Fundamentos de Programación – Programación Estructurada en C:
4.- EXPRESIONES Y OPERADORES

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

4.- EXPRESIONES Y OPERADORES

4.- EXPRESIONES Y OPERADORES

4.1.- INTRODUCCIÓN

Siguiendo con la sintaxis de C, vamos a ver los distintos operadores que se proporcionan y las expresiones que se pueden formar con ellas.

Las variables y constantes estudiadas, se pueden combinar de la forma adecuada, creando así *expresiones*. Mediante esas expresiones, se pueden calcular el resultado de cualquier tipo de operación. Para indicar las diferentes operaciones que se han de evaluar en una expresión, se usan los *operadores*.

En la sintaxis de todos los operadores de C, aparecen *operandos*, sobre los que hay que aplicar la operación indicada por el operador. Esos operadores, mientras no se especifique de forma concreta, podrán ser constantes, variables o expresiones formadas por otros operando y operadores.

Al evaluar una expresión, hay que tener en cuenta, que el procesador, al calcular una operación, necesita que todos los operandos que forman parte de la misma sean del mismo tipo de dato.. Si no lo son, se hacen conversiones, de forma que a veces, se puede perder precisión en los datos. Esto, se verá en el último punto de este tema.

4.2.- OPERADORES DE C MÁS USADOS

4.2.1.- Operador asignación

Sintaxis:

<variable>**=**<expresión>;

El funcionamiento de este operador ya ha sido explicado anteriormente (punto 1.3.1.1. Variables del tema **1.- INTRODUCCIÓN AL LENGUAJE C**). Básicamente, lo que se hace es evaluar la expresión indicada a la derecha del símbolo '=', y dar el valor obtenido a la variable indicada a la izquierda del mismo.

4.2.2.- Operadores Aritméticos

Se trata de operadores que sirven para realizar operaciones matemáticas:

Operador	Sintaxis	Descripción
+	<operando1>**+**<operando2>	Suma <operando1> y <operando2>
-	<operando1>**-**<operando2>	Resta de <operando2> a <operando1>
*	<operando1>*<operando2>	Multiplica <operando1> y <operando2>
/	<dividendo>/<divisor>	Cociente entre <dividendo> y <divisor>
%	<dividendo>**%**<divisor>	Obtiene el resto de dividir <dividendo> por <divisor>

4.2.3.- Operadores De Autoincremento y Autodecremento

Se trata de operadores que sirven para autoincrementar y autodecrementar el valor del operando al que se aplican. Cuando se usan en combinación con otros operadores, por ejemplo, el operador asignación, dependiendo de qué sintaxis se utilice, el resultado es diferente:

Operador	Sintaxis	Descripción
++	<operando>**++**	Primero se usa el valor de operando, y después incrementa su valor. Ejemplo: j=i++; ⇔ j=i; i=i+1;
	++<operando>	Primero se incrementa el valor del operando, y después usa ese valor. Ejemplo: j=++i; ⇔ i=i+1; j=i;
--	<operando>**--**	Primero se usa el valor de operando, y después decrementa su valor. Ejemplo: j=i--; ⇔ j=i; i=i-1;
	--<operando>	Primero se decrementa el valor del operando, y después usa ese valor. Ejemplo: j=--i; ⇔ i=i-1; j=i;

Ejemplo:

```
#include <stdio.h>

int main()
{
   int v1=2,v2;

   printf("\nImprimimos e incrementamos v1: %d",v1++);
   printf("\nValor de v1: %d\n",v1);

   v2=v1++;
   printf("\nValor de v2: %d",v2);
   printf("\nValor de v1: %d\n",v1);

   printf("\nIncrementamos v1: %d",++v1);
   printf("\nValor de v1: %d\n",v1);

   v2=++v1;
   printf("\nValor de v2: %d",v2);
   printf("\nValor de v1: %d",v1);

   return 0;
}
```

4.2.4.- Operadores Combinados

Se trata de operadores que facilitan operaciones combinadas del operador asignación con los operadores aritméticos:

Operador	Sintaxis	Descripción
+=	<operando1>**+=**<operando2>	<operando1>=<operando1>+<operando2>
-=	<operando1>**-=**<operando2>	<operando1>=<operando1>-<operando2>
*=	<operando1>***=**<operando2>	<operando1>=<operando1>*<operando2>
/=	<dividendo>/**=**<divisor>	<dividendo>=<dividendo>/<divisor>
%=	<dividendo>**%=**<divisor>	<dividendo>=<dividendo>%<divisor>

