6

INTEGER AND FLOAT DATA TYPES

In Chapter 5, I detailed string and boolean data types. In this chapter, I'll pivot to numeric data types, specifically the integer and floating-point data types, investigating them in great detail. Batch handles integers with ease, whether they be of the decimal, hexadecimal, or octal variants.

However, floating-point numbers are similar to booleans in that Batch doesn't actually support them explicitly as a data type. But once again, that limitation affords the imaginative Batch coder with an opportunity to be inventive, and that's exactly what we'll do before this chapter is done.

An Octals Case Study

August 1, some year in the aughts: I can't remember the exact year, but of the month and date I am quite certain, for reasons that will be clear by the end of this chapter.

I was still relatively new to Batch, but I knew more than many, so a co-worker came to me with a task with which he had been struggling. In the Batch code he needed to determine the prior day's date given only the current date. That's pretty straightforward for most days of the year, but it becomes complicated when today's date is the first of the month. Months have different lengths; New Year's Day poses a unique challenge; leap years happen every four years, except for when they don't.

This initial event occurred in February, maybe March, and it was an interesting little exercise that I coded up and tested. Like any good coder, I tested the first day of the year and the last. I also tested the first day of a handful of months, particularly the extremes, like January and December. I tested March 1 for several different years, not because I was coding this around February but because of the peculiarities of leap years. In short order, I handed over the code and moved on to other projects.

The code worked great for about six months. Then on August 1 it suddenly didn't. I don't remember the downstream consequence, but my co-worker spent a good chunk of time tracking down the root cause. He eventually zeroed in on my bat file but couldn't figure out why it stopped working on that day. His boss would hear none of it—code doesn't work for half of a year and then just blow up. My co-worker must have made some sort of change that broke the process, and he was challenged to find it.

That search ended up wasting half of his workday, but after much due diligence he finally brought the failure to me. I opened the execution log, found the results of the logic that attempted to find the date before 08/01, and . . .

I looked skyward, raised my hands, and with Shatnerian melodrama screamed "OCTAL!" I am embellishing, slightly—the moment was not as dramatic as Khan stranding Captain Kirk (played by William Shatner with Shakespearian flare) in the center of a dead planet in *Star Trek II: The Wrath of Khan*, but for me at least it was quite memorable.

What in the execution log upset me so? Let's find out, but before delving into octals, I'll start with integers.

Integers

We have already used the `set` command for alphanumeric values, but it's also used for arithmetic with the `/A` option. Recall what happens with a statement such as this:

```
set x=4+5
```

The variable denoted by `x` is set to the text `4+5`.

Using the `/A` option turns it into an *arithmetic* `set` command, so the following results in the `x` variable being set to the number `9`:

```
set /A x=4+5
```

The /A option transforms the set command into a means to perform addition and other arithmetic operations. Those previous values are obviously hardcoded as numeric.

A slightly more interesting example involves setting variables to numeric values and then adding them via the set /A command, as shown in Listing 6-1.

```
set nbr1=4
set nbr2=5
set /A sum = nbr1 + nbr2
>con echo The sum is %sum%.
```

Listing 6-1: Adding two numeric variables via the set /A command

The console output is `The sum is 9.`, and Listing 6-1 demonstrates that the /A option has altered the set command significantly—three times. First and most obviously, arithmetic is unlocked. Second, there are spaces around the equal sign, and in Chapter 2 I made a rather large point of the danger of doing that. To demonstrate, this command lacking the /A option

```
set myVar = X
```

does not set myVar to X. It sets a variable with a six-character name, myVar with a trailing space, to the two-character value of a space followed by X. By comparison, the /A option makes the set command behave more like an assignment operator of a modern language in that spaces in the command are not treated as parts of variable names or values; refreshingly, they are just spaces.

These three commands are all functionally equivalent; each sets myVar to 7:

```
set myVar=7
set /A myVar=7
set /A myVar = 7
```

To get the desired result without the /A option, spaces cannot exist around the equal sign. However, with the /A option they can exist, but they also aren't required, which is the second significant difference unlocked with the /A option.

