

# Operadores y expresiones

## Contenido

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>3.1. Operadores y expresiones</li><li>3.2. Operador de asignación</li><li>3.3. Operadores aritméticos</li><li>3.4. Operadores de incremento y decremento</li><li>3.5. Operadores relacionales</li><li>3.6. Operadores lógicos</li><li>3.7. Operadores de manipulación de bits</li><li>3.8. Operador condicional</li><li>3.9. Operador coma</li></ul> | <ul style="list-style-type: none"><li>3.10. Operadores especiales, <code>:</code>, <code>()</code>, <code>[]</code> y <code>::</code></li><li>3.11. El operador <code>sizeof</code></li><li>3.12. Conversión de tipos</li><li>RESUMEN</li><li>EJERCICIOS</li><li>PROBLEMAS</li><li>EJERCICIOS RESUELTOS</li><li>PROBLEMAS RESUELTOS</li></ul> |
|--|---|

## INTRODUCCIÓN

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Este capítulo muestra cómo C++ hace uso de los operadores y expresiones para la resolución de operaciones. Los operadores fundamentales que se analizan en el capítulo son:

- aritméticos, lógicos y relacionales;
- de manipulación de bits;
- condicionales;
- especiales.

Además, se analizarán las conversiones de tipos de datos y las reglas que seguirá el compilador cuando concurran en una misma expresión diferentes tipos de operadores. Estas reglas se conocen como *prioridad* y *asociatividad*.

## CONCEPTOS CLAVE

- Asignación.
- Asociatividad.
- Conversión explícita.
- Conversiones de tipos.
- Evaluación en cortocircuito.
- Expresión.
- Incrementación/decrementación.
- Manipulación de bits.
- Operador.
- Operador `sizeof`.
- Prioridad/precedencia.
- Tipo `bool`.

### 3.1. OPERADORES Y EXPRESIONES

Una *expresión* se compone de uno o más **operandos** que se combinan entre sí mediante **operadores**. En la práctica una expresión es una secuencia de operaciones y operandos que especifica un cálculo y en consecuencia devuelve un resultado. La forma más simple de una **expresión** consta de una única constante o variable. Las expresiones más complicadas se forman a partir de un operador y uno o más operandos.

Cada expresión produce un **resultado**. En el caso de una expresión sin operador/es el resultado es el propio operando; por ejemplo, una constante o una variable.

```
4.5           //expresión que devuelve el valor 2.5
Horas_Semana  //constante que devuelve un valor, p.e. 30
```

El resultado de expresiones que implican operadores se determina aplicando cada operador a sus operandos. Los **operadores** son símbolos que expresan operaciones que se aplican a uno o varios operandos y que devuelven un valor

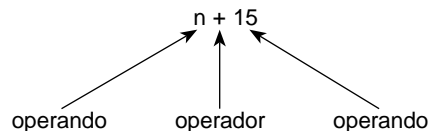


Figura 3.1. Expresión con un operador y operandos.

#### 3.1.1. Expresiones

Al igual que en otros lenguajes, C++ permite formar expresiones utilizando variables constantes y operadores aritméticos: + (suma), - (resta), × (multiplicación, / (división) y % (resto módulos). Estas expresiones se pueden utilizar en cualquier lugar que sea legal utilizar un valor del tipo resultante de la expresión.

Una **expresión** es una operación que produce un valor con la excepción de expresiones void. Casi todos los tipos de sentencias utilizan expresiones de una u otra manera.

---

#### Ejemplo

*Lista de expresiones, junto con la descripción del valor que cada una produce*

```
x           // devuelve el valor de 15
25          // devuelve 25
x + 25      // devuelve x + 25
x == 45     // comprueba la igualdad: devuelve 1 o 0
x = 50      // asignación; devuelve el valor asignado (50)
i++         // devuelve el valor de i ante la incrementación
i = num +5  // expresión compleja; devuelve un nuevo
            // valor de i
```

---

#### Ejemplo

*La declaración siguiente utiliza una expresión para su inicializador*

```
int t = (100 + 50) /2;
```

Las expresiones más simples en C++ son simplemente literales o variables. Por ejemplo:

```
1.25    false    "sierra magina"    total
```

Expresiones más interesantes se forman combinando literales, variables y los valores de retorno de las funciones con diferentes operadores para producir menos valores. Las expresiones pueden contener a su vez expresiones como miembros o partes de ellas.

---

## Ejemplo

*Diferentes expresiones C++*

```
i -> ObtenerValor(c) + 15  
P * pow (1.0 + tasa, (double) mes))  
new char[30]  
sizeof (int) + sizeof (double) + 1
```

---

## Nota

Obsérvese que las expresiones no son igual que las sentencias. Las sentencias indican al compilador que realice alguna tarea y termina con un punto y coma, mientras que las expresiones especifican un cálculo. En una sentencia puede haber varias expresiones.

## 3.1.2. Operadores

C/C++ son lenguajes muy ricos en operadores. Se clasifican en función del número de operandos sobre los que actúa y por las operaciones que realizan.

El significado de un operador —operación que realiza y tipo de resultado— depende de los tipos de sus operandos. Hasta que no se conoce el tipo de operando/s, no se puede conocer el significado de la expresión. Por ejemplo,

```
m + n
```

puede significar suma de enteros, concatenación de cadenas (`string`), suma de números en cuenta flotante, o incluso otra operación. En realidad, la expresión se evalúa y su resultado depende de los tipos de datos de `m` y `n`.

Los operadores se clasifican en: *unitarios* ("unarios"), binarios o ternarios. Operadores unitarios, tales como el operador de dirección (&) o el de *desreferencia* (\*) que actúan sobre un operando. Operadores binarios, tales como suma (+) y resta (-) que actúan sobre dos operandos. Operadores ternarios, tales como, el operador condicional (?:) que actúa sobre tres operandos.

Algunos operadores pueden actuar como "*unitarios*" o como binarios; un ejemplo es el operador \* que puede actuar como el operador de multiplicación (binario) o como operador de *desreferencia* (indirección). Los operadores imponen los requisitos sobre el tipo/s de su/s operando/s. Así, cuando un operador binario se aplica a operandos de tipos predefinidos o tipos compuestos, se requiere normalmente que sean del mismo tipo o tipos que se puedan convertir a un tipo común. Por ejemplo, se puede convertir un entero a un tipo de coma flotante y viceversa; sin embargo, no es posible convertir un tipo puntero a un tipo en coma flotante.

Los operadores se clasifican también según la posición del operador y de los operandos: *prefijo* (si va delante), *infijo* (si va en el interior) o *postfijo* (si va detras). Otra propiedad importante de los operadores es la **aridad** (número de operandos sobre los que actúa):

Unitarios (unarios o monarios) = un solo operando  
 Binarios = dos operandos  
 Ternarios = tres operandos

Cuando las expresiones contienen varios operandos, otras propiedades son muy importantes: *precedencia*, *asociatividad* y *orden de evaluación* de los operandos.

**Precedencia** (*prioridad de evaluación*). Indica la prioridad del operador respecto a otros a la hora de calcular el valor de una expresión.

**Asociatividad**. Determina el orden en que se asocian los operandos del mismo tipo en ausencia de paréntesis.

**Asociatividad por la derecha** (D-I). Si dos operandos que actúan sobre el mismo operando tienen la misma precedencia se aplica primero al operador que está más a la derecha (operadores primarios, terciarios y asignación).

**Asociatividad por la izquierda** (I-D). En este caso se aplica primero al operador que está a mano izquierda (operadores binarios).

---

## Ejemplos

$m = n = p$	<i>equivale a</i>	$m = (n = p)$	D-I
$m + n + p$	<i>equivale a</i>	$(m + n) + p$	I-D

---

**Sin asociatividad**. En algunos operadores no tiene sentido la asociatividad (éste es el caso de `sizeof`).

---

## Ejemplo

$20 * 5 + 24$

Como se verá más adelante, el operador `*` tiene mayor prioridad que `+`.

