3.2 Operaciones aritmético-lógicas en Pascal

Operadores

Los operadores sirven para combinar los términos de las expresiones. En Pascal, se manejan tres grupos de operadores :

1. ARITMÉTICOS
2. RELACIONALES
3. LÓGICOS

Operadores aritméticos

Son aquellos que sirven para operar términos numéricos. Estos operadores podemos clasificarlos a su vez como :

a. UNARIOS
b. BINARIOS

Los operadores UNARIOS son aquellos que trabajan con UN OPERANDO. Pascal permite el manejo de un operador unario llamado :

MENOS UNARIO
Este operador denota la negación del operando, y se representa por medio del signo menos (-) colocado antes del operando.

Por ejemplo :
Si x tiene asignado el valor 100, -x dará como resultado -100 ; esto es que el resultado es el inverso aditivo del operando.

Los operadores BINARIOS, son los que combinan DOS OPERANDOS , dando como resultado un valor numérico cuyo tipo será igual al mayor de los tipos que tengan los operandos.

La siguiente tabla muestra los símbolos de los operadores binarios de Pascal así como los nombres de las operaciones que realizan.

Operadores aritméticos básicos

Operador	Operación	Operandos	Ejemplo	Resultado
+	Suma	real , integer	a + b	suma de a y b
-	Resta	real , integer	a - b	Diferencia de a y b
*	Multiplicación	real , integer	a * b	Producto de a por b
/	División	real , integer	a / b	Cociente de a por b
div	División entera	integer	a div b	Cociente entero de a por b
mod	Módulo	integer	a mod b	Resto de a por b
shl	Desplazamiento a la izquierda		a shl b	Desplazar a la izquierda b bits
shr	Desplazamiento a la derecha		a shr b	Desplazar a la derecha b bits

Conviene observar lo siguiente :

1. Cuando los dos operandos sean del tipo integer, el resultado será de tipo integer.
2. Cuando cualquiera de los dos operandos, o ambos, sean del tipo real, el resultado será de tipo real.
3. Cuando, en la operación div, OPERANDO-1 y OPERANDO-2 tienen el mismo signo, se obtiene un resultado con signo positivo; si los operandos difieren en signo, el resultado es negativo y el truncamiento tiene lugar hacia el cero.

Ejemplos :

7 div 3 = 2
(-7) div (-3) = 2 (-
7) div 3 = -2
7 div (-3) = -2
15.0 div 3.0 = no válido
15 div (4/2) = no válido
La operación div almacena sólo la parte entera del resultado, perdiéndose la parte fraccionaria (truncamiento).

4. La operación MODULO está definida solamente para OPERANDO-2 positivo. El resultado se dará como el entero no negativo más pequeño que puede ser restado de OPERANDO-1 para obtener un múltiplo de OPERANDO-2 ; por ejemplo :

6 mod 3 = 0
7 mod 3 = 1
(-6) mod 3 = 0
(-7) mod 3 = -1 (-
5) mod 3 = -2
(-15) mod (-7) = -1

En la operaciones aritméticas, debe asegurarse que el resultado de sumar, restar o multiplicar dos valores, no produzca un resultado fuera de los rangos definidos por la implementación para los diferentes tipos.

Operadores relacionales

Una RELACIÓN consiste de dos operandos separados por un operador relacional. Si la relación es satisfecha, el resultado tendrá un valor booleano true ; si la relación no se satisface, el resultado tendrá un valor false. Los operadores deben ser del mismo tipo, aunque los valores de tipo real, integer y byte pueden combinarse como operandos en las relaciones. A continuación se describen los operadores relacionales utilizados en Pascal:

Símbolo	Significado
=	IGUAL que
<>	NO IGUAL que
<	MENOR que
>	MAYOR que
<=	MENOR o IGUAL que
>=	MAYOR o IGUAL que

Ejemplos:

Relación	Resultado
20 = 11	false
15 < 20	true
PI > 3.14	true
'A' < 20	false
'A' = 65	true

Operadores lógicos

Al igual que las relaciones, en las operaciones con operadores lógicos se tienen resultados cuyo valor de verdad toma uno de los valores booleanos true o false.