The third difference in Listing 6-1 is that the variables nbr1 and nbr2 are not surrounded by percent signs. Hence, the /A option allows you to resolve variables without the ubiquitous delimiters. In a nod to flexibility, you still can use the percent signs and embedded spaces, or not, so these four statements are logically equivalent:

```
set /A result = nbr1 + nbr2
set /A result = %nbr1% + %nbr2%
set /A result=nbr1+nbr2
set /A result=%nbr1%+%nbr2%
```

The spaces make the code much more readable, so I advise against the last two options in the previous code. The first option is the cleanest, but some people are so used to having percent signs surround variables that the second option might provide comforting consistency.

Let's take one more pass at the `set /A` command from Listing 6-1, but this time, executed at the very beginning of a bat file:

```
set /A sum = nbr1 + nbr2
> con echo The sum is %sum%.
```

The resulting value of `sum` written to the console will be `0`. Because `nbr1` and `nbr2` are not yet defined, and unlike unset variables used in the alphanumeric context that default to null, unset variables used in the numeric context are considered to be zero. Since neither is set, the arithmetic `0 + 0` results in `0`.

WARNING *The range of permissible integers includes the values –2,147,483,648 through 2,147,483,647, inclusive. Batch stores numbers as 32-bit signed fields, so any integer will take on one of these 2^{32} values. This rarely poses a problem, but because the code is not compiled, take care to ensure that the data being processed conforms to the limitation. The code won't abort, nor will it hang; it'll simply fail to calculate the correct value. Batch is not the preferred language for macroeconomics.*

Batch Arithmetic

Batch arithmetic does more than simple addition. The following listing shows the five primary arithmetic operations (addition, subtraction, multiplication, division, and modulo division) and their syntaxes:

```
set /A sum = nbr1 + nbr2
set /A difference = nbr1 - nbr2
set /A product = nbr1 * nbr2
set /A quotient = nbr1 / nbr2
set /A modulo = nbr1 %% nbr2
```

The operators are similar to those in other programming languages, but note the double percent sign for modulo division. The `help` command shows a single percent sign, but the correct Batch syntax requires two. (In reality the modulo character is just a single percent sign, but the first percent sign is actually *escaping* the second. If this doesn't make much sense right now, hold that thought for Chapter 14, but use two characters for now.)

Now let's execute these arithmetic commands, but first we'll define the two operands, `nbr1` and `nbr2`. The results are shown to the right of each statement as a comment (as mentioned previously, the ampersand separates two commands, and the second one can be a `rem` command):

```
set nbr1=7
set nbr2=2
set /A sum = nbr1 + nbr2          &rem sum=9
set /A difference = nbr1 - nbr2   &rem difference=5
set /A product = nbr1 * nbr2      &rem product=14
set /A quotient = nbr1 / nbr2     &rem quotient=3
set /A modulo = nbr1 %% nbr2      &rem modulo=1
```

The addition, subtraction, and multiplication operations produce no surprises, but dividing 7 by 2 returns 3 rather than 3.5, because Batch arithmetic handles only integers and truncates the decimal portion of the result. Dividing 19 by 10 doesn't yield 1.9, and it won't even return the rounded value of 2. The intermediate result of 1.9 is truncated to 1.

Modulo is a useful operator that returns the remainder. Modulo *n* returns the values 0 through *n* – 1, so the modulo 2 operation returns 0 for even numbers because 2/2, 4/2, 6/2, and so on are integers and do not produce a remainder. Odd numbers return 1, because 3/2, 5/2, 7/2, and so on all have a remainder of 1.

Oddly, Batch doesn't support the exponential or power function, which is a source of frustration for some but an impetus for creativity for others. You can create a routine that takes in a base and an exponent and returns the exponential result (and I'll do just that in Chapter 18).

Augmented Assignment Operators

Augmented assignment operators can streamline the code when you want to add a number to a variable and store the result in that same variable. The most obvious example is a simple counter where you might want to increment a variable by one for each execution of the set command, for example:

```
set /A veryVerboseTallyVariable = veryVerboseTallyVariable + 1
```

I intentionally chose a verbose and cumbersome variable name because try as we coders might, they sometimes become nearly unavoidable.

The following syntax is logically identical, condensed, and easier to comprehend:

```
set /A veryVerboseTallyVariable += 1
```

The next command adds 17 to a far more succinctly named variable:

```
set /A nbr += 17
```

Likewise, the following set commands subtract 2, multiply by 2, divide by 2, and perform modulo 2 division, respectively:

```
set /A nbr -= 2
set /A nbr *= 2
set /A nbr /= 2
set /A nbr %%= 2
```

Again, note the double percent signs for the modulo division. Many experienced Batch coders don't know that the augmented assignment operators are available in Batch, wrongly assuming that they exist only in more modern languages, but they do exist, and you should use them when appropriate.

