Lecture 6 — Relational & Bitwise Expressions

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 6, 2016

Acknowledgments: W.D. Bishop

Boolean Variables & Relational Operators

Recall from earlier that we introduced the boolean variable concept.
The boolean variable can only contain `true` or `false`.

Using an arithmetic operator on two numbers, we get a numeric result; using a relational operator, we get a boolean result.

Relational operators compare two operands and return `true` or `false`.

Example: the result of $7 > 5$ is `true`.

Relational Operators

In C++, the relational operators are:

Operator	Definition
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Use of < where <= was intended, and vice versa, is a common source of error. Same for > and >=.

Relational Operator Examples

A few relational operator examples:

`500 == 401` $\rightarrow$ `false`
`450.0 <= 450.0` $\rightarrow$ `true`
`7 != 0` $\rightarrow$ `true`
`7 != 7` $\rightarrow$ `false`

Like arithmetic expressions, variables can be in relational expressions.

`x > 0` $\rightarrow$? The result depends on the current value stored in `x`.

Comparing Numbers

Use caution when comparing numbers using `float`, `double`, and `decimal` types.

The == and != operators especially may give results that appear erroneous when using real values.

For example, 7.2000001 and 7.2 are not equivalent.

Truncation, rounding, and the internal representation in memory may result in numbers that differ slightly.

Comparing Numbers

How does this happen? Think of $\frac{1}{3}$.

The decimal (non-fractional) representation is: 0.33333333333...
It takes an infinite number of digits to represent this correctly.

Multiply the decimal representation by 3, the result is 0.99999999...
To the computer, this is not the same as 1.

This also applies in the computer's representation of numbers.

Like $\frac{1}{3}$ for us in decimal form, to accurately store `0.001d` in memory, it would take an infinite number of bits.

Comparing Numbers Solution

When comparing two real values for equality, a tolerance may help.

Instead of comparing with == or !=, compute the difference.

The absolute value of the difference must be less than a set tolerance.

The difference of 7.2000001 and 7.2 is 0.0000001.
Then compare 0.0000001 to the tolerance (for example, 0.01).

If the difference is less than the tolerance, consider the numbers equivalent.

Boolean Logic

So far: comparison operators are where we compare two numbers.

What about comparing boolean values to each other?

There is another set of operators for this: logical operators.

This follows the rules of boolean algebra, which we'll examine now.

Boolean Algebra

Like the `bool` variable, the basic values are `true` and `false`.

In many mathematical texts, false is represented as 0 and true as 1.

There are three basic operations in boolean algebra:

1. AND (also called conjunction)
2. OR (also called disjunction)
3. NOT (also called negation)

A Note about Notation

The AND, OR, and NOT operations may appear in mathematical expressions in your future courses such as ECE 103 (Discrete Math).

You may have had some experience with these already.

Operation	Math Notation	C++ Notation
AND	$\wedge$	&&
OR	$\vee$	||
NOT	$\neg$	!

In this course, we'll use the C++ notation.

Boolean Algebra: AND operation

The result of the AND operation is true only if both *x* and *y* are true.

Truth table for the AND operation:

x	**y**	**x && y**
false	false	false
false	true	false
true	false	false
true	true	true

Boolean Algebra: OR operation

The result of the OR operation is true if *x* is true or if *y* is true, or if both are true.

Truth table for the OR operation:

x	**y**	**x || y**
false	false	false
false	true	true
true	false	true
true	true	true

Boolean Algebra: NOT operation

NOT x is a unary operator.

The result of the NOT operation is true if *x* is false. It flips the value.

x	**!x**
false	true
true	false

C++ Boolean Examples

```
bool a = b && c;
bool d = e || f;
bool g = !h;
```

Short-Circuit Evaluation

In C++, the AND and OR operations use short-circuit behaviour.

What is short-circuit behaviour? When evaluating an expression, we might know the outcome partway through.

Example: *x* && *y*. If we know *x* is false, there's no need to look at *y*.
Regardless of the value of *y*, if *x* is false, the result is false.

Code example: `x > 5 && y < 7`
If `x` is less than or equal to 5, the computer won't bother evaluating if `y` is less than 7 because the outcome is false regardless.

This is an optimization: the computer doesn't have to do the work of retrieving `y` from memory and comparing it to 7.

Short-Circuit Evaluation

Short-circuit evaluation also applies to the OR operator.

Further example: `a > 1 || b < 0`
If `a` is greater than 1, the computer won't bother evaluating if `b` is less than 0 because the outcome is true regardless.

Comparison Operator Precedence

It's important to note that in C++, the `!` negation operator has higher precedence than the && and || operators.

Example: `bool x = !y && z;` is evaluated as: `(!y) && z`
and it is NOT evaluated as `!(y && z);`

As always, precedence can be specified with the use of brackets.

Bitwise Operators

Bitwise operators are uncommonly used outside of some special cases (like hardware interaction).

These operate on the bit representation of a number in memory.

Bitwise Operator Example

Let's consider an example using Bitwise AND of some numbers: 248 and 63.

248 in binary is: 1111 1000; 63 in binary is: 0011 1111.

Line them up, and compare each of the individual bits of the first number to the one immediately below.

1111 1000
0011 1111

Here we are doing a logical AND operation, and the result is:
0011 1000 (56 in decimal).

Bitwise Operators

A list of the Bitwise Operators of C++:

Operator	Definition
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
<<	Shift first operand left by number of bits in the second
>>	Shift first operand right by number of bits in the second

Some of these require a bit more explanation...

Bitwise XOR

XOR (Exclusive OR) is another operation of boolean logic, but it's not a basic one (it's derived from the three we've already seen).

(Its math symbol is $\oplus$)

The result is true if *x* and *y* have different values.

Truth table for the XOR operation:

x	**y**	**x ^ y**
false	false	false
false	true	true
true	false	true
true	true	false

Bitwise Complement

The bitwise complement is like the negation (!) operation, but at a bitwise level.

1111 1000 (248)
becomes:
0000 0111 (7)

Shift Left, Shift Right

The left shift operator updates the first operand based on the second.

If the variable `x` is 8, its bit representation is:
0000 1000

```
int y = x << 1;
```
Shifts the bits left by 1.

Then `y` is 16; its bit representation is:
0001 0000

Right shift works the same way as the left shift operator works, but in the other direction.

Relational Operators & Type Promotion

Relational and bitwise operators still follow the type promotion rules.

The compiler will convert one of the operands, if necessary.

All Assignment Operators

Now we can see the full list of the assignment operators of C++:

Operator	Definition
=	Assigns the result to the variable
+=	Adds the result to the variable
-=	Subtracts the result from the variable
*=	Multiplies the variable by the result
/=	Divides the variable by the result
%=	Assigns the remainder to the variable
&=	Assigns the bitwise AND to the variable
|=	Assigns the bitwise OR to the variable
^=	Assigns the complement to the variable
<<=	Shifts the variable left by result bits
>>=	Shifts the variable right by result bits