GCC C++: Operators

Introduction

An *operator* is a symbol that specifies which operation to perform in a statement or expression. An *operand* is one of the inputs of an operator. For example, in expression `b + c`, + is the operator and `b` and `c` are the operands.

C++ operators in specify an evaluation to be performed on one of the following:

- one operand (*unary* operator)
- two operands (*binary* operator)
- three operands (*ternary* operator)

Note that the word "binary" in this context does not relate to the binary numeral system – it specifies the number of operands as two.

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, expression `b + c * d` is evaluated as `b + (c * d)` because the `*` operator has higher precedence than the `+` operator: the `*` operator is evaluated first and the + operator second.

When an expression contains multiple operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. If the operators are left-associative, operators are evaluated from left to right: the left operator is evaluated first and the right operator last; if they are right-associative, operators are evaluated from right to left.

Precedence and associativity can be controlled using *parentheses* `()`. In the precedence example above, expression `b + c * d` is evaluated as `b + (c * d)`, where the `*` operator is evaluated first and the + operator second. Writing the expression as `( b + c ) * d` causes the + operator to be evaluated first and the `*` operator second.

Operators follow a strict precedence, which defines the evaluation order of expressions containing these operators. Operators with the same precedence associate with either the expression on their left or the expression on their right, depending on their associativity. The following table shows the precedence and associativity of C++ operators (from highest to lowest precedence).

Precedence	Operator	Description	Associativity
primary	++	postfix increment	left to right
	--	postfix decrement	
unary	++	prefix increment	right to left
	--	prefix decrement	
	+	unary plus	
	-	unary minus (two's complement)	

	~	complement (one's complement)	
	!	logical not	
multiplicative	*	multiplication	left to right
	/	division	
	%	modulus	
additive	+	addition	left to right
	-	subtraction	
shift	<<	left shift	left to right
	>>	right shift	
relational	<	less than	left to right
	>	greater than	
	<=	less than or equal	
	>=	greater than or equal	
equality	==	equality	left to right
	!=	inequality	
and	&	and	left to right
exclusive-or	^	xor (exclusive-or)	left to right
or	|	or (inclusive-or)	left to right
conditional and	&&	conditional and	left to right
conditional or	||	conditional or	left to right
assignment	=	assignment	right to left
	+=	assignment by sum	
	-=	assignment by difference	
	*=	assignment by product	
	/=	assignment by dividend	
	%=	assignment by remainder	
	<<=	assignment by left-shift	
	>>=	assignment by right-shift	
	&=	assignment by and	
	^=	assignment by xor	
	|=	assignment by or	

This article demonstrates use of each of the above operators.

Concepts

operator

Programming languages generally have a set of *operators* that are similar to operators in mathematics.

An operator is a symbol that specifies which operation to perform in a statement or expression.

The *operator* program demonstrates the commonly used operators in C++.

operand

An *operand* is one of the inputs of an operator. For example, in expression `b + c`, + is the operator and `b` and `c` are the operands.

All operators used in the *operator* program have either one or two operands.

expression

An *expression* is a programming language construct that evaluates to some quantity.

For example, `b + c` is an expression that evaluates to the sum of `b` plus `c`. If `b` = 2 and `c` = 3, expression `b + c` evaluates to 5 (2 + 3 = 5).

Use of operators within expressions is demonstrated throughout the *operator* program.

assignment

To *assign* is to set or re-set a value denoted by an identifier.

An *assignment statement* assigns a value to an entity. If the entity is a variable, assignment statements allow it to contain different values at different times during program execution. If the entity is a constant, an assignment statement can often be used to initialise its value.

Throughout the *operator* program, expressions containing operators are evaluated and assigned to variables.

precedence

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, expression `b + c * d` is evaluated as `b + (c * d)` because the `*` operator has higher precedence than the + operator: the `*` operator is evaluated first and the + operator second.

The *operator* program demonstrates operator precedece.

associativity

When an expression contains multiple operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. If the operators are left-associative, operators are evaluated from left to right: the left operator is evaluated first and the right operator last; if they are right-associative, operators are evaluated from right to left.

The *operator* program demonstrates associativity for operators having the same precedence.

binary

The *binary numeral system* (binary system), or *base-2 number system*, represents numeric values using two symbols, usually 0 and 1. Instead of using digits 0 to 9 as the decimal system does, the binary system uses digits 0 to 1 to represent numbers.

This article uses binary representation to explain the functionality of some operators, for example shift and bitwise operators.

one's complement

The *one's complement* of a binary number is the bitwise complement of the number, or the number with each bit complemented (set to `1` if it is `0` or to `0` if it is `1`).

For example, the one's complement of binary `0101` (decimal 5) is `1010`.

Adding any integer to its one's complement evaluates to an integer consisting entirely of set bits (`1111`), which is the one's complement of zero and from a one's complement perspective may be used to represent negative zero.

two's complement

The *two's complement* of a binary number is the one's complement plus one, and represents the negative value of the number.

For example, the two's complement of binary `0101` (decimal 5) is `1010 + 0001 = 1011`.

Adding any integer to its two's complement evaluates to zero (`0000`).