Order of Operation

You can do more complex arithmetic with the order of operation rules from mathematics. You might have learned the PEMDAS acronym in a pre-algebra class (or "Please Excuse My Dear Aunt Sally" as a mnemonic) for "parentheses, exponents, multiplication and division, and addition and subtraction." For Batch we have PMDAS, which is a whole lot harder to pronounce, but as mentioned, exponents aren't supported (maybe the mnemonic "Please Make Dessert Aunt Sally" will catch on). Let's take this example:

```
set /A nbr = 3 * (1 + 2) / 4 - 5
```

First the `1` and the `2` are added to make `3` because they are in parentheses, even though addition and subtraction are last in the order of operation. Multiplication and division share the same hierarchy, so the interpreter performs them from left to right. The `3` leading the expression is multiplied by the `3` from the addition, giving us `9`, and `9` is then divided by `4`, resulting in `2.25`. Actually, that's truncated, so it's simply `2`. Finally, subtract `5`, and `-3` is the result.

This example is pedagogical only, because it would be far simpler just to set `nbr` to `-3`. In practice, a mix of hardcoded numbers and variables will be used. For example:

```
set /A nbr = ((nbr1 + nbr2) * -10) / 4
```

The outer parentheses here are unnecessary by the rules of PMDAS, but they make the statement more readable.

Augmented assignment operators can also work with more complex expressions. These two statements are logically identical:

```
set /A nbr = nbr + (2 * (4 + nbr) - -5)
set /A nbr += 2 * (4 + nbr) - -5
```

In both commands the variable `nbr` is being incremented by a mathematical expression also containing `nbr`, with the only difference being that the second command uses the augmented assignment operator. Based on the order of operations, both add 4 to the variable, double it, and subtract –5. (Subtracting –5 is equivalent to adding 5.) Ultimately, the result of this expression is the amount by which `nbr` is incremented.

Octal and Hexadecimal Arithmetic

Batch supports both octal and hexadecimal arithmetic. Both number systems are more similar to the way a computer *thinks* than base 10, so it's useful for a coder to understand them and be able to use them.

The decimal number system is base 10 and uses the digits 0 to 9. There is no digit for 10; instead, there are two digits: a new place value starts with 1, while the ones place restarts at 0, hence 10. In contrast, the octal number system is base 8, using the digits 0 to 7. Adding 1 to the octal 7 does not produce 8, because 8 (and 9) are meaningless characters in the octal number system. Instead, the octal number 10 (pronounced "one-zero" because it is not "ten") is equivalent to the decimal number 8. Likewise, the octal 11 is equal to the decimal 9, and so on.

The hexadecimal number system is base 16, so it has the opposite problem of octal: it needs 16 unique digits, more than the 10 used in most human number systems on account of our having evolved to possess five digits on each of two hands. After counting from 0 to 9, we have the "numbers" A, B, C, D, E, and F. The hexadecimal number B is equal to the decimal number 11, the hexadecimal F equals the decimal 15, and the hexadecimal 10 is equal to the decimal 16.

Batch can perform arithmetic with octal, hexadecimal, and/or decimal inputs, while always returning the answer as a decimal. Hexadecimal numbers are preceded with `0x`, and octal numbers are preceded with `0` alone. Hence, these two variables are assigned octal and hexadecimal values, respectively:

```
set octalNbr=012
set hexadecimalNbr=0xB
```

Regardless of the base of the operands—decimal, octal, or hexadecimal—Batch always stores the result as a decimal. To demonstrate, first take this example:

```
set decimal7=7
set decimal1=1
set octal7=07
set octal1=01

set /A decimal = decimal7 + decimal1
set /A octal = octal7 + octal1
```

The numerals 7 and 1 are being added as decimals and octals. The decimal result is obviously 8. The sum of the two octal numbers is octal 10 ("one-zero," not decimal 10), but the interpreter immediately stores the value as a decimal 8. In this example, decimals and octals behave the same way, but that's not always true.

