3 - Acciones

```

Algunos lenguajes, C entre ellos, son lo bastante flexibles como para permitir saltarse a la torera esta estructura, pero es una buena costumbre respetarla siempre:

*La **cabecera**: contiene el nombre del programa o algoritmo.*

*Las **declaraciones**: contiene las declaraciones de variables y constantes que se usan en el algoritmo*

*Las **acciones**: son el cuerpo en sí del algoritmo, es decir, las instrucciones*

Puedes observar esta estructura en todos los ejemplos del tema.

4.6.2 Documentación

La documentación del programa comprende el conjunto de **información interna y externa** que facilita su posterior mantenimiento.

*La **documentación externa** la forman todos los documentos ajenos al programa: guías de instalación, guías de usuario, etc.*

*La **documentación interna** es la que acompaña al programa. Nosotros sólo nos ocuparemos, por ahora, de esta documentación.*

La forma más habitual de plasmar la **documentación interna** es por medio de **comentarios** significativos que acompañen a las instrucciones del algoritmo o programa. Los comentarios son **líneas de texto** insertadas entre las instrucciones, o bien al lado, que se ignoran durante la ejecución del programa y aclaran el funcionamiento del algoritmo a cualquier programador que pueda leerlo en el futuro.

Para que el ordenador sepa qué debe ignorar y qué debe ejecutar, **los comentarios se escriben precedidos de determinados símbolos** que la máquina interpreta como "principio de comentario" o "fin de comentario".

Los símbolos que marcan las zonas de comentario dependen del lenguaje de programación, como es lógico. Así, por ejemplo, en Pascal se escriben encerrados entre los símbolos (* y *):

```
(* Esto es un comentario en Pascal *)
```

El **lenguaje C**, sin embargo, utiliza los símbolos /* y */ para marcar los comentarios. Además, C++ permite emplear la doble barra (//) para comentarios que ocupen sólo una línea. Nosotros usaremos indistintamente estos dos métodos:

```
/* Esto es un comentario en C */
// Esto es un comentario en C++
```

Ejemplo Escribir un algoritmo que sume todos los números naturales de 1 hasta 1000

```
algoritmo sumar1000
/* Función: Sumar los números naturales entre 1 y 1000
Autor: Jaime Tralleta
Fecha: 12-12-04 */

variables
    cont es entero           /* variable contador */
    suma es entero           /* variable acumulador */
    N es entero

inicio
    suma = 0                 /* se pone el acumulador a 0 */
    para cont desde 1 hasta 1000 hacer
        inicio
            suma = suma + cont /* los números se suman al acumulador */
        fin
    escribir (suma)
fin
```

Este es un ejemplo de **algoritmo comentado**. Observa que los comentarios aparecen a la derecha de las instrucciones, encerrados entre llaves. A efectos de ejecución, se ignora todo lo que haya escrito entre los símbolos /* y */, pero a efectos de documentación y mantenimiento, lo que haya escrito en los comentarios puede ser importantísimo.

Una buena e interesante costumbre es incluir un **comentario al principio** de cada algoritmo que explique bien **la función** del mismo y, si se considera necesario, el autor, la fecha de modificación y cualquier otra información que se considere interesante.

Pero ¡cuidado! Comentar un programa en exceso no sólo es tedioso para el programador, sino contraproducente, porque un **exceso de documentación** lo puede hacer más ilegible. Sólo hay que insertar comentarios en los puntos que se considere que necesitan una explicación. En este sentido, el algoritmo del ejemplo está **demasiado comentado**.

4.6.3 Estilo de escritura

A lo largo de esta unidad has podido ver diversos ejemplos de algoritmos. Si te fijas en ellos, todos siguen ciertas convenciones en el uso de la tipografía, las sangrías, los espacios, etc. Escribir los algoritmos cumpliendo estas reglas es una sana costumbre.

Sangrías

Las instrucciones que aparezcan **debajo de "inicio"** deben tener una **sangría mayor** que dicha instrucción. Ésta sangría se mantendrá hasta la aparición del "fin" correspondiente. Esto es particularmente importante cumplirlo si existen varios bloques inicio–fin anidados. Asimismo, un algoritmo es más fácil de leer si los **comentarios** tienen todos la misma sangría.

Ejemplo Escribir un algoritmo que determine, entre dos números *A* y *B*, cuál es el mayor o si son iguales. Observa bien las **sangrías** de cada bloque de instrucciones, así como la posición alineada de los **comentarios**.

```

algoritmo comparar
// Función: Comparar dos números A y B
variables
  A,B son enteros
inicio
  leer (A)                // leemos los dos números del teclado
  leer (B)
  si (A == B) entonces    // los números son iguales
  inicio
    escribir ('Los dos números son iguales')
  fin
  si_no                  // los números son distintos, así que
  inicio                  // vamos a compararlos entre sí
    si (A > B) entonces
    inicio                // A es mayor
      escribir ('A es mayor que B')
    fin
    si_no
    inicio                // B es mayor
      escribir ('B es mayor que A')
    fin
  fin
fin

```

Prescindir de "inicio" y "fin"

Cuando un bloque de instrucciones **sólo contiene una instrucción**, podemos escribirla directamente, sin necesidad de encerrarla entre un "inicio" y un "fin". Esto suele redundar en una mayor facilidad de lectura.

Ejemplo Repetiremos el mismo ejemplo anterior, prescindiendo de los "inicio" y "fin" que no sean necesarios. Fíjate en que el algoritmo es más corto y, por lo tanto, más fácil de leer y entender.

```

algoritmo comparar
// Función: Comparar dos números A y B
variables
  A,B son enteros
inicio
  leer (A)                // leemos los dos números del teclado
  leer (B)
  si (A == B) entonces    // los números son iguales
    escribir ('Los dos números son iguales')
  si_no                  // los números son distintos, así que
  inicio                  // vamos a compararlos entre sí
    si (A > B) entonces    // A es mayor
      escribir ('A es mayor que B')
    si_no                // B es mayor
      escribir ('B es mayor que A')
    fin
  fin
fin

```

Tipografía

En todos los ejemplos del tema hemos resaltado las palabras del pseudocódigo en negrita, para distinguirlas de identificadores de variable, símbolos, etc. Esto también aumenta la legibilidad del algoritmo, pero, cuando utilicemos un lenguaje de programación real, no será necesario hacerlo, ya que los editores de texto que se usan en programación suelen estar preparados para resaltar las palabras reservadas.

Ahora bien, si vas a escribir un algoritmo con un procesador de texto normal o usando pseudocódigo, es conveniente que uses una **fuentes de tamaño fijo** (el tipo **Courier** va bastante bien, aunque nosotros hemos usado **Lucida Console**, que es algo más gruesa y se ve mejor en las fotocopias) y distingues en **negrita** las palabras del lenguaje para facilitar la lectura de los algoritmos.

Espacios

Otro elemento que aumenta la legibilidad es **espaciado** suficientemente (pero no demasiado) los distintos elementos de cada instrucción. Por ejemplo, esta instrucción ya es bastante complicada y difícil de leer:

```
si (a > b) y (c > d * raiz(k) ) entonces a = k + 5.7 * b
```

Pero se lee mucho mejor que esta otra, en la que se han suprimido los espacios (excepto los imprescindibles):