1. Se evalúa primero  $20 * 5$ , produce el resultado 100.
  2. Se realiza la suma  $100 + 24$  y resulta 124.
- 

Una clasificación completa de operadores se muestra a continuación:

- Operadores de resolución de ámbito (alcance).
- Operadores aritméticos.
- Operadores de incremento y decremento.
- Operadores de asignación.
- Operadores de asignación compuesta.
- Operadores relacionales.
- Operadores lógicos.
- Operadores de bits.

- Operadores condicionales.
- Operadores de dirección o de indirección.
- Operadores de tamaño (`sizeof`).
- Operadores de secuencia o de evaluación (coma).
- Operadores de conversión.
- Operadores de *moldeado* o *molde* ("`cast`").
- Operadores de construcción de tipos (`static_cast`, `reinterpret_cast`, `const_cast`, `dynamic_cast`).
- Operadores de memoria dinámica (`new` y `delete`).

## Resumen

La *prioridad* o *precedencia* de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores C++ vienen en una tabla con 16 grupos. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

- Si dos operadores se aplican al mismo operando, el operador con mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- Si dos operandos tienen igual prioridad, el operador con prioridad más alta se aplica primero.
- La asociatividad izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

Prioridad	Operadores	Asociatividad
1	<code>::</code> <code>*</code> <code>-&gt;</code> <code>[]</code> <code>()</code>	I – D
2	<code>++</code> <code>--</code> <code>~</code> <code>!</code> <code>-</code> <code>+</code> <code>&amp;</code> <code>*</code> <code>sizeof</code>	D – I
3	<code>.</code> <code>*</code> <code>-&gt;*</code>	I – D
4	<code>*</code> <code>/</code> <code>%</code>	I – D
5	<code>+</code> <code>-</code>	I – D
6	<code>&lt;&lt;</code> <code>&gt;&gt;</code>	I – D
7	<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code>	I – D
8	<code>==</code> <code>!=</code>	I – D
9	<code>&amp;</code>	I – D
10	<code>^</code>	I – D
11	<code>  </code>	I – D
12	<code>&amp;&amp;</code>	I – D
13	<code>  </code>	I – D
14	<code>?:</code> (expresión condicional)	D – I
15	<code>=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>-=</code> <code>&lt;&lt;=</code> <code>&gt;&gt;=</code> <code>&amp;=</code> <code>  =</code> <code>^=</code>	D – I
16	<code>,</code> (operador coma)	I – D

I – D : *Izquierda – Derecha.*

D – I : *Derecha – Izquierda.*

### 3.1.3. Evaluación de expresiones compuestas

Una expresión con dos o más operadores es una *expresión compuesta*. En una expresión compuesta el modo en que se agrupan los operandos a los operadores puede determinar el resultado de la expresión global. Dependiendo del modo en que agrupen los operandos el resultado será diferente.

La precedencia y asociatividad determina cómo se agrupan los operandos, aunque los programadores pueden anular estas reglas poniendo paréntesis en las expresiones particulares que seleccione.

#### Precedencia (prioridad)

Cada operador tiene una *precedencia* (prioridad). C++ utiliza las reglas de precedencia para decidir qué operador se utiliza primero. Los operadores con precedencia más alta se agrupan de modo que se evalúan lógicamente antes de los operadores con menor precedencia. La precedencia determina cómo analiza el compilador la expresión, no necesariamente el orden real de cálculo. Por ejemplo, en la expresión  $x() + b() * c()$ , la multiplicación tiene la precedencia más alta, pero  $x()$  puede ser llamado primero.

---

#### Ejemplo

¿Cual es el resultado de  $6 + 3 * 4 / 2 + 2$ ;

Dependiendo de como se agrupen las expresiones se producirá un resultado u otro. En concreto posibles soluciones son 14 (resultado real en C++), 36, 20 o 9.

$  \begin{array}{ccccccc}  6 & + & 3 & * & 4 & / & 2 & + & 2 \\  \downarrow & & & & \underbrace{\phantom{3 * 4}}_{12} & & & & \downarrow \\  6 & + & & & 6 & & & + & 2  \end{array}  $	resultado en C++ 14
--	---------------------

El resultado 14 se produce porque la multiplicación (y la división) tiene la precedencia más alta y cuando coinciden en la misma expresión se aplica primero el operador de la izquierda, ya que los operadores aritméticos tienen asociatividad a izquierda. En el ejemplo anterior se realiza primero  $3 * 4$  y luego su valor se divide por 2.

---

#### Anulación de la precedencia con paréntesis

Se puede anular la precedencia con paréntesis para tratar cada expresión entre paréntesis como una unidad y luego se aplican las reglas normales de precedencia dentro de cada expresión entre paréntesis.

---

#### Ejemplo

```

cout << 6 + 3 * 4 / 2 + 2 << endl;           //visualiza 14
cout << ((6 + ((3 * 4)) / 2)) + 2)) << endl;   //visualiza 14
cout << 6 + 3 * 4 / (2 + 2) << endl;          //visualiza 9
cout << (6 + 3) * (4 / 2 + 2) << endl;        //visualiza 36

```

---

#### Asociatividad

Algunos operadores se agrupan de izquierda a derecha y otros operadores se agrupan de derecha a izquierda.

El orden de agrupamiento se denomina *asociatividad* del operador. Cuando dos operadores tienen la misma prioridad, C++ examina a ver si sus operadores tienen asociatividad *izquierda-a-derecha* o *derecha-a-izquierda*. La asociatividad izquierda-derecha significa que si dos operadores actúan sobre el mismo operando y tienen la misma precedencia, se aplica primero el operador situado a mano izquierda; en el caso de asociatividad a derecha se aplica primero el operador situado a mano derecha.

---

### Ejemplo

- $x / y / z$  equivale a  $(x / y) / z$
- $x = y = z$  equivale a  $x = (y = z)$

```
float logos = 120/4*5;    //el valor es 150
```

---

## 3.2. OPERADOR DE ASIGNACIÓN

El operador de asignación = asigna el valor de la expresión derecha a la variable situada a su izquierda.

*variable* **operador** = *expresión*

*expresión* puede ser una variable, una constante o una expresión aritmética más complicada

```
codigo = 3467
fahrenheit = 123.456;
coordX = 525;
coordY = 725;
```

### Asignación compuesta

Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así,

```
a = b = c = 45;
```

equivale a

```
a = (b = (c = 45));
```

o dicho de otro modo, a las variables *a*, *b* y *c* se asigna el valor 45.

Esta propiedad permite inicializar varias variables con una sola sentencia

```
int a, b, c;
a = b = c = 5;    //se asigna 5 a las variables a, b y c
```

---

### Ejemplo

```
int i, j, val_m;
const int ci = i;    //inicialización, no asignación
2040 = val_m;        //error
i + j = valor_m;     //error
ci = val_m;          //error
```

---

Además del operador de asignación =, C++ proporciona cinco operadores de asignación adicionales. En la Tabla 3.1 aparecen los seis operadores de asignación.

**Tabla 3.1.** Operadores de asignación de C++.

Símbolo	Uso	Descripción
=	<code>a = b</code>	Asigna el valor de <i>b</i> a <i>a</i> .
*=	<code>a *= b</code>	Multiplica <i>a</i> por <i>b</i> y asigna el resultado a la variable <i>a</i> .
/=	<code>a /= b</code>	Divide <i>a</i> entre <i>b</i> y asigna el resultado a la variable <i>a</i> .
%=	<code>a %= b</code>	Fija a al resto de <i>a/b</i> .
+=	<code>a += b</code>	Suma <i>b</i> y <i>a</i> y lo asigna a la variable <i>a</i> .
-=	<code>a -= b</code>	Resta <i>b</i> de <i>a</i> y asigna el resultado a la variable <i>a</i> .

Estos operadores de asignación actúan como una notación abreviada para expresiones utilizadas con frecuencia. Así, por ejemplo, si se desea multiplicar 10 por *i*, se puede escribir

```
i = i * 10;
```

**Precedencia.** Muchas expresiones implican más de un operador. En estos casos es necesario saber qué operando se aplica primero para obtener el valor final.