Source Code

The source code listing is as follows:

```
/*
  operator.cpp

  Operators.

  environment: language C++
               platform Windows console
*/

#include <stdio.h>

int main()
{
  // variables

  int  n0 , n1 , n2 , ne ;
  bool b0 , b1 ,      be ;

  // assignment 1

  printf( "assignment                " ) ;

  // =

  n0 = 5                                ;
  ne = n1 = 2                           ;
  printf( "%d, %d, %d" , n0 , n1 , ne ) ;

  b0 = true                        ;
  b1 = false                       ;
  printf( ", %d, %d" , b0 , b1 ) ;

  printf( "\n" ) ;

  // sign

  printf( "sign                      " ) ;

  // +

  ne = + n0           ;
  printf( "%d" , ne ) ;

  // -
```

```
ne = - n0               ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

// arithmetic

printf( "arithmetic                 " ) ;

// +

ne = n0 + n1          ;
printf( "%d" , ne ) ;

// -

ne = n0 - 2           ;
printf( ", %d" , ne ) ;

// *

ne = 5 * 2            ;
printf( ", %d" , ne ) ;

// /

ne = 5 / 2            ;
printf( ", %d" , ne ) ;

// %

ne = 5 % 2            ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

// shift

printf( "shift                      " ) ;

n0 = 4 ;
n1 = 2 ;

// <<

ne = n0 << 1          ;
printf( "%d" , ne ) ;

// >>
```

```
ne = n0 >> n1          ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

// logical

// logical: bitwise

printf( "logical: bitwise          " ) ;

n0 =  5 ;
n1 = 12 ;

// &

ne = n0 & n1         ;
printf( "%d" , ne ) ;

// |

ne = n0 | n1           ;
printf( ", %d" , ne ) ;

// ^

ne = n0 ^ n1           ;
printf( ", %d" , ne ) ;

// ~

ne = ~ n0              ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

// logical: boolean

printf( "         boolean          " ) ;

// &

be = b0 & b1         ;
printf( "%d" , be ) ;

// |

be = b0 | b1           ;
printf( ", %d" , be ) ;
```

```
// ^

be = b0 ^ b1           ;
printf( ", %d" , be ) ;

// !

be = ! b0              ;
printf( ", %d" , be ) ;

// &&

be = b0 && b1          ;
printf( ", %d" , be ) ;

// ||

be = b0 || b1          ;
printf( ", %d" , be ) ;

printf( "\n" ) ;

// assignment 2

printf( "assignment: arithmetic    " ) ;

n0 = 5 ;
n1 = 2 ;

// +=

ne =  n0             ;
ne += n1             ;
printf( "%d" , ne ) ;

// -=

ne =  n0               ;
ne -= 2                ;
printf( ", %d" , ne ) ;

// *=

ne =  5                ;
ne *= 2                ;
printf( ", %d" , ne ) ;

// /=
```

```
ne =  5                ;
ne /= 2                ;
printf( ", %d" , ne ) ;

// %=

ne =  5                ;
ne %= 2                ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

printf( "            shift         " ) ;

n0 = 4 ;
n1 = 2 ;

// <<=

ne =   n0            ;
ne <<= 1             ;
printf( "%d" , ne ) ;

// >>=

ne =   n0              ;
ne >>= n1              ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

printf( "            logical       " ) ;

n0 =  5    ;
n1 = 12    ;
b0 = true  ;
b1 = false ;

// &=

ne =  n0             ;
ne &= n1             ;
printf( "%d" , ne ) ;

be =  b0               ;
be &= b1               ;
printf( ", %d" , be ) ;

// |=
```

```
ne =  n0              ;
ne |= n1              ;
printf( ", %d" , ne ) ;

be =  b0              ;
be |= b1              ;
printf( ", %d" , be ) ;

// ^=

ne =  n0              ;
ne ^= n1              ;
printf( ", %d" , ne ) ;

be =  b0              ;
be ^= b1              ;
printf( ", %d" , be ) ;

printf( "\n" ) ;

// increment, decrement

printf( "increment, decrement      " ) ;

ne = 5 ;

// ++

ne ++                ;
printf( "%d" , ne ) ;

++ ne                ;
printf( ", %d" , ne ) ;

// --

n0 = ne --                      ;
printf( ", %d, %d" , ne , n0 ) ;

n0 = -- ne                      ;
printf( ", %d, %d" , ne , n0 ) ;

printf( "\n" ) ;

// relational

printf( "relational: equality      " ) ;

n0 = 1 ;
n1 = 2 ;
```

```
  n2 = 1 ;

  // ==

  be = n0 == n1        ;
  printf( "%d" , be ) ;

  be = n0 == n2          ;
  printf( ", %d" , be ) ;

  // !=

  be = n0 != n1          ;
  printf( ", %d" , be ) ;

  be = n0 != n2          ;
  printf( ", %d" , be ) ;

  printf( "\n" ) ;

  printf( "            greater/less  " ) ;

  // <

  be = n0 < n1         ;
  printf( "%d" , be ) ;

  be = n0 < n2           ;
  printf( ", %d" , be ) ;

  // >

  be = n0 > n1           ;
  printf( ", %d" , be ) ;

  be = n0 > n2           ;
  printf( ", %d" , be ) ;

  // <=

  be = n0 <= n1          ;
  printf( ", %d" , be ) ;

  be = n0 <= n2          ;
  printf( ", %d" , be ) ;

  // >=

  be = n0 >= n1          ;
  printf( ", %d" , be ) ;
```

```
be = n0 >= n2          ;
printf( ", %d" , be ) ;

printf( "\n" ) ;

printf( "precedence, associativity " ) ;

// precedence

// primary, unary, multiplicative, additive, shift

n0 = 10                                ;
ne = - ++ n0 + 29 - 5 * 2 << 2 - 1 ;
printf( "%d, %d" , ne , n0 )           ;

n0 = 10                                ;
ne = - n0 ++ + 29 - 5 * 2 << 2 - 1 ;
printf( ", %d, %d" , ne , n0 )         ;

// and, xor, or

n0 = 10                                                ;
ne = 4 | 24 ^ 18 & - ++ n0 + 29 - 5 * 2 << 2 - 1 ;
printf( ", %d" , ne )                                  ;

n0 = 10                                                ;
ne = 4 | 24 ^ 18 & - n0 ++ + 29 - 5 * 2 << 2 - 1 ;
printf( ", %d" , ne )                                  ;

// relational equality, and, or

be = false || true && true == 5 < 4 ;
printf( ", %d" , be )                  ;

// associativity

ne = 4 - 1 + 2         ;
printf( ", %d" , ne ) ;

printf( "\n" ) ;

// parentheses

printf( "parentheses                " ) ;

ne = 1 + 2 * 3       ;
printf( "%d" , ne ) ;

ne = ( 1 + 2 ) * 3     ;
```

```
  printf( ", %d" , ne ) ;

  be = true | true & false ;
  printf( ", %d" , be )    ;

  be = ( true | true ) & false ;
  printf( ", %d" , be )        ;

  b1 = true ;

  b0 = false              ;
  be = b1 | ( b0 = true ) ;
  printf( ", %d" , b0 )   ;

  b0 = false               ;
  be = b1 || ( b0 = true ) ;
  printf( ", %d" , b0 )    ;
}
```

