Relational Operators

In which our hero discovers the joy and folly of comparison, and that the result, however Boolean, seems to boil down to just a zero or a one.

9.1 Must We Compare?

In life, as in computer programming, comparison forms the core of decision making. For example, if two apples, a Honeycrisp and a Red Delicious, cost the same amount of money, but the Honeycrisp is a much tastier treat than the Red Delicious\(^1\), then you should likely buy the Honeycrisp. Indeed, if you remember nothing else from these readings, it should be this: almost always, you should buy the Honeycrisp.

In C, comparison is achieved via a new set of what are called relational operators; each of these binary operators compare (or relate) two expressions and then produce a value based on the result of the comparison. The resulting value is what is generally referred to as a Boolean value, a fancy way of saying that can only be true or false.

For example, we might have the following relational expression:

\[
10 < 15
\]

This expression compares two values, the 10 on the left and the 15 on the right, with the less-than operator (\(<\)). In this case, 10 is indeed less than 15, and thus the result is true. But, as is always the case with C (and really, with any other language), there are some relevant details which we have not yet discussed, and now do. For example, what other operators are there? How does C represent these Boolean true/false values? And, most importantly, does this really count as a third question?

\(^1\)Is there a more serious misnomer than “Red Delicious”? Blech, they taste like mushy cardboard. Yes, we are very serious about our apple varietals in this book, as should you be. One new variety we are far too excited about: the Cosmic Crisp, coming soon to a store near you.
### Precedence Operator Associativity

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>postfix++/--</code></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td><code>++/--prefix unary+ unary-</code></td>
<td>Right-to-left</td>
</tr>
<tr>
<td>3</td>
<td><code>* / %</code></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td><code>+ -</code></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>5</td>
<td><code>&lt; &lt;= &gt; &gt;=</code></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>6</td>
<td><code>== !=</code></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>7</td>
<td><code>+= -= *= /= %=</code></td>
<td>Right-to-left</td>
</tr>
</tbody>
</table>

Figure 9.1: C Operator Precedence (Version 5)

#### 9.2 And The Relational Operators Are...

There are six relational operators over the integers, and you probably can guess what they are and what they do. The six operators are the previously seen **less-than** `<`, **less-than-or-equal** `<=`, **greater than** `>`, **greater-than-or-equal** `>=`, **not-equal-to** `!=`, and, finally **equal-to** `==`, or two equals symbols).

Because they are operators, they must also be assigned a precedence, and thus our precedence table must grow again. Figure 9.1 shows where the relational operators are placed; as you can see, the **relative quantity** operators `< <= > >=` are below the mathematical operators but higher than the exact equality/inequality operators `== !=`.

#### 9.3 Boolean Representation

Some programming languages have an explicit Boolean type that represents true or false values. C, however, takes a different route, and essentially produces an integer wherever you think a Boolean should arise. For example, the relational expression `10 < 15` we saw above is true, and thus would evaluate in C to the integer value 1; similarly, the expression `10 > 15` evaluates to the integer value 0.

When we use any of the relational operators above, the result will be an integer 1 or 0. However, that does not mean other numbers cannot be used where you might expect to see a Boolean value. Specifically, *any* non-zero integer value is treated as true, whereas only zero is treated as false. Later, we’ll see where this fact is put into use; the important thing now is to remember it.

#### 9.4 Example Relational Expressions

Let’s look at a few examples. Here is a list of expressions that are all true (and thus evaluate to 1):

\[
\begin{align*}
15 & < 20 \\
15 & > 10
\end{align*}
\]
RELATIONAL OPERATORS

15 >= 15  
15 <= 15  
15 == 15  
15 != 10

Pretty simple. Now, let’s examine a few stranger examples. Here is the first one; before reading further, try to think about what the value of the expression should be.

16 > 2 + 15

The answer, in this case, is 0. Why? Well, as the precedence table tells us, addition has higher precedence, and thus the addition is ordered earlier than the comparison; as 16 is not greater than 2+15, the Boolean result is false which in C is represented as a zero.

The strange nature of Boolean values in C allows comparisons to be included in what look like mathematical expressions, with predictable (but odd) results. For example, it is perfectly legal to write an expression like this:

(16 > 2) + 15

In this case, because of our use of parentheses, the comparison is evaluated first, with the result of true (1). In the next step, this integer value of 1 is added to 15, and thus we arrive at the final answer: 16.