4.2.5.- Operadores Relacionales

Los operadores relacionales, se usan para evaluar condiciones. La evaluación de una condición sólo puede dar dos respuestas, *true* (verdadero), si la condición es verdadera, y *false* (falso), si la condición es falsa. En C, un resultado *false* de una condición, se indica en tiempo de ejecución como un valor 0; un resultado *true*, sin embargo, se indica mediante cualquier valor distinto de 0, aunque normalmente se use 1. La siguiente tabla muestra los operadores relacionales más usados:

Operador	Sintaxis	Devuelve *true* si
>	<operando1>**>**<operando2>	<operando1> es mayor que <operando2>
>=	<operando1>**>=**<operando2>	<operando1> es mayor o igual que <operando2>
<	<operando1>**<**<operando2>	<operando1> es menor que <operando2>
<=	<operando1>**<=**<operando2>	<operando1> es menor o igual que <operando2>
==	<operando1>**==**<operando2>	<operando1> es igual que <operando2>
!=	<operando1>**!=**<operando2>	<operando1> es diferente de <operando2>
&&	<operando1>**&&**<operando2>	<operando1> y <operando2> son *true*
||	<operando1>||<operando2>	<operando1> u <operando2> es/son *true*
!	**!**<operando>	<operando>es *false*

Se dice que && y || realizan evaluación perezosa es decir:

- Al calcular <operando1>**&&**<operando2>, si <operando1> es *false*, <operando2> no se evalúa (NO hace falta), y el resultado es *false*.

- Al calcular <operando1>||<operando2>, si <operando1> es *true*, <operando2> no se evalúa (NO hace falta), y el resultado es *true*.

Los operadores relacionales se suelen usar en las estructuras de control de flujo, que se verán más adelante.

Ejemplo:

```
#include <stdio.h>

int main()
{
  int v1;

  v1=18;
  printf("\nEs el valor de v1 menor que 21? %d",v1<21);

  v1=27;
  printf("\nEs el valor de v1 menor que 21? %d",v1<21);

  return 0;
}
```

4.2.6.- Operadores Lógicos

Son operadores que manipulan los valores a nivel de bit:

Operador	Sintaxis	Descripción
>>	<operando1>**>>**<operando2>	Desplaza los bits de <operando1> la derecha <operando2> posiciones
<<	<operando1>**<<**<operando2>	Desplaza los bits de <operando1> la izquierda <operando2> posiciones
&	<operando1>**&**<operando2>	Operación AND entre los bits de <operando1> y <operando2>
|	<operando1>|<operando2>	Operación OR entre los bits de <operando1> y <operando2>
^	<operando1>**^**<operando2>	Operación XOR entre los bits de <operando1> y <operando2>
~	~<operando>	Operación NOT de los bits de <operando>

Ejemplos:

3<<5 ⇒ 00000011<<5 ⇒ 01100000=96
3>>5 ⇒ 00000011>>5 ⇒ 00011000=24
5&17 ⇒ 00000101&00010001⇒ 00000001=1
5|17 ⇒ 00000101|00010001⇒ 00010101=21
5^17 ⇒ 00000101|00010001⇒ 00010100=20

4.2.7.- Operador Paréntesis

Este operador sirve para agrupar operaciones.

Sintaxis:

(<expresión>**)**

4.3.- PRECEDENCIA DE LOS OPERADORES

El orden de evaluación de una expresión, en la que se incluyen más de una operación, puede variar el resultado. Por eso, en C, a cada operador se le da un nivel de prioridad, que se conoce como *precedencia*. Un operador se evaluará antes que todos los demás operadores que tengan mayor nº de orden de precedencia.

La siguiente tabla muestra el orden de precedencia de los operadores en C:

Orden de Precedencia	Operador	Forma de evaluación
1 (Primarios)	(), [], ., ->, <expresión>++, <expresión>--	De izquierda a derecha
2 (Unarios)	*<variable>,&<variable>, !,~, *casting*, ++<expresión>, --<expresión>, sizeof	De derecha a izquierda
3 (Binarios)	*, /, %	De izquierda a derecha
4 (Binarios)	+, -	
5 (Binarios)	>>, <<	
6 (Binarios)	<, <=, >, >=	
7 (Binarios)	==, !=	
8 (Binarios)	&	
9 (Binarios)	^	
10 (Binarios)	|	
11 (Binarios)	&&	
12 (Binarios)	||	
13 (Asignación)	=, *=, /=, +=, -=, %=	De derecha a izquierda

Los Operadores Primarios *[]* (corchetes), *.* (punto), *->* (flecha) y los Unarios **<variable>*, *&<variable>* y *sizeof* se verán en temas posteriores. El operador Unario *casting* se verá en el siguiente punto de este mismo tema.

