Tutorial 04 - Logic

## Introduction

In this tutorial you will be introduced to the concept of logic and how it applies to software development. Mathematical Logic and Computational Logic are the foundation for modern computing and it’s important to understand a little about logic and how to apply it. Formal Logic is itself a whole subject to study that we’re not going into.

We’re already looked at how logic is used at the CPU and mathematical level, and how to systematically decompose problems and create a logical sequence of steps that solve those problems.

The type of logic we’re concerned with now is related to specific aspects of programming. At the heart of it is the concept of **true** and **false**, which influences everything you do in programming. Remember **Sequence**, **Selection** and **Iteration**? Both **Selection** and **Iteration** rely on logic, and as we’ll see, **Sequence** follows as well.

## Boolean Logic

Boolean logic really just means that a statement of something (called an expression) is either **true** or it is **false**. No other options, no combinations or ambiguity. True and false are opposites, always. This leads to some other conclusions:

* if something is true, then it is not false.
* If something is false, then it is not true

Evaluation of simple **expressions** like these is a big part of how programs work when combined with **Selection**. An expression is evaluated, and if the result is **true**, a statement is selected and executed. If the expression is **false** (which is the same as **not true**), the statement is not selected and not executed. You can probably see how this can be applied to games - for example:

* If enemy attack hits, reduce player health by 10
* If player heath is 0, game over
* If the player clicked the right mouse button, fire secondary weapon

Based on just these simple concepts and combining similar expression, you can build more complex conditions using the fundamental logical operators **AND**, **OR** and **NOT**. These operators allow you to combine multiple expressions in different ways to compare against specific conditions.

**AND** compares the results of multiple expressions; the result of which is **true** if and only if all of the compared expressions are true. Using AND is like saying “if something A is true AND something B is true AND something C is true….”. You can compare as many expressions as you want with AND, but they all must individually be true for the result to be true.

**OR** compares the results of multiple expressions; the result of which is **true** if any of the compared expressions are true. Using OR is like saying “if something A is true OR if something B is true OR if something C is true….”. You can compare as many expressions as you want with OR, with the result being true if any are true.

**NOT** flips the logical state of an expression, allowing you to use logic more freely, especially when combined with AND and OR. You can use NOT to flip an expression to check for false: “If something A is NOT true…” or to check for true: “If something B is NOT false…”. NOT is very powerful but needs good examples to show it off.

Some examples of more complex logic might be:

* If the enemy attack hits **AND** the enemy attack rating is greater than the player defence rating, reduce player health by 10
* If the player health is 0 **OR** the enemy health is 0, game over
* If the player clicked the right mouse button **AND** the player has an item selected **AND** the selected item has a secondary attack function **AND** the select item’s secondary ammo count is **NOT** empty , use secondary attack
* If the player chose Rock **AND** the opponent chose Scissors **OR** If the player chose Paper **AND** the opponent chose Rock **OR** If the player chose Scissors **AND** the opponent chose Paper, then player wins

So all this gives you a really power set of tools – but potentially there is a weak link: the logic within each expression. You have to be careful that the expression you are evaluating is properly stated and considers all of the conditions. Perhaps you noticed a potential problem with these two earlier examples?

* If enemy attack hits, reduce player health by 10
* If player heath is 0, game over

What if the player’s health is 5 when the enemy hits? The player’s health would then be reduced to -5. The next check only looks at the player’s health being equal to 0 – oops. This is a potentially major bug! The kind that allows cheating in multiplayer games.

Perhaps a safer approach for the second line might be one of these:

* If player heath is less than 1, game over. “Less than 1” covers 0 and negative numbers.
* If player heath is less than or equal to 0, game over. “Less than or equal to 0” also covers all cases.

## Truth Tables

Sometime you have complex comparisons and it’s difficult to get your head around which resulting conditions you’re checking for and how to use NOT, and so on. Truth Tables are a simple but effective tool to help with that, by showing the results of your comparisons in a little table/chart. They list the possible states of your individual expressions, and how AND, OR and NOT affect the result.

The first table shows the result from comparing two expressions (P and Q are the two expressions – don’t ask why P and Q, it’s just the way it’s done by logic academics) using the **AND** operation, and the second table shows the **OR** operations.