Other unexpected results arise if you bring too much mathematical knowledge to the table. For example, consider the following expression:

3 > 2 > 1

If you think about this mathematically, it looks to be true, because 3 is indeed greater than 2, and 2 is greater than 1. But in C, this expression is false; can you figure out why?

The answer once again lies in the precedence table, which tells us that > is evaluated left-to-right. In this case, you must then first compare 3 and 2; because 3 is indeed greater than 2, the result is true, or 1. And now we see why the expression is false (0); 1 is not greater than 1.

Let’s look at one last example. This one also evaluates to true (1); can you see why?

1 == 6 > 5

Did you really stop and think or just read on? Come on, we need to trust each other if this is going to work. Think, then read on!

OK, we trust you. And we trust that you looked at the precedence table and noticed how equality and inequality checks are lower in
precedence, and thus first $6$ would be compared against $5$, with the result being true (1), and that then clearly the final result is true (1) because of course 1 does indeed equal itself.

**TIP: DON’T CONFUSE = AND ==**
A common bug in C programs is the accidental use of a single $=$ symbol when two were meant. Part of this, we think, arises from the fact that when you read the expression “Does $(x = y)$?” in English you say the correct thing, which is “Does $x$ equal $y$?” Unfortunately, while this reads well, it doesn’t (usually) act in the way you’d like. Thus, it’s your job to be extra careful in this specific case, and avoid the hours of debugging pain that can arise with this easy-to-make but hard-to-notice mistake.

### 9.5 The Biggest Mistake

Comparisons are mostly easy to get right. However, there is one big mistake that C programmers (even seasoned ones) make that it is worth calling it out here. The mistake is this: the accidental confusion of assignment ($=$) and comparison for equality ($==$).

Let’s assume you wish to compare the value of a variable, $x$, to some other variable, $y$. Specifically, we wish to check if they are equal. This check would be properly written as follows (either with parentheses, as shown, or without):

$$(x == y)$$

In this case, if $x$ were equal to 3, for example, and $y$ were equal to 4, the expression would evaluate to false (0), but if $y$ were also 3, to true (1).

However, think about what might happen if you typed the following expression somewhere into your C program:

$$(x = y)$$

This expression, of course, is an assignment, but recall that assignment is indeed an operator, and that it produces a value. In fact, the value of this expression is the value of the right-hand side, i.e., the value of $y$. Thus, if you are treating the result of this expression as a Boolean value, it would _almost_ always evaluate to true; in fact, the only time it would not is the case where $y$ is equal to zero.

More importantly, when writing an expression like this, you won’t get any warning or error (usually). The program will just silently assume you know what you are doing, and go about its merry way. This is C; a sharp knife that will cut you if you are not careful. And
so, the lesson: be careful, friend. Very, very careful. And realize: this is just the beginning, and it only gets worse from here².

9.6 Summary

Twain famously said that “comparison is the death of joy”³, but in computer programming, it’s the beginning of taking control of the flow of a program, which we will very shortly do. Thus, it is important understand the basics of how such comparisons work.

We’ve seen the introduction here of six new comparison operators, and the resulting Boolean values they produce. In C, we represent the results of comparisons in a primitive form, 0 or 1, false or true, but we also must remember that any non-zero value is considered to be true. Finally, we have learned to be most careful around the new equality operator, ==: dropping down to a single = when two were warranted can lead to bugs of nastiest form.

---

²Wow, that sounded more grim than we intended. It’s actually pretty fun to program in C! But yes, sometimes the mistakes you can make are irritating. Exercise a little caution and you’ll be fine, don’t worry!

³Honestly, we could fill this whole book with Twain quotes, but then the book would be called “Twain Quotes” and not really help you learn about C or how computer systems work. Anyhow, here are some more fun quotes of his: “I did not attend his funeral, but I sent a nice letter saying I approved of it.” “If you pick up a starving dog and make him prosperous he will not bite you. This is the principal difference between a dog and man.” “When I was a boy of 14, my father was so ignorant I could hardly stand to have the old man around. But when I got to be 21, I was astonished at how much the old man had learned in seven years.” “The man who does not read has no advantage over the man who cannot read.” “Never put off till tomorrow what may be done day after tomorrow just as well.” “‘Classic’ – a book which people praise and don’t read.” “Keep away from people who try to belittle your ambitions. Small people always do that, but the really great make you feel that you, too, can become great.” And it goes on and on. Check out https://www.goodreads.com/author/quotes/1244.MarkTwain for more, and waste some more of your life in this relatively good way.