CAPÍTULO 2

TIPOS DE DATOS Y VARIABLES EN C

Un **tipo de dato** define de forma explícita un conjunto de valores, denominado **dominio**, sobre el cual se pueden realizar una serie de operaciones. Un **valor** es un elemento del conjunto que hemos llamado dominio. Una **variable** es un espacio de la memoria destinada al almacenamiento de un valor de un tipo de dato concreto, referenciada por un nombre. Son conceptos sencillos, pero muy necesarios para saber exactamente qué se hace cuando se crea una variable en un programa.

Un tipo de dato puede ser tan complejo como se quiera. Puede necesitar un byte para almacenar cualquier valor de su dominio, o requerir de muchos bytes.

Cada lenguaje ofrece una colección de tipos de datos, que hemos llamado **primitivos**. También ofrece herramientas para crear tipos de dato distintos, más complejos que los primitivos y más acordes con el tipo de problema que se aborde en cada momento.

En este capítulo vamos a presentar los diferentes tipos de datos primitivos que ofrece el lenguaje C. Veremos cómo se crean (declaran) las variables, qué operaciones se pueden realizar con cada una de ellas, y de qué manera se pueden relacionar unas variables con otras para formar expresiones. Veremos las limitaciones en el uso de las variables según su tipo de dato.

Ya hemos dicho que un tipo de dato especifica un dominio sobre el que una variable de ese tipo puede tomar sus valores; y unos operadores. A lo largo del capítulo iremos presentando los distintos operadores básicos asociados con los tipos de dato primitivos del lenguaje C. Es importante entender la operación que realiza cada operador y sobre qué dominio este operador está definido.

Declaración de variables.

Antes de ver los tipos de dato primitivos, conviene saber cómo se crea una variable en C.

Toda variable debe ser declarada previa a su uso. **Declarar una variable** es indicar al programa un **identificador** o nombre para esa variable, y el **tipo de dato** para la que se crea esa variable.

La declaración de variable tiene la siguiente sintaxis:

tipo var_1 [=valor1, var_2 = valor_2, …, var_N = valor_N];

Donde *tipo* es el nombre del tipo de variable que se desea crear, y *var_1*, es el nombre o identificador de esa variable.

Aclaración a la notación: en las reglas sintácticas de un lenguaje de programación, es habitual colocar entre corchetes ([]) aquellas partes de la sintaxis que son optativas.

En este caso tenemos que en una declaración de variables se pueden declarar una o más variables del mismo tipo, todas ellas separadas por el **operador coma**. Al final de la sentencia de declaración de variables

se encuentra, como siempre será en cualquier sentencia, el operador punto y coma.

En la declaración de una variable, es posible asignarle un valor de inicio. De lo contrario, la variable creada adquirirá un valor cualquiera entre todos los explicitados por el rango del tipo de dato, desconocido para el programador.

¿Qué ocurre si una variable no es inicializada? En ese caso, al declararla se dará orden de reservar una cantidad de memoria (la que exija el tipo de dato indicado para la variable) para el almacenamiento de los valores que pueda ir tomando esa variable creada. Esa porción de memoria es un elemento físico y, como tal, deberá tener un estado físico. Cada uno de los bits de esta porción de memoria estará en el estado que se ha llamado 1, o en el estado que se ha llamado 0. Y un estado de memoria codifica una información concreta: la que corresponda al tipo de dato para el que está reservada esa memoria.

Es conveniente remarcar esta idea. No es necesario, y tampoco lo exige la sintaxis de C, dar valor inicial a una variable en el momento de su declaración. La casuística es siempre enorme, y se dan casos y circunstancias en las que realmente no sea conveniente asignar a la variable un valor inicial. Pero **habitualmente es muy recomendable inicializar las variables**. Otros lenguajes lo hacen por defecto en el momento de la declaración de variables; C no lo hace. Otros lenguajes detectan como error de compilación (errar sintáctico) el uso de una variable no inicializada; C acepta esta posibilidad.

A partir del momento en que se ha declarado esa variable, puede ya hacerse uso de ella. Tras la declaración ha quedado reservado un espacio de memoria para almacenar la información de esa variable.

Si declaramos *tipo variable = valor;* tendremos la variable <*variable, tipo, R, valor*>, de la que desconocemos su dirección de memoria. Cada vez que el programa trabaje con *variable* estará haciendo referencia a

esta posición de memoria R. Y estará refiriéndose a uno o más bytes, en función del tamaño del tipo de dato para el que se ha creado la variable.

Tipos de datos primitivos en C: sus dominios.

Los tipos de dato primitivos en C quedan recogidos en la tabla 2.1.

Las variables de tipo de dato carácter ocupan 1 byte. Aunque están creadas para almacenar caracteres mediante una codificación como la ASCII (que asigna a cada carácter un valor numérico codificado con esos 8 bits), también pueden usarse como variables numéricas. En ese caso, el rango de valores es el recogido en la tabla 2.1. En el caso de que se traten de variables con signo, entonces el rango va desde -2^7 hasta 2^7-1.

TIPOS	SIGNO	RANGO DE VALORES MENOR	MAYOR
tipos de dato CARÁCTER: **char**			
signed	**char**	-128	+127
unsigned		0	+255
tipos de dato ENTERO: **int**			
signed	**short**	-32.768	+32.767
unsigned		0	+65.535
signed	**long**	-2.147.483.648	+2.147.483.647
unsigned		0	4.294.967.295
tipos de dato CON COMA FLOTANTE			
float		-3.402923E+38	+3.402923E+38
double		-1.7976931E+308	+1.7976931E+308
long double		-1.2E+4932	+1.2E+4932

Tabla 2.1.: Tipos de datos primitivos en C.

Para crear una variable de tipo carácter en C, utilizaremos la palabra clave ***char***. Si la variable es con signo, entonces su tipo será ***signed char***, y si no debe almacenar signo, entonces será ***unsigned char***. Por

defecto, si no se especifica si la variable es con o sin signo, el lenguaje C considera que se ha tomado la variable con signo, de forma que decir ***char*** es lo mismo que decir ***signed char***.

Lo habitual será utilizar variables tipo ***char*** para el manejo de caracteres. Los caracteres simples del alfabeto latino se representan mediante este tipo de dato. El dominio de las variables ***char*** es un conjunto finito ordenado de caracteres, para el que se ha definido una correspondencia que asigna, a cada carácter del dominio, un código binario diferente de acuerdo con alguna normalización. El código más extendido es el código ASCII (American Standard Code for Information Interchange).

Las variables tipo entero, en C, se llaman ***int***. Dependiendo de que esas variables sean de dos bytes o de cuatro bytes las llamaremos de tipo ***short int*** (16 bits) ó de tipo ***long int*** (32 bits). Y para cada una de ellas, se pueden crear con signo o sin signo: ***signed short int*** y ***signed long int***, ó ***unsigned short int*** y ***unsigned long int***. De nuevo, si no se especifica nada, C considera que la variable entera creada es con signo, de forma que la palabra ***signed*** vuelve a ser opcional. En general, se recomienda el uso de la palabra ***signed***. Utilizar esa palabra al declarar enteros con signo facilita la compresión del código.

El tamaño de la variable ***int*** depende del concepto, no introducido hasta el momento, de **longitud de la palabra**. Habitualmente esta longitud se toma múltiplo de 8, que es el número de bits del byte. De hecho la longitud de la palabra viene definido por el máximo número de bits que puede manejar el procesador, de una sola vez, cuando hace cálculos con enteros.

Si se declara una variable en un PC como de tipo ***int*** (sin determinar si es ***short*** o ***long***), el compilador de C considerará que esa variable es de la longitud de la palabra: de 16 o de 32 bits. Es importante conocer ese dato (que depende del compilador), o a cambio es mejor especificar

siempre en el programa si se desea una variable corta o larga, y no dejar esa decisión al tamaño de la palabra.

Una variable declarada como de tipo ***long*** se entiende que es ***long int***. Y una variable declarada como de tipo ***short***, se entiende que es ***short int***. Muchas veces se toma como tipo de dato únicamente el modificador de tamaño, omitiendo la palabra clave ***int***.

Los restantes tipos de dato se definen para codificar valores reales. Hay que tener en cuenta que el conjunto de los reales es no numerable (entre dos reales siempre hay un real y, por tanto, hay infinitos reales). Los tipos de dato que los codifican ofrecen una codificación finita sí numerable. Esos tipos de dato codifican subconjuntos del conjunto de los reales; subconjuntos que, en ningún caso, pueden tomarse como un intervalo del conjunto de los reales.