Los operadores lógicos en Pascal son :

NOT
Sintaxis : not operando
Descripción : Invierte el valor de verdad de operando.
Ejemplo :
Si bandera tiene un valor de verdad true, not bandera produce un resultado con valor de verdad false.

AND
Sintaxis : operando.1 and operando.2
Descripción : Produce un resultado con valor de verdad true cuando ambos operandos tienen valor de verdad true; en cualquier otro caso el resultado tendrá un valor de verdad false.

OR
Sintaxis : operando.1 or operando.2
Descripción : Produce un resultado con valor de verdad false cuando ambos operadores tienen valores de verdad false; en cualquier otro caso el resultado tendrá un valor de verdad true.

XOR
Sintaxis : operando.1 xor operando.2
Descripción : Un operando debe tener valor de verdad true y el otro false para que el resultado tenga valor de verdad true.

Turbo Pascal también permite operaciones entre los bits de operandos exclusivamente de tipo entero :

AND
Sintaxis : operando.1 and operando.2
Descripción: Pone a ceros los bits de operando.2 cuyos correspondientes en operando.1 estén en ceros.
Los valores se pasan a binario, y, sobre cada bit de operando.1 se realiza la operación and lógica con el correspondiente bit de operando.2.
Ejemplo : 29 and 30 = 28
Cuya forma en binario es :

```
      0000000000011101 = 29 (operando.1)
and 0000000000011110 = 30 (operando.2)

      0000000000011100 = 28 (resultado)
```

OR (o inclusiva)
Sintaxis : operando.1 or operando.2
Descripción : Pone a uno los bits de operando.1 cuyos correspondientes bits en operando.2 están a uno. Ejemplo :
17 or 30 = 31
En binario:

```
      0000000000010001 = 17 (operando.1)
or 0000000000011110 = 30 (operando.2)

     0000000000011111 = 31 (resultado)
```

XOR (o exclusiva)
Sintaxis : operando.1 xor operando.2
Descripción : Invierte el estado de los bits de operando.1, cuyos correspondientes en operando.2 están a uno. Ejemplo :
103 xor 25 = 126
En binario:

```
      0000000001100111 = 103 (operando.1)
xor 0000000000011001 = 25  (operando.2)

      0000000001111110 = 126 (resultado)
```

SHL
Sintaxis : operando.1 shl operando.2
Descripción : Desplaza hacia la izquierda los bits de operando.1, el número de posiciones establecidas por operando.2.
Los bits que salen por el extremo izquierdo se pierden.
Ejemplo : 10 shl 2 = 40
En binario:
0000000000001010 = 10 (operando.1) shl 2
<= 0000000000101000 = 40 (resultado)
(operando.2)

SHR
Sintaxis : operando.1 shr operando.2
Descripción : Desplaza hacia la derecha los bits de operando.1 el número de posiciones establecidas por operando.2.
Los bits que salen por el extremo derecho se pierden
Ejemplo : 125 shr 3 = 15
En binario :
0000000001111101 = 125 (operando.1) shr
3 => 0000000000001111 = 15 (resultado)
(operando.2)

Expresiones

Las expresiones son secuencias de constantes y/o variables separadas por operadores válidos. Se puede construir una expresión válida por medio de :

1. Una sola constante o variable, la cual puede estar precedida por un signo + ó - .
2. Una secuencia de términos (constantes, variables, funciones) separados por operadores.

Además debe considerarse que:

- Toda variable utilizada en una expresión debe tener un valor almacenado para que la expresión, al ser evaluada, dé como resultado un valor.
- Cualquier constante o variable puede ser reemplazada por una llamada a una función.

Como en las expresiones matemáticas, una expresión en Pascal se evalúa de acuerdo a la *precedencia de operadores*. La siguiente tabla muestra la precedencia de los operadores en Turbo Pascal:

Precedencia de operadores	
5	- (Menos unario)
4	not
3	* / div mod and shl shr
2	+ - or xor
1	= <> > < >= <=

Las reglas de evaluación para las expresiones son
:

1. Si todos los operadores en una expresión tienen la misma precedencia, la evaluación de las operaciones se realiza de izquierda a derecha.
2. Cuando los operadores sean de diferentes precedencias, se evalúan primero las operaciones de más alta precedencia (en una base de izquierda a derecha), luego se evalúan las de precedencia siguiente, y así sucesivamente.
3. Las reglas 1) y 2) pueden ser anuladas por la inclusión de paréntesis en una expresión.