Compiling and Running

1. Save the source code listing into a file named `operator.cpp`. Make sure the code ends with a carriage return.

2. Launch a Windows command prompt.

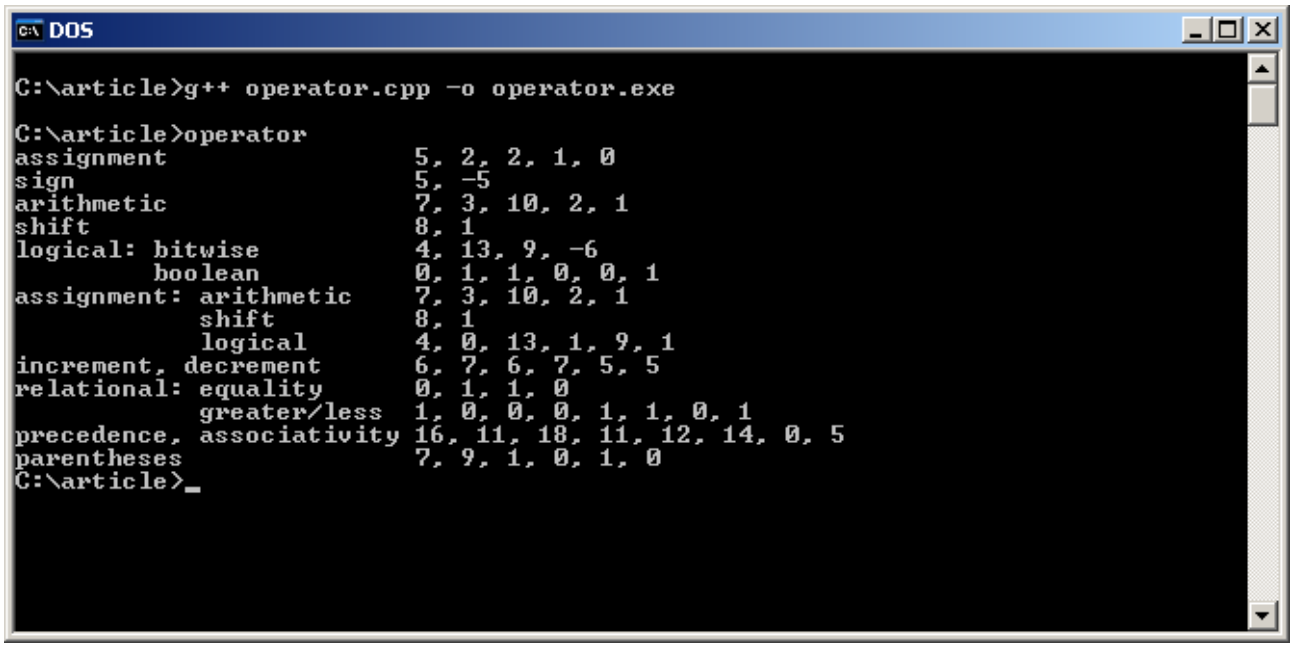
3. Navigate to the directory `operator.cpp` was saved in.

4. To compile the program type:

   ```
   > g++ operator.cpp -o operator.exe
   ```

5. To run the program type:

   ```
   > operator
   ```



Code Explanation

Assignment

An *assignment* statement uses the *assignment operator* = to set or re-set the value stored in a variable. In an assignment statement the first operand is a variable and the second operand is an expression.

=

```
n0 = 5 ;
```

This statement assigns a literal integer value to an `int` variable.

After this statement has executed, the value of variable `n0` is 5. The *operator* program reports the value of variable `n0`, displaying it in the console terminal.

```
ne = n1 = 2 ;
```

An assignment operator computes a value which can be used as the expression for another assignment.

This statement assigns a literal integer value to an `int` variable and then to another `int` variable.

After this statement has been executed, the values of variables `n1` and `ne` are 2. The *operator* program reports the values of variables `n1` and `ne`.

```
b0 = true  ;
b1 = false ;
```

These statements each assign a literal boolean value to a `bool` variable.

After these statements have been executed, the value of variable `b0` is true and the value of `b1` is false. The *operator* program reports the values of variables `b0` and `b1`.

Sign

C++ provides *sign operators* + and − which respectively evaluate to the value and the negated value of their operand. Sign operators are unary operators performing operations on a single operand which is an expression.

Prior to executing the following statements, the value of variable `n0` is 5.

\+

```
ne = + n0 ;
```

The *plus operator* + computes the value of its operand, maintaining the sign.

Expression `+ n0` evaluates to `n0`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  + n0
= + 5
= 5
```

5 is assigned to variable `ne` and the *operator* program reports its value.

\-

```
ne = - n0 ;
```

The *minus operator* − computes the *negated* value of its operand.

Expression `- n0` evaluates to the negated value of `n0`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  - n0
= - 5
= -5
```

-5 is assigned to variable `ne` and the *operator* program reports its value.

Arithmetic

C++ provides *arithmetic operators* for *addition* `+`, *subtraction* `-`, *multiplication* `*`, *division* `/` and *modulus* `%`. The modulus operation computes the remainder of division of one number by another. Arithmetic operators are binary operators performing operations on two operands which are both expressions.

Prior to executing the following statements the value of variable `n0` is 5 and the value of `n1` is 2.

`+`

```
ne = n0 + n1 ;
```

The *addition operator* `+` computes the value of the first operand *plus* the second operand.

Expression `n0 + n1` evaluates to the sum of variable `n1` added to variable `n0`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 + n1
= 5 + 2
= 7
```

7 is assigned to variable `ne` and the *operator* program reports its value.

`-`

```
ne = n0 - 2 ;
```

The *subtraction operator* `-` computes the value of the first operand *minus* the second operand.

Expression `n0 - 2` evaluates to the difference of integer literal 2 subtracted from variable `n0`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 - 2
= 5 - 2
= 3
```

3 is assigned to variable `ne` and the *operator* program reports its value.

*

```
ne = 5 * 2 ;
```

The *multiplication operator* `*` computes the value of the first operand *multiplied by* the second operand.

