Expresiones y sentencias

Expresión
Construcción (combinación de tokens)
que se evalúa para devolver un valor.

Sentencia
Representación de una acción o una secuencia de acciones.
En C, todas las sentencias terminan con un punto y coma [;].

Construcción de expresiones

- Literales y variables son expresiones primarias:

```
1.7     // Literal real de tipo double
sum     // Variable
```

  - Los literales se evalúan a sí mismos.
  - Las variables se evalúan a su valor.

- Los operadores nos permiten combinar expresiones primarias y otras expresiones formadas con operadores:

```
1 + 2 + 3*1.2 + (4 +8)/3.0
```

  Los operadores se caracterizan por:

  - El número de operandos (unarios, binarios o ternarios).
  - El tipo de sus operandos (p.ej. numéricos o booleanos).
  - El tipo del valor que generan como resultado.

Número de operandos

- Operadores unarios

Operador	Descripción
-	Cambio de signo
!	Operador NOT
~	Complemento a 1

- Operadores binarios

Operadores	Descripción
+ - * / %	Operadores aritméticos
== != < > <= >=	Operadores relacionales
&& || ^	Operadores booleanos
& | ^ << >>	Operadores a nivel de bits

Tipo de los operandos

Operadores	Descripción	Operandos
+ - * / %	Operadores aritméticos	Números (% sólo enteros)
== !=	Operadores de igualdad	Cualesquiera
< > <= >=	Operadores de comparación	Números o caracteres
¡ && || ^	Operadores booleanos	"Booleanos"
~ & | ^ << >>	Operadores a nivel de bits	Enteros

Tipo del resultado

Operadores	Descripción	Resultado
+ - * / %	Operadores aritméticos	Número*
== != < > <= >=	Operadores relacionales	"Booleano"
¡ && || ^	Operadores booleanos	
~ & | ^ << >>	Operadores a nivel de bits	Entero

Sentencias de asignación

Las sentencias de asignación constituyen el ingrediente básico en la construcción de programas con lenguajes imperativos.

Sintaxis:

```
<variable> = <expresión>;
```

Al ejecutar una sentencia de asignación:

1. Se evalúa la expresión que aparece a la derecha del operador de asignación (=).
2. El valor que se obtiene como resultado de evaluar la expresión se almacena en la variable que aparece a la izquierda del operador de asignación (=).

Restricción:

El tipo del valor que se obtiene como resultado de evaluar la expresión ha de ser compatible con el tipo de la variable.

Ejemplos

```
x = x + 1;
int miVariable = 20;          // Declaración con inicialización
otraVariable = miVariable;    // Sentencia de asignación
```

NOTA IMPORTANTE:
Una sentencia de asignación no es una igualdad matemática.

Efectos colaterales

Al evaluar una expresión, algunos operadores provocan efectos colaterales (cambios en el estado del programa; es decir, cambios en el valor de alguna de las variables del programa).

Operadores de incremento (++) *y decremento* (--)

El operador ++ incrementa el valor de una variable.
El operador -- decrementa el valor de una variable.

El resultado obtenido depende de la posición relativa del operando:

Operador	Descripción
`x++`	**Post-incremento** Evalúa primero y después incrementa
`++x`	**Pre-incremento** Primero incrementa y luego evalúa
`x--`	**Post-decremento** Evalúa primero y luego decrementa
`--x`	**Pre-decremento** Primero decrementa y luego evalúa

Ejemplo	Equivalencia	Resultado
`i=0; i++;`	`i=0; i=i+1;`	**`i=1;`**
`i=1; j=++i;`	`i=1; i=i+1; j=i;`	**`j=2;`**
`i=1; j=i++;`	`i=1; j=i; i=i+1;`	**`j=1;`**
`i=3; n=2*(++i);`	`i=3; i=i+1; n=2*i;`	**`n=8;`**
`i=3; n=2*(i++);`	`i=3; n=2*i; i=i+1;`	**`n=6;`**
`i=1; k=++i+i;`	`i=1; i=i+1; k=i+i;`	**`k=4;`**

El uso de operadores de incremento y decremento reduce el tamaño de las expresiones pero las hace más difíciles de interpretar. Es mejor evitar su uso en expresiones que modifican múltiples variables o usan varias veces una misma variable.

Operadores combinados de asignación (`op=`*)*

C define 10 operadores que combinan el operador de asignación con otros operadores (aritméticos y a nivel de bits):

Operador	Ejemplo	Equivalencia
+=	`i += 1;`	`i++;`
-=	`f -= 4.0;`	`f = f – 4.0;`
***=**	`n *= 2;`	`n = n * 2;`
/=	`n /= 2;`	`n = n / 2;`
%=	`n %= 2;`	`n = n % 2;`
&=	`k &= 0x01;`	`k = k & 0x01;`
|=	`k |= 0x02;`	`k = k & 0x02;`
^=	`k ^= 0x04;`	`k = k & 0x04;`
<<=	`x <<= 1;`	`x = x << 1;`
>>=	`x >>= 2;`	`x = x >> 2;`

La forma general de los operadores combinados de asignación es

```
variable op= expresión;
```

que pasa a ser

```
variable = variable op (expresión);
```

OJO: Las expresiones con y sin operadores combinados de asignación no son siempre equivalentes.