A esta codificación de los reales se le llama de coma flotante. Así codifica el lenguaje C (y muchos lenguajes) los valores no enteros. Tomando como notación para escribir esos números la llamada notación científica (signo, mantisa, base, exponente: por ejemplo, el numero de Avogadro, $+6,023 \cdot 10^{23}$: signo positivo, mantisa 6,023, base decimal y exponente 23), almacena en memoria, de forma normalizada (**norma IEEE754**) el signo del número, su mantisa y su exponente. No es necesario almacenar la base, que en todos los casos trabaja en la base binaria.

Los tipos de dato primitivos en coma flotante que ofrece el lenguaje C son tres: ***float***, que reserva 4 bytes para su codificación y que toma valores en el rango señalado en la tabla 2.1.; ***double***, que reserva 8 bytes; y ***long double***, que reserva 10 bytes. Desde luego, en los tres tipos de dato el dominio abarca tantos valores positivos como negativos.

Existe además un tipo de dato que no reserva espacio en memoria: su tamaño es nulo. Es el tipo de dato ***void***. No se pueden declarar variables de ese tipo. Más adelante se verá la necesidad y utilidad de

tener definido un tipo de dato de estas características. Por ejemplo es muy conveniente para el uso de funciones.

En C el carácter que indica el fin de la parte entera y el comienzo de la parte decimal se escribe mediante el carácter punto. La sintaxis no acepta interpretaciones de semejanza, y para el compilador el carácter coma es un operador que nada tiene que ver con el punto decimal. Una equivocación en ese carácter causará habitualmente un error en tiempo de compilación.

El lenguaje C dedica nueve palabras para la identificación de los tipos de dato primitivos: ***void***, ***char***, ***int***, ***float***, ***double***, ***short***, ***long***, ***signed*** e ***unsigned***. Ya se ha visto, por tanto más de un 25 % del léxico total del lenguaje C. Existen más palabras clave para la declaración y creación de variables. Se verán más adelante.

Tipos de datos primitivos en C: sus operadores.

Ya se dijo que un tipo de dato explicita un conjunto de valores, llamado dominio, sobre el que son aplicables una serie de operadores. No queda del todo definido un tipo de dato presentando sólo su dominio. Falta indicar cuáles son los operadores que están definidos para cada tipo de dato.

Los operadores pueden aplicarse a una sola variable, a dos variables, e incluso a varias variables. Llamamos **operación unaria** a la que se aplica a una sola variable. Una operación unaria es, por ejemplo, el signo del valor.

No tratamos ahora las operaciones que se pueden aplicar sobre una variable creada para almacenar caracteres. Más adelante hay un capítulo entero dedicado a este tipo de dato ***char***.

Las variables enteras, y las ***char*** cuando se emplean como variables enteras de pequeño rango, además del operador unario del signo, tienen

definidos el operador asignación, los operadores aritméticos, los relacionales y lógicos y los operadores a nivel de bit.

Los operadores de las variables con coma flotante son el operador asignación, todos los aritméticos (excepto el operador módulo o cálculo del resto de una división), y los operadores relacionales y lógicos. Las variables ***float***, ***double*** y ***long double*** no aceptan el uso de operadores a nivel de bit.

Operador asignación.

El operador asignación permite al programador modificar los valores de las variables y alterar, por tanto, el estado de la memoria del ordenador.

El carácter que representa al operador asignación es el carácter '='. La forma general de este operador es

nombre_variable = expresión;

Donde *expresión* puede ser un literal, otra variable, o una combinación de variables, literales y operadores y funciones. Podemos definirlo como una secuencia de operandos y operadores que unidos según ciertas reglas producen un resultado.

Este signo en C no significa igualdad en el sentido matemático al que estamos acostumbrados, sino asignación. No puede llevar a equívocos expresiones como la siguiente:

```
a = a + 1;
```

Ante esta instrucción, el procesador toma el valor de la variable *a*, lo copia en un registro de la ALU donde se incrementa en una unidad, y almacena (asigna) el valor resultante en la variable *a*, modificando por ello el valor anterior de esa posición de memoria. La expresión comentada no es una igualdad de las matemáticas, sino una orden para incrementar en uno el valor almacenado en la posición de memoria reservada por la variable *a*.

El operador asignación tiene dos extremos: el izquierdo (que toma el nombre *Lvalue* en mucho manuales) y el derecho (*Rvalue*). La apariencia del operador es, entonces:

LValue = RValue;

Donde *Lvalue* sólo puede ser el nombre de una variable, y nunca una expresión, ni un literal. Expresiones como *a + b = c;* ó *3 = variable*; son erróneas, pues ni se puede cambiar el valor del literal *3* que, además, no está en memoria porque es un valor literal; ni se puede almacenar un valor en la expresión *a + b*, porque los valores se almacenan en variables, y *a + b* no es variable alguna.

Un error de este estilo interrumpe la compilación del programa. El compilador dará un mensaje de error en el que hará referencia a que el *Lvalue* de la asignación no es correcto.

Cuando se trabaja con variables enteras, al asignar a una variable un valor mediante un literal (por ejemplo, *v = 3;*) se entiende que ese dato viene expresado en base 10.

Pero en C es posible asignar valores en la base hexadecimal. Si se quiere dar a una variable un valor en hexadecimal, entonces ese valor va precedido de un cero y una letra equis. Por ejemplo, si se escribe *v = 0x20*, se está asignando a la variable *v* el valor 20 en hexadecimal, es decir, el 32 en decimal, es decir, el valor 100000 en binario.

Operadores aritméticos.

Los operadores aritméticos son:

1. **Suma**. El identificador de este operador es el carácter '+'. Este operador es aplicable sobre cualquier variable primitiva de C. Si el operador '+' se emplea como operador unario, entonces es el operador de signo positivo.

2. **Resta**. El identificador de este operador es el carácter '– '. Este operador es aplicable sobre cualquier variable primitiva de C. Si el operador '–' se emplea como operador unario, entonces es el operador de signo negativo.
3. **Producto**. El identificador de este operador es el carácter '*'. Este operador es aplicable sobre cualquier variable primitiva de C.
4. **Cociente o División**. El identificador de este operador es el carácter '/'. Este operador es aplicable sobre cualquier variable primitiva de C.

   Cuando el cociente se realiza con variables enteras el resultado será también un entero, y trunca el resultado al mayor entero menor que el cociente. Por ejemplo, 5 dividido entre 2 es igual a 2. Y 3 dividido entre 4 es igual a 0. Es importante tener esto en cuenta cuando se trabaja con enteros.

   Supongamos la expresión

   ```
   sup = (1 / 2) * base * altura;
   ```

   para el cálculo de la superficie de un triángulo, y supongamos que todas las variables que intervienen han sido declaradas enteras. Así expresada la sentencia o instrucción de cálculo, el resultado será siempre el valor 0 para la variable *sup*, sea cual sea el valor actual de las variable *base* o *altura*: y es que al calcular el valor de 1 dividido entre 2, el procesador ofrece como resultado el valor 0.

   Cuando el cociente se realiza entre variables de coma flotante, entonces el resultado es también de coma flotante.

   Siempre se debe evitar el cociente en el que el denominador sea igual a cero, porque en ese caso se dará un error de ejecución y el programa quedará abortado.
5. **Módulo**. El identificador de este operador es el carácter '%'. Este operador calcula el resto del cociente entero. Por su misma

definición, no tiene sentido su aplicación entre variables no enteras: su uso con variables de coma flotante provoca error de compilación. Como en el cociente, tampoco su divisor puede ser cero.

6. **Incremento y decremento**. Estos dos operadores no existen en otros lenguajes. El identificador de estos operadores son los caracteres "++" para el incremento, y "--" para el decremento. Este operador es válido para todos los tipos de dato primitivos de C.

   La expresión "*a++;*" es equivalente a la expresión "*a = a + 1;*". Y la expresión "*a--;*" es equivalente a la expresión "*a = a - 1;*".

   Estos operadores condensan, en uno sola expresión, un operador asignación, un operador suma (o resta) y un valor literal: el valor 1. Y como se puede apreciar son operadores unarios: se aplican a una sola variable.

   Dónde se ubique el operador con respecto a la variable tiene su importancia, porque varía su comportamiento dentro del total de la expresión.

   Por ejemplo, el siguiente código

```
unsigned short int a, b = 2, c = 5;
a = b + c++;
```

   modifica dos variables: por el operador asignación, la variable *a* tomará el valor resultante de sumar los contenidos de *b* y *c*; y por la operación incremento, que lleva consigo asociado otro operador asignación, se incrementa en uno el valor de la variable *c*.

   Pero queda una cuestión abierta: ¿Qué operación se hace primero: incrementar *c* y luego calcular *b + c* para asignar su resultado a la variable *a*; o hacer primero la suma y sólo después incrementar la variable *c*?