Expression `5 * 2` evaluates to the product of integer literal 5 multiplied by integer literal 2. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  5 * 2
= 10
```

10 is assigned to variable `ne` and the *operator* program reports its value.

/

```
ne = 5 / 2 ;
```

The *division operator* `/` computes the value of the first operand *divided by* the second operand.

Expression `5 / 2` evaluates to the dividend of integer literal 5 divided by integer literal 2. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  5 / 2
= 2
```

2 is assigned to variable `ne` and the *operator* program reports its value.

%

```
ne = 5 % 2 ;
```

The *modulus operator* `%` computes the *remainder* value of the first operand divided by the second operand.

Expression `5 % 2` evaluates to the remainder of division of integer literal 5 by integer literal 2. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  5 % 2
= 1
```

1 is assigned to variable `ne` and the *operator* program reports its value.

Shift

In *bit shift* operations the binary digits that make up an operand are moved, or shifted, to the left or right.

For example, shifting binary number `00101100` (decimal 44) left by 1 evaluates to `01011000`; shifting `00101100` right by 2 evaluates to `00001011`.

C++ shift operators perform an arithmetic shift for a signed operand and a logical shift for an unsigned operand. In both cases bits that are shifted out of either end are discarded. In a left arithmetic shift, zeros are shifted in on the right; in a right arithmetic shift, if the operand is unsigned, zeros are shifted in on the left; if the operand is signed, the sign bit is shifted in on the left, thus preserving the sign of the operand. Internally the sign of an integer (whether it is positive or negative) is determined by the left-most (or most significant) bit: if the left-most bit is 1 the integer is negative; otherwise it is positive. Therefore preserving the value of the left-most bit during a right shift preserves the operand's sign.

For example, using an 8-bit signed operand, shifting binary number `11010011` left by 1 returns `10100110`; shifting `11010011` right by 1 returns `11101001`. In the first case, the left-most digit is shifted beyond the end of the register, and a new 0 is shifted into the right-most position. In the second case, the right-most digit is shifted out, and a new 1 is copied into the left-most position, preserving the sign of the number.

A left arithmetic shift by `n` is equivalent to multiplying by 2^n (provided the value does not overflow), while a right arithmetic shift by `n` of a two's complement value is equivalent to dividing by 2^n and rounding toward negative infinity. If the binary number is treated as one's complement, then the same right-shift operation results in division by 2^n and rounding toward zero.

The *bit shift operators* compute the value of the first operand shifted left or right by the number of bits specified in the second operand. Both operands are expressions.

Prior to executing the following statements the value of variable `n0` is binary `0100` (decimal 4) and the value of `n1` is 2.

<< `ne = n0 << 1 ;`

The *left shift operator* << computes the first operand *shifted left by* the number of bits specified in the second operand. The high-order bits outside the range of the type of the first operand are discarded, the remaining bits are shifted left and the low-order empty bit positions are set to zero.

Expression `n0 << 1` evaluates to variable `n0` shifted left by literal value 1, which is effectively `n0` * 2^1 which equals `n0` * 2. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 << 1
= 0100 << 1  (where 0100 is a binary number (decimal 4))
= 1000
```

Binary `1000` (decimal 8) is assigned to variable `ne` and the *operator* program reports its value.

>> `ne = n0 >> n1 ;`

The *right shift operator* >> computes the first operand *shifted right by* the number of bits specified in the second operand. The low-order bits are discarded and the remaining bits are shifted right. If the first operand is signed, the high-order empty bit positions are set to zero if the first operand is positive or one if the first operand is negative; if the first operand is unsigned, the high-order empty bit positions are set to zero.

Expression `n0 >> n1` evaluates to variable `n0` shifted right by variable `n1`, which is effectively `n0` / 2^{n1}. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 >> n1
= 0100 >> 2  (where 0100 is a binary number (decimal 4))
= 0001
```

Binary value `0001` (decimal 1) is assigned to variable `ne` and the *operator* program reports its value.

Logical: Bitwise

Bitwise logical operators perform boolean logic on corresponding bits of one or two integral expressions. Valid integral types are signed and unsigned integers. They return a compatible integral result with each bit conforming to the boolean evaluation.

Prior to executing the following statements the value of variable `n0` is binary `0101` (decimal 5) and the value of `n1` is binary `1100` (decimal 12).

`&`

```
ne = n0 & n1 ;
```

And operators `&` are predefined for integral and boolean types. For integral operands, `&` computes the logical *bitwise and* of its operands. Each bit in the result is 1 if the corresponding bits in both its operands are 1; otherwise the bit is 0.

Expression `n0 & n1` evaluates to variable `n0` and variable `n1`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 & n1
= 0101 & 1100  (where 0101 and 1100 are binary numbers (decimal 5 and 12))
= 0100
```

Binary `0100` (decimal 4) is assigned to variable `ne` and the *operator* program reports its value.

`|`

```
ne = n0 | n1 ;
```

Or operators `|` are predefined for integral and boolean types. For integral operands, `|` computes the logical *bitwise or* of its operands. Each bit in the result is 0 if the corresponding bits in both its operands are 0; otherwise the bit is 1.

Expression `n0 | n1` evaluates to variable `n0` or variable `n1`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 | n1
= 0101 | 1100  (where 0101 and 1100 are binary numbers (decimal 5 and 12))
= 1101
```

Binary `1101` (decimal 13) is assigned to variable `ne` and the *operator* program reports its value.

^

```
ne = n0 ^ n1 ;
```

Exclusive-or operators ^ are predefined for integral and boolean types. For integral operands, ^ computes the logical *bitwise exclusive-or* of its operands. Each bit in the result is 1 if the corresponding bit in exactly one of its operands is 1; otherwise the bit is 0.

Expression `n0 ^ n1` evaluates to variable `n0` exclusive-or variable `n1`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  n0 ^ n1
= 0101 ^ 1100  (where 0101 and 1100 are binary numbers (decimal 5 and 12))
= 1001
```

Binary `1001` (decimal 9) is assigned to variable `ne` and the *operator* program reports its value.

~

```
ne = ~ n0 ;
```

The bitwise *complement operator* ~ is predefined for integral types. ~ computes a *bitwise complement* of its operand, which has the effect of reversing each bit.

Expression ~ `n0` evaluates to the bitwise complement of variable `n0`. This statement assigns the value of the expression to variable `ne`.