Now take this example:

```
set decimal11=11
set decimal2=2
set octal11=011
set octal2=02

set /A decimal = decimal11 + decimal2
set /A octal = octal11 + octal2
> con echo The decimal sum is %decimal%.
> con echo The octal sum is %octal%.
```

The decimal addition yields decimal 13, while the octal addition yields octal 13 ("one-three," not decimal 13). Remember, the octal number system has no 8 or 9. Octal 10 is decimal 8, and in this example octal 13 is decimal 11. Therefore, in Batch, 11 + 2 = 13, but 011 + 02 = 013 = 11, so the following result is displayed:

```
The decimal sum is 13.
The octal sum is 11.
```

The interpreter can even handle arithmetic with a mixture of decimal and octal values. The decimal addition of 10 + 10 is 20, and the octal addition of 010 + 010 is 16. When adding a decimal and an octal, say 10 + 010, Batch gives the correct result of 18. Usually, this type of arithmetic is done by accident, but sometimes savvy coders will use this to their advantage, and it's good to know that it's possible.

In a similar fashion, these values are treated as hexadecimals:

```
set /A hexadecimalNbr = 0xA * 0x14
```

With this multiplication, `0xA` is equal to decimal 10, and `0x14` is four more than 16 when converted to decimal. After this statement executes, the variable is equal to 200, the product of 10 and 20.

Octals and hexadecimals can be powerful tools; however, be careful to ensure that there are no leading zeros if you are intending to do decimal arithmetic. Since hexadecimals start with `0x`, accidentally performing hexadecimal arithmetic is far more difficult, but unknowingly performing octal arithmetic because of a seemingly innocuous leading zero is exceedingly easy.

NOTE *Because math is all around us, you'll find boxes containing various examples of bat file arithmetic in Chapters 16, 18, and 21. Batch also has arithmetic operators for bit manipulation: bitwise and, bitwise or, bitwise exclusive or, logical shift left, and logical shift right. I'll wait until Chapter 30 to explore them because these operators use some special characters that have other uses and because many experienced coders have never manipulated a bit in compiled code, much less Batch.*

Floating-Point Numbers

Batch doesn't explicitly handle floating-point numbers—that is, non-integer rational numbers. In fact, if extensive processing is to be done on such numbers, there are better tools to use than Batch. It would be analogous to digging a foundation for a house with a spade shovel. It can be done, but only by the most austere ascetic. If the task is big enough, write some compiled code and call it from the bat file, but when some lightweight floating-point arithmetic needs to be done, Batch can handle it, just as you can use the spade shovel to plant a couple tulip bulbs in the front yard.

Keep in mind that all Batch variables are really just glorified strings. We can easily assign a couple of variables floating-point values—that is, some numbers with a period for the decimal point. Here are two amounts in dollars and cents:

```
set amt1=1.99
set amt2=2.50
```

If these were integers, we could simply add them with the `set /A` command. Let's try it and see what happens:

```
set /A sum = amt1 + amt2
```

The result is the value `3` being stored in the `sum`, not the hoped-for `4.49`. The decimal part of each number is completely ignored, resulting in the sum of the integers `1` and `2`.

We need to remove the decimal place, do the arithmetic, and restore the decimal place. Multiplying each amount by 100 would do the trick, but again, Batch isn't going to allow that. Since the floating-point value is just a disguised string, however, we can remove the decimal point with the syntax described in the previous chapter:

```
set amt1=%amt1:.=%
set amt2=%amt2:.=%
```

Now the amounts are `199` and `250`. The following `set /A` command results in `449`:

```
set /A sum = amt1 + amt2
```

To restore the decimal, we can't simply divide by 100—once again, that works only for integers, but we can use more of the string-parsing logic from the previous chapter. Using substringing, the following `set` command resets the variable to a concatenation of three items: everything but the last two bytes of the number, a hardcoded decimal place (aka dot), and the last two bytes of the number:

```
set sum=%sum:~0,-2%.%sum:~-2%
> con echo The sum is %sum%.
```

Finally, the variable written to the console has been set to `4.49`.

Multiplication works the same way. If you buy that new computer for $499 with no payments for the first year and an interest rate of 19 percent, how much will you owe a year from now? The interest rate translates to a factor of `1.19`, but again we must remove the decimal place. After finding the product of two integers, we restore the decimal place by inserting it before the last two bytes, as shown in Listing 6-2.

```
set amt=499
set factor=1.19
set factor=%factor:.=%
set /A product = amt * factor
set product=%product:~0,-2%.%product:~-2%
> con echo The product is %product%.
```

Listing 6-2: Multiplication of an integer and a floating-point number

The `product` of `593.81` might make you reconsider the financing plan.