   Eso lo indicará la posición del operador. Si el operador incremento (o decremento) precede a la variable, entonces se ejecuta antes de evaluar el resto de la expresión; si se coloca después de la variable,

entonces primero se evalúa la expresión donde está implicada la variable a incrementar o decrementar y sólo después se incrementa o decrementa esa variable.

En el ejemplo antes sugerido, el operador está ubicado a la derecha de la variable *c*. Por lo tanto, primero se efectúa la suma y la asignación sobre *a*, que pasa a valer 7; y luego se incrementa la variable *c*, que pasa a valer 6. La variable *b* no modifica su valor.

Por completar el ejemplo, si la expresión hubiera sido

```
a = b + ++c;
```

entonces, al final tendríamos que *c* vale 6 y que *a* vale 8, puesto que no se realizaría la suma y la asignación sobre *a* hasta después de haber incrementado el valor de la variable *c*.

Los operadores incremento y decremento, y el juego de la precedencia, son muy cómodos y se emplean mucho en los códigos escritos en lenguaje C.

Aunque hasta el tema siguiente no se va a ver el modo en que se pueden recibir datos desde el teclado (función ***scanf()***) y el modo de mostrar datos por pantalla (función ***printf()***), vamos a recoger a lo largo de este capítulo algunas cuestiones muy sencillas para resolver. Por ahora lo importante no es entender el programa entero, sino la parte que hace referencia a la declaración y uso de las variables.

Operadores relacionales y lógicos.

Los operadores relacionales y los operadores lógicos crean expresiones que se evalúan como verdaderas o falsas.

En muchos lenguajes existe un tipo de dato primitivo para estos valores booleanos de verdadero o falso. En C ese tipo de dato no existe.

El lenguaje C toma como falsa cualquier expresión que se evalúe como 0. Y toma como verdadera cualquier otra evaluación de la expresión. Y cuando en C se evalúa una expresión con operadores relacionales y/o lógicos, la expresión queda evaluada a 0 si el resultado es falso; y a 1 si el resultado es verdadero.

Los operadores relacionales son seis: **igual que** ("=="), **distintos** ("!=") , **mayor que** ('>'), **mayor o igual que** (">="), **menor que** ('<') y **menor o igual que** ("<=").

Todos ellos se pueden aplicar a cualquier tipo de dato primitivo de C.

Una expresión con operadores relacionales sería, por ejemplo, *a != 0*, que será verdadero si *a* toma cualquier valor diferente al 0, y será falso si *a* toma el valor 0. Otras expresiones relacionales serían, por ejemplo,

```
a > b + 2;
```

ó

```
x + y == z + t;
```

Con frecuencia interesará evaluar una expresión en la que se obtenga verdadero o falso no solo en función de una relación, sino de varias. Por ejemplo, se podría necesitar saber (obtener verdadero o falso) si el valor de una variable concreta está entre dos límites superior e inferior. Para eso necesitamos concatenar dos relacionales. Y eso se logra mediante los operadores lógicos.

Un error frecuente (y de graves consecuencias en la ejecución del programa) al programar en C ó C++ es escribir el operador asignación ('='), cuando lo que se pretendía escribir era el operador relacional "igual que" ("=="). El C ó C++ la expresión *variable = valor;* será siempre verdadera si *valor* es distinto de cero. Si colocamos una asignación donde deseábamos poner el operador relacional "igual que", tendremos dos consecuencias graves: se cambiará el valor de la variable colocada a la izquierda del operador asignación (cosa que no queríamos)

y, si el valor de la variable de la derecha es distinto de cero, la expresión se evaluará como verdadera al margen de cuáles fueran los valores iniciales de las variables.

Los operadores lógicos son: **AND**, cuyo identificador está formado por el carácter repetido "&&"; **OR**, con el identificador "||"; y el operador **negación**, cuyo identificador es el carácter de admiración final ('!').

a	b	a && b	a || b	!a
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Tabla 2.2.: Resultados de los operadores lógicos.

Estos operadores binarios actúan sobre dos expresiones que serán verdaderas (o distintas de cero), o falsas (o iguales a cero), y devuelven como valor 1 ó 0 dependiendo de que la evaluación haya resultado verdadera o falsa.

La tabla de valores para conocer el comportamiento de estos operadores está recogida en la tabla 2.2. En esa tabla se recoge el resultado de los tres operadores en función del valor de cada una de las dos expresiones que evalúan.

Por ejemplo, supongamos el siguiente código en C:

```
int a = 1 , b = 3 , x = 30 , y = 10;
int resultado;
resultado = a * x == b * y;
```

El valor de la variable resultado quedará igual a 1.

Y si queremos saber si la variable *x* guarda un valor entero positivo menor que cien, escribiremos la expresión

```
(x > 0 && x < 100)
```

Con estos dos grupos de operadores son muchas las expresiones de evaluación que se pueden generar. Quizá en este momento no adquiera mucho sentido ser capaz de expresar algo así; pero más adelante se verá cómo la posibilidad de verificar sobre la veracidad y falsedad de muchas expresiones permite crear estructuras de control condicionales, o de repetición.

Una expresión con operadores relacionales y lógicos admite varias formas equivalentes Por ejemplo, la antes escrita sobre el intervalo de situación del valor de la variable *x* es equivalente a escribir

```
!(x < 0 || x >= 100)
```

- ***Evaluar las siguientes expresiones.***

```
short a = 0, b = 1, c = 5;
a;                      // FALSO
b;                      // VERDADERO
a < b;                  // VERDADERO
5 * (a + b) == c;       // VERDADERO
```

```
float pi = 3.141596;
long x = 0, y = 100, z =1234;

3 * pi < y && (x + y) * 10 <= z / 2;          // FALSO
3 * pi < y || (x + y) * 10 <= z / 2;          // VERDADERO
3 * pi < y && !((x + y) * 10 <= z / 2);       // VERDADERO
```

```
long a = 5, b = 25, c = 125, d = 625;
5 * a == b;             // VERDADERO
5 * b == c;             // VERDADERO
a + b + c + d < 1000;   // VERDADERO
a > b || a = 10;        // VERDADERO
```

La última expresión trae su trampa: Por su estructura se ve que se ha pretendido crear una expresión lógica formada por dos sencillas enlazadas por el operador OR. Pero al establecer que uno de los extremos de la condición es *a = 10* (asignación, y no operador relacional "igual que") se tiene que en esta expresión recogida la variable *a* pasa a

valer *10* y la expresión es verdadera puesto que el valor *10* es verdadero (todo valor distinto de cero es verdadero).

Operadores a nivel de bit.

Ya se ha dicho en el capítulo primero que el lenguaje C es de medio nivel. Con eso se quiere decir que es un lenguaje de programación que tiene la capacidad de trabajar a muy bajo nivel, modificando un bit de una variable, o logrando códigos que manipulan la codificación interna de la información. Todos los operadores a nivel de bit están definidos únicamente sobre variables de tipo entero. No se puede aplicar sobre una variable ***float***, ni sobre una ***double***, ni sobre una ***long double***.

Los operadores a nivel de bit son seis:

1. Operador **AND a nivel de bit**. Su identificador es un solo carácter '&'. Se aplica sobre variables del mismo tipo, con la misma longitud de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en ese bit en esa posición si los dos bits de las dos variables sobre las que se opera valen 1; en otro caso asigna a esa posición del bit el valor 0.
2. Operador **OR a nivel de bit**. Su identificador es un solo carácter '|'. Se aplica sobre variables del mismo tipo, con la misma longitud de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en ese bit en esa posición si alguno de los dos bits de las dos variables sobre las que se opera valen 1; si ambos bits valen cero, asigna a esa posición del bit el valor 0.

   Es frecuente en C y C++ el error de pretender escribir el operador lógico "*and*" ("&&"), o el "*or*" ("||") y escribir finalmente el operador a nivel de bit ('&' ó '|'). Desde luego el significado de la sentencia o instrucción será completamente distinto e imprevisible. Será un error del programa de difícil detección.

3. Operador **OR EXCLUSIVO, ó XOR a nivel de bit**. Su identificador es un carácter '^'. Se aplica sobre variables del mismo tipo, con la misma longitud de bits. Bit a bit compara los dos de cada misma posición y asigna al resultado un 1 en ese bit en esa posición si los dos bits de las dos variables tienen valores distintos: el uno es 1 y el otro 0, o viceversa; si los dos bits son iguales, asigna a esa posición del bit el valor 0.

variable	binario	hex.	dec.
a	1010 1011 1100 1101	ABCD	43981
b	0110 0111 1000 1001	6789	26505
a_and_b	0010 0011 1000 1001	2389	9097
a_or_b	1110 1111 1100 1101	EFCD	61389
a_xor_b	1100 1100 0100 0100	CC44	52292

Tabla 2.3.: Valores del ejemplo en binario, hexadecimal y decimal. Operadores a nivel de bit.