The expression evaluates as follows:

```
  ~ n0
= ~ 0101  (where 0101 is a binary number (decimal 5))
= 1010
```

Binary `1010` (decimal -6) is assigned to variable `ne` and the *operator* program reports its value.

Logical: Boolean

Boolean logical operators perform boolean logic on boolean expressions. Binary operators evaluate the expression on the left, and then the expression on the right. Finally, the two expressions are evaluated together in the context of the boolean logical operator between them, computing a `bool` result.

Prior to executing the following statements the value of variable `b0` is true and the value of `b1` is false.

`&`

```
be = b0 & b1 ;
```

And operators `&` are predefined for integral and boolean types. For boolean operands, `&` computes the *logical and* of its operands. The result is true if both its operands are true; otherwise the result is false.

Expression `b0 & b1` evaluates to variable `b0` and variable `b1`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  b0 & b1
= true & false
= false
```

False is assigned to variable `be` and the *operator* program reports its value.

`|`

```
be = b0 | b1 ;
```

Or operators `|` are predefined for integral and boolean types. For boolean operands, `|` computes the *logical or* of its operands. The result is false if both its operands are false; otherwise the result is true.

Expression `b0 | b1` evaluates to variable `b0` or variable `b1`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  b0 | b1
= true | false
= true
```

True is assigned to variable `be` and the *operator* program reports its value.

^

```
be = b0 ^ b1 ;
```

Exclusive-or operators ^ are predefined for integral and boolean types. For boolean operands, ^ computes the *logical exclusive-or* of its operands. The result is true if exactly one of its operands is true; otherwise the result is false.

Expression `b0 ^ b1` evaluates to variable `b0` exclusive-or variable `b1`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  b0 ^ b1
= true ^ false
= true
```

True is assigned to variable `be` and the *operator* program reports its value.

!

```
be = ! b0 ;
```

The logical *not operator* ! computes the *negated value* of its operand. The result is true if the operand is false; otherwise the result is false.

Expression `! b0` evaluates to negation of variable `b0`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  ! b0
= ! true
= false
```

False is assigned to variable `be` and the *operator* program reports its value.

`&&`

```
be = b0 && b1 ;
```

The *conditional-and operator* `&&` computes the *logical and* of its operands but only evaluates its second operand if necessary. If the first operand evaluates to false, the computation will be false regardless of the value of the second operand, so the second operand is not evaluated. The `&&` operator performs the same operation as the `&` operator except that if the expression on the left is false, the expression on the right is not evaluated.

Expression `b0 && b1` evaluates to variable `b0` and variable `b1`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  b0 && b1
= true && false
= false
```

Since the expression on the left of the `&&` operator is true, the expression on the right is also evaluated. False is assigned to variable `be` and the *operator* program reports its value.

`||`

```
be = b0 || b1 ;
```

The *conditional-or operator* `||` computes the *logical or* of its operands but only evaluates its second operand if necessary. If the first operand evaluates to true, the computation will be true regardless of the value of the second operand, so the second operand is not evaluated. The `||` operator performs the same operation as the `|` operator except that if the expression on the left is true, the expression on the right is not evaluated.

Expression `b0 || b1` evaluates to variable `b0` or variable `b1`. This statement assigns the value of the expression to variable `be`.

The expression evaluates as follows:

```
  b0 || b1
= true || false
= true
```

Since the expression on the left of the `||` operator is true, the expression on the right is not evaluated. True is assigned to variable `be` and the *operator* program reports its value.

Assignment: Arithmetic

C++ provides *arithmetic assignment* operators for *addition* `+=`, *subtraction* `-=`, *multiplication* `*=`, *division* `/=` and *modulus* `%=`. In arithmetic assignment statements the first operand is a variable and the second operand is an expression.

Prior to executing each of the following statements the value of variable `ne` is 5.

`+=`

```
ne += n1 ;
```

The *assignment by sum operator* `+=` *adds* the value of the second operand to the first operand.

This statement adds the value of variable `n1` to variable `ne`.

Prior to executing this statement the value of variable `n1` is 2.

After execution the value of variable `ne` is 7. The *operator* program reports its value.

`-=`

```
ne -= 2 ;
```

The *assignment by difference* operator `-=` *subtracts* the value of the second operand from the first operand.

This statement subtracts literal value 2 from variable `ne`.

After execution the value of variable `ne` is 3. The *operator* program reports its value.

`*=`

```
ne *= 2 ;
```

The *assignment by product operator* `*=` *multiplies* the first operand by the value of second operand.

This statement multiplies variable `ne` by literal value 2.

After execution the value of variable `ne` is 10. The *operator* program reports its value.

`/=`

```
ne /= 2 ;
```

The *assignment by dividend operator* `/=` *divides* the first operand by the value of the second operand.

This statement divides variable `ne` by literal value 2.

After execution the value of variable `ne` is 2. The *operator* program reports its value.

%=
```
ne %= 2 ;
```

The *assignment by remainder operator* %= assigns the *remainder* value of the first operand divided by the second operand to the first operand.

This statement assigns the remainder value of variable ne divided by 2 to variable ne.

After execution the value of variable ne is 1. The *operator* program reports its value.

Assignment: Shift

In *bit shift assignment* operations the binary digits that make up the first operand are moved, or shifted, to the left or right by the number of bits specified in the value of the second operand. In bit shift assignment statements the first operand is a variable and the second operand is an expression.

Prior to executing each of the following statements the value of variable ne is binary 0100 (decimal 4).

<<=
```
ne <<= 1 ;
```

The *assignment by left-shift operator* <<= *shifts* the first operand *left* by the number of bits specified in the second operand. The high-order bits outside the range of the first operand are discarded, the remaining bits are shifted left and the low-order empty bit positions are set to zero.

This statement shifts variable ne left by literal value 1, which effectively multiplies variable ne by 2^1 (or 2).

After execution the value of variable ne is binary 1000 (decimal 8). The *operator* program reports its value.

>>=
```
ne >>= n1 ;
```

The *assignment by right-shift operator* >>= *shifts* the first operand *right* by the number of bits specified in the second operand. The low-order bits are discarded and the remaining bits are shifted right. If the first operand is signed, the high-order empty bit positions are set to zero if the first operand is positive or one if the first operand is negative; if the first operand is unsigned, the high-order empty bit positions are set to zero.

This statement shifts variable ne right by the value of variable n1, which effectively divides ne by 2^{n1}.

Prior to executing this statement the value of variable n1 is 2.

After execution the value of variable ne is binary 0001 (decimal 1). The *operator* program reports its value.

Assignment: Logical

Logical assignment operators perform boolean logic on the first operand. In logical assignment statements the first operand is a variable and the second operand is an expression.

Prior to executing each of the following statements the value of variable `ne` is binary `0101` (decimal 5), `n1` is binary `1100` (decimal 12), `be` is true and `b1` is false.

`&=`