NOTA: En esta tabla, como en el tema en sí, no se han incluido todos los operadores de C, sólo los que se van a usar en la asignatura.

Ejemplo:

Supongamos que en un programa en C, tenemos una expresión como la siguiente:

-(2+5)*6+(4+3*(2+3))

Los pasos que en ejecución se siguen para evaluar la expresión, se dan siguiendo el orden de precedencia de los operadores:

Paso 1: -7*6+(4+3*(2+3))
Paso 2: -7*6+(4+3*5)
Paso 3: -7*6+(4+15)
Paso 4: -7*6+19
Paso 5: -42+19
Paso 6: -23

En este ejemplo, todos los valores son constantes enteras, es decir, datos de tipo *int*. Si en algún paso, los operandos de la operación a evaluar, no fueran del mismo tipo de dato, el procesador haría una conversión, en la que se puede perder precisión.

4.4.- CONVERSIONES DE TIPOS DE DATOS

Tal y como hemos adelantado en la introducción del presente tema, al ejecutar un operador, el procesador necesita que los operandos sobre los que se ejecuta sean del mismo tipo de dato. Si no lo son, tendrá que hacer lo que se llama una *conversión de tipo de dato*, convirtiendo el valor de uno de los

operandos al tipo de dato del otro. Esa conversión, a veces se puede hacer sin ningún problema, pero otras veces puede suponer la pérdida de precisión en un valor. Cuando esto último ocurre, el compilador da un *aviso* (*warning*) como resultado de la compilación.

Para entender las conversiones, hay que saber primero que los tipos de dato tienen un concepto llamado rango, relacionado con el nº de bytes que ocupan en memoria. El orden de los tipos de datos según el rango, de menor a mayor, es el que se indica a continuación:

char<short<unsigned short<int<unsigned int<float<double

Así, un valor en un programa en C, puede sufrir dos clases de conversiones:

- **Conversión implícita**: Este tipo de conversión la realiza el procesador de forma automática, sin que sea necesario que el programador se dé cuenta de ello al programar, y sin ninguna pérdida de precisión.
  Hay casos en los que se puede dar conversión implícita:
  - Conversión de rango menor a mayor: Cuando el tipo de dato de un valor ha de pasar de un tipo de rango menor a rango mayor. Se realiza de forma implícita siempre.
  - Conversión de rango mayor a menor, entre tipos de dato compatibles: Cuando el tipo de dato de un valor ha de pasar de un tipo de rango mayor a rango menor, sólo puede ocurrir de forma implícita, cuando los tipos de dato implicados sean compatibles para el valor a convertir.
    Ejemplo:

    El valor *3* puede considerarse de tipo *int* (porque es un número entero), pero también de tipo *float* o incluso *double* (porque los números enteros son también números reales). Si siendo ese *3* el valor de una variable de tipo *float*, hubiera que convertirlo en *int*, el procesador lo haría de forma implícita sin ningún problema, ya que es una conversión de mayor a menor rango entre tipos compatibles. Incluso, podría convertirse en un valor *char*, ya que el *3* es un código válido de la tabla ASCII.

    El valor *3.1* sin embargo, se podría convertir de *double* a *float* sin ningún problema, porque ambos tipos de dato son compatibles con el valor. Sin embargo, no podría convertirse de forma implícita a un *int*, por ejemplo, porque el tipo de dato *int* no puede representar números con coma flotante (no es compatible con el valor).

- **Conversión explícita**: Este tipo de conversión NO la realiza el procesador de forma automática, sino porque el programador se lo ha indicado expresamente. Por eso, se le suele llamar también *conversión forzada por el programador* o *casting*. Su uso es obligatorio para las conversiones que no se puedan realizar de forma implícita, es decir, para conversiones de rango mayor a menor entre tipos de dato NO compatibles para el valor a convertir. Siempre suponen pérdida de precisión en el valor del dato convertido, es inevitable, pero si no se indica de forma explícita en el fichero fuente, el compilador dará un aviso de pérdida de precisión.

La forma de hacer una conversión explícita o casting, es escribiendo entre paréntesis el nuevo tipo de dato, justo antes del valor a convertir, y se considera como Operador Unario con precedencia 2.

Ejemplo:

```
int k,altura;

k=1.7f+altura;
/*En este ejemplo es obligatorio hacer un casting para no tener un warning de pérdida de precisión.*/
```