Por ejemplo, y antes de seguir con los otros tres operadores a nivel de bit, supongamos que tenemos el siguiente código:

```
unsigned short int a = 0xABCD, b = 0x6789;
unsigned short int a_and_b = a & b;
unsigned short int a_or_b = a | b;
unsigned short int a_xor_b = a ^ b;
```

La variable *a* vale, en hexadecimal ABCD, y en decimal 43981. La variable *b* 6789, que en base diez es 26505. Para comprender el comportamiento de estos tres operadores, se muestra ahora en la tabla 2.3. los valores de *a* y de *b* en base dos, donde se puede ver bit a bit de ambas variables, y veremos también el bit a bit de las tres variables calculadas.

En la variable *a_and_b* se tiene un 1 en aquellas posiciones de bit donde había 1 en *a* y en *b*; un 0 en otro caso. En la variable *a_or_b* se tiene un 1 en aquellas posiciones de bit donde había al menos un 1 entre *a* y *b*; un 0 en otro caso. En la variable *a_xor_b* se tiene un 1 en aquellas posiciones de bit donde había un 1 en *a* y un 0 en *b*, o

un 0 en *a* y un 1 en *b*; y un cero cuando ambos bits coincidían de valor en esa posición.

La tabla de valores de estos tres operadores queda recogida en la tabla 2.4.

		and	or	xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabla 2.4.: Valores que adoptan los tres operadores a nivel de bit

4. Operador **complemento a uno**. Este operador unario devuelve el complemento a uno del valor de la variable a la que se aplica. Su identificador es el carácter '~'. Si se tiene que a la variable *x* de tipo ***short*** se le asigna el valor hexadecimal ABCD, entonces la variable *y*, a la que se asigna el valor *~x* valdrá 5432. O si *x* vale 578D, entonces *y* valdrá A872. Puede verificar estos resultados calculando los complementos a uno de ambos números.

5. Operador **desplazamiento a izquierda**. Su identificador es la cadena "<<". Es un operador binario, que realiza un desplazamiento de todos los bits de la variable o valor literal sobre la que se aplica un número dado de posiciones hacia la izquierda. Los bits más a la izquierda (los más significativos) se pierden; a la derecha se van introduciendo tantos bits puestos a cero como indique el desplazamiento.

   Por ejemplo, si tenemos el siguiente código:

```
short int var1 = 0x7654;
short int var2 = var1 << 3;
```

La variable *var2* tendrá el valor de la variable *var1* a la que se le aplica un desplazamiento a izquierda de 3 bits.

Si la variable *var1* tiene el valor, en base binaria

0111 0110 0101 0100 (estado de la variable *var1*)

entonces la variable *var2*, a la que se asigna el valor de la variable *var1* al que se han añadido tres ceros a su derecha y se le han eliminado los tres dígitos más a la izquierda queda:

1011 0010 1010 0000 (estado de la variable *var2*).

Es decir, *var2* valdrá, en hexadecimal, B2A0.

Una observación sobre esta operación. Introducir un cero a la derecha de un número es lo mismo que multiplicarlo por la base.

En el siguiente código

```
unsigned short int var1 = 12;
unsigned short int d = 1;
unsigned short int var2 = var1 << d;
```

var2 será el doble que *var1*, es decir, 24. Y si *d* hubiera sido igual a 2, entonces *var2* sería cuatro veces *var1*, es decir, 48. Y si *d* hubiera sido igual a 3, entonces *var2* sería ocho veces *var1*, es decir, 96.

Si llega un momento en que el desplazamiento obliga a perder algún dígito 1 a la izquierda, entonces ya habremos perdido esa progresión, porque la memoria no será suficiente para albergar todos sus dígitos y la cifra será truncada.

Si las variables *var1* y *var2* están declaradas como ***signed***, y si la variable *var1* tiene asignado un valor negativo (por ejemplo, -7), también se cumple que el desplazamiento equivalga a multiplicar por dos. Es buen ejercicio de cálculo de complementos y de codificación de enteros con signo verificar lo datos que a continuación se presentan:

var1 = -7;: estado de memoria FFF9

var2 = var1 << 1;: estado de memoria para la variable *var2* será FFF2, que es la codificación del entero -14.

6. Operador **desplazamiento a derecha**. Su identificador es la cadena ">>". Es un operador binario, que realiza un desplazamiento de todos los bits de la variable o valor literal sobre la que se aplica un número dado de posiciones hacia la derecha. Los bits más a la derecha (los menos significativos) se pierden; a la izquierda se van introduciendo tantos bits como indique el desplazamiento. En esta ocasión, el valor de los bits introducidos por la izquierda dependerá del signo del entero sobre el que se aplica el operador desplazamiento. Si el entero es positivo, se introducen tantos ceros por la izquierda como indique el desplazamiento. Si el entero es negativo, se introducen tantos unos por la izquierda como indique el desplazamiento. Evidentemente, si el entero sobre el que se aplica el desplazamiento a derecha está declarado como **unsigned**, únicamente serán ceros lo que se introduzca por su izquierda, puesto que en ningún caso puede codificar un valor negativo.

   Si desplazar a la izquierda era equivalente a multiplicar por la base, ahora, desplazar a la derecha es equivalente a dividir por la base (división entera, sesgando el valor al entero mayor, menor que el resultado de dicho cociente). Y el hecho de que el desplazamiento a derecha considere el signo en el desplazamiento permite que, en los valores negativos, siga siendo equivalente desplazar a derecha que dividir por la base.

   Por ejemplo, si tenemos el siguiente código

```
signed short int var1 = -231;
signed short int var2 = var1 >> 1;
```

   Entonces, el estado que codifica el valor de *var1* es, expresado en hexadecimal, FF19. Y el valor que codifica entonces *var2*, si lo hemos desplazado un bit a la derecha, será FF8C, que es la codificación del entero negativo -116.

Los operadores a nivel de bit tienen una gran potencialidad. De todas formas no son operaciones a las que se está normalmente habituado, y eso hace que no resulte muchas veces evidente su uso. Los operadores a nivel de bit operan a mucha mayor velocidad que, por ejemplo, el operador producto o cociente. En la medida en que se sabe, quien trabaja haciendo uso de esos operadores puede lograr programas notablemente más veloces en ejecución.

Operadores compuestos.

Ya se ha visto que una expresión de asignación en C trae, a su izquierda (*Lvalue*), el nombre de una variable y, a su derecha (*Rvalue*) una expresión a evaluar, o un literal, o el nombre de otra variable. Y ocurre frecuentemente que la variable situada a la izquierda forma parte de la expresión de la derecha. En estos casos, y si la expresión es sencilla, todos los operadores aritméticos y los operadores a nivel de bit binarios (exceptuando, por tanto, los operadores de signo, incremento, decremento y complemento) pueden presentar otra forma, en la que se asocia el operador con el operador asignación. Son los llamados operadores de asignación compuestos:

+=	*x += y;*	es lo mismo que decir *x = x + y;*
–=	*x -= y;*	es lo mismo que decir *x = x – y;*
*=	*x *= y;*	es lo mismo que decir *x = x * y;*
/=	*x /= y;*	es lo mismo que decir *x = x / y;*
%=	*x %= y;*	es lo mismo que decir *x = x % y;*
>>=	*x >>= y;*	es lo mismo que decir *x = x >> y;*
<<=	*x <<= y;*	es lo mismo que decir *x = x << y;*
&=	*x &= y;*	es lo mismo que decir *x = x & y;*
|=	*x |= y;*	es lo mismo que decir *x = x | y;*
^=	*x ^= y;*	es lo mismo que decir *x = x ^ y;*

Puede parecer que estos operadores no facilitan la comprensión del código escrito. Quizá una expresión de la forma *F*=n--;* no tenga una presentación muy clara. Pero de hecho estos operadores compuestos se usan frecuentemente y, quien se habitúa a ellos, agradece que se hayan definido para el lenguaje C.

Por cierto, que la expresión del párrafo anterior es equivalente a escribir estas dos líneas de código: *F = F * n;* y *n = n – 1;*.

Operador ***sizeof***.

Ya sabemos el número de bytes que ocupan en memoria todas las variables de tipo de dato primitivo en C: 1 byte las variables tipo ***char***; 2 las de tipo ***short***; 4 las de tipo ***long*** y ***float***; 8 las de tipo ***double***, y 10 las variables ***long double***.

Pero ya se ha dicho que además de estos tipos primitivos, C permite la definición de otros tipos diferentes, combinación de esos primitivos. Y los tamaños de esos tipos definidos pueden ser tan diversos como diversas pueden ser las definiciones de esos nuevos tipos. no es extraño trabajar con tipos cuyas variables ocupan 13 bytes, ó 1045,ó cualquier otro tamaño.