```
ne &= n1 ;
be &= b1 ;
```

The *assignment by and operator* `&=` *ands* the first operand with the second operand.

After execution the value of `ne` is binary `0100` (decimal 4) and the value of `be` is false. The *operator* program reports the values of `ne` and `be`.

`|=`

```
ne |= n1 ;
be |= b1 ;
```

The *assignment by or operator* `|=` *ors* the first operand with the second operand.

After execution the value of `ne` is binary `1101` (decimal 13) and the value of `be` is true. The *operator* program reports the values of `ne` and `be`.

`^=`

```
ne ^= n1 ;
be ^= b1 ;
```

The *assignment by xor operator* `^=` *exclusive-ors* the first operand with the second operand.

After execution the value of `ne` is binary `1001` (decimal 9) and the value of `be` is true. The *operator* program reports the values of `ne` and `be`.

Increment, Decrement

The *increment* operator ++ adds 1 to its operand; the *decrement* operator -- subtracts 1 from its operand. The increment and decrement operators are unary operators in which the operand is a variable.

The increment and decrement operators can appear before or after the operand. If the operator appears before the operand it performs a prefix operation: the result of the operation is the value of the operand after it has been incremented or decremented. If the operator appears after the operand it performs a postfix operation: the result of the operation is the value of the operand before it has been incremented or decremented.

Prior to executing the following statements the value of variable `ne` is 5.

++

```
ne ++ ;
```

The *increment operator* ++ *adds 1* to its operand.

This statement increments variable `ne` from 5 to 6.

The *operator* program reports the value of variable `ne`.

```
++ ne ;
```

The ++ operator can appear before or after the operand.

This statement increments variable `ne` from 6 to 7.

The *operator* program reports the value of variable `ne`.

--

```
n0 = ne -- ;
```

The *decrement operator* -- *subtracts 1* from its operand. If the operator appears after the operand it performs a postfix decrement operation: the result of the operation is the value of the operand before it has been decremented.

This statement performs a postfix decrement operation, assigning the value of `ne` before it has been decremented to `n0`.

The value of `ne` is decremented from 7 to 6 and the value of `ne` before it has been decremented, 7, is assigned to `n0`. The *operator* program reports the values of `ne` and `n0`.

```
n0 = -- ne ;
```

If the decrement operator appears before the operand it performs a prefix decrement operation: the result of the operation is the value of the operand after it has been decremented.

This statement performs a prefix decrement operation, assigning the value of `ne` after it has been decremented to `n0`.

The value of `ne` is decremented from 6 to 5 and the value of `ne` after it has been decremented, 5, is assigned to `n0`. The *operator* program reports the values of `ne` and `n0`.

Relational: Equality

The *equality* operators == and `!=` compare two expressions for equality or inequality.

Prior to executing the following statements the value of variable `n0` is 1, the value of `n1` is 2 and the value of `n2` is 1.

==

```
be = n0 == n1 ;
be = n0 == n2 ;
```

The *equality operator* == returns true if its operands are equal; otherwise it returns false.

Expression `n0 == n1` evaluates to equality between `n0` and `n1`; expression `n0 == n2` evaluates to equality between `n0` and `n2`. These statements assign the values of the expressions to variable `be`.

The expressions evaluate as follows:

```
  n0 == n1
= 1 == 2
= false

  n0 == n2
= 1 == 1
= true
```

False and then true are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

!=

```
be = n0 != n1 ;
be = n0 != n2 ;
```

The *inequality operator* != returns false if its operands are equal; otherwise it returns true.

Expression `n0 != n1` evaluates to inequality between `n0` and `n1`; expression `n0 != n2` evaluates to inequality between `n0` and `n2`. These statements assign the values of the expressions to variable `be`.

The expressions evaluate as follows:

```
  n0 != n1
= 1 != 2
= true

  n0 != n2
= 1 != 1
= false
```

True and then false are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

Relational: Greater/Less

Relational operators return a boolean value resulting from comparison between the values of the first expression and the second expression. C++ provides relational operators for *less than* <, *greater than* >, *less than or equal* <= and *greater than or equal* >=.

Prior to executing the following statements the value of variable `n0` is 1, the value of `n1` is 2 and the value of `n2` is 1.

<

```
be = n0 < n1 ;
be = n0 < n2 ;
```

The *less than operator* < returns true if the first operand is *less than* the second; otherwise it returns false.

Expression `n0 < n1` evaluates to true only if `n0` is less than `n1`; expression `n0 < n2` evaluates to true only if `n0` is less than `n2`.

The expressions evaluate as follows:

```
  n0 < n1
= 1 < 2
= true

  n0 < n2
= 1 < 1
= false
```

True and then false are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

>

```
be = n0 > n1 ;
be = n0 > n2 ;
```

The *greater than operator* > returns true if the first operand is *greater than* the second; otherwise it returns false.

Expression `n0 > n1` evaluates to true only if `n0` is greater than `n1`; expression `n0 > n2` evaluates to true only if `n0` is greater than `n2`.

The expressions evaluate as follows:

```
  n0 > n1
= 1 > 2
= false

  n0 > n2
= 1 > 1
= false
```

False and then false are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

<=

```
be = n0 <= n1 ;
be = n0 <= n2 ;
```

The *less than or equal operator* <= returns true if the first operand is *less than or equal to* the second; otherwise it returns false.

Expression `n0 <= n1` evaluates to true only if `n0` is less than or equal to `n1`; expression `n0 <= n2` evaluates to true only if `n0` is less than or equal to `n2`.

The expressions evaluate as follows:

```
  n0 <= n1
= 1 <= 2
= true

  n0 <= n2
= 1 <= 1
= true
```

True and then true are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

>=

```
be = n0 >= n1 ;
be = n0 >= n2 ;
```

The *greater than or equal operator* >= returns true if the first operand is *greater than or equal to* the second; otherwise it returns false.

Expression `n0 >= n1` evaluates to true only if `n0` is greater than or equal to `n1`; expression `n0 >= n2` evaluates to true only if `n0` is greater than or equal to `n2`.

The expressions evaluate as follows:

```
  n0 >= n1
= 1 >= 2
= false

  n0 >= n2
= 1 >= 1
= true
```

False and then true are assigned to `be`. The *operator* program reports the value of `be` after each statement executes.

Precedence

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, expression `b + c * d` is evaluated as `b + (c * d)` because the `*` operator has higher precedence than the `+` operator: the `*` operator is evaluated first and the + operator second.

Individual operators are evaluated in the following order of categories:

1. primary
2. unary
3. multiplicative
4. additive
5. shift
6. relational
7. equality
8. and
9. exclusive-or
10. or
11. conditional and
12. conditional or
13. assignment

Primary, Unary, Multiplicative, Additive, Shift

```
ne = - ++ n0 + 29 - 5 * 2 << 2 - 1 ;
```

This statement demonstrates precedence for expressions containing operators which are evaluated in the following order:

1. unary
2. multiplicative
3. additive
4. shift

Prior to executing this statement the value of variable `n0` is 10.

Expression `- ++ n0 + 29 - 5 * 2 << 2 - 1` is evaluated as follows:

`- ++ n0 + 29 - 5 * 2 << 2 - 1`	
`= - ++ 10 + 29 - 5 * 2 << 2 - 1`	evaluating `n0`
`= -11 + 29 - 5 * 2 << 2 - 1`	evaluating unary operators
`= -11 + 29 - 10 << 2 - 1`	evaluating multiplicative operator
`= 8 << 1`	evaluating additive operators
`= 16`	evaluating shift operator

Variable `n0` is incremented to 11; then the expression is evaluated and 16 is assigned to variable `ne`. The *operator* program reports the values of `ne` and `n0`.

```
ne = - n0 ++ + 29 - 5 * 2 << 2 - 1 ;
```

This statement demonstrates precedence for expressions containing operators which are evaluated in the following order:

1. primary
2. unary
3. multiplicative
4. additive
5. shift

Prior to executing this statement the value of variable `n0` is 10.

Expression `- n0 ++ + 29 - 5 * 2 << 2 - 1` is evaluated as follows:

`- n0 ++ + 29 - 5 * 2 << 2 - 1`	
`= - 10 ++ + 29 - 5 * 2 << 2 - 1`	evaluating `n0`
`= - 10 + 29 - 5 * 2 << 2 - 1`	evaluating primary operator
`= -10 + 29 - 5 * 2 << 2 - 1`	evaluating unary operator
`= -10 + 29 - 10 << 2 - 1`	evaluating multiplicative operator
`= 9 << 1`	evaluating additive operators
`= 18`	evaluating shift operator

The expression is evaluated and 18 is assigned to variable `ne`; then variable `n0` is incremented to `11`. The *operator* program reports the values of `ne` and `n0`.

And, Exclusive-or, Or

```
ne = 4 | 24 ^ 18 & - ++ n0 + 29 - 5 * 2 << 2 - 1 ;
```

This statement demonstrates precedence for expressions containing operators which are evaluated in the following order:

1. unary
2. multiplicative
3. additive
4. shift
5. and
6. exclusive-or
7. or

Prior to executing this statement the value of variable `n0` is 10.

Expression `4 | 24 ^ 18 & - ++ n0 + 29 - 5 * 2 << 2 - 1` is evaluated as follows:

`4 | 24 ^ 18 & - ++ n0 + 29 - 5 * 2 << 2 - 1`	
`= 4 | 24 ^ 18 & - ++ 10 + 29 - 5 * 2 << 2 - 1`	evaluating `n0`
`= 4 | 24 ^ 18 & -11 + 29 - 5 * 2 << 2 - 1`	unary
`= 4 | 24 ^ 18 & -11 + 29 - 10 << 2 - 1`	multiplicative
`= 4 | 24 ^ 18 & 8 << 1`	additive
`= 4 | 24 ^ 18 & 16`	shift
`= 00000100 | 00011000 ^ 00010010 & 00010000`	binary
`= 00000100 | 00011000 ^ 00010000`	and
`= 00000100 | 00001000`	exclusive-or
`= 00001100`	or
`= 12`	decimal

12 is assigned to variable `ne` and the *operator* program reports its value.

```
ne = 4 | 24 ^ 18 & - n0 ++ + 29 - 5 * 2 << 2 - 1 ;
```

This statement demonstrates precedence for expressions containing operators which are evaluated in the following order:

1. primary
2. unary
3. multiplicative
4. additive
5. shift
6. and
7. exclusive-or
8. or

Prior to executing this statement the value of variable `n0` is 10.

Expression `4 | 24 ^ 18 & - n0 ++ + 29 - 5 * 2 << 2 - 1` is evaluated as follows:

`4 | 24 ^ 18 & - n0 ++ + 29 - 5 * 2 << 2 - 1`	
`= 4 | 24 ^ 18 & - 10 ++ + 29 - 5 * 2 << 2 - 1`	evaluating `n0`
`= 4 | 24 ^ 18 & - 10 + 29 - 5 * 2 << 2 - 1`	primary
`= 4 | 24 ^ 18 & -10 + 29 - 5 * 2 << 2 - 1`	unary
`= 4 | 24 ^ 18 & -10 + 29 - 10 << 2 - 1`	multiplicative
`= 4 | 24 ^ 18 & 9 << 1`	additive
`= 4 | 24 ^ 18 & 18`	shift
`= 00000100 | 00011000 ^ 00010010 & 00010010`	binary
`= 00000100 | 00011000 ^ 00010010`	and
`= 00000100 | 00001010`	exclusive-or
`= 00001110`	or
`= 14`	decimal

14 is assigned to variable `ne` and the *operator* program reports its value.

Relational Equality, Conditional And, Conditional Or

```
be = false || true && true == 5 < 4 ;
```

This statement demonstrates precedence for expressions containing operators which are evaluated in the following order:

1. relational
2. equality
3. conditional and
4. conditional or

Expression `false || true && true == 5 < 4` is evaluated as follows:

```
  false || true && true == 5 < 4
= false || true && true == false     evaluating relational operator
= false || true && false             evaluating equality operator
= false || false                     evaluating and operator
= false                              evaluating or operator
```

False is assigned to variable `be` and the *operator* program reports its value.

Associativity

When an expression contains multiple operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

- Except for assignment operators, all binary operators are left-associative.
- Assignment operators are right-associative.

