10.1 Truisms And Falsehoods

Remember our dear pal George Boole, whom we most recently discussed when thinking about relational operators? Well, he’s back at it again, and this time, with a new set of boolean operators that work their magic in combining small-scale truisms and falsehoods into full blown logical expressions.

Logical expressions are quite common in C programs, and help govern all sorts of behavior in programs, as we will soon see when we discuss decisions and control. For example, we might want to have a program print out your name only if your first name is “Remzi” AND your last name is “Arpaci-Dusseau”\(^1\). In this example, the AND is a binary boolean operator (known as logical-and), which takes two boolean parameters and produces a single boolean result.

In this chapter, we’ll provide a quick overview of Boolean logic and then show how to utilize said logic within C. We’ll also point out a few oddities that can (and do) arise.

10.2 Basic Logic

Let’s first introduce the operators we are interested in. They are logical-and (AND), logical-or (OR), and logical-not, a.k.a., logical negation, (NOT). They work as you might expect: \(x \text{ AND } y\) is true if and only if both \(x\) and \(y\) are each true; \(x \text{ OR } y\) is true if either \(x\) or \(y\) is true; finally, \(NOT\ x\) is true if \(x\) is false and false if \(x\) is true. As usual, the basics are pretty easy.

\(^1\)Names chosen randomly; no nepotism or tampering has been detected.
We often show how said operations work via what are called \textit{truth tables}. These tables simply enumerate all possible inputs and show the value of the operator as output. See Figure 10.1 for the complete truth tables for logical and, logical or, and logical not.

### 10.3 Boolean Operators In C

As we established earlier, Boolean values are represented in C as integer values, where 0 means false and any other integer value represents true (though we often think of Boolean true as 1). To operate on these values, we need to know what the C version of those logical operators are; how do you express $\textit{AND}$, $\textit{OR}$, and $\textit{NOT}$ in C?

In C, we have the following three boolean operators: `&&` (two ampersands) for logical-and, `||` (two vertical bars) for logical-or, and `!` (an exclamation point) for logical-not.

With this knowledge, we can now evaluate a few C boolean expressions. Let’s start with the truth table for logical and:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$\text{AND}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

As you should know from the truth table above (and, from general knowledge), these four expressions evaluate to 0, 0, 0, and 1, respectively; only when both parameters to `&&` are 1 is the resulting output also 1 (true).

The truth table for logical or is similarly simple to express, with each of the following evaluating to true (1) except for the first row (i.e., `0 || 0` evaluates to 0, or false).

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$\text{OR}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Finally, we show the C truth table for logical-not. You can probably guess that `!0` evaluates to 1, and that `!1` to 0.
Because we have added new operators, we must also update the precedence chart. Figure 10.2 does exactly that, showing where the new logical operators slide in. Logical not has very high precedence and associates right-to-left; logical-and and logical-or are relatively low with logical-and having the higher precedence.

### 10.4 More Complex Logical Expressions

Programs commonly combine relational comparisons (as seen in the last chapter) with logical operators (as seen in this one) to form more complex logical expressions. For example, let’s look at the following expression:

\[ 3 > 2 \land \land 0 < 1 \]

From Figure 10.2, we know that the comparisons \(<\) and \(>\) have higher precedence than \(\land\), and thus we can think of this expression as this:

\[(3 > 2) \land (0 < 1)\]

Rewritten as such, it should be clear that the expression evaluates to 1 (true), as three is indeed greater than two and zero is less than one. If we (for some odd reason) wanted to override the normal precedence rules, we could write something like this instead:

\[3 > (2 \land \land 0) < 1\]

Doing so would cause 2 (which is logically true, recall, as it is non-zero) and 0 to be and’d together first (result: 0, or false). Then, we have the expression:

\[3 > 0 < 1\]

This expression we evaluate left-to-right, and thus we first establish that three is greater than zero (result: 1, or true); finally, we are left with 1<1, which is clearly false (result: 0).