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P AND Q** |
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P OR Q** |
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

**NOT** is a simple case, shown here:

|  |  |
| --- | --- |
| **P** | **NOT P** |
| F | T |
| T | F |

Two additional operations will be thrown in here, just so you have seen them all. These are very important for computing hardware, but not as much for code.

**XOR** is the exclusive or. It means “if P is true or if Q is true, but not if both are true”.

**NAND** means “not and”, and it’s essentially the same as NOT AND: “if P is not true AND if Q is not true”. Or more precisely, “NOT (if P is true AND Q is true)”.

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P XOR Q** |
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | F |

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P NAND Q** |
| F | F | T |
| F | T | T |
| T | F | T |
| T | T | F |

Note the parentheses on the NAND – they are important because of **precedence** and **associativity** when you’re writing code, just as with some math operators. It doesn’t matter as much when you’re using words because our brains automatically interpret at least some of this correctly, but it’s very important in programming because you cannot rely on the computer/compiler to know what it is you want.

NOT is evaluated first, then AND, and lastly OR.

If the parentheses were omitted in the NAND explanation, it could have been interpreted as “if P is NOT true AND Q is true”.

The earlier complex example could (maybe should) be rewritten with parentheses to show clearly how it should be evaluated:

* (If the player chose Rock **AND** the opponent chose Scissors) **OR** (if the player chose Paper **AND** the opponent chose Rock) **OR** (if the player chose Scissors **AND** the opponent chose Paper), then player wins.

As this last example shows, sometimes you will have complex and difficult logical. Truth Tables are a useful when planning out the code to handle these situations to ensure that you check for exactly the right conditions, so your code works properly without bugs. They are perfectly valid design tools to include with your projects, as are the types of statement used in this section as part of your pseudocode.

## Relational Operators

Now to start bringing these logical concepts into actual programming. You have already looked at the **Boolean** data type which holds the values **true** and **false**. You’ve just gone through an introduction to logic and now you need to start putting it together.

So how to apply these logical expressions and use Boolean logic effectively? There are a few ways, but the main approach is to use **relational** **operators**, which are used to *compare* values, the result of which is always a boolean **true** or **false**.

In programming languages, **operators** are symbols that represent ways to modify or compare data, usually stored in variables. You have already used some of the **mathematical** operators with the fundamental data types, such as ‘**=**’ (assignment), ‘**+**’ (addition) and even ‘**<<**’ (stream insertion).

The most basic example of a relational operator is the **equality operator '**==**'**, which consists of two equal signs. Using the equality operator in expressions results in a boolean:

(5 == 5) is true

(5 == 6) is false

Of course, comparing two literals is fairly pointless, so generally the relational operators are used in combination with variables, so for instance if you had a variable score that was assigned 200, and high\_score that was assigned 300, then:

(score == 200) is true

(score == 999) is false

(score == high\_score) is false

C/C++ uses several other relational operators, listed in the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator | Meaning | Examples | | |
| == | equal to | (5 == 5) is true | (5 == 6) is false |  |
| != | not equal to | (5 != 6) is true | (5 != 5) is false |  |
| > | greater than | (5 > 4) is true | (4 > 5) is false |  |
| < | less than | (2 < 7) is true | (7 < 2) is false |  |
| >= | greater than or equal | (4 >= 3) is true | (3 >= 4) is false | (5 >= 5) is true |
| <= | less than or equal | (1 <= 9) is true | (9 <= 1) is false | (6 <= 6) is true |

You can look back in at the previous section on boolean logic and see where the relational operators would be used in the code that matches the pseudocode in the examples.