Ejemplos :

f **3 + 2*5 {*,+}**
f **4 + 10 =14**
f **20*4 div 5**
f **{Igual prioridad de izquierda a derecha : *,div}**
f **div 5 = 16**
f **3 - 5 * (20+(6/2))**
f **3 - 5 * (20+(6/2)) = 3 - 5 * (20 + 3)**
f **{paréntesis más interno}**
f **= 3 - 5 * 23 {segundo paréntesis}**
f **= 3 - 115 {Multiplicación}**
f **= -112 {resta}**

3.2. Operaciones aritmético-lógicas en Lenguaje C

Operadores

Un ***operador*** *es un carácter o grupo de caracteres que actúa sobre una, dos o más variables para realizar una determinada* ***operación*** *con un determinado* ***resultado***. Ejemplos típicos de operadores son la *suma* (+), la *diferencia* (-), el *producto* (*), etc. Los operadores pueden ser ***unarios***, ***binarios*** y ***ternarios***, según actúen sobre uno, dos o tres operandos, respectivamente. En C existen muchos operadores de diversos tipos (éste es uno de los puntos fuertes del lenguaje), que se verán a continuación.

OPERADORES ARITMÉTICOS

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

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Resto: %

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.

El único operador que requiere una explicación adicional es el operador ***resto*** %. En realidad su nombre completo es ***resto de la división entera***. Este operador se aplica solamente a constantes, variables o expresiones de tipo ***int***. Aclarado esto, su significado es evidente: 23%4 es 3, puesto que el resto de dividir 23 por 4 es 3. Si ***a%b*** es cero, ***a*** es múltiplo de ***b***.

Como se verá más adelante, una ***expresión*** es un conjunto de variables y constantes –y también de otras expresiones más sencillas– relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es el siguiente polinomio de grado 2 en la variable ***x***:

```
5.0 + 3.0*x - x*x/2.0
```

Las expresiones pueden contener ***paréntesis*** (...) que agrupan a algunos de sus términos. Puede haber paréntesis contenidos dentro de otros paréntesis. El significado de los paréntesis coincide con el habitual en las expresiones matemáticas, con algunas características importantes que se verán más adelante. En ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones.

OPERADORES DE ASIGNACIÓN

Los ***operadores de asignación*** atribuyen a una variable –es decir, depositan en la zona de memoria correspondiente a dicha variable– el resultado de una expresión o el valor de otra variable (en realidad, una variable es un caso particular de una expresión).

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

```
nombre_de_variable = expresion;
```

cuyo funcionamiento es como sigue: se evalúa **expresion** y el resultado se deposita en **nombre_de_variable**, sustituyendo cualquier otro valor que hubiera en esa posición de memoria anteriormente. Una posible utilización de este operador es como sigue:

```
variable = variable + 1;
```

Desde el punto de vista matemático este ejemplo no tiene sentido (¡Equivale a 0 = 1!), pero sí lo tiene considerando que en realidad ***el operador de asignación (=) representa una sustitución***; en efecto, se toma el valor de **variable** contenido en la memoria, se le suma una unidad y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador **variable**, sustituyendo al valor que había anteriormente. El resultado ha sido incrementar el valor de **variable** en una unidad.

Así pues, una variable puede aparecer a la izquierda y a la derecha del operador (=). Sin embargo, *a la izquierda del operador de asignación (=) no puede haber nunca una expresión*: tiene que ser necesariamente el nombre de una variable[5]. Es incorrecto, por tanto, escribir algo así como:

```
a + b = c;      // incorrecto
```

Existen otros cuatro operadores de asignación (`+=, -=, *= y /=`) formados por los 4 operadores aritméticos seguidos por el carácter de igualdad. Estos operadores simplifican algunas operaciones recurrentes sobre una misma variable. Su forma general es:

```
variable op= expresion;
```

donde ***op*** representa cualquiera de los operadores (`+ - * /`). La expresión anterior es equivalente a:

```
variable = variable op expresion;
```