The goal of every coder should be to write "bullet-proof" code—unfortunately, the previous offering is more of a cotton mesh than Kevlar, and there are a number of batveats to discuss. We've made several assumptions, and if any one of them is violated, the code will break. The addition assumes that both numbers have two decimal places; 1.9 instead of 1.90 will throw off the result by a factor of 10. A non-numeric character, other than the decimal place, will cause issues, and a leading zero on the value will trigger octal arithmetic. The multiplication is even more complicated. Listing 6-2 contains an integer amount, but if `amt` had been expressed in dollars and cents, the product would have resulted in four decimal places, not two. To represent the result as dollars and cents, the last two bytes should be truncated—or better yet, rounded.

I won't go into these nuances here for the simple reason that if the inputs are not consistent and data validation is required, Batch floating-point arithmetic may not be the optimal solution. Coding for all possible situations would be tedious at best. What's important is that the coder understands the options at hand. If all the values have a consistent number of decimal places, one can do the arithmetic with just a few lines of code. On the rare instance when I have resorted to using the floating-point data type in Batch, it has been for a very specific task involving consistent data. Break out that spade shovel, but only when appropriate.

An Octals Case Study, Continued

So, what exactly did I find in that execution log on that first day of August of a year early in the millennium? In the bat file, today's date was formatted as CCYYMMDD, for instance `20050801`, which was broken down into three discrete fields:

```
todaysYear = 2005
todaysMonth = 08
todaysDay = 01
```

If `todaysDay` is anything other than `01`, we simply subtract 1 from the eight-digit number and move on. But when it is `01`, we need to do some additional arithmetic. Considering just the month logic (and understanding that there'll be some special logic for January), we must subtract 1 to determine the prior month:

```
set /A month = todaysMonth - 1
```

When `todaysMonth` is `03`, the `month` is `2`; when `todaysMonth` is `07`, the `month` is `6`. But when `todaysMonth` is `08` as it is on August 1, the `month` in the previous arithmetic resolves to the value of `-1`.

The interpreter sees the leading `0` and treats the arithmetic as octal arithmetic. Octal understands only the digits `0` through `7`, so when the interpreter sees `8`, it considers the character to be as foreign as "ohkuh" (the numeral corresponding to eight in the Vulcan language) and simply ignores it. Ultimately, the `set /A` command assigns the mathematical result of what remains of the expression, which is `-1`, to the `month` variable. This value ends up breaking the date logic, and we fail to get the desired date of July 31.

"OCTAL!"

Using substringing and the `if` command, I inserted this one-line fix to strip the leading zero, if present, off the value of the `todaysMonth` variable:

```
if %todaysMonth:~0,1% equ 0  set todaysMonth=%todaysMonth:~1%
```

The code worked fine for years to come, even on the firsts of August and September. If the original code hadn't been run on August 1, it would have failed if run on September 1, since September is denoted by `09`. But what if the code hadn't been run on either of those days? When would it fail next? On October 1, the month would be denoted as `10`. The interpreter would have treated that like a decimal, and the code would have performed as expected. So, the firsts of August and September are the only dates capable of breaking the code.

Be very aware of octal.

Summary

In this chapter, I discussed numeric data types and how they are treated in Batch. Unlike most other languages, Batch variables are not defined as a certain data type. Intrinsically, all variables are simple strings, but when that string contains a number, it can be treated as numeric.

Addition, subtraction, multiplication, division, and even modulo division work on decimal integers with relative ease, using the order of operation rules you likely learned in school. Octal and hexadecimal integers are also supported, although octal arithmetic can all too easily be invoked in error. Take it from my personal experience and ensure that your decimal integers are not prefixed with any zeros. Augmented assignment operators offer a handy and underutilized tool for incrementing integers.

The floating-point numeric data type isn't supported in Batch, but you've learned that with a little work, you can perform some lightweight arithmetic on numbers with a decimal point.

Changing gears, I'll discuss file movements in the next chapter. An immensely useful feature of Batch is the creating, copying, moving, renaming, and deleting of files and directories.