In which our hero finally takes control of a program and makes it do different things, based on whether certain expressions are true or false.

11.1 Don’t Blink

In the book “Blink”, Malcolm Gladwell presents a fun, entertaining, and (probably) occasionally wrong book on the human ability to make decisions [G07]. Specifically, he talks about the different types of speeds at which human beings can make decisions: one is slow, conscious decision making; the other is the “adaptive unconscious”, which operates at much higher velocities: “The only way that human beings could ever have survived as a species for as long as we have is that we’ve developed another kind of decision-making apparatus that’s capable of making very quick judgments based on very little information.”

Computers also make decisions all the time, and honestly, all of them happen very quickly, at least at the lowest levels of the system. With modern processors that operate at billions of operations per second, perhaps this is no surprise to you, but it’s still pretty cool. We won’t yet talk about the very lowest-level instructions that the machine executes to make decisions, called branch instructions, but we will now discuss how you can express decision-making within C programs today, with the if statement and its old contrarian pal, else.

11.2 The First Example

Let’s dive in and look at a simple example. Just by reading it in English, you’ll probably understand what it means. Try to follow the program as it executes, and, specifically, predict the final value of the variable \( y \).
int x, y;

x = 11;  // Line 1
y = 0;   // Line 2

if (x > 10) {  // Line 3
    y = 1;   // Line 4
}
...  // Line 5

First, the program defines two integers, \( x \) and \( y \), making space for them in memory somewhere. Then, the C abstract machine begins to run (Line 1), and assigns values to both \( x \) (11) at Line 1 and \( y \) (0) at Line 2.

Finally, we hit a decision point in the program (Line 3): if the value of \( x \) is greater than 10, the program will execute Line 4, and set \( y \) to the value 1. If the value of \( x \) is not greater than 10, the program will skip ahead in the code (to Line 5), and thus \( y \) will remain at the value 0. And, as you’ve probably already figured out, \( y \) does indeed get set to 1, as \( x \) was initialized to 11. Easy!

### 11.3 Official If Format

We can now show you the general form of an **if statement**. There are actually two forms you will encounter in C code. The first is the longer form, and it looks like this:

```c
if (expression) {
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

The expression shown between the (required) parentheses\(^1\) is just any old expression as you’ve already seen; if the value of that expression is non-zero, it is true, and if zero, false. When the expression is true, the program will execute the statements between the curly braces (\( \text{statement}_1 \) through \( \text{statement}_n \)); otherwise, it will skip over them and execute the line after the braces end.

We call a block of code between curly braces (i.e., between \{ and \}) a **compound statement**, which is just a fancy way of saying it’s a statement that’s actually made up of a bunch of statements. We’ll see statements grouped together like this all the time in C code (after if

---

\(^1\)Some languages, like Python, do not require parentheses around an expression such as this one. C is not such a language, so don’t get too excited.
Once again, remember our goal: readable, concise code. Whitespace is your friend; packing code together does not make it more readable, usually. Find some reasonable style guide (the Google one for C++ is a good read, although not a perfect match for C [G17]) and try to follow it. Your goal is not to save space in the text of your code.

Statements, `while` loops, and even function definitions). In fact, you can even just throw curly braces around any piece of code if you’d like – something we’ll talk about more when we discuss scope in a few chapters.

The `if` statement also has a short form, which you can use to make code more compact when only a single statement follows it:

```c
if (expression)
    statement;
```

Note that none of these forms are particularly sensitive to whitespace or newline characters. For example, we could rewrite the most recent example as follows:

```c
if (expression) statement;
```

We could also write the longer form as follows, if we really wanted to smash things together:

```c
if(expression){x=1;y=2;}
```

Generally, don’t do these things – they are ugly and hard to read. Whitespace is your friend.

### 11.4 Altering The Flow Of Control

When you are thinking about the C Abstract Machine (CAM) and how C programs logically execute, you now have a new construct to think about thanks to the `if` statement: **control flow**, or the **flow of control**. All imperative programming languages like C (in which you write code that tells the computer exactly what to do, one step at a time) have an obvious order in which statements are executed; that is what we call the flow of control.

Before the `if`, you just had **straightline code** to worry about: the abstract machine executed one statement after the other, in order, much like a simple recipe to follow, executing (in this example) statements 1 through 5 in order:

---

2 A contrast to an imperative language like C is a declarative language like SQL, in which you write down what you want and the system figures out how best to get it for you; take a database class to learn more!
statement_1;
statement_2;
statement_3;
statement_4;
statement_5;

With if statements, we have now introduced conditional execution: some instructions only execute if certain conditions are true. Otherwise, the CAM skips right over them, advancing the conceptual C instruction pointer (CIP) just past the body following the if.

statement_1;
statement_2;
if (expression) {
    statement_3;
    statement_4;
}
statement_5;

In the example above, there are two possible control flows for this program. If the expression is true (i.e., it evaluates to non-zero), statements 1, 2, 3, 4, and 5 will be executed, in order (with the expression itself getting evaluated between statements 2 and 3). If the expression is false, a different flow takes place: 1, 2, evaluate the expression, and then 5, skipping right over 3 and 4.

And thus our C Abstract Machine grows more complex, with the ability to skip over pieces of code conditionally. This functionality lies at the heart of decision making within programs.

11.5 The Other Case: Else

The if statement gives us everything you need to make decisions in programs. Truly, if you had to, you wouldn’t need any other primitive to write correct and fully-functional code.

However, sometimes it would be a pain to use, if left all to itself. For example, let’s say you wanted to either do one thing or the other, based on whether something was true or false. For example, let’s say you wanted to compare x to 10. If x were greater than 10, you wanted to code to do one thing (e.g., statement_1); otherwise, you wanted to code to do something else (e.g., statement_2). You could write this as follows:

if (x > 10) {
    statement_1;
}
if (x <= 10) {
    statement_2;
}
Yuck! Not only is this code less clear than it needs to be, it also is error prone. Because you repeat the value 10 twice, if you change it in one place and not the other, you'll introduce a bug into your logic.

Fortunately, there is a better way, which comes in the form of the constant companion to if, the optional else. With else, we can express the otherwise clause readily:

```c
if (x > 10) {
    statement_1;
} else {
    statement_2;
}
```

The result is clarity and simplicity as compared to only having if available. The else makes code clearer, easier to read, and less error prone; what a great invention!

The flow of control for if and else should be clear. The expression is evaluated first. If it is true (non-zero), the first compound statement is executed (the if block); if it is not true (zero), the second compound statement is executed (the else block).

The short form without curly braces also works:

```c
if (x > 10)
    statement_1;
else
    statement_2;
```

### 11.6 Nesting And Recursive Structure

As will be the case with many C constructs, arbitrary nesting of code is possible, a comment that only became relevant once we introduced compound statements. For example, it is quite logical (and common) to nest one if statement within another:

```c
if (expression_1) {
    statement_1;
    if (expression_2) {
        statement_2;
    } else {
        statement_3;
    }
    statement_4;
} statement_5;
```

In this nesting, first expression_1 is evaluated. If false (zero), C will skip the entire compound statement and execute statement_5;
if true (non-zero), the code will enter compound statement and execute statement_1, and then evaluate the expression expression_2. These next steps proceed as you would guess: if that expression is true (non-zero), statement_2 is executed, otherwise statement_3 is. Either way, statement_4 is executed, and finally statement_5. All together, these three possible control flows are possible: e1 (short for expression_1), s5 (short for statement_5) or e1, s1, e2, s2, s4, s5 or e1, s1, e2, s3, s4, s5.

One way to think about this is that a compound statement consists of other statements; these statements can also be other compound statements. With this recursive structuring, you can build arbitrary complexity into your code to match the problem you need to solve. This doesn’t mean you should make your code have intricate and complicated structure (rather, you should do the opposite), but the fact that you can is powerful and useful.

11.7 If And Else If And ... And Else

Sometimes, you’ll want code to have a richer possible control flow than simply one if followed by a single else. For example, you might want to do the following:

```c
if (x < 5) {
    statement_1;
} else {
    if (x < 10) {
        statement_2;
    } else {
        if (x < 20) {
            statement_3;
        } else {
            statement_4;
        }
    }
}
```

This code executes statement_1 if x is less than five, executes statement_2 if x is greater than or equal to five but less than ten, executes statement_3 if x is greater than or equal to 10 but less than twenty, and finally executes statement_4 otherwise. Wow, a little hard to read, no?

Fortunately, it is considered good C to write this type of conditional in a different, more compact form. Thus, you can (and should!) rewrite the above code as follows:
if (x < 5) {
    statement_1;
} else if (x < 10) {
    statement_2;
} else if (x < 20) {
    statement_3;
} else {
    statement_4;
}

While these code snippets are logically identical, the latter is easier to read and much preferred. By avoiding the many levels of indentation, the code became cleaner while achieving the same goals. See if you can convince yourself that the code snippets are indeed identical.

There is another way to write conditional code flows like this one, by using something called a switch statement. We’ll see such switch statements later on as needed, as in some cases they are more convenient to use than the equivalent if/else if/else statement used above.

11.8 Short Circuiting

There are a few special cases we should discuss before wrapping up. The first is a feature we introduced briefly when presenting boolean expressions earlier, and it is the notion of short circuiting. Let’s look at the following piece of code:

```c
int x = 9;
int y = 1;
int output = 0;

if ((y = y + 1) && x == 10) {
    output = 1;
}
```

What do you think the final values of y and output will be? (think about it)

Did you guess y would be 2 and output would be 0? If so, you are right! Well done. If not, think about it some more.

The reason the value of y changes is the assignment statement on the left of the expression: it increments y by 1 and assigns that new value (2) to y. The reason the value of output does not change is the expression on the right: x does not equal 10, and thus the code skips the if body.
Now consider the exact same expression, but with the left part and the right part switched (seemingly showing the same boolean expression):

```c
if (x == 10 && (y = y + 1)) {
    output = 1;
}
```

What do you think the final values will be now? *(Again, think about it, please)*

In this case, `output` is once again zero (the expression is still false), but, critically, `y` is still equal to 1. The reason for this is the short circuiting behavior of C expression evaluation.

Short circuiting, which primarily is of concern with boolean-and (`&&`) and boolean-or (`||`), evaluates expressions differently than what you might expect. Specifically, the left side of either of these operators is guaranteed to be fully evaluated before the right side. With `&&`, if the left side is false (zero), it is further guaranteed that the right side will not even be evaluated; what is the point, after all, given that `0 && x` is 0 (false) for any `x`? The corollary is true for `||`; if the left-side evaluates to true (non-zero), the right side is not evaluated either as the expression now must be true.

The fancy term for this behavior is that the operator `&&` (or `||`) is a **sequence point** [S17]. A sequence point is a spot in the program where all previous evaluations will have been performed and no effects from subsequent expressions can be felt. These operators (as well as a few others) behave in this manner, and thus, if you are not careful, can lead to surprises.

Short circuiting is efficient – the code doesn’t evaluate latter part of the expression if the outcome is known, which is especially important with long expressions (e.g., `a && b && c && ...`). Short circuiting also enables code of this nature to exist in C:

```c
if (check_if_something_is_safe_to_do() && do_it()) {
    ....
}
```

The only way the code executes a potentially dangerous operation is when the safety check has passed. Now that you know about short circuiting, you’ll know how to interpret code that is written with safety in mind.

### 11.9 If/Else Short Form

There is one last form of conditional you’ll see in C code; as the legends of C (and Unix) really hated typing too much, you’ll not be surprised to hear that this other form is a short form, called a **ternary conditional operator**. Here is the generic form of such an expression:
You might use such an expression as follows:

```c
x = y > 0 ? 1 : 0;
```

The way to read this expression: if \( y \) is greater than zero, set \( x \) to 1, otherwise set \( x \) to 0. Yes, we could have just written this same code as follows, perhaps with more clarity:

```c
if (y > 0) {
    x = 1;
} else {
    x = 0;
}
```

11.10 Common Mistakes

Before closing, let’s look at a few common mistakes people make when writing conditional code. The first we have already presaged when discussing the certain operators. See if you can spot the potential bug in the following code; do this before reading on! it is more fun that way. Specifically, what will the value of \( y \) be when the code below is finished executing? How about \( x \)?

```c
x = 5;
y = 0;
if (x = 10) {
    y = 1;
}
```

The mistake in this code is simple: using `=` instead of `==` in the expression after the `if`. If the desire is compare whether two values are equal, then use the `==` operator. If you use the `=` operator instead, you are generating perfectly valid C, but perhaps not getting the program to do what you want. With this operator, the expression `x=10` evaluates to the value on the right, which is 10 (and, as a side effect, sets \( x \) to the value 10); because this is non-zero, the expression is true, and thus \( y \) gets assigned the value of 1. After the code has run \( x \) is equal to 10 and \( y \) to 1.

There are cases where you will find the ability to assign a value within an expression quite useful, so this “feature” is actually a feature much of the time. But it can lead to tricky bugs, and C itself

---

3Use `==` to compare integers, and for other basic types that we will soon see, such as characters and floating point values. Generally, do not use it for strings and more complex structures; we will learn why in a bit.
Usually, you should put curly braces around conditional blocks of code; doing so avoids silly bugs that might arise when moving code around. One other way to steer clear of these problems is by using a good editor, which will automatically indent your code properly.

will offer nary a complaint, letting you stare hopelessly at this code, trying to figure out if you’ve discovered a bug in C, until the truth hits you and you smile and add that extra = to make it work like you wanted.

One other mistake can arise if you are not careful, and particularly so if you cut and paste code. See if you can spot it:

\[
\begin{align*}
x &= 5; \\
y &= 0; \\
z &= 0; \\
if (x > 10) \\
&\quad y = 1; \\
&\quad z = 2;
\end{align*}
\]

Did you figure it out?

Here, the problem lies in the indentation of the code, which leads you to believe that the assignment \( z = 2 \) is conditionally executed based on whether \( x \) is greater than 10. However, the short form of if is in use, and thus the more proper indentation would be:

\[
\begin{align*}
if (x > 10) \\
&\quad y = 1; \\
&\quad z = 2;
\end{align*}
\]

As is readily apparent from this writing of the code, \( z \) is always assigned to 2. One way to avoid this type of bug is to use curly braces around conditional blocks unless you really are greatly improving the readability of the code without them.

11.11 Summary

We have seen the addition of the conditional expression into C programs. The if statement lets us powerfully decide which code to execute based on current conditions within the program; such conditional execution is at the heart of most any interesting program.
References

[G07] “Blink”  
Malcolm Gladwell  
Back Bay Books, 2007  
Gladwell’s books are fun, but often frustrating, containing a fascinating blend of summaries of good science and not-so-good science. But, probably worth reading nonetheless. Want a book closer to the actual science? Kahneman’s “Thinking Fast and Slow” is likely the way to go.

Engineers at Google  
google.github.io/styleguide/cppguide.html  
The Google C++ style guide. If there’s one thing that Google cares deeply about (besides not being evil and all that), it’s code quality. Thus, whether you adopt anything from this guide or not, you will learn something by reading it.

[S17] “Sequence Points”  
https://en.wikipedia.org/wiki/Sequence_point  
Wikipedia provides a nice overview of sequence points (now “sequencing” in C++) and some of the other cases where they arise.