C ofrece un operador que devuelve la cantidad de bytes que ocupa una variable o un tipo de dato concreto. El valor devuelto es tomado como un entero, y puede estar presente en cualquier expresión de C. Es el operador ***sizeof***. Su sintaxis es:

sizeof*(nombre_variable);* ó ***sizeof****(nombre_tipo_de_dato);*

ya que se puede utilizar tanto con una variable concreta como indicándole al operador el nombre del tipo de dato sobre el que queramos conocer su tamaño. No es válido utilizar este operador indicando entre paréntesis el tipo de dato ***void***: esa instrucción daría error en tiempo de compilación.

Con este operador aseguramos la portabilidad, al no depender la aplicación del tamaño del tipo de datos de la máquina que se vaya a usar. Aunque ahora mismo no se ha visto en este texto qué utilidad puede tener en un programa conocer, como dato de cálculo, el número

de bytes que ocupa una variable, la verdad es que con frecuencia ese dato es muy necesario.

Ejemplo: Podemos ver el tamaño de los diferentes tipos de datos primitivos de C. Basta teclear este código en nuestro editor:

```
#include <stdio.h>
main()
{
      printf("int           => %d\n",sizeof(int));
      printf("char          => %d\n",sizeof(char));
      printf("short         => %d\n",sizeof(short));
      printf("long          => %d\n",sizeof(long));
      printf("float         => %d\n",sizeof(float));
      printf("double        => %d\n",sizeof(double));
}
```

(La función *printf* quedará presentada y explicada en el próximo capítulo.)

Expresiones en las que intervienen variables de diferente tipo.

Hay lenguajes de programación que no permiten realizar operaciones con valores de tipos de dato distintos. Se dice que son lenguajes de tipado fuerte, que fuerzan la comprobación de la coherencia de tipos en todas las expresiones, y lo verifican en tiempo de compilación.

El lenguaje C NO es de esos lenguajes, y permite la compilación de un programa con expresiones que mezclan los tipos de datos.

Y aunque en C se pueden crear expresiones en las que intervengan variables y literales de diferente tipo de dato, el procesador trabaja de forma que todas las operaciones que se realizan en la ALU sean con valores del mismo dominio.

Para lograr eso, cuando se mezclan en una expresión diferentes tipos de dato, el compilador convierte todas las variables a un único tipo compatible; y sólo después de haber hecho la conversión se realiza la operación.

Esta conversión se realiza de forma que se no se pueda perder información: en una expresión donde intervienen elementos de diferentes dominios, todos los valores se codifican de acuerdo con el tipo de dato de mayor rango.

La ordenación de rango de los tipos de dato primitivos de C es, de menor a mayor, la siguiente:

char – ***short*** – ***long*** – ***float*** – ***double*** – ***long double***

Así, por ejemplo, si se presenta el siguiente código:

```
char ch = 7;
short sh = 2;
long ln = 100, ln2;
double x = 12.4, y;
y = (ch * ln) / sh – x;
```

La expresión para el cálculo que se almacenará en la variable *y* va cambiando de tipo de dato a medida que se va realizando: en el producto de la variable ***char*** con la variable ***long***, se fuerza el cambio de la variable de tipo ***char***, que se recodificará y así quedará para su uso en la ALU, a tipo ***long***. Esa suma será por tanto un valor ***long***. Luego se realizará el cociente con la variable ***short***, que deberá convertirse en ***long*** para poder dividir al resultado ***long*** antes obtenido. Y, finalmente, el resultado del cociente se debe convertir en un valor ***double***, para poder restarle el valor de la variable x.

El resultado final será pues un valor del tipo de dato ***double***. Y así será almacenado en la posición de memoria de la variable *y* de tipo ***double***.

Si la última instrucción hubiese sido

```
ln2 = (ch * ln) / sh – x;
```

todo hubiera sido como se ha explicado, pero a la hora de almacenar la información en la memoria reservada para la variable ***long*** *ln2*, el resultado final, que venía expresado en formato ***double***, deberá recodificarse para ser guardado como ***long***. Y es que, en el trasiego de la memoria a los registros de la ALU, bien se puede hacer un cambio de

tipo y por tanto un cambio de forma de codificación y, especialmente, de número de bytes empleados para esa codificación. Pero lo que no se puede hacer es que en una posición de memoria como la del ejemplo, que dedica 32 bits a almacenar información, se quiera almacenar un valor de 64 bits, que es lo que ocupan las variables ***double***.

Ante el operador asignación, si la expresión evaluada, situada en la parte derecha de la asignación, es de un tipo de dato diferente al tipo de dato de la variable indicada a la izquierda de la asignación, entonces el valor del lado derecho de la asignación se convierte al tipo de dato del lado izquierdo. En este caso el forzado de tipo de dato puede consistir en llevar un valor a un tipo de dato de menor rango. Y ese cambio corre el riesgo de perder —truncar— la información.

Operador para forzar cambio de tipo.

En el epígrafe anterior se ha visto el cambio o conversión de tipo de dato que se realiza de forma implícita en el procesador cuando encuentra expresiones que contienen diferentes tipos de dato. También existe una forma en que programador puede forzar un cambio de tipo de forma explícita. Este cambio se llama cambio por promoción, o ***casting***. C dispone de un operador para forzar esos cambios.

La sintaxis de este operador unario es la siguiente:

(tipo) nombre_variable;

El operador de promoción de tipo, o *casting*, precede a la variable. Se escribe entre paréntesis el nombre del tipo de dato hacia donde se desea forzar el valor codificado en la variable sobre la que se aplica el operador.

La operación de conversión debe utilizarse con cautelas, de forma que los cambios de tipo sean posibles y compatibles. No se puede realizar cualquier cambio. Especialmente cuando se trabaja con tipos de dato

creados (no primitivos), que pueden tener una complejidad grande. También hay que estar vigilante a los cambios de tipo que fuerzan a una disminución en el rango: por ejemplo, forzar a que una variable ***float*** pase a ser de tipo ***long***. El rango de valores de una variable ***float*** es mucho mayor, y si el valor de la variable es mayor que el valor máximo del dominio de los enteros de 4 bytes, entonces el resultado del operador forzar tipo será imprevisible. Y tendremos entonces una operación que no ofrece problema alguno en tiempo de compilación, pero que bien puede llevar a resultados equivocados en tiempo de ejecución.

de tipo…	al tipo…	Posibles pérdidas
char	***signed char***	Si el valor inicial es mayor de 127, entonces el nuevo valor será negativo.
short	***char***	Se pierden los 8 bits más significativos.
long int	***char***	Se pierden los 24 bits más significativos.
long int	***short***	Se pierden los 16 bits más significativos.
float	***int***	Se pierde la parte fraccional y más información.
double	***float***	Se pierde precisión. El resultado se presenta redondeado.
long double	***double***	Se pierde precisión. El resultado se presenta redondeado.

Tabla 2.5.: Pérdidas de información en los cambios de tipo con disminución de rango.

No se pueden realizar conversiones del tipo ***void*** a cualquier otro tipo, pero sí de cualquier otro tipo al tipo ***void***. Eso se entenderá mejor más adelante.

En la tabla 2.5. se muestran las posibles pérdidas de información que se pueden producir en conversiones forzadas de tipo de dato. Esas pérdidas

se darán tanto si la conversión de tipo de dato viene forzada por el operador conversor de tipo, como si es debido a exigencias del operador asignación. Evidentemente, salvo motivos intencionados que rara vez se han de dar, este tipo de conversiones hay que evitarlas. Los compiladores no interrumpen el trabajo de compilación cuando descubren, en el código, alguna conversión de esta índole, pero sí advierten de aquellas expresiones donde puede haber un cambio forzado de tipo de dato que conduzca a la pérdida de información.

Propiedades de los operadores.

Al evaluar una expresión formada por diferentes variables y literales, y por diversos operadores, hay que lograr expresar realmente lo que se desea operar. Por ejemplo, la expresión *a* + *b* * *c*… ¿Se evalúa como el producto de la suma de *a* y *b*, con *c*; o se evalúa como la suma del producto de *b* con *c*, y *a*?

Para definir unas reglas que permitan una interpretación única e inequívoca de cualquier expresión, se han definido tres propiedades en los operadores:

1. su **posición**. Establece dónde se coloca el operador con respecto a sus operandos. Un operador se llamará **infijo** si viene a colocarse entre sus operandos; y se llamará **prefijo** se el operador precede al operando.
2. su **precedencia**. Establece el orden en que se ejecutan los distintos operadores implicados en una expresión. Existe un orden de precedencia perfectamente definido, de forma que en ningún caso una expresión puede tener diferentes interpretaciones. Y el compilador de C siempre entenderá las expresiones de acuerdo con su orden de precedencia establecido.
3. su **asociatividad**. Esta propiedad resuelve la ambigüedad en la elección de operadores que tengan definida la misma precedencia.

