In which our hero actually gets to see a useful part of a computer program cobbled together, one which can readily manipulate memory. Our hero also learns of a new and quite important operator (=), which enables one to achieve these wonders.

8.1 Our Friend, The Equals Sign

Thus far, you’ve seen how to create mathematical expressions, and, quite recently, how to define a variable. In this chapter, you’ll put these two concepts together, thus allowing you to read and update memory with complex mathematical formulas (or even, with simple ones). All of this wonder will be achieved with the addition of a single new and powerful operator: the assignment operator, which is written in C code with the = symbol.

Here is a simple example, in which we define an integer, and then set its value with a mathematical expression:

```c
int a;

a = 10 + 17 / 2;
```

Once again, simple! Let’s dig a little deeper to make sure this most basic and fundamental operator well.

8.2 Does Equals Equal Equals?

According to Wikipedia (which, despite some rumblings now and then, is an excellent source of information on many topics), the equals sign was invented in 1557 by Robert Recorde, in the book “The Whetstone of Witte” [R57]. However, as is often a little misleading for those new to programming, our usage here is different.

In math, when you write \( A = B \), you mean that the thing on the left has the same value as the thing on the right; it is a statement
of truth. You could similarly write \( x + 3 = y + 4 \) and have a valid mathematical statement.

In C programming (and many other languages), the equals symbol is used not to state a truth but rather to cause action; specifically, when you write \( A = B \); in a C program, you mean that we should take the value of the expression B and put it into the variable (container) named A. A is thus assigned the value of expression B.

Thus, the left side of the equals operator is restricted: it must be the name of some kind of container. In contrast to math, \( x + 3 = y + 4 \) is not legal C. There is a name for this that programming language folks use: a locator value, or lvalue for short; basically, as Bendersky succinctly states, an lvalue is “represents an object that occupies some identifiable location in memory (i.e. has an address).” [B11] For now, the only thing that we know about that qualifies is a defined integer variable; later, we’ll see some other things that qualify as lvalues.

### 8.3 A Memory-Based View

We know variables are just named containers. And now, with assignment, we can access (read) and update (write) those spots in memory rather easily. In this example, assume we have defined an integer named \( x \):

```
int x;
```

For the example, assume that four bytes of memory have been allocated at address \( 1016 \), and this is where C integer \( x \) resides. Assume further that \( x \) is initially set to zero. Figure 8.1 shows a depiction of the part of memory where \( x \) is found, at address \( 1016 \) (we use decimal for both addresses and values in memory for simplicity).

We now perform an assignment, as follows:

```
x = x + 10;
```

The intent of this statement is to first read (load) the current value of \( x \) into the processor, add ten to it, and then write (store) the new value of \( x \) back into memory in the same location, at address 1016.
When we are finished, the memory has been updated, as shown in Figure 8.2. The assignment statement thus makes it trivial for us to update memory wherever we have a variable defined.

8.4 Assignment Can Ease Initialization

Assignment immediately makes one aspect of C programming much more concise: initial assignment of a variable. The longer form of definition and subsequent initialization is as follows:

```c
int x;
x = 33;
```

C, however, allows a shorter form, combining the assignment operator directly into the line that defines the variable:

```c
int x = 33;
```

In general, C provides many such shortcuts to make code shorter without any loss of readability. In fact, writing code in such a manner makes the code more readable; furthermore, as a missing initialization is a common bug (which we’ll discuss further below), doing both the definition and initialization together reduces the likelihood of making this mistake.

8.5 Assignment Is An Operator

One aspect of assignment that can be confusing (though almost never at first) is that assignment is an operator, just like the operators we’ve seen thus far. The first implication of this fact is that it too has precedence, as shown in our revised precedence table (Figure 8.3).

You might have guessed that the precedence of assignment would be low; if so, you were right. If not, what were you thinking? This enables us to write code in a simple and intuitive fashion. Imagine the following line:

```
x = (a * a * 20) + (a * 10) + 150;
```
Because assignment has such low precedence, the entire expression on the right of it will be evaluated first. Only after this value is computed will it be stored in \( x \), as desired.

Assignment, like addition, multiplication, etc., is a binary operator (i.e., it operates on two parameters); the result of this operator has a value, which is equal to the value of the expression on the right side of the \( = \) symbol. This fact leads a further implication: you can use assignment expressions in the middle of more complex expressions.

Here is the simplest example of such a usage, involving two variables, \( a \) and \( b \):

\[
a = 7; \\
b = 7;
\]

C allows us another shorthand here:

\[
a = b = 7;
\]

To understand how this will be evaluated, examine the precedence table, and see if you have a guess. Go on, do it! You’ll learn more by pausing and thinking than just reading the answer.  *think pause inserted here; music from Jeopardy playing; finally, the bell rings, and Alex Trebek asks you for an answer*

The answer, as you figured out already, is that \( = \) associates right-to-left; thus, the above equation can be thought of as:

\[
a = (b = 7);
\]

Thus, \( b \) is first set to 7. The result of that expression is also the value 7; thus, next, \( a \) is set to that same value, 7. In memory, this looks like what you see in Figure 8.4. As you can see therein, \( a \) and \( b \) are just two independent integers, each of which were assigned the value 7, with one fancy C statement.

