Numbers, Operators, and Expressions

In which our hero remembers Maths from middle school, and learns that the world within a computer system is not as perfect as one might think, even for something as simple as addition, subtraction, and the like.

5.1 Begin With Things You Know

A wise old Italian professor (Professor G) once told us, when you are teaching someone something new, start with what they know, and then build from there. When teaching Italian, for example, Professor G asks students what they think these words mean: “musico”, “familiare”, “maturo”, and “broccoli”. Suddenly, the students think they know Italian, and answer “music”, “familiar”, “mature”, and “broccoli.” By starting with what you already know, you get a quick boost of confidence, encouraging you to go on; more importantly, the old information in your head is connected to the new, thus broadening your knowledge in a rational and coherent fashion. What a teacher, Professor G! And now the downside: he’s not your teacher.

Thus, to learn C, we start with something we assume you, dear reader, already are familiar with: math. That’s right, computer programs are often filled with mathematical manipulations of various kinds, so what better place to start learning? But, as we’ll soon see, there are some tricky aspects to dealing with math in a C program (and, on a computer, more generally), of which any sage programmer must be quite aware, lest poor results arise (i.e., bugs, security flaws, and other terrible messes).

5.2 Just Numbers

We start with the simplest of constructs: the integer [N69]. As you might recall from math class years ago, integers are whole numbers that range from negative infinity \(-\infty\) to \(\infty\). That’s a lot of numbers! Don’t try to count them; it will take a long time.
Integers thus look like this (as if you need reminding):

8
265
0
400232

C programs are often filled with numbers like this. When you see a number like this in a C program, the fancy programming language gurus call it an **integer literal**, but we will probably just call it a number. These **literals**, in general, represent constant values in the program text; we’ll see other types of literals later on.

We can also write these numbers in octal or hexadecimal form, if we’d like to, when writing a C program. For example, our C program could include any of the following:

0x01
0xFFFF0000
010
022

The first two are numbers shown in hexadecimal form – the leading 0x tells us this (as well as the ability to use A ... F as numbers). The last two in this list look innocently like normal numbers in decimal form, but are actually octal because of the leading zero. Thus, the value 010 is not ten but rather octal ten which is decimal eight.

### 5.3 Operator, Can You Help Me?

There isn’t too much more we can say about just numbers, yet. After all, they are just numbers! It’s what we can do with them that is more interesting. To do something with numbers, we need a new type of entity within our C program text, and that is something called an **operator**. We’ll introduce some very familiar operators shortly (to help us do math), but we’ll also see a number of interesting other operators later on that will reveal the full power of the C language.

The first set of operators we focus on are those which let us do something with integers. The most basic of these are these five basic operations over integers: **addition** (+), **subtraction** (−), **multiplication** (*), **integer division** (/), and **integer remainder** or **modulo** (%).

We’ll now use these operators to create what we call, in C, an **expression**. The type of expression we will create here just combine integer literals and operators in a legal way to produce a value (in this case, an integer value). Here are some possible C expressions:

5 + 10
10 % 3
4 + 6 / 2
When a program runs, these expressions are evaluated, which means the computer performs some instructions to compute the resulting value of the expression. In the first example above (5+10), you (hopefully!) know from your experience with math that the result will be 15 – not too exciting.

Same thing with the next example (10%3). If you remember how modulo works, it outputs the remainder when dividing the numerator (10, shown on the left) and the denominator (3, on the right). As you know, 3 goes into 10 three times, leaving a remainder of 1. And thus, 10%3 is equal to 1.

That leaves us with our final expression, the seemingly simple: 4+6/2. The value of this expression, as you may have guessed, is 7, but to know this you have to know something about how the expression is evaluated. For example, do we just proceed right to left, adding 4 to 6 and the dividing by 2? Or do we first perform the division and then the addition?

How exactly an expression is evaluated is determined by something called the rules of precedence. Like most programming languages, C has a (mostly) well-defined set of rules that govern this behavior. Specifically, operator precedence determines which part of an expression is treated as a parameter for each operator [PC15], e.g., that 6 is the correct numerator in the example above, not (4+6).