```
si (a>b)y(c>d*raiz(k))entonces a=k+5.7*b
```

Al ordenador le dará igual si escribimos $(a > b)$ o $(a>b)$, pero a cualquier programador que deba leer nuestro código le resultará mucho más cómoda la primera forma.

Por la misma razón, también es conveniente dejar **líneas en blanco** entre determinadas instrucciones del algoritmo cuando se considere que mejora la legibilidad.

Identificadores

A la hora de elegir identificadores de variables (o de constantes) es muy importante **utilizar nombres que sean significativos**, es decir, que den una idea de la información que almacena esa variable. Por ejemplo, si en un programa de nóminas vamos a guardar en una variable la edad de los empleados, es una buena ocurrencia llamar a esa variable "edad", pero no llamarla "X", "A" o "cosa".

Ahora bien, dentro de esta política de elegir identificadores significativos, es conveniente optar por aquellos que sean **lo más cortos posible**, siempre que sean descifrables. Así, un identificador llamado "edad_de_los_empleados" es engorroso de escribir y leer, sobre todo si aparece muchas veces en el algoritmo, cuando probablemente "edad_empl" proporciona la misma información. Sin embargo, si lo acortamos demasiado (por ejemplo "ed_em") llegará un momento en el que quede claro lo que significa.

Toda esta idea de significación de los identificadores es **extensible** a los nombres de los algoritmos, de las funciones, de los procedimientos, de los archivos y, en general, de **todos los objetos** relacionados con un programa.

Por último, señalar que muchos lenguajes de programación **distinguen entre mayúsculas y minúsculas**, es decir, que para ellos no es lo mismo el identificador "edad" que "Edad" o "EDAD". Es conveniente, por tanto, ir acostumbrándose a esta limitación. Nosotros preferiremos usar **identificadores en minúscula**, por ser lo más habitual entre los programadores de lenguaje C.

5 Programación modular

Podemos definir la **programación modular** como aquella que afronta la solución de un problema descomponiéndolo en subproblemas más simples, cada uno de los cuales se resuelve mediante un algoritmo o **módulo** más o menos independiente del resto (de ahí su nombre: "programación modular")

Las **ventajas** de la programación modular son varias:

- * *Facilita la **comprensión** del problema y su resolución escalonada*
- * *Aumenta la **claridad y legibilidad** de los programas*
- * *Permite que **varios programadores** trabajen en el mismo problema a la vez, puesto que cada uno puede trabajar en uno o varios módulos de manera bastante independiente*
- * *Reduce el **tiempo** de desarrollo, **reutilizando** módulos previamente desarrollados*
- * *Mejora la **fiabilidad** de los programas, porque es más sencillo diseñar y depurar módulos pequeños que programas enormes*
- * *Facilita el **mantenimiento** de los programas*

Resumiendo, podemos afirmar sin temor a equivocarnos que es virtualmente **imposible** escribir un programa de grandes dimensiones si no procedemos a **dividirlo en fragmentos más pequeños**, abarcables por nuestro pobre intelecto humano.

Recuérdese que la programación modular y la estructurada no son técnicas incompatibles, sino más bien complementarias. Todos los programas que desarrollemos de ahora en adelante serán, de hecho, al mismo tiempo modulares y estructurados.

Pero expliquemos más despacio que es eso de "descomponer un problema en subproblemas simples"...

5.1 Descomposición modular: ¡divide y vencerás!

La forma más habitual de diseñar algoritmos para resolver problemas de cierta envergadura se suele denominar, muy certeramente, **divide y vencerás** (en inglés, **divide and conquer** o simplemente **DAC**). Fíjate que hemos dicho "diseñar" algoritmos: estamos adentrándonos, al menos en parte, en la fase de diseño del ciclo de vida del software. Si no recuerdas lo que es, repasa el apartado correspondiente.

El método **DAC** consiste en dividir un problema complejo en **subproblemas**, y tratar cada subproblema del mismo modo, es decir, dividiéndolo a su vez en subproblemas. Así sucesivamente hasta que obtengamos problemas lo suficientemente sencillos como para escribir algoritmos que los resuelvan. Dicho de otro modo: problemas que se parezcan en complejidad a los que hemos venido resolviendo hasta ahora. Llamaremos **módulo** a cada uno de estos algoritmos que resuelven los problemas sencillos.

Una vez resueltos todos los subproblemas, es decir, escritos todos los módulos, es necesario combinar de algún modo las soluciones para generar la solución global del problema.

Esta forma de diseñar una solución se denomina **diseño descendente o top-down**. No es la única técnica de diseño que existe, pero sí la más utilizada. Resumiendo lo dicho hasta ahora, el diseño descendente debe tener dos fases:

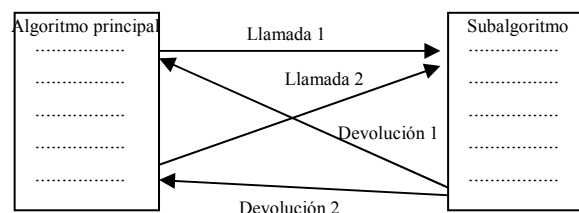
- * **La identificación de los subproblemas más simples y la construcción de algoritmos que los resuelvan (módulos)**
- * **La combinación de las soluciones de esos algoritmos para dar lugar a la solución global**

La mayoría de lenguajes de programación, incluyendo por supuesto a C, permiten aplicar técnicas de diseño descendente mediante un proceso muy simple: independizando fragmentos de código en subprogramas o módulos denominados **procedimientos y funciones**, que más adelante analizaremos en profundidad.

5.1.1 Algoritmo principal y subalgoritmos

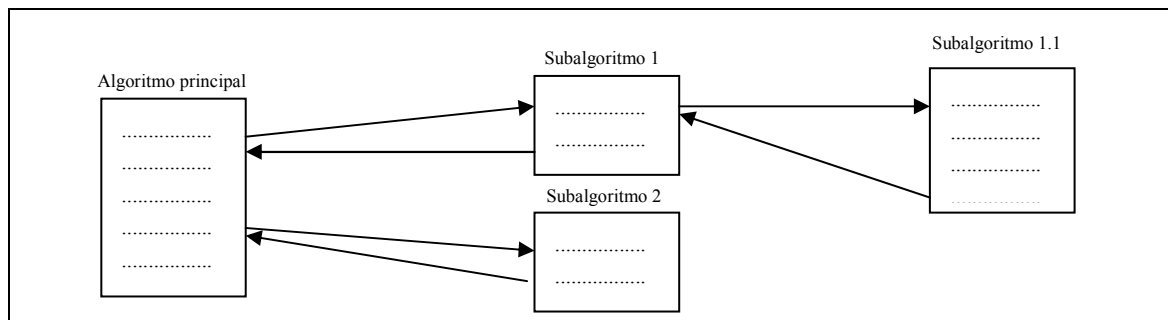
En general, el problema principal se resuelve en un algoritmo que denominaremos **algoritmo o módulo principal**, mientras que los subproblemas sencillos se resolverán en **subalgoritmos**, también llamados **módulos a secas**. Los subalgoritmos están subordinados al algoritmo principal, de manera que éste es el que decide cuándo debe ejecutarse cada subalgoritmo y con qué conjunto de datos.

El algoritmo principal realiza **llamadas o invocaciones** a los subalgoritmos, mientras que éstos **devuelven resultados** a aquél. Así, el algoritmo principal va recogiendo todos los resultados y puede generar la solución al problema global.



Cuando el algoritmo principal hace una llamada al subalgoritmo (es decir, lo **invoca**), se empiezan a ejecutar las instrucciones del subalgoritmo. Cuando éste termina, **devuelve los datos** de salida al algoritmo principal, y la ejecución continúa por la instrucción siguiente a la de invocación. También se dice que el subalgoritmo **devuelve el control** al algoritmo principal, ya que éste toma de nuevo el control del flujo de instrucciones después de habérselo cedido temporalmente al subalgoritmo.

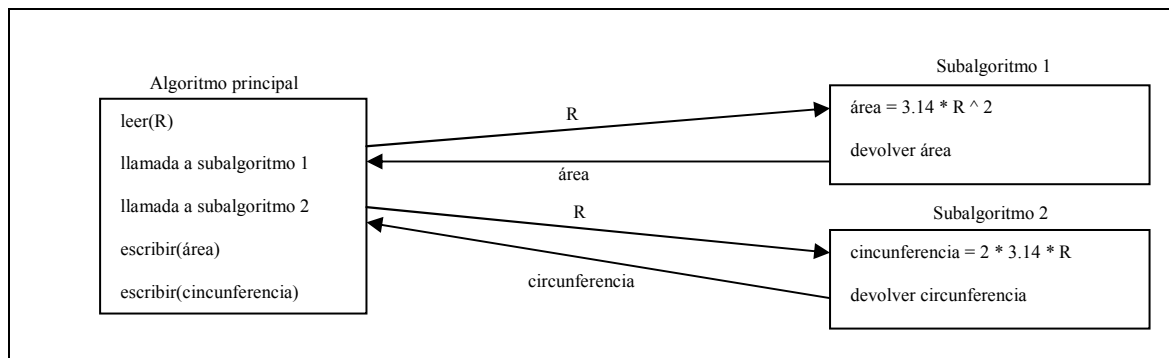
El programa principal puede invocar a cada subalgoritmo el número de veces que sea necesario. A su vez, cada subalgoritmo puede invocar a otros subalgoritmos, y éstos a otros, etc. **Cada subalgoritmo devolverá los datos y el control al algoritmo que lo invocó.**



Los subalgoritmos pueden hacer las mismas operaciones que los algoritmos, es decir: entrada de datos, proceso de datos y salida de datos. La diferencia es que los datos de entrada se los proporciona el algoritmo que lo invoca, y los datos de salida son devueltos también a él para que haga con ellos lo que considere oportuno. No obstante, un subalgoritmo también puede, si lo necesita, tomar datos de entrada desde el teclado (o desde cualquier otro dispositivo de entrada) y enviar datos de salida a la pantalla (o a cualquier otro dispositivo de salida).

Ejemplo Diseñar un algoritmo que calcule el área y la circunferencia de un círculo cuyo radio se lea por teclado. Se trata de un problema muy simple que puede resolverse sin aplicar el método divide y vencerás, pero lo utilizaremos como ilustración.

Dividiremos el problema en dos subproblemas más simples: por un lado, el cálculo del área, y, por otro, el cálculo de la circunferencia. Cada subproblema será resuelto en un subalgoritmo, que se invocará desde el algoritmo principal. La descomposición en algoritmos y subalgoritmos sería la siguiente (se indican sobre las flechas los datos que son intercambiados entre los módulos):



Lógicamente, los subalgoritmos deben tener asignado un nombre para que puedan ser invocados desde el algoritmo principal, y también existe un mecanismo concreto de invocación/devolución. Todo esto lo veremos con detalle en los siguientes epígrafes.

5.1.2 Técnicas de descomposición modular

Nivel de descomposición modular

Los problema complejos, como venimos diciendo, se descomponen sucesivamente en subproblemas más simples cuya solución combinada dé lugar a la solución general. Pero, **¿hasta dónde es necesario descomponer?** O, dicho de otro modo, ¿qué se puede considerar un “problema simple” y qué no?

La respuesta se deja al sentido común y a la experiencia del diseñador del programa. Como regla general, digamos que un módulo no debería constar de más de **30 ó 40 líneas de código**. Si obtenemos un módulo que necesita más código para resolver un problema, probablemente podamos dividirlo en dos o más subproblemas. Por supuesto, esto no es una regla matemática aplicable a todos los casos. En muchas ocasiones no estaremos seguros de qué debe incluirse y qué no debe incluirse en un módulo.

Tampoco es conveniente que los módulos sean **excesivamente sencillos**. Programar módulos de 2 ó 3 líneas daría lugar a una descomposición excesiva del problema, aunque habrá ocasiones en las que sea útil emplear módulos de este tamaño.

Diagrama de estructura modular

La **estructura modular**, es decir, el conjunto de módulos de un programa y la forma en que se invocan unos a otros, se puede representar gráficamente mediante un **diagrama de estructura modular**¹⁰. Esto es particularmente útil si el programa es complejo y consta de muchos módulos con relaciones complicadas entre sí.

En el diagrama se representan los módulos mediante **cajas**, en cuyo interior figura el nombre del módulo, unidos por líneas, que representan las interconexiones entre ellos. En cada línea se pueden escribir los parámetros de invocación y los datos devueltos por el módulo invocado.

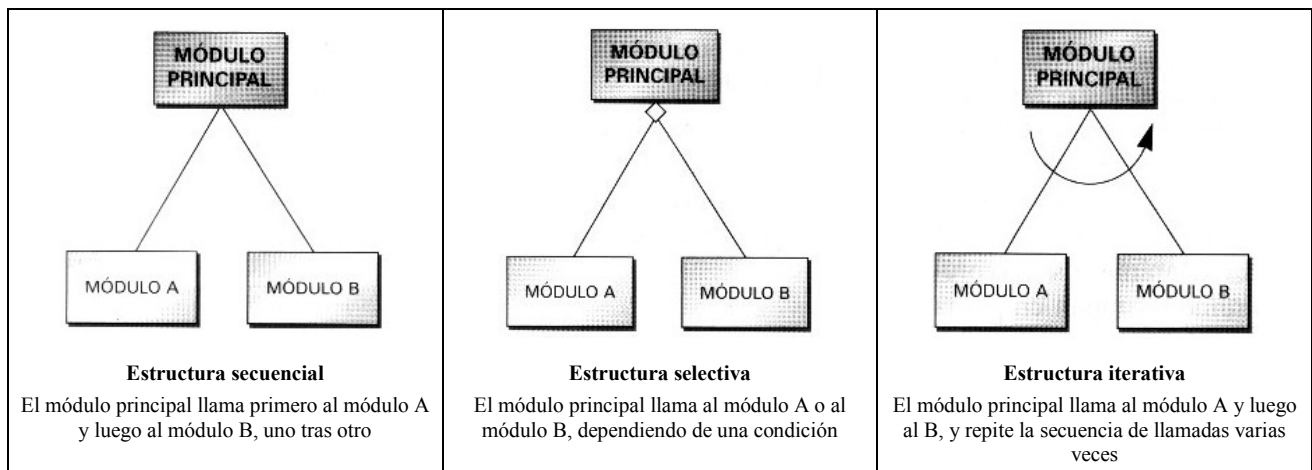
El diagrama de estructura siempre tiene forma de **árbol invertido**. En la raíz figura el módulo principal, y de él “cuelgan” el resto de módulos en uno o varios niveles.

En el diagrama también se puede representar el **tipo de relación entre los módulos**. Las relaciones posibles se corresponden exactamente con los tres tipos de estructuras básicas de la programación estructurada:

- ✱ **Estructura secuencial:** cuando un módulo llama a otro, después a otro, después a otro, etc.
- ✱ **Estructura selectiva:** cuando un módulo llama a uno o a otro dependiendo de alguna condición
- ✱ **Estructura iterativa:** cuando un módulo llama a otro (o a otros) en repetidas ocasiones

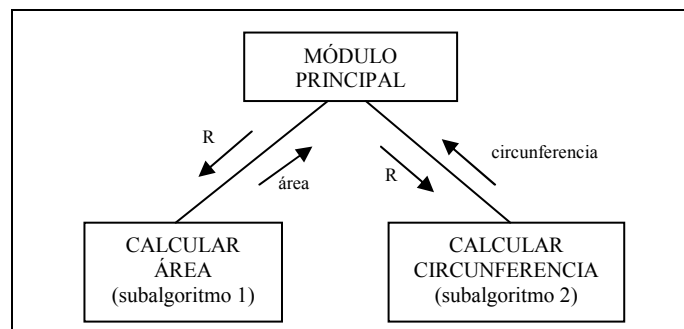
Las tres estructuras de llamadas entre módulos se representan con tres símbolos diferentes:

¹⁰ Estos diagramas se usan profusamente en la etapa de diseño del ciclo de vida. Nosotros, que no pretendemos convertirnos en diseñadores, los emplearemos sólo cuando nos enfrentemos con diseños modulares complejos. También se denominan *Diagramas de Cuadros de Constantine*, debido al clásico libro sobre diseño estructurado: CONSTANTINE, L.; YOURDON, E., *Structured design: fundamentals of a discipline of computer programs and system design*, Prentice-Hall, 1979.



Ejemplo Diagrama de estructura del algoritmo que calcula el área y la circunferencia de un círculo, que vimos como ejemplo en la página 35. La descomposición modular que hicimos en aquel ejemplo consistía en un algoritmo principal que llamaba a dos subalgoritmos: uno para calcular el área y otro para calcular la circunferencia.

Los dos subalgoritmos (o módulos) son llamados en secuencia, es decir, uno tras otro, por lo que lo representaremos con la estructura secuencial. El módulo principal pasará a los dos subalgoritmos el radio (R) del círculo, y cada subalgoritmo devolverá al módulo principal el resultado de sus cálculos. Observa que ese trasiego de información también puede representarse en el diagrama de estructura:



Escritura del programa

Una vez diseñada la estructura modular, llega el momento de **escribir los algoritmos y subalgoritmos** mediante pseudocódigo, diagramas de flujo o cualquier otra herramienta. Lo más conveniente es comenzar por los módulos (subalgoritmos) de nivel inferior e ir ascendiendo por cada rama del diagrama de estructura.

Lo dicho hasta ahora respecto de algoritmos y subalgoritmos se puede traducir en **programas y subprogramas** cuando pasemos del pseudocódigo a lenguajes de programación concretos, como C. Los subprogramas, en general, se pueden dividir en dos tipos, muy similares pero con alguna sutil diferencia: las **funciones** y los **procedimientos**, que estudiaremos en los siguientes epígrafes del tema.

5.2 Funciones

Las **funciones** son subalgoritmos (o módulos) que resuelven un problema sencillo y **devuelven un resultado** al algoritmo que las invoca.

Las funciones pueden tener **argumentos**, aunque no es obligatorio. Los argumentos son **los datos que se proporcionan** a la función en la invocación, y que la función utilizará para sus cálculos.

Además, las funciones tienen, obligatoriamente, que **devolver un resultado**. Este resultado suele almacenarse en una variable para usarlo posteriormente.

Ejemplo Paso de parámetros a funciones de biblioteca. Cuando utilizamos las funciones matemáticas de biblioteca, siempre escribimos algún dato entre paréntesis para que la función realice sus cálculos con ese dato. Pues bien, ese dato es el **argumento o parámetro de entrada**:

```
A = raíz(X)
B = redondeo(7.8)
N = aleatorio(100)
```

En estas tres instrucciones de asignación, se invoca a las funciones **raíz()**, **redondeo()** y **aleatorio()**, pasándoles

los argumentos X y 7.8. Recuerda que estas son funciones que los lenguajes de programación incorporan por defecto, junto con muchas otras que iremos descubriendo con el uso.

Ambas funciones devuelven un resultado; el resultado de la función **raiz()** se almacena en la variable A, el de **redondeo()** en la variable B y el de la función **aleatorio()** en la variable N.

5.2.1 Declaración de funciones

No sólo de funciones de biblioteca vive el programador. Como es lógico, también podemos crear nuestras propias funciones para invocarlas cuando nos sea necesario.

Recuerda que una función no es más que un **módulo**, es decir, un subalgoritmo que depende, directamente o a través de otro subalgoritmo, del algoritmo principal. Por tanto, su estructura debe ser similar a la de los algoritmos que hemos manejado hasta ahora.

La **sintaxis** en pseudocódigo de una función es:

```

tipo_resultado función nombre_función(lista_de_argumentos)
constantes
    lista_de_constantes
variables
    lista_de_variables
inicio
    acciones
    devolver (expresión)
fin
  
```

Observa que es exactamente igual que los algoritmos que conoces, excepto por la primera línea, que ya no contiene la palabra "algoritmo" e incluye algunos elementos nuevos:

- * El **tipo_resultado** es el tipo de datos del resultado que devuelve la función
- * El **nombre_función** es el identificador de la función
- * La **lista_de_argumentos** es una lista con los parámetros que se le pasan a la función

También aparece una nueva sentencia, **devolver** (expresión), justo al final de la función. La expresión se evalúa y su resultado es devuelto al algoritmo que invocó a la función. El tipo de la expresión debe coincidir con el de **tipo_resultado**.

De todos estos elementos nuevos, el más complejo con diferencia es la **lista de argumentos**, ya que pueden existir argumentos de entrada, de salida y de entrada/salida. El problema de los argumentos lo trataremos en profundidad en el epígrafe 5.4 (pág. 40), cuando ya nos hayamos acostumbrado al uso de funciones. Por ahora, diremos que es una lista de esta forma:

```

parámetro_1 es tipo_de_datos_1, parámetro_2 es tipo_de_datos_2, etc.
  
```

Ejemplo Declaración de una función que calcule el área de un círculo. El radio se pasa como argumento de tipo real.