C++ proporciona un operador abreviado de asignación (\*=), que realiza una asignación equivalente:

```
i *= 10;      equivale a      i = i * 10;
```

**Tabla 3.2.** Equivalencia de operadores de asignación.

Operador	Sentencia abreviada	Sentencia no abreviada
+=	<code>m += n</code>	<code>m = m + n;</code>
-=	<code>m -= n</code>	<code>m = m - n;</code>
*=	<code>m *= n</code>	<code>m = m * n;</code>
/=	<code>m /= n</code>	<code>m = m / n;</code>
%=	<code>m %= n</code>	<code>m = m % n;</code>

Estos operadores de asignación no siempre se utilizan, aunque algunos programadores C++ se acostumbran a su empleo por el ahorro de escritura que suponen.

## Operadores de asignación compuesta

C++ proporciona operadores de asignación compuesta para cada uno de los operadores. La sintaxis general de un operador de asignación compuesta es

```
a op= b;
```

donde `op=` puede ser cualquiera de los siguientes diez operadores:

```
+=    -=    *=    /=    %=    //operadores aritméticos
<<=   >>=   &=    ^=    |=    //operadores de bits
```



En esencia, cada operador compuesto equivale a:

```
a = a op b;
```

*b* puede ser una expresión *a op= expresión* y entonces la operación equivale a

```
a = a op(expresión)
```

Ejemplo

*Equivalente a*

cuenta += 5;	cuenta = cuenta + 5;
total -= descuento;	total = total - descuento;
cambio %= 100;	cambio = cambio % 100;
cantidad *= c1 + c2;	cantidad = cantidad * (c1 + c2);
m -= 8;	m = m - 8;
m *= 6;	m = m * 6;

3.3. OPERADORES ARITMÉTICOS

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Los operadores aritméticos C++ siguen las reglas algebraicas típicas de jerarquía o prioridad. Estas reglas especifican la precedencia de las operaciones aritméticas.

3.3.1. Precedencia

Considere la expresión

```
3 + 5 * 2
```

¿Cuál es el valor correcto,  $16 = (8 * 2)$  o  $13 = (3 + 10)$ ? De acuerdo a las citadas reglas, la multiplicación se realiza antes que la suma. Por consiguiente, la expresión anterior equivale a:

```
3 + (5 * 2)
```

En C++ las expresiones interiores a paréntesis se evalúan primero; a continuación, se realizan los operadores unitarios, seguidos por los operadores de multiplicación, división, resto (módulo), suma y resta.

Tabla 3.3. Operadores aritméticos.

Operador	Tipos enteros	Tipos reales	Ejemplo
+	Suma	Suma	4 + 5
-	Resta	Resta	7 - 3
*	Producto	Producto	4 * 5
/	División entera: cociente	División en coma flotante	8 / 5
%	División entera: resto	División en coma flotante	12 % 5

**Tabla 3.4.** Precedencia de operadores aritméticos básicos.

Operador	Operación	Nivel de precedencia
+, -, *, /, %	+25, -6.745	1
	5*5 es 25	2
	25/5 es 5	
	25%6 es 1	
+, -	2+3 es 5	3
	2-3 es -1	

Obsérvese que los operadores + y -, cuando se utilizan delante de un operador, actúan como operadores unitarios más y menos.

```
+75      // significa que es positivo
-154     // significa que es negativo
```

**Ejemplo 3.1**

1. ¿Cuál es el resultado de la expresión:  $6 + 2 * 3 - 4 / 2$ ?

$$\begin{array}{r} 6 + \underbrace{2 * 3} - 4 / 2 \\ 6 + 6 - \underbrace{4 / 2} \\ \underbrace{6 + 6} - 2 \\ \underbrace{12} - 2 \\ 10 \end{array}$$

2. ¿Cuál es el resultado de la expresión:  $5 * 5 (5 + (6 - 2) + 1)$ ?

$$\begin{array}{r} 5 * (5 + \underbrace{(6 - 2)} + 1) \\ 5 * (\underbrace{5 + 4 + 1}) \\ 5 * 10 \\ 50 \end{array}$$

**3.3.2. Asociatividad**

En una expresión tal como

$$3 * 4 + 5$$

el compilador realiza primero la multiplicación —por tener el operador \* prioridad más alta— y luego la suma, por tanto produce 17. Para forzar un orden en las operaciones se deben utilizar paréntesis

$$3 * (4 + 5)$$

produce 27, ya que 4+5 se realiza en primer lugar.

La *asociatividad* determina el orden en que se agrupan los operadores de igual prioridad; es decir, de izquierda a derecha o de derecha a izquierda. Por ejemplo,

$$10 - 5 + 3 \text{ se agrupa como } (10 - 5) + 3$$

ya que  $-$  y  $+$ , que tienen igual prioridad, tienen asociatividad de izquierda a derecha. Sin embargo,

$$x = y = z$$

se agrupa como

$$x = (y = z)$$

ya que su asociatividad es de derecha a izquierda.

**Tabla 3.5.** Prioridad y asociatividad.

Prioridad (mayor a menor)	Asociatividad
$+$ , $-$ (unitarios)	izquierda-derecha (I-D)
$*$ , $/$ , $\%$	izquierda-derecha (I-D)
$+$ , $-$	izquierda-derecha (I-D)

### Ejemplo 3.2

¿Cuál es el resultado de la expresión:  $70 - 5 \% 3 * 4 + 9$ ?

Existen tres operadores de prioridad más alta ( $*$ ,  $\%$  y  $+$ )

$$70 - 5 \% 3 * 4 + 9$$

La asociatividad es a izquierda, por consiguiente se ejecuta a continuación  $\%$

$$70 - 2 * 4 + 9$$

y la segunda multiplicación se realiza a continuación, produciendo

$$70 - 8 + 9$$

Las dos operaciones restantes son de igual prioridad y como la asociatividad es a izquierda, se realizará la resta primero y se obtiene el resultado

$$62 + 9$$

y por último se realiza la suma y se obtiene el resultado final de

$$71$$

### 3.3.2. Uso de paréntesis

Los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su prioridad y asociatividad. Las subexpresiones entre paréntesis se evalúan en primer lugar

según el modo estándar y los resultados se combinan para evaluar la expresión completa. Si los paréntesis están *anidados* —es decir, un conjunto de paréntesis contenido en otro— se ejecutan en primer lugar los paréntesis más internos.

Por ejemplo, considérese la expresión  $(7 * (10 - 5) \% 3) * 4 + 9$ . La subexpresión  $(10 - 5)$  se evalúa primero, produciendo

$$(7 * 5 \% 3) * 4 + 9$$

A continuación, se evalúa de izquierda a derecha la subexpresión  $(7 * 5 \% 3)$

$$(35 \% 3) * 4 + 9$$

seguida de

$$2 * 4 + 9$$

Se realiza a continuación la multiplicación, obteniendo

$$8 + 9$$


y la suma produce el resultado final

$$17$$

### Precaución

Se debe tener cuidado en la escritura de expresiones que contengan dos o más operaciones para asegurarse que se evalúan en el orden previsto. Incluso aunque no se requieran paréntesis, deben utilizarse para clarificar el orden concebido de evaluación y escribir expresiones complicadas en términos de expresiones más simples. Es importante, sin embargo, que los paréntesis estén equilibrados —cada paréntesis a la izquierda tiene un correspondiente paréntesis a la derecha que aparece posteriormente en la expresión— ya que si existen paréntesis desequilibrados se producirá un error de compilación.

$$((8 - 5) + 4 - (3 + 7))$$

 error de compilación, falta paréntesis final a la derecha

## 3.4. OPERADORES DE INCREMENTO Y DECREMENTO

De las muchas características de C++ heredadas de C, una de las más útiles son los operadores de incremento ++ y decremento --. Los operadores ++ y --, denominados de *incrementación* y *decrementación*, suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable.