```
ne = 4 - 1 + 2 ;
```

This statement demonstrates associativity for expressions containing operators that both have additive precedence.

Because the operators are left-associative expression `4 - 1 + 2` performs the operations from left to right, evaluating to 5. If the operators were right-associative the expression would evaluate to 1. The expression is evaluated as follows:

```
  4 - 1 + 2
= 3 + 2          4 – 1 = 3
= 5              3 + 2 = 5
```

5 is assigned to variable `ne` and the *operator* program reports its value.

Parentheses

Precedence and associativity can be controlled using *parentheses* `()`.

```
ne = 1 + 2 * 3     ;
ne = ( 1 + 2 ) * 3 ;
```

These statements demonstrate use of parentheses to control the order in which operations with different precedence are performed on integral values.

Expression `1 + 2 * 3` evaluates `2 * 3` first because the multiplication operator `*` has a higher precedence than the addition operator `+`. The expression evaluates as follows:

```
  1 + 2 * 3
= 1 + 6          2 * 3 = 6
= 7              1 + 6 = 7
```

7 is assigned to variable `ne` and the *operator* program reports its value.

Expression `( 1 + 2 ) * 3` evaluates `1 + 2` first because expression `1 + 2` is enclosed in parentheses. The expression evaluates as follows:

```
  ( 1 + 2 ) * 3
= 3 * 3            1 + 2 = 3
= 9                3 * 3 = 9
```

9 is assigned to variable `ne` and the *operator* program reports its value.

```
be = true | true & false     ;
be = ( true | true ) & false ;
```

These statements demonstrate use of parentheses to control the order in which operations with different precedence are performed on boolean values.

Expression `true | true & false` evaluates `true & false` first because the and operator `&` has a higher precedence than the or operator `|`. The expression is evaluated as follows:

```
      true | true & false
    = true | false          true & false = false
    = true                  true | false = true
```

True is assigned to variable `be` and the *operator* program reports its value.

Expression `( true | true ) & false` evaluates `true | true` first because expression `true | true` is enclosed in parentheses. The expression is evaluated as follows:

```
      ( true | true ) & false
    = true & false              true | true = true
    = false                     true & false = false
```

False is assigned to variable `be` and the *operator* program reports its value.

```
be = b1 |  ( b0 = true ) ;
be = b1 || ( b0 = true ) ;
```

These statements demonstrate use of parentheses to control the order of execution of assignment statements and also demonstrate the difference between logical operators and conditional logical operators.

Prior to executing each of these statements the value of variable `b0` is false and the value of `b1` is true.

Expression `b1 | ( b0 = true )` evaluates `b1`, which evaluates to true. Expression `b0 = true` is then evaluated, assigning true to `b0` and evaluating to true. The overall expression is thus `true | true`, which is evaluates to true. During evaluation of this expression, the value of `b0` changes from false to true.

The *operator* program reports the value of `b0`.

Expression `b1 || ( b0 = true )` evaluates `b1`, which evaluates to true. Since one of the operands in the overall or expression is true, the expression will evaluate to true regardless of the value of the remaining operand, and the remaining operand is thus not evaluated. During evaluation of this expression, expression `b0 = true` is not evaluated and the value of `b0` therefore does not change, but remains false.

The *operator* program reports the value of `b0`.

Further Comments

C++ provides a large set of operators, some of which are outwith the scope of this article and have therefore not been included in the previous sections.

The following table shows the precedence and associativity of all C++ operators (from highest to lowest precedence). Operators with the equal precedence are evaluated in the given order in an expression unless explicitly forced by parentheses.

Precedence	Operator	Description	Associativity
scope resolution	`::`	scope resolution	left to right
primary	`++`	postfix increment	left to right
	`--`	postfix decrement	
	`()`	function call	
	`[]`	array subscript	
	`.`	member selection by reference	
	`->`	member selection through pointer	
	`typeid()`	run-time type information	
	`const_cast`	type cast	
	`dynamic_cast`	type cast	
	`reinterpret_cast`	type cast	
	`static_cast`	type cast	
unary	`++`	prefix increment	right to left
	`--`	prefix decrement	
	`+`	unary plus	
	`-`	unary minus (two's complement)	
	`~`	complement (one's complement)	
	`!`	logical not	
	`( type )`	type cast	
	`*`	indirection, dereference	
	`&`	reference	
	`sizeof`	size-of	
	`new`	dynamic memory allocation	
	`delete`	dynamic memory deallocation	
pointer	`.`	member	left to right
	`->`	indirect member	
multiplicative	`*`	multiplication	left to right

	/	division	
	%	modulus	
additive	+	addition	left to right
	-	subtraction	
shift	<<	left shift	left to right
	>>	right shift	
relational	<	less than	left to right
	>	greater than	
	<=	less than or equal	
	>=	greater than or equal	
equality	==	equality	left to right
	!=	inequality	
and	&	and	left to right
exclusive-or	^	xor (exclusive-or)	left to right
or	|	or (inclusive-or)	left to right
conditional and	&&	conditional and	left to right
conditional or	||	conditional or	left to right
conditional	? :	ternary conditional	right to left
assignment	=	assignment	right to left
	+=	assignment by sum	
	-=	assignment by difference	
	*=	assignment by product	
	/=	assignment by dividend	
	%=	assignment by remainder	
	<<=	assignment by left-shift	
	>>=	assignment by right-shift	
	&=	assignment by and	
	^=	assignment by xor	
	|=	assignment by or	
exception	throw	throw exception	
comma	,	comma	left to right