```

real función área_círculo (radio es real)
variables
    área es real
inicio
    área = 3.14 * radio ^ 2
    devolver (área)
fin
  
```

Fíjate en que la función no es más que un algoritmo normal y corriente, salvo por dos detalles:

- * La primera línea. En ella aparece más información: el **tipo de valor devuelto** por la función (**real**, puesto que calcula el área del círculo), el **nombre de la función** (**área_círculo**) y la **lista de argumentos**. En esta función sólo hay un argumento, llamado **radio**. Es de tipo real.
- * La penúltima línea (antes de fin). Contiene el valor que la función **devuelve**. Debe ser una expresión del mismo tipo que se indicó en la primera línea (en este ejemplo, real).

5.2.2 Invocación de funciones

Para que las instrucciones escritas en una función sean ejecutadas es necesario que la función se llame o invoque desde otro algoritmo.

La **invocación** consiste en una mención al **nombre de la función** seguida, entre paréntesis, de los valores que se desan asignar a los **argumentos**. Deben aparecer tantos valores como argumentos tenga la función, y además coincidir en tipo. Estos valores se asignarán a los argumentos y se podrán utilizar, dentro de la función, como si de variables se tratase.

Como las funciones **devuelven valores**, es habitual que la invocación aparezca junto con una asignación a variable para guardar el resultado y utilizarlo más adelante.

Ejemplo 1 Escribir un algoritmo que calcule el área de un círculo mediante el empleo de la función vista en el ejemplo anterior. La función `área_círculo()` que acabamos de ver puede ser invocada desde otro módulo, igual que invocamos las funciones de biblioteca como `raiz()` o `redondeo()`

```

algoritmo círculo
variables
    A, B, R son reales
inicio
    leer(R)
    A = área_círculo(R)
    escribir(A)
fin

```

Este fragmento de código **invocará la función `área_círculo()`** con el argumento **R**. La función se ejecutará con el valor de **R** asociado al identificador **radio**, exactamente igual que si éste fuera una variable y hubiéramos hecho la asignación `radio = R`. Una vez calculado el resultado, la función lo devuelve al módulo que la invocó, y por tanto el valor del área se asigna a la variable **A**. Por último, el valor de **A** se escribe en la pantalla.

Ejemplo 2 Escribir un algoritmo que calcule el cuadrado y el cubo de un valor *X* introducido por teclado, utilizando funciones. Aunque el algoritmo es simple y podría resolverse sin modularidad, forzaremos la situación construyendo dos funciones, `cuadrado()` y `cubo()`:

```

algoritmo cuadrado_cubo
variables
    N, A, B son reales
inicio
    leer(N)
    A = cuadrado(N)
    B = cubo(N)
    escribir("El cuadrado es ", A)
    escribir("El cubo es ", B)
fin

real función cuadrado (número es real)           // Devuelve el cuadrado de un número
inicio
    devolver (número ^ 2)
fin

real función cubo (número es real)               // Devuelve el cubo de un número
inicio
    devolver (número ^ 3)
fin

```

Fíjate en que hemos escrito las funciones *después* del algoritmo principal. Esto puede variar dependiendo del lenguaje utilizado.

5.3 Procedimientos

Las funciones son muy útiles como herramientas de programación, pero tienen una seria limitación: sólo pueden devolver **un** resultado al algoritmo que las invoca. Y en muchas ocasiones es necesario devolver **más de un** resultado.

Para eso existen los **procedimientos**, también llamados **subrutinas**, que son, en esencia, iguales a las funciones, es decir:

- * *son algoritmos independientes que resuelven algún problema sencillo*
- * *pueden recibir datos de entrada del algoritmo que los invoca*
- * *el algoritmo que los invoca queda momentáneamente en suspenso mientras se ejecuta el procedimiento y, cuando éste termina, el algoritmo principal continúa ejecutándose*

Pero existe una **diferencia** fundamental entre las funciones y los procedimientos:

- * *los procedimientos pueden devolver 0, 1 o más resultados, mientras que las funciones siempre devuelven uno.*

Los procedimientos son, por lo tanto, módulos más generales que las funciones. La **declaración** de un procedimiento es similar a la de una función, pero sustituyendo la palabra **función** por **procedimiento** y sin indicar el tipo de datos del resultado; tampoco tienen sentencia **devolver** al final del código:

```

procedimiento nombre_procedimiento(lista_de_argumentos)
constantes
    lista_de_constantes
variables
    lista_de_variables
inicio
    acciones
fin

```

Pero, si no tienen sentencia **devolver**, ¿cómo devuelve un procedimiento los resultados al algoritmo que lo invoca? La única posibilidad es **utilizar los argumentos** como puerta de dos direcciones, es decir, que no solo sirvan para que el algoritmo comunique datos al subalgoritmo, sino también para comunicar datos desde el subalgoritmo hacia el algoritmo.

Para ello necesitamos saber más cosas sobre el paso de parámetros, que es lo que estudiamos en el siguiente epígrafe:

5.4 Paso de parámetros

El paso de parámetros, o comunicación de datos del algoritmo invocante al subalgoritmo invocado, puede hacerse mediante, al menos, dos métodos:

- ★ *Paso de parámetros por valor, que es la forma más sencilla pero no permite al subalgoritmo devolver resultados en los parámetros.*
- ★ *Paso de parámetros por referencia, que es más complejo pero permite a los subalgoritmos devolver resultados en los parámetros.*

Veamos cada método detenidamente.

5.4.1 Paso de parámetros por valor

Los subalgoritmos/subprogramas, como hemos visto, pueden tener una serie de parámetros en su declaración. Estos parámetros se denominan **parámetros formales**.

Ejemplo Una función que calcula la potencia de un número elevado a otro

```
real función potencia(base es real, exponente es real)
inicio
    devolver (base ^ exponente)
fin
```

En esta declaración de función, `base` y `exponente` son parámetros formales.

Cuando el subalgoritmo es invocado, se le pasan entre paréntesis los valores de los parámetros. A éstos se les denomina **parámetros actuales**; por ejemplo:

```
A = 5
B = 3
C = potencia(A,B)
```

En esta invocación de la función `potencia()`, los parámetros actuales son A y B, es decir, 5 y 3.

Al invocar un subalgoritmo, **los parámetros actuales son asignados a los parámetros formales** en el mismo orden en el que fueron escritos. Dentro del subalgoritmo, los parámetros se pueden utilizar **como si fueran variables**. Así, en el ejemplo anterior, dentro de la función `potencia()`, el parámetro `base` puede usarse como una variable a la que se hubiera asignado el valor 5, mientras que `exponente` es como una variable a la que se hubiera asignado el valor 3.

Cuando el subalgoritmo termina de ejecutarse, sus parámetros formales `base` y `exponente` dejan de existir y se devuelve el resultado (en nuestro ejemplo, 5^3), que se asigna a la variable C.

5.4.2 Paso de parámetros por referencia

En el paso de parámetros por referencia **se produce una ligadura entre el parámetro actual y el parámetro formal**, de modo que si el parámetro formal se modifica dentro del subalgoritmo, el parámetro actual, propio del algoritmo principal, también será modificado.

Los argumentos pasan sus parámetros por valor excepto cuando indiquemos que el paso es por referencia colocando el símbolo ***** (asterisco) delante del nombre del argumento.

Ejemplo Escribiremos el mismo subalgoritmo de antes, pero utilizando un procedimiento (que, en principio, no devuelve resultados) en lugar de una función.

```
procedimiento potencia(base es real, exponente es real, *resultado es real)
inicio
    resultado = base ^ exponente
fin
```

Observa el símbolo ***** delante del nombre del argumento `resultado`: esa es la señal de que el paso de parámetros será por referencia para ese argumento. Si no aparece el símbolo *****, el paso será por valor, como es el caso de los argumentos `base` y `exponente`.

La invocación del subalgoritmo se hace del mismo modo que hasta ahora, pero delante del parámetro que se pasa por referencia debe colocarse el símbolo **&**:

```
A = 5
B = 3
```



```
C = 0
potencia(A, B, &C)
```

En este caso, pasamos tres parámetros actuales, ya que el subalgoritmo tiene tres parámetros formales. El tercero de ellos, C, se **pasa por referencia** (para señalar esta circunstancia, se antepone el símbolo &), y por lo tanto **queda ligado al parámetro formal resultado**.

El parámetro formal es modificado en la instrucción `resultado = base ^ exponente`, y como está ligado con el parámetro actual C, el valor de **la variable C también se modifica**. Por lo tanto, C toma el valor 5^3 .

Cuando el subalgoritmo termina de ejecutarse, dejan de existir todos sus parámetros formales (`base`, `exponente` y `resultado`), pero la ligadura de `resultado` con la variable C hace que esta variable conserve el valor 5^3 incluso cuando el parámetro `resultado` ya no exista.

5.4.3 Diferencias entre los métodos de paso de parámetros

La utilidad del método de paso de parámetros **por referencia** es evidente: **un subalgoritmo puede devolver tantos resultados como argumentos tenga**, y no tiene que limitarse a un único resultado, como en el caso de las funciones.

El paso de parámetros por referencia suele, por lo tanto, **usarse en procedimientos que tienen que devolver muchos resultados** al algoritmo que los invoca. Cuando el resultado es sólo uno, lo mejor es emplear una función. Esto no quiere decir que las funciones no puedan tener argumentos pasados por referencia: al contrario, a veces es muy útil.

Expresado de otro modo:

- * *el paso por valor es unidireccional, es decir, sólo permite transmitir datos del algoritmo al subalgoritmo a través de los argumentos.*
- * *el paso por referencia es bidireccional, es decir, permite transmitir datos del algoritmo al subalgoritmo, pero también permite al subalgoritmo transmitir resultados al algoritmo.*

5.5 El problema del ámbito

5.5.1 Variables locales

Se llama **ámbito** de una variable a la parte de un programa donde dicha variable puede utilizarse.

En principio, todas las variables declaradas en un algoritmo son locales a ese algoritmo, es decir, no existen fuera del algoritmo, y, por tanto, no pueden utilizarse más allá de las fronteras marcadas por *inicio* y *fin*. **El ámbito de una variable es local al algoritmo donde se declara.**

Cuando el algoritmo comienza, las variables se crean, reservándose un espacio en la memoria RAM del ordenador para almacenar su valor. Cuando el algoritmo termina, todas sus variables se destruyen, liberándose el espacio en la memoria RAM. Todos los resultados que un algoritmo obtenga durante su ejecución, por lo tanto, se perderán al finalizar, salvo que sean devueltos al algoritmo que lo invocó o sean dirigidos a algún dispositivo de salida (como la pantalla). Esta forma de funcionar ayuda a que los algoritmos sean módulos independientes entre sí, que únicamente se comunican los resultados de sus procesos unos a otros.

Ejemplo Calcular el cuadrado de un valor X introducido por teclado utilizando diseño modular.

```
algoritmo cuadrado
variables
    N, result son reales
inicio
    leer(N)
    calcular_cuadrado()
    escribir("El cuadrado es ", result)
fin

procedimiento calcular_cuadrado ()           // Calcula el cuadrado de un número
inicio
    result = N ^ 2
fin
```

En este algoritmo hay un **grave error**, ya que se han intentado utilizar las variables `result` y `N`, que son locales al algoritmo principal, en el subalgoritmo `cuadrado()`, desde donde no son accesibles.

Es importante señalar que en algunos lenguajes de programación, y bajo determinadas circunstancias, cuando un algoritmo invoca a un subalgoritmo, puede que todas las variables locales del algoritmo estén disponibles en el subalgoritmo. Así, el ejemplo anterior podría llegar a ser correcto. Esto no ocurre en C, debido a que no se pueden anidar funciones dentro de funciones, pero debe ser tenido en cuenta por el alumno/a si en algún momento debe programar en otro lenguaje. El problema que surge en esas situaciones es similar al de las variables globales que tratamos a continuación.

5.5.2 Variables globales

En ocasiones es conveniente utilizar **variables cuyo ámbito exceda el del algoritmo donde se definen** y puedan utilizarse en varios algoritmos y subalgoritmos. Las variables globales implican una serie de riesgos, como veremos más adelante, por lo que no deben utilizarse a menos que sea estrictamente necesario. A pesar de los riesgos, la mayoría de los lenguajes de programación disponen de algún mecanismo para manejar variables globales.

Aunque ese mecanismo varía mucho de un lenguaje a otro, diremos como regla general que las variables globales deben declararse en el algoritmo principal, anteponiendo el identificador **global** al nombre de la variable, siendo entonces accesibles a todos los algoritmos y subalgoritmos que conformen el programa.

Ejemplo *Calcular el cuadrado de un valor X introducido por teclado utilizando diseño modular.*

```

algoritmo cuadrado
variables
    global N es real
    global result es reales
inicio
    leer(N)
    calcular_cuadrado()
    escribir("El cuadrado es ", result)
fin

procedimiento calcular_cuadrado ()           // Calcula el cuadrado de un número
inicio
    result = N ^ 2
fin

```

El error que existía antes ya no ocurre, porque ahora las variables `result` y `N` han sido declaradas como globales en el algoritmo principal, y por lo tanto pueden utilizarse en cualquier subalgoritmo, como `cuadrado()`.

Pudiera ocurrir que **una variable global tenga el mismo nombre que una variable local**. En ese caso, el comportamiento depende del lenguaje de programación (los hay que ni siquiera lo permiten), pero lo habitual es que la variable local sustituya a la global, haciendo que ésta última sea inaccesible desde el interior del subalgoritmo. Al terminar la ejecución del subalgoritmo y destruirse la variable local, volverá a estar accesible la variable global que, además, habrá conservado su valor, pues no ha podido ser modificada desde el subalgoritmo.

De todas formas, y puestos a evitar la utilización de variables globales (a menos que no quede otro remedio), con más razón aún evitaremos usar variables locales que tengan el mismo nombre que las globales.

5.5.3 Los efectos laterales

Al utilizar variables globales, muchas de las ventajas de la programación modular (que puedes repasar en la página 34) desaparecen.

Efectivamente, la filosofía de la programación modular consiste en diseñar **soluciones sencillas e independientes** (llamadas **módulos**) para problemas sencillos, haciendo que los módulos se comuniquen entre sí sólo mediante el paso de parámetros y la devolución de resultados.

Cuando empleamos variables globales como en el ejemplo anterior, se crea una comunicación alternativa entre módulos a través de la variable global. Ahora un módulo puede influir por completo en otro modificando el valor de una variable global. Los módulos dejan de ser "*compartimentos estanco*" y pasan a tener **fuertes dependencias mutuas** que es necesario controlar. Cuando el programa es complejo y consta de muchos módulos, ese control de las dependencias es cada vez más difícil de hacer.

Cualquier comunicación de datos entre un algoritmo y un subalgoritmo al margen de los parámetros y la devolución de resultados se denomina **efecto lateral**. Los efectos laterales, como el ilustrado en el ejemplo anterior, son peligrosísimos y fuente habitual de malfuncionamiento de los programas. Por esa razón, debemos tomar como norma:

- * *Primero, evitar la utilización de variables globales*
- * *Segundo, si no quedara más remedio que emplear variables globales, no hacer uso de ellas en el interior de los procedimientos y las funciones, siendo preferible pasar el valor de la variable global como un parámetro más al subalgoritmo*

5.6 La reutilización de módulos

El diseño modular tiene, entre otras ventajas, la posibilidad de **reutilizar módulos previamente escritos**. Es habitual que, una vez resuelto un problema sencillo mediante una función o un procedimiento, ese mismo problema, o uno muy parecido, se nos presente más adelante, durante la realización de otro programa. Entonces nos bastará con volver a utilizar esa función o procedimiento, sin necesidad de volver a escribirlo.

Es por esto, entre otras razones, que **los módulos deben ser independientes** entre sí, comunicándose con otros módulos únicamente mediante los datos de entrada (paso de parámetros por valor) y los de salida (devolución de resultados – en

las funciones – y paso de parámetros por referencia). Los módulos que escribamos de este modo nos servirán probablemente para otros programas, pero no así los módulos que padezcan **efectos laterales**, pues sus relaciones con el resto del programa del que eran originarios serán diferentes y difíciles de precisar.

Es habitual **agrupar varios algoritmos** relacionados (por ejemplo: varios algoritmos que realicen diferentes operaciones matemáticas) en un mismo archivo, formando lo que se denomina una **biblioteca** de funciones. Cada lenguaje trata las librerías de manera distinta, de modo que volveremos sobre este asunto al estudiar el lenguaje C.

Por último, señalemos que, para reutilizar con éxito el código, es importante que esté **bien documentado**. En concreto, en cada algoritmo deberíamos documentar claramente:

- ★ *la **función** del algoritmo, es decir, explicar qué hace*
- ★ *los parámetros de **entrada***
- ★ *los datos de **salida**, es decir, el resultado que devuelve o la forma de utilizar los parámetros por referencia*

Ejemplo Documentaremos la función `potencia()`, que hemos utilizado como ejemplo en otras partes de esta unidad didáctica. Es un caso exagerado, pues la función es muy sencilla y se entiende sin necesidad de tantos comentarios, pero ejemplifica cómo se puede hacer la documentación de una función.