Por consiguiente,

**Tabla 3.6.** Operadores de incrementación (++) y decrementación (--).

Incrementación	Decrementación
++n	--n
n += 1	n -= 1
n = n + 1	n = n - 1

```
a++
```

es igual que

```
a+1
```

Las sentencias

```
++n;  
n++;
```

tienen el mismo efecto; así como

```
--n;  
n--;
```

Sin embargo, cuando se utilizan como expresiones tales como

```
m = n++;
```

o bien,

```
cout << --n;
```

`++n` produce un valor que es mayor en uno que el de `n++`, y `--n` produce un valor que es menor en uno que el valor de `n--`.

```
int a = 1, b;  
b = a++;           //b vale 1 y a vale 2  
  
int a = 1, b;  
b = ++a;           //b vale 2 y a vale 2
```

Si los operadores `++` y `--` están de prefijos, la operación de incremento se efectúa antes que la operación de asignación; si los operadores `++` y `--` están de sufijos, la asignación se efectúa en primer lugar y la incrementación o decrementación a continuación.

### Ejemplo

```
int i = 10;  
int j;  
...  
j = i++;
```

La variable `j` vale 10, ya que cuando aparece `++` después del operando (la variable `i`) el valor que se asigna a `j` es el valor de `i` (10) y luego posteriormente se incrementa a `i`, que toma el valor 11.

---

### Ejemplo 3.3

*Demostración del funcionamiento de los operadores de incremento/decremento.*

```
#include <iostream>  
using namespace std;
```

```
// Test de operadores ++ y --
main()
{
    int m = 45, n = 75;
    cout << " m = " << m << " n = " << n << endl;
    ++m;
    --n;
    cout << " m = " << m << ", n = " << n << endl;
    m++;
    n--;
    cout << "m=" << m << ", n=" << n << endl;
    return 0;
}
```

---

### Ejecución

```
m = 45,    n = 75
m = 46,    n = 74
m = 46,    n = 73
```

### Ejemplo 3.4

*Diferencias entre operadores de preincremento y postincremento.*

```
#include <iostream>
using namespace std;

// test de operadores incremento y decremento
int main()
{
    int m = 99, n;
    n = ++m;
    cout << "m = " << m << ", n = " << n << endl;
    n = m++;
    cout << "m = " << m << ", n = " << n << endl;
    cout << "m = " << m++ << endl;
    cout << "m = " << m << endl;
    cout << "m = " << ++m << endl;
    return 0;
}
```

---

### Ejecución

```
m = 100,    n = 100
m = 101,    n = 100
m = 101
m = 102
m = 103
```

### Ejemplo 3.5

*Orden de evaluación no predecible en expresiones.*

```
#include <iostream>
using namespace std;

void main()
{
    int n = 5, t;
    t = ++n * --n;
    cout << "n= " << n << ", t = " << t << endl;
    cout << ++n << " " << ++n << " " << ++n << endl;
}
```

### Ejecución

```
n = 5,  t = 25
6  7  8
```

Aunque parece que aparentemente el resultado de *t* será 30, en realidad es 25, debido a que en la asignación de *t*, *n* se incrementa a 6 y, a continuación, se decrementa a 5 antes de que se evalúe el operador producto, calculando 5 \* 5. Por último, las tres subexpresiones se evalúan de derecha a izquierda y como la asociatividad a izquierda del operador de salida << no es significativa, el resultado será 6 7 8, al contrario de 8 7 6, que es lo que parece que aparentemente se producirá.

## 3.5. OPERADORES RELACIONALES

ANSI C++ soporta el tipo `bool` que tiene dos literales `false` y `true`. Una expresión *booleana* es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores `true` y `false`.

C++ no tiene tipos de datos lógicos o booleanos, como Pascal, para representar los valores verdadero (*true*) y falso (*false*). En su lugar se utiliza el tipo `int` para este propósito, con el valor entero 0 que representa a falso y distinto de cero a verdadero.

<i>falso</i>	cero
<i>verdadero</i>	distinto de cero

Operadores tales como `>=` y `==` que comprueban una relación entre dos operandos se llaman operadores relacionales y se utilizan en expresiones de la forma

$$\text{expresión}_1 \text{ operador\_relacional } \text{expresión}_2$$

<i>expresión<sub>1</sub></i>	<i>expresión<sub>2</sub></i>	expresiones compatibles C++
<i>operador_relacional</i>		un operador de la tabla 3.7

Los operadores relacionales se usan normalmente en sentencias de selección (`if`) o de iteración (`while`, `for`), que sirven para comprobar una condición. Utilizando operadores relacionales se realizan

operaciones de igualdad, desigualdad y diferencias relativas. La Tabla 3.7 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: `char`, `int`, `float`, `double`, etc.

Cuando se utilizan los operadores en una expresión, el operador relacional produce un 0, o un 1, dependiendo del resultado de la condición. 0 se devuelve para una condición *falsa*, y 1 se devuelve para una condición *verdadera*. Por ejemplo, si se escribe

```
c = 3 < 7;
```

la variable `c` se pone a 1, dado que como 3 es menor que 7, entonces la operación `<` devuelve un valor de 1, que se asigna a `c`.

### Precaución

Un error típico, incluso entre programadores experimentales, es confundir el operador de asignación (`=`) con el operador de igualdad (`==`).

**Tabla 3.7.** Operadores relacionales de C++.

Operador	Significado	Ejemplo
<code>==</code>	<i>Igual a</i>	<code>a == b</code>
<code>!=</code>	<i>No igual a</i>	<code>a != b</code>
<code>&gt;</code>	<i>Mayor que</i>	<code>a &gt; b</code>
<code>&lt;</code>	<i>Menor que</i>	<code>a &lt; b</code>
<code>&gt;=</code>	<i>Mayor o igual que</i>	<code>a &gt;= b</code>
<code>&lt;=</code>	<i>Menor o igual que</i>	<code>a &lt;= b</code>

### Ejemplo

- Si `x`, `a`, `b` y `c` son de tipo `double`, `número` es `int` e `inicial` es de tipo `char`, las siguientes expresiones booleanas son válidas:

```
x < 5.75
b * b >= 5.0 * a * c
numero == 100
inicial != '5'
```

- En datos numéricos, los operadores relacionales se utilizan normalmente para comparar. Así, si

```
x = 3.1
```

la expresión

```
x < 7.5
```

produce el valor `true`. De modo similar si

```
numero = 27
```

la expresión

```
numero == 100
```

produce el valor *false*.



- Los caracteres se comparan utilizando los códigos numéricos. (Véase Apéndice D, código ASCII, en página web del libro).

'A' < 'C' es verdadera (*true*), ya que A es el código 65 y es menor que el código 67 de C.  
 'a' < 'c' es verdadera (*true*): a (código 97) y b (código 99)  
 'b' < 'B' es false (*false*) ya que b (código 98) no es menor que B (código 66).

Los operadores relacionales tienen menor prioridad que los operadores aritméticos, y asociatividad de izquierda a derecha. Por ejemplo,

$m + 5 \leq 2 * n$  equivale a  $(m + 5) \leq (2 * n)$

Los operadores relacionales permiten comparar dos valores. Así, por ejemplo, (*if* significa *si*, se verá en el Capítulo 4)

```
if (Nota_asignatura < 9)
```

comprueba si *Nota\_asignatura* es menor que 9. En caso de desear comprobar si la variable y el número son iguales, entonces utilizar la expresión

```
if (Nota_asignatura == 9)
```

Si por el contrario, se desea comprobar si la variable y el número no son iguales, entonces utilice la expresión

```
if (Nota_asignatura != 9)
```

### 3.6. OPERADORES LÓGICOS

Además de los operadores matemáticos, C++ tiene también *operadores lógicos*. Estos operadores se utilizan con expresiones para devolver un valor verdadero (cualquier entero distinto de cero) o un valor falso (0). Los operadores lógicos se denominan también *operadores booleanos*, en honor de George Boole, creador del álgebra de Boole.