En la práctica habitual de un programador, se acude a dos reglas para lograr escribir expresiones que resulten correctamente evaluadas:

1. Hacer uso de **paréntesis**. De hecho los paréntesis son un operador más, que además son los primeros en el orden de ejecución. De acuerdo con esta regla, la expresión antes recogida podría escribirse (a + b) * c; ó a + (b * c); en función de cuál de las dos se desea. Ahora, con los paréntesis, estas expresiones no llevan a equívoco alguno.
2. Conocer y aplicar las **reglas de precedencia** y de asociación por izquierda y derecha. Este orden podrá ser siempre alterado mediante el uso de paréntesis. Según esta regla, la expresión antes recogida se interpreta como *a + (b * c)*.

   Se considera buena práctica de programación conocer esas reglas de precedencia y no hacer uso abusivo de los paréntesis. De todas formas, cuando se duda sobre cómo se evaluará una expresión, lo habitual es echar mano de los paréntesis. A veces una expresión adquiere mayor claridad si se recurre al uso de los paréntesis.

Las reglas de precedencia son las que se recogen el la tabla 2.6. Cuanto más alto en la tabla esté el operador, más alta es su precedencia, y antes se evalúa ese operador que cualquier otro que esté más abajo en la tabla. Y para aquellos operadores que estén en la misma fila, es decir, que tengan el mismo grado de precedencia, el orden de evaluación, en el caso en que ambos operadores intervengan en una expresión, viene definido por la asociatividad: de derecha a izquierda o de izquierda a derecha.

Existen 16 categorías de precedencia, y todos los operadores colocados en la misma categoría tienen igual precedencia que cualquiera otro de la misma categoría. Algunas de esas categorías tan solo tienen un operador.

Cuando un operador viene duplicado en la tabla, la primera ocurrencia es como operador unario, la segunda como operador binario.

Cada categoría tiene su regla de asociatividad: de derecha a izquierda (anotada como D – I), o de izquierda a derecha (anotada como I – D).

() [] -> .	I – D
! ~ ++ -- + - * &	D – I
.* ->*	I – D
* / %	I – D
+ -	I – D
<< >>	I – D
> >= < <=	I – D
== !=	I – D
&	I – D
^	I – D
|	I – D
&&	I – D
||	D – I
?:	I – D
= += -= *= /= %= &= |= <<= >>=	D – I
,	I – D

Tabla 2.6.: Precedencia y Asociatividad de los operadores.

Por ejemplo, la expresión

```
a * x + b * y - c / z:
```

se evalúa en el siguiente orden: primero los productos y el cociente, y ya luego la suma y la resta. Todos estos operadores están en categorías con asociatividad de izquierda a derecha, por lo tanto, primero se efectúa el producto más a la izquierda y luego el segundo, más al centro

de la expresión. Después se efectúa el cociente; luego la suma y finalmente la resta.

Todos los operadores de la tabla 2.6. que faltan por presentar en el manual se emplean para vectores y cadenas y para operatoria de punteros. Más adelante se conocerán todos ellos.

Valores fuera de rango en una variable.

Ya hemos dicho repetidamente que una variable es un espacio de memoria, de un tamaño concreto en donde la información se codifica de una manera determinada por el tipo de dato que se vaya a almacenar en esa variable.

Espacio de memoria limitado.

Es importante conocer los límites de nuestras variables. Esos límites ya venían presentados en la tabla 2.1.

Cuando en un programa se pretende asignar a una variable un valor que no pertenece al dominio, el resultado es habitualmente extraño. Se suele decir que es imprevisible, pero la verdad es que la electrónica del procesador actúa de la forma para la que está diseñada, y no son valores aleatorios los que se alcanzan entonces, aunque sí, muchas veces, valores no deseados, o valores erróneos.

Se muestra ahora el comportamiento de las variables ante un desbordamiento (que así se le llama) de la memoria. Si la variable es entera, ante un desbordamiento de memoria el procesador trabaja de la misma forma que lo hace, por ejemplo, el cuenta kilómetros de un vehículo. Si en un cuenta kilómetros de cinco dígitos, está marcado el valor 99.998 kilómetros, al recorrer cinco kilómetros más, el valor que aparece en pantalla será 00.003. Se suele decir que se le ha dado la vuelta al marcador. Y algo similar ocurre con las variables enteras. En la

tabla 2.7. se muestran los valores de diferentes operaciones con desbordamiento.

signed short	32767 + 1 da el valor –32768
unsigned short	65535 + 1 da el valor 0
signed long	2147483647 +1 da el valor -2147483648
unsigned long	4294967295 + 1 da el valor 0

Tabla 2.7.: valores de desbordamiento en las variables de tipo entero.

Si el desbordamiento se realiza por asignación directa, es decir, asignando a una variable un literal que sobrepasa el rango de su dominio, o asignándole el valor de una variable de rango superior, entonces se almacena el valor truncado. Por ejemplo, si a una variable ***unsigned short*** se le asigna un valor que requiere 25 dígitos binarios, únicamente se quedan almacenados los 16 menos significativos. A eso hay que añadirle que, si la variable es ***signed short***, al tomar los 16 dígitos menos significativos, interpretará el más significativo de ellos como el bit de signo, y según sea ese bit, interpretará toda la información codificada como entero negativo en complemento a la base, o como entero positivo.

Hay situaciones y problemas donde jugar con las reglas de desbordamiento de enteros ofrece soluciones muy rápidas y buenas. Pero, evidentemente, en esos casos hay que saber lo que se hace.

Si el desbordamiento se realiza con variables en coma flotante el resultado es mucho más complejo de prever, y no resulta sencillo pensar en una situación en la que ese desbordamiento pueda ser deseado.

Si el desbordamiento es por asignación, la variable desbordada almacenará un valor que nada tendrá que ver con el original. Si el desbordamiento tiene lugar por realizar operaciones en un tipo de dato de coma flotante y en las que el valor final es demasiado grande para

ese tipo de dato, entonces el resultado es completamente imprevisible, y posiblemente se produzca una interrupción en la ejecución del programa. Ese desbordamiento se considera, sin más, error de programación.

Constantes. Directiva ***#define***.

Cuando se desea crear una variable a la que se asigna un valor inicial que no debe modificarse, se la precede, en su declaración, de la palabra clave de C ***const***.

const tipo var_1 = val_1[,var_2 = val_2, …, var_N = val_N];

Se declara con la palabra reservada ***const***. Pueden definirse constantes de cualquiera de los tipos de datos simples.

```
const float DOS_PI = 6.28;
```

Como no se puede modificar su valor, la constante no puede ser un *Lvalue* (excepto en el momento de su inicialización). Si se intenta modificar el valor de la constante mediante el operador asignación el compilador dará un error y no se creará el programa.

Otro modo de definir constantes es mediante la directiva de preprocesador o de compilación **#define**. Ya se ha visto en el primer capítulo otra directiva que se emplea para indicar al compilador de qué bibliotecas se toman funciones ya creadas y compiladas: la directiva **#include**.

```
#define DOS_PI 6.28
```

Como se ve, la directiva **#define** no termina en punto y coma. Eso es debido a que las directivas de compilación no son instrucciones de C, sino órdenes que se dirigen al compilador. La directiva **#define** se ejecuta previamente a la compilación, y sustituye, en todas las líneas de código posteriores a la aparición de la directiva, cada aparición de la

primera cadena de caracteres por la segunda cadena: en el ejemplo antes presentado, la cadena DOS_PI por el valor 6.28.

Ya se verá cómo esta directiva pude resultar muy útil en ocasiones.

Intercambio de valores de dos variables

Una operación bastante habitual en un programa es el **intercambio de valores** entre dos variables. Supongamos el siguiente ejemplo:

<variable1, int, R, 10> y <variable2, int, R, 20>

Si queremos que variable1 almacene el valor de variable2 y variable2 el de variable1, es necesario acudir a una variable auxiliar. El proceso es así:

```
auxiliar  = variable1;
variable1 = variable2;
variable2 = auxiliar;
```

Porque no podemos copiar el valor de variable2 en variable1 sin perder con esta asignación el valor de variable1 que queríamos guardar en variable2.