```
{ Función: potencia() --> Calcula una potencia de números enteros
  Entrada: base      --> Base de la potencia
           exponente --> Exponente de la potencia
  Salida:  base elevado a exponente }
```

```
real función potencia(base es real, exponente es real)
inicio
  devolver (base ^ exponente)
fin
```

6 Actividades

Actividades introductorias

1. Investiga y responde a las siguientes cuestiones, indicando la fuente dónde encuentres la información:
 - a) ¿Por qué los ordenadores digitales sólo utilizan ceros y unos, es decir, códigos binarios, en lugar de manejar internamente códigos decimales, como hacemos los humanos?
 - b) ¿Por qué los humanos estamos habituados a un sistema de numeración basado en 10 símbolos y no a cualquier otro, por ejemplo, uno con 8 símbolos, o con 5, o con 12?
 - c) Ya debes conocer la diferencia entre hardware y software. ¿Cuál es más importante de los dos? ¿Cuál piensas suele evolucionar antes? ¿Cuál es más difícil de diseñar y fabricar?
 - d) ¿Qué es el firmware?
2. Tenemos un viejo ordenador con una capacidad de almacenamiento en la memoria principal de 2 MB. Suponiendo que un nombre ocupe 30 caracteres y un apellido ocupe 25, ¿cuántos nombres y apellidos puede almacenar este ordenador?
3. Convierte las siguientes cantidades de información:

a) 30 GB a MB	c) 2 MB a bits
b) 128 KB a bits	d) 64512 KB a MB
4. Convierte los siguientes números al sistema de numeración indicado:

a) 100101_2 a decimal	c) 1111111_2 a decimal
b) 254_{10} a binario	d) 191_{10} a binario
5. Convierte los siguientes números entre los sistemas hexadecimal y binario, utilizando la tabla de conversión que hemos visto en este tema. Si puedes hacerlo sin mirar la tabla, mucho mejor.

a) 10011101_2 a hexadecimal	c) $38C_{16}$ a binario
b) 110100111011001101_2 a hexadecimal	d) $FDCA_{16}$ a binario
6. ¿A qué tipo pertenecen los siguientes datos?

a) 0	d) -90.0	g) falso
b) -90.234	e) 'A'	h) "falso"
c) -90	f) "Almería"	i) "-90"
7. Indica cuál es la mantisa y cuál el exponente de los siguientes números reales:
 - a) 0.75
 - b) -12.5
 - c) 0.0000000023
 - d) 5873000000000000000000.0
8. ¿Cuáles de los siguientes identificadores de variable no son correctos y por qué?

a) XYZ	b) _xyz	c) 'valor'
--------	---------	------------

- | | | |
|--------------|---------------|---------------|
| d) 56vértice | f) año | h) año_actual |
| e) indice28 | g) año&actual | i) ZZZZ |

9. Calcula el valor de estas expresiones, sabiendo que $A = 2$, $B = 5$ y $C = 8$:

- | | |
|--|--|
| a) $4/2 * 3/6 + 6/2/1/5^2/4 * 2$ | h) $7 * 10 - 50 \% 3 * 4 + 9$ |
| b) $3 * A - 4 * B / A^2$ | i) $(7 * (10 - 5) \% 3) * 4 + 9$ |
| c) $B * A - B^2/4 * C$ | j) $7 \% 5 \% 3$ |
| d) $((B + C) / 2 * A + 10) * 3 * B) - 6$ | k) $\text{raiz}(B^*B)$ |
| e) $7 \text{ div } 2$ | l) $\text{raiz}(B^*B)$ |
| f) $7 \% 2$ | m) $\text{trunc}(81.5) + \text{redondeo}(81.5)$ |
| g) $0 \% 5$ | n) $\text{trunc}(\text{raiz}(C)) > \text{abs}(-(A^2))$ |

10. Convierte estas expresiones algebraicas a su notación informática:

a) $\frac{M}{N} + P$

c) $\frac{x+y}{a-b}$

$$\text{e) } \frac{m + \frac{n}{p}}{q - \frac{r}{s}}$$

b) $M + \frac{N}{P-Q} + P$

d) $2 \cdot \frac{\sin(x) + \cos(x)}{\tan(x)}$

$$\text{f) } \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

11. Se tienen las siguientes variables: A y B de tipo entero; C y D de tipo real; E y F de tipo carácter; y G de tipo lógico. Señala cuáles de las siguientes asignaciones no son válidas y por qué:

- a) $A = 20$ d) $B = 5500$ g) $E = 'F'$ j) $G = \text{verdadero}$
b) $B = 12$ e) $C = 0$ h) $E = F$ k) $G = \text{'falso'}$
c) $F = '0'$ f) $D = C$ i) $A = 12.56$ l) $F = G$

12. ¿Cuáles son los valores de las variables A, B y C después de la ejecución de estas expresiones?

```
A, B, C son enteros
A = 3
B = 4
C = A + 2 * B
C = C + B
B = C - A
A = B * C
```

13. ¿Cuál es el valor de la variable X después de la ejecución de estas expresiones?

```
X es real
X = 2.0
X = (X + X)^2
X = raiz(X + raiz(X) + 5)
```

14. ¿Cuál es el valor de las variables X, Y y Z después de estas instrucciones?

```
X, Y son enteros
Z es lógico
X = 5
Y = X - 2
X = Y^2 + 1
Z = (X > (Y + 5))
```

15. ¿Cuál es el valor de las variables A y B después de la ejecución de estas expresiones?

```
A, B son enteros
A = 10
B = 5
A = B
B = A
```

16. Se tienen dos variables, A y B. Escribe las asignaciones necesarias para intercambiar sus valores, sean cuales sean.

17. Se tienen tres variables, A, B y C. Escribe las asignaciones necesarias para intercambiar sus valores, sean cuales sean, de manera que:

- B tome el valor de A
- C tome el valor de B
- A tome el valor de C

18. Escribe, en español (ya que aún no conocemos el pseudocódigo) un algoritmo para hacer una tortilla francesa.

19. Escribe, en español, un algoritmo que determine, dados dos números A y B, cuál es el mayor de los dos.

20. Escribe en español un algoritmo que, dados tres números A, B y C, averigüe cuál es el mayor de los tres.

Actividades sobre programación estructurada

Nota: En los ejercicios siguientes hay que diseñar un algoritmo que resuelva el problema propuesto. Todos se deben describir con pseudocódigo y con diagramas de flujo, salvo que en el ejercicio se indique otra cosa.

21. Calcular el área y la circunferencia de un círculo cuyo radio será proporcionado a través del teclado. Recuerda que $\text{área} = \pi r^2$ y $\text{circunferencia} = 2 \pi r$
22. Determinar si un número leído del teclado es positivo o negativo.
23. Calcular la raíz cuadrada de un número introducido por teclado. Hay que tener la precaución de comprobar que el número sea positivo.
24. Leídos dos números por teclado, A y B, calcular la resta del mayor menos el menor. Por ejemplo, si $A = 8$ y $B = 3$, el resultado debe ser $A - B$, es decir, 5. Pero si $A = 4$ y $B = 7$, el resultado debe ser $B - A$, es decir, 3.
25. Determinar si un año es bisiesto o no (los años bisiestos son múltiplos de 4; utilícese el operador módulo)
26. Averiguar si un número real introducido por teclado tiene o no parte fraccionaria (utilícese la función `trunc()` que aparece descrita en los apuntes)
27. Leer un número real y un tipo de moneda, que puede ser "euro" o "peseta". Convertir la cantidad al tipo de moneda indicado, suponiendo que está expresada en la otra. Por ejemplo, si la cantidad es 15 y la moneda es "peseta", se supondrá que se trata de 15 € y que hay que convertirlos a pesetas y, por lo tanto, el resultado debe ser 2495.
28. Leer tres números por teclado, X, Y y Z, y decidir si están ordenados de menor a mayor.
29. Como el anterior, pero para averiguar si los números son consecutivos.
30. Determinar el número de cifras de un número entero. El algoritmo debe funcionar para números de hasta 5 cifras, considerando los negativos. Por ejemplo, si se introduce el número 5342, la respuesta del programa debe ser 4. Si se introduce -250, la respuesta debe ser 3.
31. Calcular las dos soluciones de una ecuación de segundo grado, del tipo $ax^2 + bx + c = 0$. Los coeficientes a, b y c deberá introducirlos el usuario a través del teclado.
32. Dados tres números enteros, A, B, C, determinar cuál es el mayor, cuál el menor y cuál el mediano. Sólo pseudocódigo.
33. Escribir todos los números pares entre 1 y 1000
34. Escribir todos los números impares entre dos números A y B introducidos por teclado. Antes habrá que comprobar cuál de los dos números A y B es mayor.
35. Calcular la suma de todos los números pares entre 1 y 1000. Es decir, $2 + 4 + 6 + \dots + 998 + 1000$.
36. Mostrar el mensaje "¿Desea terminar? (S/N)" y leer la respuesta del usuario. Si es "S", el programa terminará. Si es "N", volverá a formular la pregunta.
37. Calcular el valor medio de una serie de valores enteros positivos introducidos por teclado. Para terminar de introducir valores, el usuario debe teclear un número negativo.
38. El usuario de este programa será un profesor, que introducirá las notas de sus 30 alumnos de una en una. El algoritmo debe decirle cuántos suspensos y cuántos aprobados hay.
39. Calcular el valor máximo de una serie de 10 números introducidos por teclado. Sólo en pseudocódigo.
40. Generalizar el ejercicio anterior para que también se averigüe el valor mínimo y el medio. Sólo en pseudocódigo.
41. Calcular el factorial de un número entero N. Recuerda que el factorial de un número es el producto de ese número por todos los enteros menores que él. Por ejemplo, el factorial de 5 (simbolizado 5!) se calcula como: $5! = 5 \times 4 \times 3 \times 2 \times 1$. Sólo en pseudocódigo.
42. Determinar si un número N introducido por teclado es o no primo. Recuerda que un número primo es aquél que sólo es divisible por sí mismo y por la unidad. Sólo en pseudocódigo.
43. Generalizar el algoritmo anterior para averiguar todos los números primos que existen entre 2 y 1000. Sólo en pseudocódigo.
44. Introducida una hora por teclado (horas, minutos y segundos), se pretende sumar un segundo a ese tiempo e imprimir en la pantalla la hora que resulta (también en forma de horas, minutos y segundos). Sólo pseudocódigo.
45. Generar combinaciones al azar para la lotería primitiva (6 números entre 1 y 49). Debes utilizar la función `aleatorio(x)` que vimos en los apuntes. Por ahora, no te preocupes porque los números puedan repetirse.
46. Generar combinaciones al azar para la quiniela (14 valores dentro del conjunto 1, X o 2)
47. La calculadora. Diseñar un algoritmo que lea dos números, A y B, y un operador (mediante una variable de tipo carácter), y calcule el resultado de operar A y B con esa operación. Por ejemplo, si $A = 5$ y $B = 2$, y operación = "+", el resultado debe ser 7. El algoritmo debe seguir pidiendo números y operaciones indefinidamente, hasta que el usuario decida terminar (utilizar un valor centinela para ello)

48. Juego del número secreto. El ordenador elegirá un número al azar entre 1 y 100. El usuario irá introduciendo números por teclado, y el ordenador le irá dando pistas: "mi número es mayor" o "mi número es menor", hasta que el usuario acierte. Entonces el ordenador le felicitará y le comunicará el número de intentos que necesitó para acertar el número secreto. Sólo en pseudocódigo.
49. El siguiente algoritmo intenta calcular la tabla de multiplicar de un número N introducido por teclado, pero tiene dos errores que debes corregir.