Los operadores lógicos de C++ son: **not (!)**, **and (&&)** y **or (||)**. El operador lógico **!** (**not**, *no*) produce *falso* si su operando es *verdadero* (distinto de cero) y viceversa. El operador lógico **&&** (**and**, *y*) produce verdadero *sólo* si ambos operandos son *verdadero* (no cero); si cualquiera de los operandos es falso produce *falso*. El operador lógico **||** (**or**, *o*) produce verdadero si cualquiera de los operandos es verdadero (distinto de cero) y produce falso sólo si ambos operandos son falsos. La Tabla 3.8 muestra los operadores lógicos de C++.

**Tabla 3.8.** Operadores lógicos.

Operador	Operación lógica	Ejemplo
Negación (!)	No lógica	!(x >= y)
y disfunción lógica (&&)	operando_1 && operando_2	m < n && i > j
o disfunción lógica (  )	operando_1    operando_2	m = 5    n != 10

**Tabla 3.9.** Tabla de verdad del operador lógico NOT (!).

Operando (a)	NOT a (!a)
Verdadero (1)	Falso (0)
Falso (0)	Verdadero (1)

**Tabla 3.10.** Tabla de verdad del operador lógico AND (&&).

Operandos		
A	B	a && b
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Falso (0)
Falso (0)	Verdadero (1)	Falso (0)
Falso (0)	Falso (0)	Falso (0)

Los valores booleanos (bool) son verdadero y falso.  
true y false son constantes predefinidas de tipo bool.

**Tabla 3.11.** Tabla de verdad del operador lógico OR(||).

Operandos		
A	B	a    b
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Verdadero (1)
Falso (0)	Verdadero (1)	Verdadero (1)
Falso (0)	Falso (0)	Falso (0)

Al igual que los operadores matemáticos, el valor de una expresión formada con operadores lógicos depende de: (a) el operador y (b) sus argumentos. Con operadores lógicos existen sólo dos valores posibles para expresiones: *verdadero* y *falso*. La forma más usual de mostrar los resultados de operaciones lógicas es mediante las denominadas *tablas de verdad*, que muestran cómo funcionan cada uno de los operadores lógicos (Tablas 3.9, 3.10 y 3.11).

## Ejemplos

```
!(7 == 5)
(aNum > 5) && (Nombre == "Mortimer")
(bNum > 3) || (Nombre == "Mortimer")
```

Los operadores lógicos se utilizan en expresiones condicionales y mediante sentencias if, while o for, que se analizarán en capítulos posteriores. Así, por ejemplo, la sentencia if (*si la condición es verdadera/falsa...*) se utiliza para evaluar operadores lógicos.

```
1. if ((a < b) && (c > d))
    {
        cout << "Los resultados no son válidos";
    }
```

Si la variable *a* es menor que *b* y, al mismo tiempo, *c* es mayor que *d*, entonces visualizar el mensaje: Los resultados no son válidos.

```
2. if ((ventas > 50000) || (horas < 100))
    {
        prima = 100000;
    }
```

Si la variable *ventas* es mayor 50000 o bien la variable *horas* es menor que 100, entonces asignar a la variable *prima* el valor 100000.

```
3. if (!(ventas < 2500))
    {
        prima = 12500;
    }
```

En este ejemplo, si *ventas* es mayor que o igual a 2500, se inicializará *prima* al valor 12500.

El operador `!` tiene prioridad más alta que `&&`, que a su vez tiene mayor prioridad que `||`. La asociatividad es de izquierda a derecha.

La precedencia de los operadores es: los operadores matemáticos tienen precedencia sobre los operadores relacionales, y los operadores relacionales tienen precedencia sobre los operadores lógicos.

La siguiente sentencia:

```
if (ventas < sal_min * 3 && anios > 10 * iva)...
```

equivale a

```
if ((ventas < (sal_min * 3)) && (anios > (10 * iva)))...
```

### 3.6.1. Evaluación en cortocircuito

En C++ los operandos de la izquierda de `&&` y `||` se evalúan siempre en primer lugar; si el valor del operando de la izquierda determina de forma inequívoca el valor de la expresión, el operando derecho no se evalúa. Esto significa que si el operando de la izquierda de `&&` es falso o el de `||` es verdadero, el operando de la derecha no se evalúa. Esta propiedad se denomina *evaluación en cortocircuito* y se debe a que si *p* es falso, la condición *p* && *q* es falsa, con independencia del valor de *q*, y de este modo C++ no evalúa *q*. De modo similar, si *p* es verdadera, la condición *p* || *q* es verdadera, con independencia del valor de *q*, y C++ no evalúa a *q*.

---

#### Ejemplo 3.6

Supongamos que se evalúa la expresión

```
(x >= 0.0) && (sqr(x) >= 2)
```

---

Dado que en una operación lógica `Y` (`&&`) si el operando de la izquierda ( $x \geq 0.0$ ) es falso (*x es negativo*), la expresión lógica se evalúa a falso y, en consecuencia, no es necesario evaluar el segundo operando. En el ejemplo anterior la expresión evita calcular la raíz cuadrada de números (*x*) negativos.

La evaluación en **cortocircuito** tiene dos beneficios importantes:

1. Una expresión *booleana* se puede utilizar para *guardar* una operación potencialmente insegura en una segunda expresión *booleana*.
2. Se puede ahorrar una considerable cantidad de tiempo en la evaluación de condiciones complejas.

---

### Ejemplo 3.7

1. *Los beneficios anteriores se aprecian en la expresión booleana*

```
(n != 0) && (x < 1.0 / n)
```

ya que no se puede producir un error de división por cero al evaluar esta expresión, pues si *n* es 0, entonces la primera expresión

```
n != 0
```

es falsa y la segunda expresión

```
x < 1.0 / n
```

no se evalúa.

De modo similar, tampoco se producirá un error de división por cero al evaluar la condición

```
(n == 0) || (x >= 5.0 / n)
```

ya que si *n* es 0, la primera expresión

```
n == 0
```

es verdadera y entonces no se evalúa la segunda expresión

```
(x >= 5.0 / n)
```

### Aplicación

Dado el test condicional

```
if ((7 > 5) || (ventas < 30) && (30 != 30))...
```

C++ examina sólo la primera condición, ( $7 > 5$ ), ya que como es verdadera, la operación lógica `||` (o) será verdadera, sea cual sea el valor de la expresión que le sigue.

Otro ejemplo es el siguiente:

```
if ((8 < 4) && (edad > 18) && (letra_inicial == 'Z'))...
```

En este caso, C++ examina la primera condición y su valor es falso; por consiguiente, sea cual sea el valor que sigue al operador `&&`, la expresión primitiva será falsa y toda la subexpresión a la derecha de ( $8 < 4$ ) no se evalúa por C++.

Por último, en la sentencia

```
if ((10 > 4) || (num = 0)) ...
```

la sentencia `num = 0` nunca se ejecutará.

---

### 3.6.2. Asignaciones *booleanas* (lógicas)

Las sentencias de asignación se pueden escribir de modo que se puede dar un valor de tipo `bool` o una variable `bool`.

#### Ejemplo

<code>MayorDeEdad = true;</code>	<i>asigna el valor true MayorDeEdad</i>
<code>MayorDeEdad = (x == y);</code>	<i>asigna el valor de <code>x == y</code> a MayorDeEdad</i>
<code>MayorDeEdad</code>	<i>cuando <code>x</code> e <code>y</code> son iguales, MayorDeEdad es true y si no false.</i>

---

#### Ejemplo 3.8

Las sentencias de asignación siguientes asignan valores a los dos tipos de variables `bool`, `rango` y `es_letra`. La variable `rango` es verdadera (`true`) si el valor de `n` está en el rango `-100` a `100`; la variable `es_letra` es verdadera si `car` es una letra mayúscula o minúscula

```
rango    = (n > -100) && (n < 100);
esletra  = (('A' <= car) && (car <= 'Z')) ||
          (('a' <= car) && (car <= 'z'));
```

La expresión de *a* es `true` si `n` cumple las condiciones expresadas (`n` mayor de `-100` y menor de `100`); en caso contrario es `false`. La expresión *b* utiliza los operadores `&&` y `||`. La primera subexpresión (antes de `||`) es `true` si `car` es una letra mayúscula; la segunda subexpresión (después de `||`) es `true` si `car` es una letra minúscula. En resumen, `esLetra` es verdadera (`true`) si `car` es una letra, y `false` en caso contrario.