## Logical Expressions and **if()** Statements

In order to do anything useful in a program we need a way of changing what happens based on a specific condition (or set of conditions) being **true** or **false**. Just like the pseudocode in the first sections, in C++ we use an "if" statement to evaluate an determine which code to run next

The if() statement works as follows

if(*expression* is true)

{

// do something in this statement block

}

The ***expression*** can be anything that gives a true or false value (like we've seen earlier), and if it is true then any code in the statement block following the if() is executed, otherwise it is skipped. Here are some examples with different kinds of expressions:

if(true)

{

cout << "this ALWAYS executes");

}

if(false)

{

cout << "this NEVER executes" << endl;

}

and here with operators:

int score = 500;

int high\_score = 400;

if(score > high\_score) // "if score **is greater than** high\_score"

{

cout << "You've got the new high score!" << endl;

}

int lives = 3;

if(lives != 0) // "if lives **is not equal** to zero"

{

cout << "You're still alive!" << endl;

}

int my\_height = 175, your\_height = 175;

if(my\_height <= your\_height) // "if my\_height **is less than or equal to** to your\_height"

{

cout << "You are not taller than me.");

}

There are a couple of potential problems you may come across when using if() statements, especially if you are new to C/C++. One difficult to spot error is a semi-colon added after the if() statement, for instance:

if (false); // DON'T DO THIS!

{

cout << "this should NEVER execute" << endl;

}

This code will compile and run, but unfortunately you will see "this should NEVER execute" displayed. This is because C/C++ allows you to use **empty statements**. This is a statement which does **nothing**, this may sound strange but there are uses for this in more advanced code. In C/C++, they are represented somewhat confusingly as a semi-colon on its own. For now all you need to know is that if you use the above code then the if() statement is associated with the empty statement represented by the semi-colon, and it is the empty statement that doesn't get executed. Therefore it jumps to the next statement, which happens to be the cout that should never be executed in the following statement block.

One of the biggest cause of logical errors in C/C++ programs is confusing '=' and '==':

int score = 400;

int high\_score = 500;

if (score = high\_score) // DON'T DO THIS! Use == in if() statements

{

cout << "You've got the new high score!" << endl;

}

This code will also compile and run, but will not do as expected. Because the assignment operator has been used '=', instead of a comparison being made the value of score is set to the value of high\_score, which is probably not what was intended.

Always check your if() and assignment statements are using the correct operator, especially if things aren't working as expected.

**Best Practice**

Notice the different use of the equal signs in the first assignment statement. The single '=' sign represents an **assignment** that sets a variable, and the double '==' sign is the **equality** relational operator doing a comparison. Confusing these two is a common cause of errors in C/C++.

If you catch yourself making this mistake often, try this trick: if you are comparing a variable to a constant or literal value, always put the variable on the *right* of the operator:

if (200 == score) …

If you accidentally use a single '=' (assignment) operator, it won't compile – the compiler catches the mistake for you.

## **if..else()** Statements

Sometimes you may want to also perform alternate operations if the expression resolves to **false**, and the way to do this is to use if..else() statements. Say, for instance you wanted to extend the situation above where a new score is compared to the high score, and always print a message telling the user how their score compares. Pseudocode for this would be something like:

* If player score is greater than the high score, print "You've got the new high score!"
* Otherwise, print "You didn't get the high score"

This could be done with two if() statements alone:

if (score > high\_score)

{

cout << "You've got the new high score!" << endl;

}

if (score <= high\_score)

{

cout << "You didn't get the high score" << endl;

}

Notice that the second if uses the opposite test, i.e. **less-than-or-equal** instead of **greater-than**. But the second test is unnecessary – we already know that because the first comparison was false, the second *must* be true.

Rather than performing two tests it's clearer and more efficient to use the if..else()statement, which works as follows:

if (*expression* is true)

{

// do something

}

else

{

// do something different

}

This can be understood as "**if** the expression is true do something, **otherwise** do something different". Note that **only one** of the statement blocks will ever be executed - if..else() creates a do "one or the other" situation.

Here is the flow diagram that demonstrates a simple if..else()statement, comparing the player's score against the recorded high score. Note the connection point where the logic meets.

**FALSE**

**TRUE**

**score > high\_score**

**Display**

**"No High Score"**

**Display**

**"High Score"**

The score comparison code can now be more efficiently written:

if (score > high\_score)

{

cout << "You've got the new high score!" << endl;

}

else // if (score <= high\_score)

{

cout << "You didn't get the high score" << endl;

}

#### Avoiding side effects

There's another good reason for using the if..else() format in situations where different sections of code are executed on the differing outcomes of a comparison, and that is to **avoid side effects**. Consider the following code that keeps track of the highest number input by the user:

if (score > high\_score)

{

cout << "You've got the new high score!" << endl;

high\_score = score;

}

if (score <= high\_score)

{

cout << "You didn't get the high score" << endl;

}

First off, this code is trying to do something properly – the previous example printed a message when a score beat the high score, but it didn't update the high score to the new score. Fixing this oversight is a good thing.