A continuación se presentan algunos ejemplos con estos operadores de asignación:

```
distancia += 1;      equivale a:      distancia = distancia + 1;
```

```
rango /= 2.0          equivale a:      rango = rango /2.0
x *= 3.0 * y - 1.0    equivale a:      x = x * (3.0 * y - 1.0)
```

OPERADORES INCREMENTALES

Los ***operadores incrementales*** (++) y (--) son operadores unarios que incrementan o disminuyen ***en una unidad*** el valor de la variable a la que afectan. Estos operadores pueden ir inmediatamente delante o detrás de la variable. Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión. A continuación se presenta un ejemplo de estos operadores:

---

[5] También podría ir una dirección de memoria -o una expresión cuyo resultado fuera una dirección de memoria-, precedida del *operador indirección (*)*. Esto es lo que en C se llama *left-value* o *l-value* (algo que puede estar a la izquierda del oprador (=)). Más adelante, al hablar de ***punteros***, quedará más claro este tema.

```
i = 2;
j = 2;
m = i++;      // despues de ejecutarse esta sentencia m=2 e i=3
n = ++j;      // despues de ejecutarse esta sentencia n=3 y j=3
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados **m** y **n** del ejemplo anterior son diferentes.

OPERADORES RELACIONALES

Este es un apartado especialmente importante para todas aquellas personas sin experiencia en programación. Una característica imprescindible de cualquier lenguaje de programación es la de ***considerar alternativas***, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los ***operadores relacionales*** permiten estudiar si se cumplen o no esas condiciones. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (***yes***, ***no***), (***on***, ***off***), (***true***, ***false***), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (***true***, ***false***). Si una condición se cumple, el resultado es ***true***; en caso contrario, el resultado es ***false***.

En C un 0 representa la condición de ***false***, y cualquier número distinto de 0 equivale a la condición ***true***. Cuando el resultado de una expresión es ***true*** y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad. Los ***operadores relacionales*** de C son los siguientes:

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

Todos los ***operadores relacionales*** son operadores ***binarios*** (tienen dos operandos), y su forma general es la siguiente:

```
expresion1 op expresion2
```

donde ***op*** es uno de los operadores (`==, <, >, <=, >=, !=`). El funcionamiento de estos operadores es el siguiente: se evalúan **expresion1** y **expresion2**, y se comparan los valores resultantes. *Si la condición representada por el operador relacional se cumple, el resultado es 1; si la condición no se cumple, el resultado es 0.*

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

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

OPERADORES LÓGICOS

Los ***operadores lógicos*** son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc. El lenguaje C tiene dos operadores lógicos: el operador ***Y*** (&&) y el operador ***O*** (||). En inglés son los operadores ***and*** y ***or***. Su forma general es la siguiente:

```
expresion1 || expresion2
expresion1 && expresion2
```

El operador **&&** devuelve un 1 si tanto **expresion1** como **expresion2** son verdaderas (o distintas de 0), y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0); por otra parte, el operador **||** devuelve 1 si al menos una de las expresiones es cierta. Es importante tener en cuenta que los compiladores de C tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no deseados. Por ejemplo: para que el resultado del operador **&&** sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa **expresion1** y es falsa, ya no hace falta evaluar **expresion2**, y de hecho no se evalúa. Algo parecido pasa con el operador **||**: si **expresion1** es verdadera, ya no hace falta evaluar **expresion2**.

Los operadores **&&** y **||** se pueden combinar entre sí –quizás agrupados entre paréntesis–, dando a veces un código de más difícil interpretación. Por ejemplo:

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

OTROS OPERADORES

Además de los operadores vistos hasta ahora, el lenguaje C dispone de otros operadores. En esta sección se describen algunos ***operadores unarios*** adicionales.

– Operador ***menos*** (–).

El efecto de este operador en una expresión es cambiar el signo de la variable o expresión que le sigue. Recuérdese que en C no hay constantes numéricas negativas. La forma general de este operador es:

```
- expresion
```

– Operador ***más*** (+).

Este es un nuevo operador unario introducido en el ANSI C, y que tiene como finalidad la de servir de complemento al operador (–) visto anteriormente. Se puede anteponer a una variable o expresión como operador unario, pero en realidad no hace nada.

– Operador ***sizeof***().

Este es el operador de C con el nombre más largo. Puede parecer una función, pero en realidad es un operador. La finalidad del operador **sizeof()** es devolver el tamaño, en ***bytes***, del tipo de variable introducida entre los paréntesis. Recuérdese que este tamaño depende del compilador y del tipo de computador que se está utilizando, por lo que es necesario disponer de este operador para producir código portable. Por ejemplo:

```
var_1 = sizeof(double)           // var_1 contiene el tamaño
                                 //   de una variable double
```

– Operador ***negación lógica*** (!).

Este operador devuelve un cero (*false*) si se aplica a un valor distinto de cero (*true*), y devuelve un 1 (*true*) si se aplica a un valor cero (*false*). Su forma general es:

```
!expresion
```

– Operador ***coma*** (,).

Los operandos de este operador son expresiones, y tiene la forma general:

```
expresion = expresion_1, expresion_2
```

En este caso, **expresion_1** se evalúa primero, y luego se evalúa **expresion_2**. El resultado global es el valor de la segunda expresión, es decir de **expresion_2**. Este es el operador de menos precedencia de todos los operadores de C. Como se explicará más adelante, su uso más frecuente es para introducir expresiones múltiples en la sentencia ***for***.

– Operadores ***dirección*** (&) e ***indirección*** (*).

Aunque estos operadores se introduzcan aquí de modo circunstancial, su importancia en el lenguaje C es absolutamente esencial, resultando uno de los puntos más fuertes –y quizás más difíciles de dominar– de este lenguaje. La forma general de estos operadores es la siguiente:

```
*expresion;
&variable;
```

El ***operador dirección* &** devuelve la dirección de memoria de la variable que le sigue. Por ejemplo:

```
variable_1 = &variable_2;
```

Después de ejecutarse esta instrucción **variable_1** contiene la dirección de memoria donde se guarda el contenido de **variable_2**. Las variables que almacenan direcciones de otras variables se denominan ***punteros*** (o apuntadores), deben ser declaradas como tales, y tienen su propia aritmética y modo de funcionar. Se verán con detalle un poco más adelante.

No se puede modificar la dirección de una variable, por lo que no están permitidas operaciones en las que el operador **&** figura a la izda del operador (=), al estilo de:

```
&variable_1 = nueva_direccion;
```

El ***operador indirección*** ***** es el operador complementario del **&**. Aplicado a una expresión que represente una dirección de memoria (***puntero***) permite hallar el contenido o valor almacenado en esa dirección. Por ejemplo:

```
variable_3 = *variable_1;
```

El contenido de la dirección de memoria representada por la variable de tipo ***puntero*** **variable_1** se recupera y se asigna a la variable **variable_3**.

Como ya se ha indicado, las ***variables puntero*** y los operadores ***dirección*** (&) e ***indirección*** (*) serán explicados con mucho más detalle en una sección posterior.

Expresiones

Ya han aparecido algunos ejemplos de expresiones del lenguaje C en las secciones precedentes. Una expresión es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos. Por ejemplo, 1+5 es una expresión formada por dos *operandos* (1 y 5) y un *operador* (el +); esta expresión es equivalente al valor 6, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos. En C existen distintos tipos de expresiones.

EXPRESIONES ARITMÉTICAS

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (`+, -, *, /, %, ++, --`). Como se ha dicho, también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, la solución de la ecuación de segundo grado:

se escribe, en C en la forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El conjunto de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una ***sentencia***. En la expresión anterior aparece la llamada a la ***función de librería sqrt()***, que tiene como ***valor de retorno*** la raíz cuadrada de su único ***argumento***. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
```

EXPRESIONES LÓGICAS

Los elementos con los que se forman estas expresiones son ***valores lógicos***; *verdaderos* (***true***, o distintos de 0) y *falsos* (***false***, o iguales a 0), y los ***operadores lógicos*** **||**, **&&** y **!**. También se pueden emplear los ***operadores relacionales*** (`<, >, <=, >=, ==, !=`) para producir estos valores lógicos a partir de valores numéricos.
Estas expresiones equivalen siempre a un valor 1 (***true***) o a un valor 0 (***false***). Por ejemplo:

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

donde de nuevo la ***expresión lógica*** es lo que está entre el operador de asignación (=) y el (;). La variable **a** valdrá 1 si **b** es mayor que **c** *y* **c** mayor que **d**, ó si **c** es igual a **e** ó **e** es igual a **b**.

