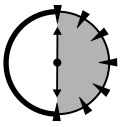
SESSION


# Logical Operations

## Session Checklist

- ✔ Using simple logical operators and variables
- ✔ Working with binary numbers
- ✔ Performing bitwise operations
- ✔ Creating logical assignment statements



*30 Min. To Go*

Other than a few oddballs such as increment and decrement, the C++ mathematical operators are common operations that you encounter in everyday life. By comparison, the C++ logical operators are unknown.

It isn't that people don't deal with logical operations. If you think back to our mechanical tire changer in Session 1, the capability to express logical computations such as "if the tire is flat AND I have a lug wrench. . ." is a must. People compute AND and OR constantly; it's just that they aren't accustomed to writing them down.

Logical operators fall into two types. One type, which I call simple logical operators, are operators that are common in everyday life. The second type, the bitwise logical operators, is unique to the computer world. I consider the simple operators before jumping off into the more complex bitwise versions.

## Simple Logical Operators

The simple logical operators represent the same kind of logical decisions you'd make in daily life: such as considering whether something is true or false, or comparing two things to determine whether they're the same. The simple logical operators are shown in Table 7-1.

**Table 7-1**
*Simple Logical Operators*

| Operator | Meaning |
|---|---|
| == | equality; true if the left hand argument has the same value as the right |
| != | inequality; opposite of equality |
| >, < | greater than, less than; true if the left hand argument is greater than/less than the right hand argument |
| >=, <= | greater than or equal to, less than or equal to; true if either > or == is true, /< or == is true |
| && | AND; true if both the left and right hand arguments are true |
| \|\| | OR; true if either the left or the right hand arguments are true |
| ! | NOT; true if it's argument is false |

The first six entries in Table 7-1 are the comparison operators. The equality operator is used to compare two numbers. For example, the following is true if the value of `nVariable` is 0.

```
nVariable == 0;
```


**Don't confuse the equality operator == with the assignment operator =. Not only is this is a common mistake, but it's a mistake that the C++ compiler cannot catch.**

```
nVariable = 0;   // programmer meant to say nVariable == 0
```

The greater than (>) and less than (<) operators are similarly common in everyday life. The following logical comparison expression is true:

```
int nVariable1 = 1;
int nVariable2 = 2;
nVariable1 < nVariable2;
```

The greater than or equal to (>=) and less than or equal to (<=) are similar except that they include equality, whereas the other operators do not.

The && (AND) and || (OR) are equally common. These operators are typically combined with the other logic operators:

```
// true if nV2 is greater than nV1 but smaller than nV3
(nV1 < nV2) && (nV2 < nV3);
```


**Be careful when using the comparison operators on floating-point numbers. Consider the following example:**

```
float fVariable1 = 10.0;
float fVariable2 = (10 / 3) * 3;
fVariable1 == fVariable2;   // are these two equal?
```

**The comparison in the above example is not necessarily true. 10 / 3 is equal to 0.333.... C++ cannot represent an infinite number of digits. Stored as a float,** `10/3` **becomes something like 0.333333. Multiplying this number by 3 gives a result of 0.999999 which is not exactly equal to 10.**

**Just as** `(10.0 / 3) * 3` **is not exactly identical to 10.0, the results of a floating-point calculation may be off by some normally tiny amount. Such small differences may be unnoticeable to you and me, but not to the computer. Equality means exactly that, exact equality.**

**The safer comparison is as follows:**

```
float fDelta = fVariable1 - fVariable2;
fDelta < 0.01 && fDelta > -0.01;
```

**This comparison is true if** `fVariable1` **and** `fVariable2` **are within some delta of each other. (The term "some delta" is intentionally vague — it should represent acceptable error.)**


**The numerical processor in your PC goes to great pains to avoid floating round off errors such as the one above — depending on the situation, the numeric processor might provide a tiny delta for you automatically; however, you should never count on such aids.**

## Short circuits and C++

The && and || operators perform what is called *short circuit evaluation*. Consider the following:

```
condition1 && condition2;
```

If `condition1` is not true, then the result is not true no matter what the value of `condition2` (that is, `condition2` could be true or false without changing the result). Similarly, in the following

```
condition1 || condition2;
```

if `condition1` is true then the result is true no matter what the value of `condition2`.