But because this code considers both cases separately, if a new high score is reached, the first if() statement will be true, so the **"You've got the new high score!"** text will be output and the high\_score variable will be updated. However this means that when the second if()statement is evaluated it will also be true, as high\_score has already been updated to equal the score, so **less-than-or-equals** check will be **true** and the **"You didn't get the high score**" will **also** be output which is obviously incorrect. This is called a **side-effect**, and obviously this should be avoided. You can sometimes get around these problems by reorganising the code, but this can be difficult to manage and error prone. In this simple case, a better way would be:

if (score > high\_score)

{

cout << "You've got the new high score!" << endl;

high\_score = score;

}

else //if (score <= high\_score)

{

cout << "You didn't get the high score" << endl;

}

Only one of the statements can be executed for any given test, so no side effect will occur.

## Using Boolean Variables with Logical Expressions

In a previous tutorial you learned about the boolean data type that holds the values true and false. It might not have seemed very useful then, but hopefully you will see the value now: you can use boolean variables to store the result of a logical expression or a comparison using relational operators:

bool my\_bool = true;

if(my\_bool == true) // "if my\_bool **is equal** to true"

{

cout << "my\_bool is true, so this code executes" << endl;

}

if(my\_bool == false) // "if my\_bool **is equal** to false"

{

cout << "If my\_bool is true, this DOESN'T execute" << endl;

}

Boolean variables are extremely useful, to store results of comparisons for later evaluation or to record the status of something in your program. You will use boolean variables like: has\_weapon\_fired, player\_is\_dead, is\_game\_running, did\_attack\_hit. Note how these variables are named – they contain words like "is" and "has", which help make it obvious they are Booleans. The value they hold (true or false) should effectively answer the question the variable name is asking.

This longer example shows use of a boolean to record whether a new high score has been set by the player:

int score = 200; // a real value would probably come from playing the game

int high\_score = 300; // a real value would probably come from a high score list

bool is\_new\_high\_score = false; // initialize to false as a default

is\_new\_high\_score = (score > high\_score); // is\_new\_high\_score is true if score > high\_score

if (true == is\_new\_high\_score) // see how the order is reversed? true is on the left!

{

cout << "You've got the new high score!" << endl;

high\_score = score;

}

else //if (score <= high\_score)

{

cout << "You didn't get the high score" << endl;

}

## Logical Operators

Using expressions as part of selection to determine program flow is a key part of programming. The expressions used in the code examples so far are very normal and similar to what you can expect. But they have mostly been very simple – sometimes you need more complex logic to precisely determine which blocks of code to execute.

Remember this bit of pseudocode from Boolean Logic?

* If the enemy attack hits **AND** the enemy attack rating is greater than the player defence rating, reduce player health by 10

Using normal if() statements, this would be written as two comparisons – if the first condition is true, then evaluate condition is true, and if it is also true, then perform the action, something like:

if (true == did\_enemy\_attack\_hit)

{

if (enemy\_attack\_rating > player\_defence\_rating)

{

player\_health -= 10;

}

}

This works, but it's a little crude. And it would get very ugly if you have more conditions to check:

* If the player chose Rock **AND** the opponent chose Scissors **OR** If the player chose Paper **AND** the opponent chose Rock **OR** If the player chose Scissors **AND** the opponent chose Paper, then player wins

**Logical operators** allow you to readily evaluate these complex situations that involve combinations of expressions. The words **AND** and **OR** above are simply replaced by the symbols used in C++ to represent those logical operations:

&&- thedouble ampersand is logical **AND** operator

|| - the double vertical bar is the logical **OR** operator

The format of the **AND** operator && is:

(expression1) && (expression2)

The && operator evaluates to **true** only if **both** expressions evaluate to **true**. If either or both are **false** then it evaluates to **false**. This can be summarised in the following table:

|  |  |  |
| --- | --- | --- |
| **Logical AND (&&)** | | |
| **expression1** | **expression2** | **expression1 && expression2** |
| **true** | **true** | **true** |
| **true** | **false** | **false** |
| **false** | **true** | **false** |
| **false** | **false** | **false** |

Logical operators are not limited to connecting two expressions – any number of expressions can be 'chained' together with the && operator. The same rules apply regardless of how many expressions you connected this way..