---

### 3.6.3. Expresiones booleanas

La mayoría de las sentencias de bifurcación que se verán en los Capítulos 4 y 5 están controladas por expresiones booleanas o lógicas. Una *expresión booleana* es cualquier expresión que es verdadera o falsa. La forma más simple de una expresión booleana consta de dos expresiones, tales como números o variables que se comparan con uno de los operadores de comparación o relacionales. Observe que algunos de los operadores se escriben con dos símbolos, `==`, `!=`, `<=`, `>=`, etc. y también, que se utiliza un signo doble igual para el signo igual y que se utilizan dos símbolos `!=` para el signo no igual (Tabla 3.7).

#### Operador "and" &&

Se pueden combinar dos comparaciones utilizando el operador "and" que se escribe `&&`.

---

#### Ejemplo

La expresión `(2 < x) && (x < 7)` es verdad si `x` es mayor que 2 y `x` es menor que 7, es caso contrario es falsa

```
(exp_booleana_1) && (expresión_booleana_2)
```

**Aplicación**

```
if (nota > 0) && (nota < 10)
    cout << "la calificación está entre 0 y 10.\n";
else
    cout << "la calificación no está entre 0 y 10.\n";
```

si el valor de nota es mayor que 0 y menor de 10, se ejecuta la primera sentencia cout, en caso contrario se ejecuta la segunda sentencia cout.

**Ejemplo**

*Combinación de dos operadores "or" (acentos ||) en C++*

```
(y < 0) || (y > 12)
```

es verdad si y es menor que 0 o y es mayor que 12.

**Ejemplo**

*Se puede negar cualquier expresión booleana utilizando el operador !. Si desea negar una expresión booleana, sitúe la expresión entre paréntesis y ponga el operador ! delante de ella.*

!(x < y)	significa	"x es no menor que y"
!(x < y)	es equivalente a	x >= y

**Regla**

*Si desea utilizar una cadena de desigualdades tales como  $x < z < y$ , será mejor utilice la siguiente expresión  $(x < z) \&\& (z < y)$ .*

**Operador or**

Se puede formar una expresión booleana «or» combinando dos expresiones booleanas y utilizando el operador or (||).

```
(expresión_booleana_1) || (expresión_booleana_2)
```

**Aplicación**

```
if ((x == 1) || (x == y))
    cont << "x es 1 o x es igual a y.\n";
else
    cont << "x ni es 1 ni igual a y.\n";
```

### 3.7. OPERADORES DE MANIPULACIÓN DE BITS

Una de las razones por las que C y C++ se han hecho tan populares en computadoras personales es que el lenguaje ofrece muchos operadores de manipulación de bits a bajo nivel.

Los operadores de manipulación o tratamiento de bits (*bitwise*) ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Estas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador.

Cada operador de manipulación de bits realiza una operación lógica bit a bit sobre datos internos. Los operadores de manipulación de bits se aplican sólo a variables y constantes `char`, `int` y `long`, y no a datos en coma flotante. Dado que los números binarios constan de 1,s y 0,s (denominados *bits*), estos 1 y 0 se manipulan para producir el resultado deseado para cada uno de los operadores.

Las siguientes tablas de verdad describen las acciones que realizan los diversos operadores sobre los diversos patrones de bit de un dato `int` (`char` o `long`).

**Tabla 3.12.** Operadores lógicos bit a bit.

Operador	Operación
&	<b>Y (AND)</b> <i>lógica bit a bit</i>
	<b>O (OR)</b> <i>lógica (inclusiva) bit a bit</i>
^	<b>O (XOR)</b> <i>lógica (exclusiva) bit a bit (OR exclusive, XOR)</i>
~	<i>Complemento a uno (inversión de todos los bits)</i>
<<	<i>Desplazamiento de bits a izquierda</i>
>>	<i>Desplazamiento de bits a derecha</i>

A&B == C	A  B == C	A^B == C	A	~A
0&0 == 0	0  0 == 0	0^0 == 0	1	0
0&1 == 0	0  1 == 1	0^1 == 1	0	1
1&0 == 0	1  0 == 1	1^0 == 1		
1&1 == 1	1  1 == 1	1^1 == 0		

#### Ejemplo

- Si se aplica el operador & de manipulación de bits a los números 9 y 14, se obtiene un resultado de 8. La Figura 3.2 muestra cómo se realiza la operación.
- |                 |   |      |      |      |               |
|-----------------|---|------|------|------|---------------|
| (&) 0x3A6B      | = | 0011 | 1010 | 0101 | 1011          |
| 0x00F0          | = | 0000 | 0000 | 1111 | 0000          |
| <hr/>           |   |      |      |      |               |
| 0x3A6B & 0x00F0 | = | 0000 | 0000 | 0101 | 0000 = 0x0050 |
- |                 |   |      |      |      |               |
|-----------------|---|------|------|------|---------------|
| (  ) 152 0x0098 | = | 0000 | 0000 | 1001 | 1000          |
| 5 0x0005        | = | 0000 | 0000 | 0000 | 0101          |
| <hr/>           |   |      |      |      |               |
| 152    5        | = | 0000 | 0000 | 1001 | 1101 = 0x009d |
- |        |     |      |            |      |
|--------|-----|------|------------|------|
| (^)    | 83  | =    | 0101       | 0011 |
|        | 204 | =    | 1100       | 1100 |
| <hr/>  |     |      |            |      |
| 83^204 | =   | 1001 | 1101 = 157 |      |

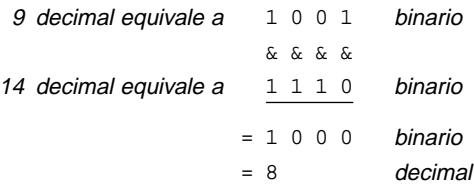


Figura 3.2. Operador & de manipulación de bits.

3.7.1. Operadores de asignación adicionales

Al igual que los operadores aritméticos, los operadores de asignación abreviados están disponibles también para operadores de manipulación de bits. Estos operadores se muestran en la Tabla 3.13.

Tabla 3.13. Operadores de asignación adicionales.

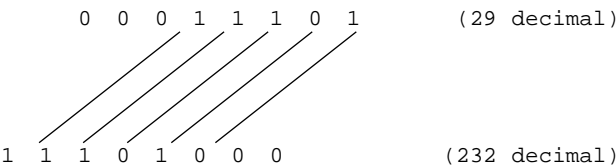
Símbolo	Uso	Descripción
<=	a <= b	Desplaza a a la izquierda b bits y asigna el resultado a a.
>>=	a >>= b	Desplaza a a la derecha b bits y asigna el resultado a a.
&=	a &= b	Asigna a a el valor a & b.
^=	a ^= b	Establece a a a ^ b.
=	a  = b	Establece a a a   b.

3.7.2. Operadores de desplazamiento de bits (>>, <<)

Equivalen a la instrucción SHR (>>) y SHL (<<) de los microprocesadores 80x86. Efectúa un desplazamiento a la derecha (>>) o a la izquierda (<<) de *n* posiciones de los bits del operando, siendo *n* un número entero. El número de bits desplazados depende del valor a la derecha del operador. Los formatos de los operadores de desplazamiento son:

- 1. valor << numero\_de\_bits;
- 2. valor >> numero\_de\_bits;

El *valor* puede ser una variable entera o carácter, o una constante. El *numero\_de\_bits* determina cuántos bits se desplazarán. La Figura 3.2 muestra lo que sucede cuando el número 29 (binario 00011101) se desplaza a la izquierda tres bits con un desplazamiento a la izquierda bit a bit (<<).