```

algoritmo tabla_multiplicar
variables
  N es entero
inicio
  leer(N)
  cont = 1
  mientras (cont <= 10) hacer
    inicio
      escribir(N*cont)
    fin
  fin

```

50. El siguiente algoritmo lee dos números por teclado, A y B, y determina si B es divisor de A, pero cuando B = 0 se produce un error de intento de división entre 0 en la instrucción A mod B. Corrígelo para que no ocurra:

```

algoritmo divisor
variables
  A,B son enteros
inicio
  leer(A,B)
  si (A mod B == 0) entonces
    escribir('B es divisor de A')
  si_no
    escribir('A es divisor de B')
  fin

```

51. El siguiente algoritmo sirve para contar hacia atrás desde un número N hasta 1, pero tiene algunos fallos. Corrígelos:

```

algoritmo contar_hacia_atrás
variables
  N, cont son enteros
inicio
  leer(N)
  repetir
    inicio
      cont = cont - 1
      escribir(cont)
    fin
  mientras que (cont <= 1)
  fin

```

52. Averigua qué hace este algoritmo:

```

algoritmo misterioso
variables
  A, límite, cont son enteros
inicio
  leer (A)
  leer (límite)
  para cont desde A hasta límite hacer
    inicio
      escribir (A*cont)
    fin
  fin

```

53. Averigua qué hace este algoritmo:

```

algoritmo misterioso_2
variables
  A, B, C son enteros
inicio
  leer(A,B,C)
  si (A > B) y (A > C) entonces
    inicio
      si (B > C) entonces
        inicio
          escribir (A-B-C)
        fin
      fin
    fin
  si_no
    inicio

```

```

    si (A < B) y (A < C) entonces
    inicio
        si (B < C) entonces
        inicio
            escribir (A+B+C)
        fin
    fin
inicio
fin

```

54. Averigua qué hace este algoritmo.

```

algoritmo misterioso_3
variables
    nombre es cadena
    dinero es entero
    valor1, valor2, valor3 son enteros
    continuar es carácter
inicio
    escribir('Introduzca su nombre')
    leer(nombre)
    escribir('Bienvenido al juego, ', nombre)
    dinero = 50
    repetir
    inicio
        valor1 = aleatorio(9)
        valor2 = aleatorio(9)
        valor3 = aleatorio(9)
        escribir('Su jugada es:')
        escribir(valor1, valor2, valor3)
        si (valor1 == valor2) y (valor1 == valor3) entonces
        inicio
            dinero = dinero + 100
            escribir ('¡Enhorabuena, ', nombre, '! Ha ganado 100€')
        fin
        si_no
        inicio
            si (valor1 == valor2) o (valor1 == valor3) o (valor2 == valor3) entonces
            dinero = dinero + 20
        si_no
            dinero = dinero - 10
        fin
        escribir('Su saldo actual es de ', dinero, ' euros')
        escribir('¿Desea seguir jugando? (S/N)')
        leer(continuar)
    fin
    mientras que (continuar == 'S')
        escribir('¡Hasta la próxima!')
fin

```

55. Introduce las siguientes modificaciones en el algoritmo anterior:

- Al finalizar el juego, imprimir un texto con la cantidad de dinero que le queda al jugador
- Si en algún momento el jugador se queda sin saldo, el juego debe terminar automáticamente, sin siquiera preguntar si desea continuar.
- Introducir un nivel de dificultad seleccionable por el jugador, de manera que la probabilidad de acertar sea proporcional a ese nivel de dificultad (modificando las funciones aleatorias).

Actividades sobre programación modular

56. Averigua qué hace este programa modular:

```

algoritmo ejercicio_1
variables
    operacion es carácter
    A, B, result son enteros
inicio
    repetir
    inicio
        escribir("Introduzca la operación (Q para terminar)")
        leer(operacion)
        si (operacion != 'Q') entonces
        inicio
            escribir("Introduzca los operandos")
            leer(A, B)
            si (operacion == '+') entonces

```

```

        result = sumar(A, B)
    si (operacion == '-') entonces
        result = restar(A, B)
    si (operacion == '+') o (operacion == '-') entonces
        escribir("El resultado es: ", result)
    si_no
        escribir("Error: operación no reconocida")
    fin
fin
mientras que (operacion != 'Q')
fin

entero función sumar (N1, N2 son enteros)
variables
    s es entero
inicio
    s = N1 + N2
    devolver (s)
fin

entero función restar (N1, N2 son enteros)
variables
    r es entero
inicio
    r = N1 - N2
    devolver (N1 - N2)
fin

```

57. Escribe un programa modular que pregunte al usuario si desea ver los números pares o impares y que, dependiendo de la respuesta, muestre en la pantalla los números pares o impares entre 1 y 1000.
58. Escribe una función salario() que calcule el dinero que debe cobrar un trabajador a la semana, pasándole como parámetros el número de horas semanales que ha trabajado y el precio que se le paga por cada hora. Si ha trabajado más de 40 horas, el salario de cada hora adicional es 1,5 veces el de las horas convencionales.
59. Escribe dos versiones de la función: una con paso de parámetros por valor y otra por referencia.
60. Modifica el programa anterior para que calcule mediante subprogramas:
- El **salario bruto mensual**, sabiendo que todas las horas que excedan de 40 semanales se consideran horas extra. Las primeras 5 horas extra se cobran a 1,5 veces la tarifa normal, y las demás al doble de la tarifa normal.
 - Los descuentos por **impuestos**: se le descontará un 10% si gana menos de 1000 € al mes, y un 15% si gana más de esa cantidad.
 - El **salario neto**, es decir, el dinero que cobrará después de descontar los impuestos

61. Cuando se trabaja con polígonos regulares, conociendo su número de lados, la longitud de cada lado y la apotema¹¹, se puede calcular el área y el perímetro según estas expresiones:

$$\begin{aligned}\text{Área} &= n^{\circ} \text{ lados} \times \text{longitud del lado} \times \text{apotema} / 2 \\ \text{Perímetro} &= n^{\circ} \text{ de lados} \times \text{longitud del lado}\end{aligned}$$

Escribe un programa modular que pregunte esos tres valores y calcule el área y el perímetro de cualquier polígono regular, y además escriba su denominación (triángulo, rectángulo, pentágono, hexágono, etc, hasta polígonos de 12 lados)

62. Escribe un programa que simule el mecanismo de devolución de monedas de una máquina expendedora. El programa preguntará una cantidad en euros y luego calculará qué monedas debería devolver la máquina para completar esa cantidad. Por ejemplo, si la cantidad es 1,45 €, la respuesta del programa debe ser: una moneda de 1 €, dos de 20 céntimos y una de 5 céntimos.

Utiliza programación modular. Puedes elegir entre estos dos enfoques:

- Escribir **un único subalgoritmo** que se encargue del calcular todas las monedas que hacen falta.
- Escribir **varios subalgoritmos**, uno para cada tipo de moneda, y que cada uno se encargue de determinar cuántas monedas de ese tipo hacen falta para realizar la devolución.

63. Escribe un programa para predecir el tiempo que va a hacer mañana a partir de varios datos atmosféricos suministrados por el usuario. Estos datos son:

¹¹ La apotema es un segmento que va del centro del polígono a la mitad de uno de sus lados

- a. La **presión atmosférica**: puede ser alta, media o baja.
- b. La **humedad relativa**: también puede ser alta, media o baja

Tienes que hacer tres subalgoritmos. El primero se encargará de calcular la probabilidad de lluvia, el segundo la probabilidad de sol y el tercero la probabilidad de que haga frío, mediante estos cálculos:

- **Para calcular la probabilidad de lluvia:**

Presión	Humedad	Probabilidad de lluvia
Baja	Alta	Muy alta
Baja	Media	Alta
Baja	Baja	Media
Media	Media	Media
En cualquier otro caso		Baja

- **Para calcular la probabilidad de que haga sol:**

Presión	Humedad	Probabilidad de que haga sol
Baja	Alta	Baja
Baja	Media	Media
Baja	Alta	Media
Media	Media	Media
En cualquier otro caso		Alta

- **Para calcular la probabilidad que haga frío:**

Presión	Humedad	Probabilidad de que haga frío
Baja	Alta	Alta
Baja	Media	Alta
Media	Alta	Alta
Media	Media	Media
En cualquier otro caso		Baja

64. Escribe un programa que simule el funcionamiento de un **reloj**. El usuario introducirá la hora actual a través de teclado y, a partir de entonces, el programa incrementará esa hora cada segundo. Supondremos que existe una función llamada `esperar()`, que hace que el ordenador espere la cantidad de segundos indicada entre paréntesis antes de pasar a la siguiente instrucción.

Por ejemplo, `esperar(5)` detiene momentáneamente la ejecución del programa durante 5 segundos. Del mismo modo, `esperar(1)` la detiene durante 1 segundo.

Utiliza un subalgoritmo para actualizar el tiempo una vez por segundo.

65. Vamos a suponer que existe una función de biblioteca que escriba el carácter correspondiente a cierto código ASCII. La función la llamaremos `carácter_ascii()`. Por ejemplo, al escribir la siguiente asignación, la variable `letra` tomará el valor "A", que es el carácter correspondiente al código ASCII 65:

```
letra es carácter
letra = carácter_ascii(65)
```

Escribe un programa que permita al usuario introducir cualquier número entre 0 y 255 y muestre el carácter al que corresponde ese código, con las siguientes salvedades:

- Los caracteres **del 0 al 31** no son imprimibles, sino que son caracteres de control especiales (como, por ejemplo, "fin de línea"). Si el usuario introduce un código en este rango, el programa debe informarle de su error.
 - Cualquier número **fuera del rango de 0 a 255** debe provocar que se escriba toda la tabla de caracteres ASCII desde el 32 hasta el 255.
66. Escribe un programa modular que permita convertir una cantidad de información expresada en cualquier unidad de medida (bit, byte, KB, MB y GB) a cualquier otra. El programa, por tanto, debe preguntar la cantidad y la unidad en la que está expresada, así como la unidad a la que se desea convertir.

Por ejemplo, la siguiente podría ser una secuencia de acciones del programa:

```
Escriba una cantidad de información
> 2
¿En qué unidad está expresada esta cantidad de información (bit,
```

```

byte, KB, MB o GB)?
> MB
¿A qué unidad desea convertirla (bit, byte, KB, MB o GB)?
> KB
La solución es: 2048 KB

```

67. Escribe un programa modular que pregunte al usuario su fecha de nacimiento y la fecha del día de hoy, y calcule la edad del usuario en años.

Este programa se puede mejorar haciendo que calcule la edad en años, meses y días (¡incluso en horas, minutos y segundos!), pero es una labor por ahora solo apta para los/las más valientes.

68. Escribe un programa modular que puedan utilizar en una tienda para calcular el descuento de los artículos. En esta tienda aplican un descuento del 15% en todos los artículos vendidos a los mayores de 65 años, y de un 10% a los menores de 25.

El programa debe preguntar, al inicio, la fecha del día de hoy, y no volver a preguntarla más. Después irá preguntando por el precio del artículo que se vende y la fecha de nacimiento del cliente. Entonces calculará la edad y, a partir de ella, determinará el descuento que se debe aplicar al artículo.

El proceso se repetirá hasta que se introduzca un precio de artículo negativo.

Para calcular la edad de los clientes puedes reutilizar el subalgoritmo que escribiste en el ejercicio anterior (he aquí una de las grandes ventajas de utilizar subalgoritmos: el código se puede reutilizar fácilmente).

69. Escribe un programa que lea tres números, A, B y C, y que los muestre en la pantalla ordenados de menor a mayor, utilizando para ello un módulo que los ordene, intercambiando sus valores si fuera necesario. Usa el método de paso de parámetros por referencia.
70. Escribe un programa modular que lea un número de hasta 5 cifras por teclado y lo escriba en forma de letra. Por ejemplo, si se introduce el número 1980, la salida del programa debe ser "mil novecientos ochenta". Utiliza una función para cada posición del número (unidades, decenas, centenas, etc)
71. Escribe un programa modular que lea un número binario de hasta 8 cifras y lo convierta a decimal.
72. Estudia detenidamente el siguiente programa sobre un juego de dados y averigua en qué consiste exactamente, porque luego lo vas a tener que modificar.

```

algoritmo dados
variables
    puntos_ordenador, puntos_humano son enteros
    turno es cadena
    tirada es carácter
    dado1, dado2, dado3 son enteros
    premio es entero
inicio
    puntos_ordenador = 0
    puntos_humano = 0
    turno = "HUMANO"
repetir
inicio
    si (turno == "HUMANO") entonces
        inicio
            escribir ("Es tu turno. Pulsa Enter para tirar los dados o Q para terminar")
            leer (tirada)
        fin
        si_no
            escribir ("Es mi turno.")
        si (tirada != 'Q') entonces
            inicio
                dado1 = aleatorio(5) + 1
                dado2 = aleatorio(5) + 1
                dado3 = aleatorio(5) + 1
                escribir("La tirada ha sido: ", dado1, dado2, dado3)
                premio = 0
                si (dado1 == dado2) o (dado1 == dado3) entonces
                    premio = 10
                si (dado1 == dado2) y (dado1 == dado3) entonces
                    inicio
                        si (dado1 == 6) entonces
                            premio = 50
                        si_no
                            premio = 20
                    fin
                si (turno == "HUMANO") entonces
                    inicio
                        puntos_humano = puntos_humano + premio

```

```

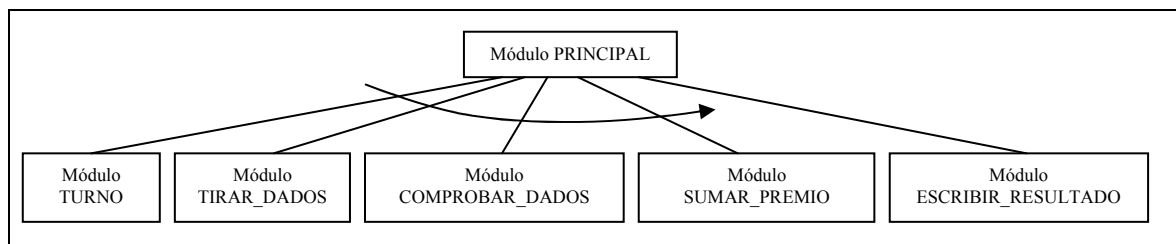
    turno = "ORDENADOR"
  fin
  si_no
  inicio
    puntos_ordenador = puntos_ordenador + premio
    turno = "HUMANO"
  fin
fin
fin
fin
mientras que (tirada != 'Q')
  escribir ("Tu puntuación final: ", puntos_humano)
  escribir ("Mi puntuación final: ", puntos_ordenador)
  si (puntos_humano > puntos_ordenador) entonces
    escribir ("¡Has ganado, enhorabuena!")
  si_no
    si (puntos_ordenador < puntos_humano) entonces
      escribir ("Te he ganado. Inténtalo en otra ocasión.")
    si_no
      escribir ("Hemos empatado. Intenta ganarme otro día.")
fin

```

Este es un ejemplo de **algoritmo demasiado complejo** y, por lo tanto, demasiado difícil de entender, de modo que lo vamos a descomponer en subproblemas más sencillos. Los subproblemas (módulos) serán:

- * **Módulo turno:** escribe en la pantalla de quién es el turno y lee la tirada en caso de que el turno sea del jugador humano
- * **Módulo tirar_dados:** tira los tres dados y escribe el resultado en la pantalla
- * **Módulo comprobar_dados:** compara el resultado de los dados y determina qué premio corresponde a la tirada
- * **Módulo sumar_premio:** suma el premio de la tirada a la puntuación del jugador que tiene el turno
- * **Módulo escribir_resultado:** muestra en la pantalla la puntuación final de cada jugador y un mensaje diciendo quién ha ganado.

Reescribe el **programa** completo, añadiendo comentarios donde consideres necesario. Respeta el siguiente diagrama de descomposición modular y complétalo señalando el trasiego de información entre los módulos:



73. Averigua para qué sirve este programa y conviértelo en modular. Diseña tú la estructura de módulos y dibuja el diagrama de estructura.

```

algoritmo notas
variables
  contraseña es cadena
  n_alumnos, alum, asig son enteros
  nota, nota_media es real
inicio
  escribir("Bienvenido al sistema. Por favor, escriba la contraseña.")
  repetir
    leer(contraseña)
  mientras que (contraseña=="12345")
    escribir("Introduzca el número de alumnos")
    leer (n_alumnos)
    para alum desde 1 hasta n_alumnos hacer
      inicio
        escribir ("Introduciendo notas del alumno/a nº ", alum)
        nota_media = 0.0
        para asig desde 1 hasta 5 hacer
          inicio
            según (asig) hacer
              inicio
                1: escribir("Escriba la nota de los ejercicios")
                2: escribir("Escriba la nota del examen")
                3: escribir("Escriba la nota de clase")
                4: escribir("Escriba la nota del trabajo obligatorio")
                5: escribir("Escriba la nota de aptitud")
              fin
            leer (nota)
          según (asig) hacer

```

```
        inicio
        1: nota_media = nota_media + (nota * 0.10)
        2: nota_media = nota_media + (nota * 0.30)
        3: nota_media = nota_media + (nota * 0.15)
        4: nota_media = nota_media + (nota * 0.25)
        5: nota_media = nota_media + (nota * 0.20)
        fin
    fin
    escribir(nota_media)
    si (nota_media >= 5) entonces
        escribir ("El alumno/a está APROBADO/A")
    si_no
        escribir ("El alumno/a está SUSPENSO/A")
    fin
fin
```