5.4 Operator Precedence

The first rule of C operator precedence that we will learn is the one you learned in elementary school: that operations such as multiply, divide, and modulo have higher precedence than addition and subtraction. Envision the the following expression:

\[ 8 \times 3 + 6 / 2 - 10 \% 2 \]

Precedence tells us that we can think of the expression as equivalent to \((8 \times 3) + (6/2) - (10\%2)\). Thus, we first evaluate the sub-expressions within the parentheses, as follows: first, the multiplication \(8 \times 3\) (result: 24), then the division \(6/2\) (result: 3), and then the modulo \(10\%2\) (result: 0). With these sub-expressions evaluated, we can now perform the addition \(24 + 3\) (result: 27) and subsequent subtraction \(27 - 0\) (final result: 27).

Note that simply stating that \(*, /, \%\) have higher precedence than \(+, -\) is not enough; we also need to know what order to execute operators that are at the same level of precedence. For example, to what do you think the following expression evaluates?

\[ 2 \times 7 / 2 \]

As you know, both multiply and integer divide are at the same precedence level, and thus we need more information to evaluate this expression. Why? Well, if we first perform the multiply, we
obtain a result of $14/2$ which equals 7. However, if we first perform the integer divide, we instead get $2*3$ with a result of 6 (!), because with integers, 7 divided by 2 is 3 (remainder 1).

To resolve this dilemma, there is another concept we need, referred to as operator associativity (or just associativity). There are two types of associativity we will see: left-to-right associativity and right-to-left associativity, and both work like you might think they do. With left-to-right, if we have the expression $X \ op1 \ Y \ op2 \ Z$, where $op1$ and $op2$ are operators at the same level of precedence, then we first group $X \ op1 \ Y$; thus, we can think of the entire expression as $(X \ op1 \ Y) \ op2 \ Z$. With right-to-left associativity, the opposite takes place: $X \ op1 \ (Y \ op2 \ Z)$.

With these rules in mind, we can put together our first precedence table, as seen in Figure 5.1. We’ll add to this table over time as we learn new operators; for a full list, see an online source [C17].

### 5.5 Breaking Precedence With Parentheses

Sometimes, when writing an expression, you’ll want to have the value computed in a manner different than what would naturally arise from the rules of precedence. In such a case, we can use parentheses to force C to group parameters and operators differently, in just the way you might expect from your knowledge of algebra. Parentheses are not operators (per se), but can be used to alter precedence to match programmer intent, and thus you must understand them too.

For example, earlier we examined this expression: $4+6/2$. What if we wanted to first add 4 to 6, before the division took place? Easy! Just use parentheses: $(4+6)/2$. Evaluating the expression now results in the desired outcome of 5 (i.e., $10/2$).

As you may know from your math days, parentheses can be nested arbitrarily; in those cases, we evaluate the expression from inner-to-outer, while keeping with precedence rules otherwise. Let’s look at the following expression:

$$(10 \times (8 - 2 \times (6 + 1) + 3) - 2)$$

To evaluate it, we first find the innermost grouping and use normal precedence operators within to evaluate it. In this case, that would be the sub-expression $6+1$, which of course is 7. We are thus left with:

$$(10 \times (8 - 2 \times 7 + 3) - 2)$$

The next focus would thus be $(8 - 2 \times 7 + 3)$, inside which we must first perform the multiplication, and then the left-to-right...
Given that parentheses can be used to change how an expression is evaluated, there are certainly occasions when you simply must use them to get the expression to evaluate in the manner that you desire. However, sometimes it is also useful to use parentheses just to make your code more clear in its intent. We’ll see more examples of this later when we introduce more operators, but for now, it’s safe to say that you should feel comfortable adding parentheses if you think it makes the code clearer and easier to understand. Don’t worry about efficiency; adding parentheses just for clarity (and not to change evaluation) will not slow your code down.

subtraction then addition, resulting in the value $-3$. This step leaves us with our near-final result: $(10 \times (-3) - 2)$; we first multiply (result: $-30$) then subtract 2 to get the final outcome: $-32$.