Con el operador or exclusivo se puede hacer intercambio de valores sin acudir, para ello, a una variable auxiliar. El procedimiento es el siguiente:

```
variable1 = variable1 ^ variable2;
variable2 = variable1 ^ variable2;
variable1 = variable1 ^ variable2;
```

que, con operadores compuestos queda de la siguiente manera:

```
variable1 ^= variable2;
variable2 ^= variable1;
variable1 ^= variable2;
```

Veamos un ejemplo para comprobar que realmente realiza el intercambio:

```
short int variable1 = 3579;
short int variable2 = 2468;
```

en base binaria el valor de las dos variables es:

```
                variable1  0 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1
                variable2  0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 0
   variable1 ^= variable2  0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 1
   variable2 ^= variable1  0 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1
   variable1 ^= variable2  0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 0
```

Al final del proceso, el valor de *variable1* es el que inicialmente tenía *variable2*, y al revés. Basta comparar valores. Para verificar que las operaciones están correctas (que lo están) hay que tener en cuenta que el proceso va cambiando los valores de *variable1* y de *variable2*, y esos cambios hay que tenerlos en cuenta en las siguientes operaciones or exclusivo a nivel de bit.

Ayudas On line.

Muchos editores y compiladores de C cuentan con ayudas en línea abundante. Todo lo referido en este capítulo puede encontrarse en ellas. Es buena práctica de programación saber manejarse por esas ayudas, que llegan a ser muy voluminosas y que gozan de buenos índices para lograr encontrar el auxilio necesario en cada momento.

Recapitulación.

Después de estudiar este capítulo, ya sabemos crear y operar con nuestras variables. También conocemos muchos de los operadores definidos en C. Con todo esto podemos realizar ya muchos programas sencillos.

Si conocemos el rango o dominio de cada tipo de dato, sabemos también de qué tipo conviene que sea cada variable que necesitemos. Y estaremos vigilantes en las operaciones que se realizan con esas variables, para no sobrepasar ese dominio e incurrir en un overflow.

También hemos visto las reglas para combinar, en una expresión, variables y valores de diferente tipo de dato. Es importante conocer bien

todas las reglas que gobiernan estas combinaciones porque con frecuencia, si se trabaja sin tiento, se llegan a resultados erróneos.

Con los ejercicios que se proponen a continuación se pueden practicar y poner a prueba nuestros conceptos adquiridos.

Ejemplos y ejercicios propuestos.

1. ***Escribir un programa que realice las operaciones de suma, resta, producto, cociente y módulo de dos enteros introducidos por teclado.***

```
#include <stdio.h>
void main(void)
{
      signed long a, b;
      signed long sum, res, pro, coc, mod;

      printf("Introduzca el valor del 1er. operando ... ");
      scanf("%ld",&a);
      printf("Introduzca el valor del 2do. operando ... ");
      scanf("%ld",&b);

// Cálculos
      sum = a + b;
      res = a - b;
      pro = a * b;
      coc = a / b;
      mod = a % b;

// Mostrar resultados por pantalla.
      printf("La suma es igual a %ld\n", sum);
      printf("La resta es igual a %ld\n", res);
      printf("El producto es igual a %ld\n", pro);
      printf("El cociente es igual a %ld\n", coc);
      printf("El resto es igual a %ld\n", mod);
}
```

Observación: cuando se realiza una operación de cociente o de resto es muy recomendable antes verificar que, efectivamente, el divisor no es igual a cero. Aún no sabemos hacerlo, así que queda el programa un poco cojo. Al ejecutarlo será importante que el usuario no introduzca un valor para la variable *b* igual a cero.

2. ***Repetir el mismo programa para números de coma flotante.***

```
#include <stdio.h>
void main(void)
{
      float a, b;
      float sum, res, pro, coc;

      printf("Introduzca el valor del 1er. operando ... ");
      scanf("%f",&a);
      printf("Introduzca el valor del 2do. operando ... ");
      scanf("%f",&b);

// Cálculos
      sum = a + b;
      res = a - b;
      pro = a * b;
      coc = a / b;
// mod = a % b; : esta operación no está permitida
// Mostrar resultados por pantalla.
      printf("La suma es igual a %f\n", sum);
      printf("La resta es igual a %f\n", res);
      printf("El producto es igual a %f\n", pro);
      printf("El cociente es igual a %f\n", coc);
}
```

En este caso se ha tenido que omitir la operación módulo, que no está definida para valores del dominio de los números de coma flotante. Al igual que en el ejemplo anterior, se debería verificar (aún no se han presentado las herramientas que lo permiten), antes de realizar el cociente, que el divisor era diferente de cero.

3. ***Escriba un programa que resuelva una ecuación de primer grado ($a \cdot x + b = 0$). El programa solicita los valores de los parámetros a y b y mostrará el valor resultante de la variable x.***

```
#include <stdio.h>
void main(void)
{
      float a, b;
      printf("Introduzca el valor del parámetro a ... ");
      scanf("%f",&a);
      printf("Introduzca el valor del parámetro b ... ");
      scanf("%f",&b);
      printf("x = %f",-b / a);
}
```

De nuevo sería más correcto el programa si, antes de realizar el cociente, se verificase que la variable *a* es distinta de cero.

4. ***Escriba un programa que solicite un entero y muestre por pantalla su valor al cuadrado y su valor al cubo.***

```
#include <stdio.h>
void main(void)
{
      short x;
      long cuadrado, cubo;
      printf("Introduzca un valor ... ");
      scanf("%hi",&x);
      cuadrado = (long)x * x;
      cubo = cuadrado * x;
      printf("El cuadrado de %hd es %li\n",x, cuadrado);
      printf("El cubo de %hd es %li\n",x, cubo);
}
```

Es importante crear una presentación cómoda para el usuario. No tendría sentido comenzar el programa por la función *scanf*, porque en ese caso el programa comenzaría esperando un dato del usuario, sin aviso previo que le indicase qué es lo que debe hacer. En el siguiente

tema se presenta con detalle las dos funciones de entrada y salida por consola.

La variable *x* es **short**. Al calcular el valor de la variable *cuadrado* forzamos el tipo de dato para que el valor calculado sea ***long*** y no se pierda información en la operación. En el cálculo del valor de la variable *cubo* no es preciso hacer esa conversión, porque ya la variable *cuadrado* es de tipo **long**. En esta última operación no queda garantizado que no se llegue a un desbordamiento: cualquier valor de *x* mayor de 1290 tiene un cubo no codificable con 32 bits. Se puede probar qué ocurre introduciendo valores mayores que éste indicado.

5. ***Escriba un programa que solicite los valores de la base y de la altura de un triángulo y que imprima por pantalla el valor de la superficie.***

```
#include <stdio.h>
void main(void)
{
      double b, h, S;
      printf("Introduzca la base ... ");
      scanf("%lf",&b);
      printf("Introduzca la altura ... ");
      scanf("%lf",&h);
      S = b * h / 2;
      printf("La superficie del triangulo de ");
      printf("base %.2lf y altura %.2lf ",b,h);
      printf("es %.2lf",S);
}
```

Las variables se han tomado ***double***. Así no se pierde información en la operación cociente. Puede probar a declarar las variables como de tipo ***short***, modificando también algunos parámetros de las funciones de entrada y salida por consola:

```
void main(void)
{
      short b, h, S;
```

```
      printf("Introduzca la base ... ");
      scanf("%hd",&b);
      printf("Introduzca la altura ... ");
      scanf("%hd",&h);
      S = b * h / 2;
      printf("La superficie del triangulo de ");
      printf("base %hd y altura %hd ",b,h);
      printf("es %hd",S);
}
```

Si al ejecutar el programa el usuario introduce los valores 5 para la base y 3 para la altura, el valor de la superficie que mostrará el programa al ejecutarse será ahora de 7, y no de 7,5.

6. ***Escriba un programa que solicite el valor del radio y muestre la longitud de su circunferencia y la superficie del círculo inscrito en ella.***

```
#include <stdio.h>
#define PI 3.14159
void main(void)
{
      signed short int r;
      double l, S;
      const double pi = 3.14159;
      printf("Indique el valor del radio ... ");
      scanf("%hd",&r);
      printf("La longitud de la circunferencia");
      printf(" cuyo radio es %hd",r);
      l = 2 * pi * r;
      printf(" es %lf. \n",l);
      printf("La superficie de la circunferencia");
      printf(" cuyo radio es %hd",r);
      S = PI * r * r;
      printf(" es %lf. \n",S);
}
```

En este ejemplo hemos mezclado tipos de dato. El radio lo tomamos como entero. Luego, en el cálculo de la longitud *l*, como la expresión tiene el valor de la constante *pi*, que es ***double***, se produce una conversión implícita de tipo de dato, y el resultado final es ***double***.

Para el cálculo de la superficie, en lugar de emplear la constante *pi* se ha tomado un identificador PI definido mediante la directiva ***#define***. El valor de PI también tiene forma de coma flotante, y el resultado será por tanto de este tipo.