EXPRESIONES GENERALES

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C es su flexibilidad para combinar expresiones y operadores de distintos tipos en una expresión que se podría llamar *general*, aunque es una expresión absolutamente ordinaria de C.

Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un
0); esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico: ***true*** si es distinto de 0 y ***false*** si es igual a 0. Esto permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:

```
(a - b*2.0) && (c != d)
```

A su vez, *el operador de asignación* (=), además de introducir un nuevo valor en la variable que figura a su izda, *deja también este valor disponible para ser utilizado* en una expresión más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables **a**, **b** y **c**:

```
a = b = c = 1;
```

que equivale a:

```
a = (b = (c = 1));
```

En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a **c**; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a **b**; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable **a**.

Reglas de precedencia y asociatividad

El resultado de una expresión depende del orden en que se ejecutan las operaciones. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

```
3 + 4 * 2
```

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11. Con objeto de que el resultado de cada expresión quede claro e inequívoco, es necesario definir las reglas que definen el orden con el que se ejecutan las expresiones de C. Existe dos tipos de reglas para determinar este orden de evaluación: las reglas de ***precedencia*** y de ***asociatividad***. Además, el orden de evaluación puede modificarse por medio de paréntesis, pues *siempre se realizan primero las operaciones encerradas en los paréntesis más interiores*. Los distintos operadores de C se ordenan según su distinta ***precedencia*** o prioridad; para operadores de la misma precedencia o prioridad, en algunos el orden de ejecución es de izquierda a derecha, y otros de derecha a izquierda (se dice que *se asocian* de izda a dcha, o de dcha a izda). A este orden se le llama ***asociatividad***.

En la Tabla 4.1 se muestra la precedencia –disminuyendo de arriba a abajo– y la asociatividad de los operadores de C. En dicha Tabla se incluyen también algunos operadores que no han sido vistos hasta ahora.

Tabla 4.1. Precedencia y asociatividad de los operadores de C.

Precedencia	**Asociatividad**
`() [] -> .`	izda a dcha
`++ -- ! sizeof (tipo)` `+(unario) -(unario) *(indir.) &(dirección)`	dcha a izda
`* / %`	izda a dcha
`+ -`	izda a dcha
`< <= > >=`	izda a dcha
`== !=`	izda a dcha
`&&`	izda a dcha
`||`	izda a dcha
`?:`	dcha a izda
`= += -= *= /=`	dcha a izda
`, (operador coma)`	izda a dcha

En la Tabla anterior se indica que el operador (*) tiene precedencia sobre el operador (+). Esto quiere decir que, en ausencia de paréntesis, el resultado de la expresión `3+4*2` es 11 y no 14. Los operadores binarios (+) y (-) tienen igual precedencia, y asociatividad de izda a dcha. Eso quiere decir que en la expresión,

```
a-b+d*5.0+u/2.0                 // (((a-b)+(d*5.0))+(u/2.0))
```

el orden de evaluación es el indicado por los paréntesis en la parte derecha de la línea (Las últimas operaciones en ejecutarse son las de los paréntesis más exteriores).

Sentencias

Las ***expresiones*** de C son unidades o componentes elementales de unas entidades de rango superior que son las ***sentencias***. Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

SENTENCIAS SIMPLES

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

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

SENTENCIA VACÍA Ó NULA

En algunas ocasiones es necesario introducir en el programa una sentencia *que ocupe un lugar, pero que no realice ninguna tarea*. A esta sentencia se le denomina ***sentencia vacía*** y consta de un simple carácter (;). Por ejemplo:

```
;
```

SENTENCIAS COMPUESTAS O BLOQUES

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de ***sentencias compuestas***. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves **`{...}`**. También se conocen con el nombre de ***bloques***. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas. Un ejemplo de sentencia compuesta es el siguiente:

```
{
     int i = 1, j = 3, k;
     double masa;

     masa = 3.0;
     k = y + j;
}
```

Las sentencias compuestas se utilizarán con mucha frecuenciaal introducir las sentencias que permiten modificar el flujo de control del programa.