5.6 Two More Operators: Plus and Minus, Again?

Thus far, we’ve primarily looked at operators such as multiplication, addition, subtraction, etc. Each of these operators work on two inputs: one on the left of the operator, and one on the right. For this reason, we call them **binary operators** (not because they work on binary numbers, but because they take two inputs).

However, there is another way in which two of our symbols, the addition and subtraction symbols, are used that is slightly different, and that is as **unary operators**, meaning that they are applied to a single input.

For example, assume we have the following expression:

$$6 \times -3$$

We know from our math training that the answer is $-18$, but with C (or any language) we need to precisely specify how to evaluate the expression. The way C will approach this is to first examine the integer literal 3, and then apply the unary operator $-$ to it (negation). Only after this negation is performed will the multiplication take place.

And thus, the careful reader (you!) can deduce that the unary operators $+$ and $-$ have higher precedence than multiplication (and than division and modulo, too, naturally). We thus can show a second version of our precedence table with these unary operators near the top (Figure 5.2).

We’d like to comment on one aspect of unary operators that diverges a bit from mathematical representation. Specifically, in math, we can put as many plus or minus signs in front of number and it is acceptable; for example, the mathematical expression $- - - 7$ is valid.
(and can be reduced to just \(-7\)); similarly, in math, you can write \(+ + + 7\), or put any combination of addition and subtraction symbols in front of a number legally.

In C, however, you have to be a little careful when using multiple negative or positive signs in front of a number. If you try to put the expression \(-- -7\) in your C program, you’ll get some kind of weird error like this:

```
error: lvalue required as decrement operand
```

The reason for this is a bit hard to understand (at this moment) for you, the reader, but has to do with the difficulty of determining the intent of the programmer. Specifically, there is another operator that we haven’t seen yet, \(--\) (as well as \(++\)). As a result, three negative signs placed together as such are hard to interpret; do you mean the \(--\) operator followed by a unary negation, or three unary negations? Thus, to string together negations, if for some reason you want to, you must use extra spaces. The resulting expression \(-- -7\) is perfectly valid C (if a bit odd).

### 5.7 Summary

We have seen the first and simplest aspect of a C program, the integer, and started adding the ability to compute on these integers with operators. We have also seen that operators follow some guidelines for order evaluation, which (mostly) match our intuition from math class. Finally, when precedence rules don’t quite get us what we want, we can apply parentheses to alter the order of evaluation.

Figure 5.3 shows a slight update to the C Abstract Machine. Our basic components remain the same: code and data in a large memory (sometimes called the address space of the program); the CIP to tell us which C statement to operate upon next; a basic data type, the C integer; operators to apply to the basic data type, enabling us to perform mathematical calculations.

But can we do any sort of math we want with integers, and get the expected result? As it turns out, not with C integers, which are an imperfect representation of the pure mathematical concept. And thus, on to the next topic of discussion...
Figure 5.3: The C Abstract Machine: Ints and Operators
References

[C17] “C Operator Precedence”
A comprehensive C operator precedence guide, ironically upon the C++ reference page (cppreference.com). Note that this table includes some modernisms from C99 and C11, which we will ignore (mostly).

[N69] “The Exact Sciences in Antiquity”
Otto Neugebauer
Dover Publications, 1969
No, the integer was not invented in 1969. But this book was published in that year, and it is a good one. Basically this is an excellent presentation on the science and math of the olden times from famed scientific historian Otto Neugebauer. Read online about his life, too, if interested; he led quite an amazing one.

[PC15] “C in a Nutshell”
Peter Prinz and Tony Crawford
O’Reilly, 2015
Probably the best modern comprehensive guide to C, in all its gory detail. The more you want to know about C, the more you’re going to want to have this book on your shelf.