(expression1) && (expression2) && (expression3) && (expression4) …

The logical **OR** operator is represented by || (These characters are not lowercase 'L' or numeric '1' but the vertical line character '|' (On a UK keyboard it is usually above the '\' character).

The format of this operator is:

(expression1) || (expression2)

The || operator evaluates to **true** if **either** expression is **true**. This can be summarised as follows:

|  |  |  |
| --- | --- | --- |
| **Logical OR (||)** | | |
| **expression1** | **expression2** | **expression1 || expression2** |
| **true** | **true** | **true** |
| **true** | **false** | **true** |
| **false** | **true** | **true** |
| **false** | **false** | **false** |

Just like the && operator, the || operator can connect as many expressions as you need to evaluate more complex logical conditions. Using the Rock. Paper, Scissors example, the logic to determine if the player wins combines both && and ||:

if ( ((player\_chose\_rock == true) && (opponent\_chose\_scissors == true)) ||

((player\_chose\_paper == true) && opponent\_chose\_rock == true)) ||

((player\_chose\_scissors == true) && (opponent\_chose\_paper == true)))

{

player\_wins = true;

}

**Best Practices**

Notice the use of parentheses (or brackets) to ensure the logical operations are done in the right order. Each of the expression connected by **&&** are evaluated before those connected by **||** because **&&** has higher **precedence**. The parentheses help make what you intend clear, and serve as reminders in case the code has to be changed. It is highly advisable to always use brackets in logical expressions in the same way as any other expression, even if it's possible to write it correctly without.

The last logical operator is **NOT**, represented by the ! symbol. This is used with a single expression in the format:

!(expression)

and simply inverts the result, so applying **NOT** to a **true** expression returns **false**, and vice versa. The table summarising its operation is:

|  |  |
| --- | --- |
| **Logical NOT (!)** | |
| **expression** | **!(expression)** | |
| **true** | **false** | |
| **false** | **true** | |

The following code demonstrates its use, checking if integer1 is **not** 3 **or** 7:

if( !((integer1 == 3) || (integer1 == 7)) )

{

cout << "Integer1 is NOT equal to 3 nor equal to 7" << endl;

}

The **NOT** operator has a higher precedence than the other logical operators, so if brackets were not used:

if( !(integer1 == 3) || (integer1 == 7) ) // missing brackets

{

cout << "Integer1 is NOT equal to 3 nor equal to 7" << endl; // wrong - will be output if integer1 = 7

}

The code would not work correctly when 7 is input, as !(integer1 == 3) would be evaluated first, which would resolve to **false**, then this **false** would be **OR'd** with (integer1 == 7), which would resolve to **true**.

The **NOT** operator can be used to negate the evaluation of any number of expressions. The example above used two expressions which are **OR'd** – more expressions which are **AND'd** or **OR'd** with any complexity could be negated in the same way – but be careful with the parentheses.

**NOT** can be used to negate itself even – a double negation:

! ( ! expression ) == expression

## Iteration

As you've seen, **selection** statements allow a program to choose **whether** to execute a section of code.

A **loop**, which is the name generally used in programming for the **iteration construct**, allows a program to ***repeat*** a section of code. Just like with selection, a loop requires a boolean value to determine whether to repeat the loop. Each time through the loop is called an **iteration of the loop** (to iterate means to repeat a process).

Most algorithms you come across or design will require some tasks to be carried out more than once, and loops allow you to perform these repetitive tasks by repeating the same code over and over on different sets of data or on data that is changing, as opposed to writing the code out multiple times. Another advantage is that often you will not know until the program is running how many times to repeat an action. It would be almost impossible to do this without a loop, but with a loop you can adjust the number of iterations of the loop as the program is executing.