To save time, C++ evaluates `condition1` first. C++ does not evaluate `condition2` if `condition1` is false in the case of && or `condition1` is true in the case of ||.

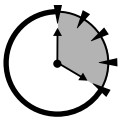
## Logical variable types

If > and && are operators, then a comparison such as `a > 10` must be an expression. Clearly, the result of such an expression must be either TRUE or FALSE.

You may have noticed already that no Boolean variable type was mentioned in Session 5. That is, there is no variable type that can have the value TRUE or FALSE and nothing else. Then what is the type of an expression such as `a > 10`?

C++ uses the type *int* to store Boolean values. The value 0 is taken to be FALSE. Any value other than 0 is TRUE. An expression such as `a > 10` evaluates to either 0 (FALSE) or 1 (TRUE).


**Microsoft Visual Basic also uses an integer to hold TRUE and FALSE values; however, in Visual Basic a comparison operation returns either a 0 (FALSE) or a –1 (TRUE).**



20 Min. To Go

## Binary Numbers

The so-called bitwise logical operators operate on their arguments at the bit level. To understand how they work, let's first examine binary numbers, which are how computers store values.

The numbers that we are familiar with are known as decimal numbers because they are based on the number 10. A number such as 123 refers to 1 × 100 plus 2 × 10 plus 3 × 1. Each of these base numbers, 100, 10, and 1 are powers of 10.

$123_{10}$ = 1 * 100 + 2 * 10 + 3 * 1

The use of a base number of 10 for our counting system most likely stems from the fact that humans have 10 fingers, the original counting tools. If our numbering scheme had been invented by dogs, it might well have been based on the number 8 (1 "finger" of each paw is out of sight on the back part of the leg). Such an octal system would have worked just as well:

$123_{10}$ = $173_8$ = 1 * $64_{10}$ + 7 * $8_{10}$ + 3

The small 10 and 8 here refer to the numbering system, 10 for decimal (base 10) and 8 for octal (base 8). A counting system may use any base other than 1.

Computers have essentially two fingers. Because of the way they are built, they prefer counting using base 2. The number $123_{10}$ is expressed as

0*128 + 1*64 + 1*32 + 1*16 + 1*8 + 0*4 +1*2 + 1*1

or $01111011_{2.}$

It is convention to always express binary numbers using either 4, 8, 32, or 64 binary digits even if the leading digits are zero. This is also because of the way computers are built internally.


**Because the term "digit" refers to a multiple of 10, a binary digit is called a *bit*. Bit stems from *b*inary dig*it*. Eight bits make up a *byte*.**

With such a small base, it is necessary to use a large number of digits to express numbers. It is inconvenient to use an expression such as $01111011_2$ to express such a mundane value as $123_{10}$. Programmers prefer to express numbers by units of 4 bits.

A single 4-bit digit is essentially base 16, because 4 bits can express any value from 0 to 15. Base 16 is known as the hexadecimal number system. *Hexadecimal* is often contracted to *hex*.

Hexadecimal uses the same digits as decimal for the numbers 0 through 9. For the digits between 9 and 16, hexadecimal uses the first 6 letters of the alphabet: A for 10, B for 11, and so on. Thus, $123_{10}$ becomes $7B_{16}$.

```
7 * 16 + B (i.e. 11) * 1 = 123
```

Because programmers prefer to express numbers in 4, 8, 32, or 64 bits, they similarly prefer to express hexadecimal numbers in 1, 2, 4, or 8 hexadecimal digits, even when the leading digits are 0.