A diferencia de los ejemplos anteriores, en éste hemos guardado los resultados obtenidos en variables.

7. ***Escriba un programa que solicite el valor de la temperatura en grados Fahrenheit y muestre por pantalla el equivalente en grados Celsius. La ecuación que define esta trasformación es:***

Celsius = (5 / 9) · (Fahrenheit – 32).

```
#include <stdio.h>
void main(void)
{
      double fahr, cels;
      printf("Temperatura en grados Fahrenheit ... ");
      scanf("%lf",&fahr);
      cels = (5 / 9) * (fahr - 32);
      printf("La temperatura en grados Celsius ");
      printf("resulta ... %lf.",cels);
}
```

Tal y como está escrito el código parece que todo ha de ir bien. Si ensayamos el programa con la entrada en grado Fahrenheit igual a 32, entonces la temperatura en Celsius resulta 0, que es correcto puesto que esas son las temperaturas, en las dos tablas, en las que se derrite el hielo.

Pero, ¿y si probamos con otra entrada?... ¡También da 0! ¿Por qué?

Pues porque *(5 / 9)* es una operación cociente entre dos enteros, cuyo resultado es un entero: el truncado, es decir, el mayor entero menor que el resultado de la operación; es decir, 0.

¿Cómo se debería escribir la operación?

```
cels = (5 / 9.0) * (fahr - 32);
cels = (5.0 / 9) * (fahr - 32);
cels = ((double)5 / 9) * (fahr - 32);
cels = (5 / (double)9) * (fahr - 32);
cels = 5 * (fahr - 32) / 9;
```

Hay muchas formas: lo importante es saber en todo momento qué operación realizará la sentencia que escribimos. Hay que saber escribir expresiones que relacionan valores de distinto tipo para no perder nunca información por falta de rango al elegir mal uno de los tipos de alguno de los literales o variables.

Es importante, en este tema de los lenguajes, no perder de vista la realidad física que subyace en toda la programación. Un lenguaje de programación no es una herramienta que relacione variables en el sentido matemático de la palabra. Relaciona elementos físicos, llamados posiciones de memoria. Y es importante, al programar, razonar de acuerdo con la realidad física con la que trabajamos.

8. Rotaciones de bits dentro de un entero.

Una operación que tiene uso en algunas aplicaciones de tipo criptográficos es la de rotación de un entero. Rotar un número x posiciones hacia la derecha consiste en desplazar todos sus bits hacia la derecha esas x posiciones, pero sin perder los x bits menos significativos, que pasarán a situarse en la parte más significativa del número. Por ejemplo, el número a = 1101 0101 0011 1110 rotado 4 bits a la derecha queda 1110 1101 0101 0011, donde los cuatro bits menos significativos (1110) se han venido a posicionar en la izquierda del número.

La rotación hacia la izquierda es análogamente igual, donde ahora los x bits más significativos del número pasan a la derecha del número. El número a rotado 5 bits a la izquierda quedaría: 1010 0111 1101 1010.

Qué órdenes deberíamos dar para lograr la rotación de números a la izquierda:

```
unsigned short int a, b, despl;
a = 0xABCD;
despl = 5;
b = ((a << despl) | (a >> (8 * sizeof(short) - despl)));
```

Veamos cómo funciona este código:

```
                          a:  1010 1011 1100 1101
                      despl:  5
                 a << despl:  0111 1001 1010 0000     ←
     8 * sizeof(short) - despl:  8 * 2 - 5 = 11
                  a >> 11:  0000 0000 0001 0101     ←
                          b:  0111 1001 1011 0101
```

Y para lograr la rotación de números a la derecha:

```
unsigned short int a, b, despl;
a = 0xABCD;
despl = 5;
b = ((a >> despl) | (a << (8 * sizeof(short) - despl)));
```

9. ***Indicar el valor que toman las variables en las siguientes asignaciones.***

```
int a = 10, b = 5, c, d;
float x = 10.0 ,y = 5.0, z, t, v;
c = a/b;
d = x/y;
z = a/b;
t = (1 / 2) * x;
v = (1.0 / 2) * x;
```

10. ***Escribir el siguiente programa y justificar la salida que ofrece por pantalla.***

```
#include <stdio.h>
void main(void)
{
      char a = 127;
      a++;
      printf("%hd", a);
}
```

La salida que se obtiene con este código es… -128. Intente justificar por qué ocurre. No se preocupe si aún no conoce el funcionamiento de la función *printf()*. Verdaderamente la variable *a* ahora vale -128. ¿Por qué?

Si el código que se ejecuta es el siguiente, explique la salida obtenida. ¿Sabría adivinar qué va a salir por pantalla antes de ejecutar el programa?

```
#include <stdio.h>
void main(void)
{
      short sh = 0x7FFF;
      long ln = 0x7FFFFFFF;
      printf("\nEl valor inicial de sh es ... %hd", sh);
      printf("\nEl valor inicial de ln es ... %ld", ln);
      sh++;
      ln++;
      printf("\nEl valor final de sh es ... %hd", sh);
      printf("\nEl valor final de ln es ... %ld", ln);
}
```

11. ***Escribir un programa que indique cuántos bits y bytes ocupa una variable long.***

```
#include <stdio.h>
void main(void)
{
```

```
      short bits, bytes;
      bytes = sizeof(long);
      bits = 8 * bytes;
      printf("BITS = %hd - BYTES = %hd.", bits, bytes);
}
```

12. ***Escribir el siguiente programa y explique las salidas que ofrece por pantalla.***

```
#include <stdio.h>

void main(void)
{
      char a = 'X', b = 'Y';
      printf("\nvalor (caracter) de a: %c", a);
      printf("\nvalor (caracter) de b: %c", b);

      printf("\nvalor de a (en base 10): %hd", a);
      printf("\nvalor de b (en base 10): %hd", b);

      printf("\nvalor (caracter) de a + b: %c", a + b);
      printf("\nvalor (en base 10) de a + b: %hd", a + b);

      printf("\nvalor (caracter) de a+b + 5: %c", a + b + 5);
      printf("\nvalor (en base 10) de a+b+5: %hd", a+b + 5);
}
```

La salida que se obtiene por pantalla es:

```
valor (caracter) de a: X
valor (caracter) de b: Y
valor de a (en base 10): 88
valor de b (en base 10): 89
valor (caracter) de a + b: ▒
valor (en base 10) de a + b: 177
valor (caracter) de a + b + 5: Â
valor (en base 10) de a + b + 5: 182
```

13. ***Escriba un programa que calcule la superficie y el volumen de una esfera cuyo radio es introducido por teclado.***

$S = 4 \cdot \pi \cdot r^2 \quad V = 4/3 \cdot \pi \cdot r^3$

(Recuerde que, en C, la expresión 4/3 no significa el valor racional resultante del cociente, sino un nuevo valor entero, que es idénticamente igual a 1.)

14. ***El número áureo (α) es aquel que verifica la propiedad de que al elevarlo al cuadrado se obtiene el mismo valor que al sumarle 1. Haga un programa que calcule y muestre por pantalla el número áureo.***

```
#include <stdio.h>
#include <math.h>
void main(void)
{
      double aureo;
      printf("Número AUREO: tal que x + 1 = x * x.\n");
// Cálculo del número Aureo
// x^2 = x + 1 ==> x^2 - x - 1 = 0 ==> x = (1 + sqrt(5)) / 2.
      aureo = (1 + sqrt(5)) / 2;
      printf("El número AUREO es .. %lf\n",aureo);
      printf("aureo + 1 ........... %lf\n",aureo + 1);
      printf("aureo * aureo ....... %lf\n", aureo * aureo);
}
```

La función *sqrt()* está definida en la biblioteca ***math.h***. Calcula el valor de la raíz cuadrada de un número. Espera como parámetro una variable de tipo double, y devuelve el valor en este formato o tipo de dato.

El ejercicio es muy sencillo. La única complicación (si se le puede llamar complicación a esta trivialidad) es saber cómo se calcula el número aureo a partir de la definición aportada. Muchas veces el problema de la programación no está en el lenguaje, sino en saber expresar una solución viable de nuestro problema.

15. ***Escriba un programa que calcule a qué distancia caerá un proyectil lanzado desde un cañón. El programa recibe desde teclado el ángulo inicial de salida del proyectil (α) y su velocidad inicial (V_0).***

Tenga en cuenta las siguientes ecuaciones que definen el comportamiento del sistema descrito:

$V_x = V_0 \cdot \cos(\alpha)$ $\qquad$ $V_y = V_0 \cdot sen(\alpha) - g \cdot t$

$x = V_0 \cdot t \cdot \cos(\alpha)$ $\qquad$ $y = V_0 \cdot t \cdot sen(\alpha) - \frac{1}{2} \cdot g \cdot t^2$