You can even write stranger lines of code, once you accept that assignment is really an operator. Consider the following:

\[
a = (b = 7) + 1;
\]

What do you think this sets \( a \) and \( b \) to? The answer is left as an exercise to readers, all three of you.
8.6 Other Forms Of Assignment

In many different types of code, it is common to see the following type of assignment, which uses the same variable both as a source and a target:

\[
\text{sum} = \text{sum} + \text{count};
\]

So common, in fact, that there is a shorthand in C for it:

\[
\text{sum} += \text{count};
\]

Thus, a new family of operators are available to you. For each of the mathematical operators you have seen thus far (\(+, /, \%, +, -\)), there is a corollary shorthand assignment (\(+=, /=, %=, +=, -=\)) that does what you would expect. For example, if you wrote \(\text{sum} *= 2\), its just shorthand for \(\text{sum} = \text{sum} * 2\).

8.7 And One Weirder Shortcut: ++ and --

We’ve been trying to keep things simple thus far. But C won’t have it. No, C is just full of little surprises and tricks, so you’ve got to pay attention to keep up, even when doing something basic as math!

The tricky new thing here are two more unary operators, ++ and --. On the surface, they seem simple. For example, we could place the following in C code:

\[
i++;\]

This statement is perfect shorthand for \(i = i + 1\);, or even the slightly shorter \(i += 1\);. We can also write it with the operator on the other side of the variable, with exactly the same effect:

\[
++j;
\]
The tricky part with these new operators shows up when you use the value in an expression. For example, in the following code, what do you think the value of $x$ will be after the code executes?

```
i = 1;
int x = i++;
```

The answer depends on which side of the variable you put the `++` or `--` operator. When you write `i++`, you are using what is called the postfix increment; the value of the expression is thus the value before the increment has taken place. For the example above, $x$ thus is assigned to the value 1, but after the statement `int x = i++` is executed, $i$ will be assigned the value 2.

If we instead had used the prefix increment version, the increment takes place first, and thus $x$ is set to 2. The prefix decrement and postfix decrement operators (`--`) have similar semantics.

```
i = 1;
int x = ++i;
```

Figure 8.5 shows the precedence of these new operators. The shortcuts for assignment are at the same level as the old `=` operator; the new postfix and prefix operators are quite high in precedence, in contrast. Specifically, the postfix increment and decrement are at the highest precedence level, whereas prefix increment and decrement are at the same level as the unary `+` and `-`. We’ll see, somewhat later in the book, why this is so.

### 8.8 Common Mistakes

Before closing, let’s discuss some common mistakes that you should try your best to avoid. The biggest mistake to avoid is accessing a variable that hasn’t been set to anything yet, or what we call **initialized**. For example, this code is bad code (next page):

---

1. This explains the joke people used to say about C++, that it should have been called `++C`, if it really was an improvement.
Even if your code might work without doing so, never forget to initialize a variable before using it in an expression. Relying on automatic initialization, which does take place in some cases, is risky; imagine if you later cut and paste the code into a different context, and the assumption of auto-initialization is gone. Avoid that problem and always update each variable before using it.

```c
int a;
int b;
a = b + 10;
```

The reason you shouldn’t write it is that \( b \) is read before it has ever been written. Such uninitialized access can lead to a lot of problems in your code, so it’s best to avoid it.

What’s even worse about this type of code is that C generally will let you write and run code like this (in some cases, as we will see later, C will at least warn you about it). The results are often known as undefined behavior [R10], which means we can’t expect any particular result when running such a piece of code. Later on, we’ll talk more about undefined behavior more, but the safe thing to say now is: avoid it if you can!

For example, if we run this code snippet on a Mac, and then show the values of \( a \) and \( b \), we see:

\[
a = -1074009462 \\
b = -1074009472
\]

On Linux, we get something different (and perhaps more reasonable):

\[
a = 10 \\
b = 0
\]

However, in both cases, because our code is buggy, it is unreliable. While it may work on some platforms some of the time, it is not code you can depend on.

Note that there are some cases where you can assume a variable gets initialized to some kind of zero value automatically [W16]. However, we believe this is error prone and likely to lead to bugs. So the simple lesson here is this: always, always initialize variables before you use them.
A C program will occasionally give you the feeling that it is working, because, well, it worked once or twice. Don’t be fooled: there are many C programs that are filled with bugs and problems but often run seemingly correctly. A correct run does not prove anything!

8.9 Summary

We have seen the introduction of a new and powerful operator, assignment. Assignment allows us to write meaningful pieces of code that compute expressions and then record them in memory in a variable, which is a fundamental part of many sorts of calculations. We also saw that we need to be very careful to avoid forgetting to initialize a variable before using it elsewhere; doing so can lead to hard to debug problems in your code. Don’t be lazy (except when you are about to do something wrong).
References

[B11] “Understanding lvalues and rvalues in C and C++”
Eli Bendersky
December, 2011
eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c
A nice random article about lvalues and their counterpart, rvalues. Read it now and read it again
later after we have learned more.

[R57] “The Whetstone of Witte”
Robert Recorde
1557
We really just wanted to cite a book from 1557, because that is pretty cool. It is also the first
recorded use of the equals sign, so there is that too.

John Regehr
http://blog.regehr.org/archives/213
A terrific introduction to undefined behavior in C. We’ll discuss this in more detail later, as it is
(unfortunately) important.

[W16] “Uninitialized Variable”
en.wikipedia.org/wiki/Uninitialized_variable
A really boring wiki page about this topic.