Después de tres desplazamientos

Figura 3.2. Desplazamiento a la izquierda tres posiciones de los bits del número binario equivalente a 29.



Supongamos que la variable `num1` contiene el valor 25, si se desplaza tres posiciones (`num << 3`), se obtiene el nuevo número 104 (11001000 en binario).

```
int num1 = 25;           //00011001 binario
int despl, desp2;

despl = num1 << 3;       //11001000 binario
```

### 3.7.3. Operadores de direcciones

Son operadores que permiten manipular las direcciones de los objetos:

```
*expresión
&valor_i (lvalue)
objeto.miembro
puntero_hacia_objeto -> miembro
```

**Tabla 3.14.** Operadores de direcciones.

Operador	Acción
*	Lee o modifica el valor apuntado por la expresión. Se corresponde con un puntero y el resultado es del tipo apuntado.
&	Devuelve un puntero al objeto utilizado como operando, que debe ser un <i>lvalue</i> (variable dotada de una dirección de memoria). El resultado es un puntero de tipo idéntico al del operando.
.	Permite acceder a un miembro de un objeto agregado (unión, estructura o clase).
->	Accede a un miembro de un objeto agregado (unión, estructura o clase) apuntado por el operando de la izquierda.

## 3.8. OPERADOR CONDICIONAL

El operador condicional, `?:`, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas.

Al ser un operador ternario requiere tres operandos. El operador *condicional* se utiliza para reemplazar a la sentencia `if-else` lógica en algunas situaciones. El formato del operador condicional es:

```
expresión_c ? expresión_v : expresión_f;
```

Se evalúa `expresión_c` y su valor (cero = falso, distinto de cero = verdadero) determina cuál es la expresión a ejecutar; si la condición es verdadera se ejecuta `expresión_v` y si es falsa se ejecuta `expresión_f`.

La Figura 3.3 muestra el funcionamiento del operador condicional.

Otros ejemplos del uso del operador `?:` son:

```
n >= 0 ? 1 : -1    //1 si n es positivo, -1 si es negativo

m >= n ? m : n     //devuelve el mayor valor de m y n
```

```
(ventas > 150000) ? comisión = 100 : comisión = 0;
```

<i>si ventas es mayor que 150.000 es</i>	<i>si ventas no es mayor mayor que 150.000 ejecuta:</i>	<i>ejecutar</i>
comision = 100	comision = 0	

Figura 3.3. Formato de un operador condicional.

La precedencia de ? y : es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es de derechas.

### 3.9. OPERADOR COMA

El *operador coma* ( , ) de secuencia o evaluación permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha. La expresión más a la derecha determina el resultado global. El uso del operador coma es como sigue:

*expresión1, expresión2, expresión3, ..., expresiónn*

Cada expresión se evalúa comenzando desde la izquierda y continuando hacia la derecha. Por ejemplo, en

```
int i = 10, j = 25;
```

dado que el operador coma se asocia de izquierda a derecha, la primera variable está declarada e inicializada antes que la segunda variable. Otros ejemplos son:

<i>i++, j++</i>	<i>equivale a</i>	<i>i++; j++;</i>
<i>i++, j++, k++</i>	<i>equivale a</i>	<i>i++; j++; k++;</i>

El operador coma tiene prioridad de todos los operadores C++, y se asocia de izquierda a derecha.

El resultado de la expresión global se determina por el valor de *expresión*. Por ejemplo,

```
int i, j, resultado;
int i;
resultado = j = 10; i = j; i++;
```

El valor de esta expresión es 11. En primer lugar, a j se asigna el valor 10, a continuación a i se asigna el valor de j. Por último, i se incrementa a 11.

La técnica del operador coma permite operaciones interesantes.

```
i = 10;
j = (i = 12, i + 8);
```

Cuando se ejecute la sección de código anterior, j vale 20, ya que i vale 10 en la primera sentencia, en la segunda toma i el valor 12 y al sumar i + 8 resulta 20.



### 3.11. EL OPERADOR SIZEOF

Con frecuencia, su programa necesita conocer el tamaño en bytes de un tipo de dato o variable. C++ proporciona el operador `sizeof`, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, array, registro, etc.). El formato del operador es

```
sizeof(nombre_variable tipo_dato)
sizeof expresión
```

---

#### Ejemplo 3.9

*Si se supone que el tipo `int` consta de cuatro bytes y el tipo `double` consta de ocho bytes, las siguientes expresiones proporcionarán los valores 1, 4 y 8 respectivamente.*

```
sizeof (char)
sizeof (unsigned int)
sizeof (double).
```

El operador **sizeof** se puede aplicar también a expresiones. Así se puede escribir

```
cout << "La variable K es " << sizeof k << " bytes";
cout << "La expresión a + b es " << sizeof (a + b) << " bytes"
```

El operador `sizeof` es un operador unitario, ya que opera sobre un valor único. Este operador produce un resultado que es el tamaño, en bytes, del dato o tipo de dato especificados. Debido a que la mayoría de los tipos de datos y variables requieren diferentes cantidades de almacenamiento interno en computadores diferentes, el operador `sizeof` permite consistencia de programas en diferentes tipos de computadores.

El operador `sizeof` se denomina también *operador en tiempo de compilación*, ya que, en tiempo de compilación, el compilador sustituye cada ocurrencia de `sizeof` en su programa con un valor entero sin signo (unsigned). El operador `sizeof` se utiliza en programación avanzada.

---

#### Ejercicio 3.1

*Suponga que se desea conocer el tamaño, en bytes, de variables de coma flotante de su computadora. El siguiente programa realiza esta tarea:*

```
// Nombre del archivo LONGBYTE.CPP
// Imprime el tamaño de valores de coma flotante

#include <iostream>
using namespace std;

int main()
{
    cout << "El tamaño de variables de coma flotante";
    cout << "de esta computadora es:" << sizeof(float)
        << '\n';
    return 0;
}
```

Este programa producirá diferentes resultados en diferentes clases de computadores. Compilando este programa bajo C++, el programa produce la salida siguiente:

```
El tamaño de variables de coma flotante de esta computadora es: 4
```

---

### 3.12. CONVERSIONES DE TIPOS

En C++ se puede convertir un valor de un tipo en un valor de otro tipo. Tal acción se denomina *conversión de tipos*. También se puede definir sus propios nombres de tipos utilizando la palabra reservada `typedef`. Esta característica es necesaria en numerosas expresiones aritméticas y asignaciones, debido a que las operadoras binarias requieren operandos del mismo tipo. Si no es este el caso, el tipo de un operando debe de ser convertido para coincidir con el otro operando. Cuando se llama a una función, el tipo de argumento debe coincidir con el tipo del parámetro; si no es así, el argumento se convierte de modo que coincidan sus tipos. C++ tiene operadores de conversión (`cast`) que le permiten definir una conversión explícitamente, o bien el compilador puede convertir automáticamente el tipo para su programa. C++ realiza muchas conversiones automáticamente:

- C++ convierte valores cuando se asigna un valor de un tipo aritmético a una variable de otro tipo aritmético.
- C++ convierte valores cuando se combinan tipos mezclados en expresiones.
- C++ convierte valores cuando se pasan argumentos a funciones.

En la práctica se realizan dos tipos de conversiones *implícitas* (ejecutadas automáticamente, algunas de las citadas anteriormente), o *explícitas* (solicitadas especialmente por el programador C++).

#### **Conversión implícita**

Los tipos fundamentales (básicos) pueden mezclarse libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operadores de tipo de precisión más baja (más pequeña) se convierten en tipos de más alta precisión. Esta conversión se denomina *promoción integral*.

Cuando se intenta realizar operaciones con tipos de datos de diferente precisión, el compilador define automáticamente —sin intervención del programador— de modo que convierte los operandos a un tipo común antes de realizar cualquier operación.