## **while()** Loops

The most basic form of loop is the while() loop. This loop evaluates a conditional expression, and if it is **true** then the associated statement block will be executed. Once the end of the statement block is reached, the program **jumps back** (or **loops back**) to the conditional expression, and **re-evaluates** it to see if the statement block should be executed again. This iteration continues until the expression evaluates to **false** then the associated statement block is skipped, and program execution continues from the first statement **after** the loop.

This is the format of a while() loop:

while(*expression* is true)

{

// repeat the code in this statement block

}

The format of a while() loop is very similar to an if() statement. The conditional expression that is evaluated works in exactly the same way in that it resolves to **true** or **false**, and the code in the statement block immediately after the while() loop will only execute if the condition is **true**. The difference with the while() loop is that once the program reaches the final brace '}' the execution **jumps back up** to the while() statement and the expression is tested again. This process will repeat until the expression becomes **false**.

Here is a simple flow diagram of a while() loop. Try to figure out what it does before going to the next page.

**my\_integer = 0**

**my\_integer < 10 high\_score**

**FALSE**

**TRUE**

**my\_integer ++**

**cout << my\_integer**

Here's the corresponding C++ code:

int my\_integer = 0;

while (my\_integer < 10)

{

cout << my\_integer << ', ';

my\_integer++;

}

This while() loop will continue iterating as long as the value of my\_integer is **less than** 10. Inside the loop statement block (also called the **loop body**), my\_integer is incremented, increasing its value by 1 with each iteration of the loop. This continues until the value of my\_integer **reaches** **10** when the expression becomes **false**, the loop ceases to iterate and the code execution carries on with the next statement after the end of the loop body. When this code is run, the values of my\_integer will be 0, 1, 2, 3 ... 9 consecutively as the loop repeats, so this code will output the numbers 0 to 9, separated by commas.

It's generally advisable to use a **control variable** that is evaluated in the expression in order to determine when the loop should end. In the above example my\_integer was used as the control variable, and its changing value was used to determine when the loop ended. Here's a loop that uses my\_integer as a control variable to count down instead of up:

int my\_integer = 100;

while (my\_integer >= 0)

{

cout << my\_integer << endl;

my\_integer -= 5;

}

The loop will continue to iterate as long as the value of my\_integer is greater than or equal to zero, which is **true** at the start of the first iteration as my\_integer starts with a value of 100. Inside the loop, my\_integer is decremented by 5 each iteration. So the values of my\_integer will be 100, 95, 90, 85, 80 ... 10, 5, 0 in the loop, terminating when my\_integer is equal to -5, and these values will be displayed each iteration

It's also common to use a boolean variable as the control, especially if there are multiple conditions within the loop to check that would trigger it to end:

bool is\_game\_over = false;

while (false == is\_game\_over)

{

… // a bunch of game code runs here

if (player\_is\_dead == true)

is\_game\_over = true;

… // more game code runs here

if (enemy\_is\_dead == true)

is\_game\_over = true;

}

#### Avoiding Unintended Infinite Loops

If there isn't some kind of variable or condition that changes within the loop, then the loop expression will **never become false** and you will end up with an **infinite loop**. An infinite loop can be useful in certain more advanced code, but in most situations infinite loops are not what you want. For example, in the following code the variable being tested in the expression is **not** updated in the loop:

int my\_integer = 0;

int my\_test = 0;

while(my\_test < 10) //logic error! my\_test never changes, loops forever

{

cout << my\_integer << endl;

my\_integer++;

}

The my\_test variable isn't updated inside the loop, therefore the condition can never become **false**, so this loop will **repeat forever** - essentially the program will appear to have stopped working. Another way of ending up with an infinite loop is if the condition in your expression can never be reached, for instance

int my\_integer = 6;

while(my\_integer != 5) //logic error! my\_integer will never be 5, loops (almost) forever

{

cout << my\_integer << endl;

my\_integer++;

}

The first type of unintended infinite loop is relatively easy to track down as it's usually obvious from looking at the statements in the loop body, or by using the debugger to step through several iterations of the loop, that the variable being tested is not being updated. The second type can be harder to spot, as they often occur because a variable outside of the loop has been changed to an unexpected value. In this case the debugger is your friend again as you can stop the code just before the loop starts and see if the variable being used in the condition has a valid value.