It’s true you probably won’t encounter expressions like the last one in any real code. However, because it’s valid C, you should be able to reason through what it will evaluate to, just to make sure you understand how the language works.
10.5 De Morgan’s Laws

We can also mix logical-not into some more complex expressions. For example, let’s look at the following expression:

\[ ! (x \& \& y) \]

This expression is equivalent to the negation of logical-and, and thus is true unless both \( x \) and \( y \) are true (e.g., 1). The expression can also be rewritten as follows:

\[ !x \mid \mid !y \]

The equivalence of these two expressions was discovered many years ago by Augustus De Morgan; hence, we still refer to it as De Morgan’s law\(^2\).

See if you can prove to yourself that the two are equivalent using truth tables to enumerate all possible values. It’s not too hard, and this style of proof (by exhaustive or brute force techniques) is one of the easiest proof types you’ll ever encounter\(^3\).

There is a corollary to the equivalence above. Namely, \( ! (x \mid \mid y) \) can be rewritten as well using only negation (\( ! \)) and logical-and (\( \& \& \)). Can you figure out what this alternate expression is? If so, you’re just as smart as De Morgan, probably.

10.6 Potential Confusions And Other Stuff

There are a few odds and ends worth understanding in slightly more detail before closing. The first is a subtle confusion with some other operators that we haven’t yet introduced: the bitwise operators \& (called bitwise-and), | (called bitwise-or), and \~ (the tilde symbol, called bitwise-complement). These operations, which we will delve into later, look syntactically similar but perform a much different function; for example, bitwise-and takes two inputs and performs a logical-and over each pair of bits. Do not accidentally use these bitwise operations in places where you are looking to use logical operators.

If you are familiar with logic, you know there is a fourth operator, the exclusive or, or XOR. The logical-xor operator returns true if either input is true but not both. In C, however, there is no logical-xor. There are some interesting reasons for this, as described elsewhere [P04]. One simple reason: you can simply write it as \( x \neq y \) (assuming that \( x \) and \( y \) are either 0 or 1; if they aren’t, a slightly more general expression is required).

\(^2\)It was a lucky time to be alive in the mid 1800’s, with so little known and so much to yet be discovered, particularly if you could ignore the fact that you were likely to live a short, disease-ridden, and unhappy life.

\(^3\)Indeed, even systems-types like the authors can manage this style of proof, usually.
Finally, there is an interesting short cut of sorts that C takes when evaluating certain types of expressions known as **short circuiting** [M+62]. We’ll learn more about short circuits when we (finally) get to decision-making within programs (very shortly), but the basic idea is simple. Assume we have an expression such as `$x && y$`. If we evaluate `$x$` first, and see that it is zero, we know at that point that the expression will evaluate to zero, because logical-and requires both inputs to be true (`1`) for the output to be true (`1`). Thus, in this example, when it is determined that `$x$` is zero, C does not evaluate the latter part of the expression; we call this a **short circuit**\(^4\).

Short circuiting is useful for a few reasons. First, it can improve efficiency, as unneeded computation is avoided. Second, it can be used in constructs to build safety checks into code; the right part of the expression is only run, for example, if the left part (the check) says that it is safe to do so. If this doesn’t completely make sense at this point, it will soon enough, so be patient (or not, your choice!).

### 10.7 Summary

We have introduced logical operators for and, or, and not, and shown how to express them in C. Thus, with our knowledge of comparison operators, we can write complex and rich logical expressions that evaluate to true or false. But why do we need these expressions? Well, not surprisingly, they form the core of decisions programs routinely need to make, and decision making is (finally) our very next topic. If, that is, you decide to continue...

\(^{4}\)A parallel construction exists for logical-or, and the first value evaluating to `1` (true); in that case, you know the logical-or is going to be true so no need to evaluate the second part of the expression.
References

[P04] “Why is there no ‘logical exclusive or’ operator?”
Ben Pfaff
http://benpfaff.org/writings/clc/logical-xor.html
A nice short answer to the question.

[M+62] “LISP 1.5 PROGRAMMER’S MANUAL”
John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin
August, 1962
McCarthy and friends did pioneering work in AI, invented the Lisp programming language, and
even showcased how to do short circuit evaluation (sometimes thus called McCarthy evaluation,
but not that often). McCarthy even claims to have invented time sharing and multiprogramming,
an approach now commonly used in operating systems, but you’ll have to read some other book
to learn about that.