Por ejemplo, considere

```
int n = 0;
pi = 3.141592 + 3;
```

esta asignación a `pi`, de una suma de un entero y de uno real (`float`), dos tipos diferentes, puede producir como valor de `pi`, 6, o bien 6.141592 que sería lo lógico. El compilador realiza conversiones entre los tipos aritméticos que se definen para preservar, si es posible, la precisión. Lo normal sería que el entero se convierta en coma flotante y entonces se obtendrá el resultado 6.141592 que es de tipo `float` (`double`). El siguiente paso que realizará el compilador será asignar el valor `double` a `n`, que es un entero. En el caso de la operación de asignación, prevalece el tipo del operando de la izquierda ya que no es posible cambiar el tipo del objeto que está en el lado izquierdo. Cuando los tipos izquierdo y derecho de una asignación difieren, el lado derecho se convierte al tipo del lado izquierdo. En el ejemplo anterior, si el valor real (`double`) se convierte a `int` el compilador producirá un mensaje de error similar a:

```
warning = assignment to 'int' from 'double'
```

que alertará al programador.

## Reglas

1. En una expresión aritmética, los operadores binarios requieren operandos del mismo tipo. Si no coinciden, el tipo de un operando se debe convertir al tipo del otro operando.
2. En el caso de llamada a una función, el tipo de argumento debe corresponder con el tipo de parámetro; si no es así el argumento se convierte para que coincida su tipo.
3. C++ tiene operadores de *moldeado* de tipos (*moldes*, «*cast*»), que permiten definir una conversión de tipos, explícitamente, o bien el compilador puede convertir automáticamente el tipo si lo desea el programador.

## Conversiones aritméticas

El lenguaje define un conjunto de conversiones entre los tipos incorporados. Entre éstas las más típicas son las *conversiones aritméticas* que aseguran que los operandos de un operador binario, tal como un operador aritmético o lógico, se convierten a un tipo común antes de que se evalúe el operador. Este tipo común es el tipo del resultado de la expresión. Las reglas definen una jerarquía de conversiones de tipos en la cual, los operandos, se convierten a los tipos de mayor precisión en las expresiones. Es decir, si un operando es del tipo `long double`, el otro operando también se convierte a `long double` sea cual sea su tipo.

La Tabla 3.15 considera el orden de los tipos de datos a efectos de conversión automática de tipos con indicación del tipo de dato de mayor precisión (el tamaño ocupado en bytes por el tipo de datos).

**Tabla 3.15.** Orden de los tipos de datos.

Tipo de dato	Precisión	Orden
<code>long double</code>	10 bytes	Más alta (19 dígitos)
<code>double</code>	8 bytes	15 dígitos
<code>float</code>	4 bytes	7 dígitos
<code>long</code>	4 bytes	No aplicable
<code>int</code>	4 bytes	No aplicable
<code>short</code>	2 bytes	No aplicable
<code>char</code>	1 bytes	Más baja
<code>bool</code>	No aplicable	

La *promoción de tipos* es una conversión de tipos aritméticos de modo que se convierte un tipo «más pequeño» a un tipo «mayor».

Por ejemplo, si un operando es de tipo `long double`, entonces el otro operando se convierte a `long double` independientemente del tipo que contiene. Cada uno de los tipos integrales más pequeños que `int` (`char`, `signed char`, `unsigned char`, `short` y `unsigned short`) se promueven a `int`, siempre que todos los valores posibles quepan en un `int`; en caso contrario se promueve a `unsigned int`. En el caso de los tipos lógicos (`bool`) se promueven a `int`, de modo que un valor *falso* se promueve a cero y un valor *verdadero* se promueve a uno.

El compilador ANSI/ISO C++ acepta definiciones de números con signo (`signed`) y sin signo (`unsigned`). En estos casos cuando un valor `unsigned` está implicado en una expresión, las reglas de conversión se definen para preservar el valor de los operandos, aunque en estos casos los tamaños relativos dependen de la máquina.

## Ejemplo

```

bool    bandera;
char    car;
short   num_s;
int     num_i;
long    num_l;
float   num_f;
double  num_d;
long double num_ld;
//algunas conversiones automáticas de tipos
num_d + num_i;    // se convierte a double
num_d + num_f;    // se convierte a double
num_i + num_d;    // se convierte a double
num_s + num_f;    // se convierte a int y luego a float
bandera = num_d;  // si num_d es 0, bandera es falso
car + num_f;      // car se convierte a int y después a float

```

## Conversiones explícitas

Una *conversión de tipos “cast”* modelado de tipos (*moldes*) es una conversión explícita de tipos. C++ ofrece diferentes formas de modelar una expresión a un tipo diferente.

### Notación compatible con C y versiones antiguas de C++

- `tipo (expr)` //notación conversión de tipos, estilo función
- `(tipo) expr;` //notación conversión estilo C

### Notación compatible con estandar ANSI/ISO C++

C++ soporta los siguientes operadores de *molde*:

- `const_cast <tipo> (expr)`
- `dynamic_cast <tipo> (expr)`
- `reinterpret_cast <tipo> (expr)`
- `static_cast <tipo> (expr)`

## Ejemplos

1. `(float)i` // convierte i a float
2. `float (i)` // convierte i a float
3. `varCar = static_cast <char> (varEntera);`

tipo al que se  
convierte

variable  
a convertir

varEntera se convierte a char antes de ser asignada a varCar.

4. La conversión con C y compiladores antiguos de C++ de:

```
varCar = static_cast <char> (varEntera);
```

también se puede hacer con

```
varCar = (char) varEntera;
```

o alternativamente

```
varCar = char (varEntera);
```

## Resumen

El compilador automáticamente realizará las conversiones de acuerdo a ciertas reglas. Para proteger la información, normalmente, sólo se permiten conversiones de menor tamaño (menos ricas) a mayor tamaño (más ricas) y no se pierde información; esta operación se llama *promoción*; por ejemplo, de `char` a `int` (no se pierde información). En otros casos se pierde información, como la conversión de `float` a `int`.

Si el compilador no puede efectuar la conversión por dudas en la misma, el programador puede hacerla explícita.

## Ejemplo

```
int i = 7, j = 2;
double m = i/j;           // truncamiento m == 3.0
m = (double) (i/j);       // m == (double) 3 == 3.0
m = (double) i/j;         // m == 3.5 i se convierte a double
m = i/(double) j;         // bien ahora j se convierte a double
m = (double) i/ (double) j; // correcto
```

## RESUMEN

Este capítulo examina los siguientes temas:

- Concepto de operadores y expresiones.
- Operadores de asignación: básicos y aritméticos.
- Operadores aritméticos, incluyendo +, -, \*, / y % (módulos).
- Operadores de incremento y decremento. Estos operadores se aplican en formatos *pre* (anterior) y *post* (posterior). C++ permite aplicar estos operadores a variables que almacenan caracteres, enteros e incluso números en coma flotante.
- Operadores relacionales y lógicos que permiten construir expresiones lógicas. C++ no soporta un tipo lógico (*boolean*) predefinido y en su lugar considera 0 (cero) como *falso* y cualquier valor distinto de cero como *verdadero*.
- Operadores de manipulación de bits que realizan operaciones bit a bit (*bitwise*), AND, OR, XOR y NOT. C++ soporta los operadores de desplazamiento de bits << y >>.
- Operadores de asignación de manipulación de bits que ofrecen formatos abreviados para sentencias simples de manipulación de bits.
- El operador coma, que es un operador muy especial, separa expresiones múltiples en las mismas sentencias y requiere que el programa evalúe totalmente una expresión antes de evaluar la siguiente.
- La expresión condicional, que ofrece una forma abreviada para la sentencia alternativa simple-doble *if-else*, que se estudiará en el Capítulo 4.
- Operadores especiales: `()`, `[]` y `::`.
- Conversión de tipos (*typecasting*) o moldeado, que permite forzar la conversión de tipos de una expresión.
- Reglas de prioridad y asociatividad de los diferentes operadores cuando se combinan en expresiones.
- El operador `sizeof`, que devuelve el tamaño en bytes de cualquier tipo de dato o una variable.