Finally, it is inconvenient to express a hexadecimal number such as $7B_{16}$ using a subscript because terminals don't support subscripts. Even on a word processor such as the one I am using now, it is inconvenient to change fonts to and from subscript mode just to type two digits. Therefore, programmers use the convention of beginning a hexadecimal number with a "0x" (the reason for such a strange convention goes back to the early days of C). Thus, 7B becomes 0x7B. Using this convention, 0x123 is most definitely not the same number as 123. (In fact, 0x123 equals 291 in decimal.)

## Bitwise Logical Operations

All of the operators defined so far can be performed on hexadecimal numbers in the same way that they are applied to decimal numbers. The reason that we cannot perform a multiplication such as 0xC × 0xE in our head has to do with the multiplication tables we learned in school and not any limitations of the operators themselves.

In addition to the mathematical operators, there is a set of operations based on single-bit operators. These base operators are only defined for 1-bit numbers.

### The single-bit operators

The bitwise operators perform logic operations on single bits. If you consider 0 to be false and 1 to be true (it doesn't have to be his way, but that is the common convention), then you can say things such as the following for the bitwise AND operator:

```
1 (true) AND 1 (true) is 1 (true)
1 (true) AND 0 (false) is 0 (false)
```

And similarly for the OR operator:

```
1 (true) OR 0 (false) is 1 (true)
0 (false) OR 0 (false) is 0 (false)
```

Written in tabular form it looks like Tables 7-2 and 7-3.

**Table 7-2**
*Truth Table for the AND Operator*

| **AND** | **1** | **0** |
|---|---|---|
| **1** | 1 | 0 |
| **0** | 0 | 0 |

**Table 7-3**
*Truth Table for the OR Operator*

| **OR** | **1** | **0** |
|---|---|---|
| **1** | 1 | 1 |
| **0** | 1 | 0 |

In Table 7-2, the result of 1 AND 0 is shown in row 1 (corresponding to the 1 row head) and column 2 (corresponding to the 0 column head).

One other logical operation, which is not so commonly used in day-to-day living, is the "or else" operator, which is commonly contracted to XOR. XOR is true if either argument is true, but *not true* if both are true. The truth table for XOR is shown in Table 7-4.

**Table 7-4**
*Truth Table for the XOR Operator*

| **XOR** | **1** | **0** |
|---|---|---|
| **1** | 0 | 1 |
| **0** | 1 | 0 |

Armed with these single bit operators, we can take on the C++ bitwise logical operations.

## The bitwise operators

The C++ bitwise operators perform bit operations on every bit of their arguments. The individual operators are shown in Table 7-5.

**Table 7-5**
*C++ Bitwise Operators*

| Operator | Function |
|---|---|
| ~ | NOT: toggle each bit from 1 to 0 and from 0 to 1 |
| & | AND each bit of the left hand argument with that on the right |
| \| | OR |
| ^ | XOR |

The NOT operator is the easiest to understand: NOT converts a 1 into 0 and a 0 into 1. (That is, 0 is NOT 1 and 1 is NOT 0.)

```
~0110₂ (0x6)
 1001₂ (0x9)
```

The NOT operator is the only unary bitwise logical operator. The following calculation demonstrates the & operator:

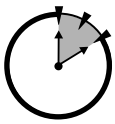
```
  0110₂
&
  0011₂
  0010₂
```

Proceeding from left to right, 0 AND 0 is 0. In the next bit, 1 AND 0 is 0. In bit 3, 1 AND 1 is 1. In the least significant bit, 0 AND 1 is 0.

The same calculation can be performed on numbers displayed in hexadecimal by first converting the number in binary, performing the operation, and then reconverting the result to hexadecimal.

```
0x6          0110₂
 &     →      &
0x3          0011₂
             0010₂      →  0x2
```


**Such conversions back and forth are much easier to perform using hexadecimal instead of decimal numbers. Believe it or not, with practice you can perform bitwise operations in your head.**



*10 Min. To Go*

## A simple test

We need an example program to test your ability to perform bitwise operations first on paper and, later, in your head. Listing 7-1 is a program that inputs two hex values from the keyboard and outputs the results of ANDing, ORing, and XORing.