Ejemplo

`v[i++]+= 2;` y `v[i++]=v[i++]+2;` no son equivalentes.

Conversión de tipos

En determinadas ocasiones, nos interesa convertir el tipo de un dato en otro tipo para poder operar con él.

Ejemplo

Convertir un número entero en un número real
para poder realizar divisiones en coma flotante

C permite realizar conversiones entre datos de tipo numérico (enteros y reales), así como trabajar con caracteres como si fuesen números enteros:

- La conversión de un tipo con menos bits a un tipo con más bits es automática (vg. de `short` a `long`, de `float` a `double`), ya que el tipo mayor puede almacenar cualquier valor representable con el tipo menor (además de valores que "no caben" en el tipo menor).

- La conversión de un tipo con más bits a un tipo con menos bits se realizar de forma explícita con "castings". Como se pueden perder datos en la conversión, un buen compilador de C nos avisa de posibles conversiones erróneas (*warnings*)

```
int i;
char b;

i = 13;      // No se realiza conversión alguna

b = 13;      // Se permite porque 13 está dentro
             // del rango permitido de valores

b = i;       // Podría ser un error

b = (char) i;         // Fuerza la conversión

i = (int) 14.456;     // Almacena 14 en i

i = (int) 14.656;     // Sigue almacenando 14
```

Castings

Para realizar una conversión explícita de tipo (un "casting") basta con poner el nombre del tipo deseado entre paréntesis antes del valor que se desea convertir:

```
char   c;
short  x;
long   k;
double d;
```

Sin conversión de tipo:

```
c = 'A';
x = 100;
k = 100L;
d = 3.0;
```

Conversiones implícitas de tipo (por asignación):

```
x = c;          // char → short
k = 100;        // short → long
k = x;          // short → long
d = 3;          // short → double
```

Conversiones implícitas de tipos (por promoción aritmética)

```
c+1             // char → short
x / 2.0f        // short → float
```

Posibles errores de conversión (no siempre detectados):

```
x = k;          // long → short
x = 3.0;        // double → short
x = 5 / 2.0f;   // float → short
```

Conversiones explícitas de tipo (castings):

```
x = (short) k;
x = (short) 3.0;
x = (short) (5 / 2.0f);
```

Algunas conversiones de interés

De números en coma flotante a números enteros

Al convertir de número en coma flotante a entero,
el número se trunca (redondeo a cero).

En la biblioteca `math.h` existen funciones
que nos permiten realizar el redondeo de otras formas:

```
floor(x), ceil(x)
```

El redondeo al que estamos acostumbrados
tendremos que implementarlo nosotros:

```
int i = x+0.5;
```

Conversión entre caracteres y números enteros

Como cada carácter tiene asociado un código ASCII,
los caracteres pueden interpretarse como números enteros

```
int i;
char c;
```

Ejemplo	Equivalencia	Resultado
`i = 'a';`	`i = (int) 'a';`	**`i = 97`**
`c = 97;`	`c = (char) 97;`	**`c = 'a'`**
`c = 'a'+1;`	`c = (char) ((int)'a'+1);`	**`c = 'b'`**
`c = i+2;`	`c = (char) (i+2);`	**`c = 'c'`**

Resumen de conversión de tipos

Conversiones implícitas

long double > double > float > unsigned long > long > unsigned short > short > char

Conversiones explícitas (castings)

`(tipo) expresión`

Evaluación de expresiones

- La **precedencia** de los operadores determina el orden de evaluación de una expresión (el orden en que se realizan las operaciones):

  3*4+2 es equivalente a (3*4)+2
  porque el operador * es de mayor precedencia que el operador +

- Cuando en una expresión aparecen dos operadores con el mismo nivel de precedencia, la **asociatividad** de los operadores determina el orden de evaluación.

  a – b + c – d es equivalente a ((a – b) + c) – d
  porque *los operadores aritméticos*
  son asociativos de izquierda a derecha

  a = b += c = 5 es equivalente a a = (b += (c = 5))
  porque *los operadores de asignación*
  son asociativos de derecha a izquierda

- La precedencia y la asociatividad determinan el orden de los operadores, pero no especifican el orden en que se evalúan los **operandos** de un operador binario (un operador con dos operandos):

  En C, los operandos se evalúan de izquierda a derecha:
  el operando de la izquierda se evalúa primero.

  Si los operandos no tienen efectos colaterales (esto es, no cambian el valor de una variable), el orden de evaluación de los operandos es irrelevante.

NOTA: **Siempre es recomendable el uso de paréntesis.**
Los paréntesis nos permiten especificar el orden de evaluación de una expresión, además de hacer su interpretación más fácil.