**Listing 7-1**
*Bitwise operator testing*

```
// BitTest - input two hexadecimal values from the
//           keyboard and then output the results
//           of applying the &, | and ^ operations
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* nArgs[])
{
    // set output format to hexadecimal
    cout.setf(ios::hex);

    // input the first argument
    int nArg1;
    cout << "Enter arg1 as a four-digit hexadecimal: ";
    cin  > nArg1;

    int nArg2;
    cout << "Enter arg2: ";
    cin  > nArg2;

    cout << "nArg1 & nArg2 = 0x"
              << (nArg1 & nArg2) << "\n";
```

*Continued*

**Listing 7-1** *Continued*

```
      cout << "nArg1 | nArg2 = 0x"
                << (nArg1 | nArg2) << "\n";
      cout << "nArg1 ^ nArg2 = 0x"
                << (nArg1 ^ nArg2) << "\n";

      return 0;
}
```

**Output:**

```
Enter arg1 as a four-digit hexadecimal: 0x1234
Enter arg2: 0x00ff
nArg1 & nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
```

The first statement, which appears as `cout.setf(ios::hex);`, sets the output format from the default decimal to hexadecimal (you'll have to trust me that it works for now).

The remainder of the program is straightforward. The program reads `nArg1` and `nArg2` from the keyboard and then outputs all combinations of bitwise calculations.

Executing the program on the values 0x1234 and x00ff results in the output shown at the end of the listing.


**Hexadecimal numbers appear with a preceding 0x.**

## *Why?*

The purpose for most operators is clear. No one would quarrel with the need for the plus or minus operators. The use for the < or > operators is clear. It may not be so clear to the beginner when and why one would use the bitwise operators.

The AND operator is often used to mask out information. For example, suppose that we wanted to extract the least significant hex digit from a four-digit number:

```
0x1234      0001 0010 0011 0100
   &        → &
0x000F      0000 0000 0000 1111
            0000 0000 0000 0100  →  0x0004
```

Another use is that of setting and extracting individual bits.

Suppose that we were using a single byte to store information about a person in a database that we were building. The most significant bit might be set to 1 if the person is male, the next set to 1 if a programmer, the next set to 1 if the person is attractive, and the least significant bit set to 1 if the person has a dog.

| Bit | Meaning |
|---|---|
| 0 | 1→male |
| 1 | 1→programmer |
| 2 | 1→attractive |
| 3 | 1→owns a dog |

This byte would be encoded for each database and stored along with name, social security number, and any number of other illegal information.

An ugly male programmer who owns a dog would be coded as $1101_2$. To test all records in the database by searching for unattractive programmers who don't own dogs irrespective of gender we would use the following comparison.


```
(databaseValue & 0x0110) == 0x0100
                  *^^*          ^ -> 0 = not attractive
                                ^     1 = is a programmer
                       * -> no interest
                       ^ -> interested
```


**In this case, the 0110 value is known as a *mask* because it masks away bit properties of no interest.**

## REVIEW

You learned the mathematical operators presented in Chapter 6 in grade school. You probably did not learn the simple logical operators then, but they are certainly common to everyday use. Operators such as AND and OR are understandable practically without explanation. Since C++ has no logical variable type — it uses 0 to mean FALSE and everything else to be TRUE.

By comparison, the binary operators are something new. These operators perform the same logical AND and OR type operators but on each bit separately.

## QUIZ YOURSELF

1. What is the difference between the && and & operators? (See "Simple Logical Operators and Bitwise Logical Operations.")
2. What is the value of (1 && 5)? (See "Simple Logical Operators.")
3. What is the value of 1 & 5? (See "Bitwise Logical Operations.")
4. Express 215 as the sum of powers of ten. (See "Binary Numbers.")
5. Express 215 as the sum of powers of two. (See "Binary Numbers.")
6. What is 215 in hexadecimal? (See "Binary Numbers.")