Arithmetic in C

+, -, *, /, %

In C, your programs will often perform very basic mathematical operations in order for you to solve your problem. Less often you will be required to do some very complex mathematical operations, in which case you will be using functions that have been written for you, and are available in the **math.h** Standard C Library for your use. Those are not the functions that I am referring to in this section. In this section, we will discuss the very basic mathematical operators available to you for performing simple math in your program. Keep in mind, most C programs perform some type of arithmetic calculation discussed below.

In C we use familiar arithmetic operators (similar to regular math). The following table shows the math operators used, and their purpose:

OPERATION	OPERATOR
Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Modulus (remainder)	**%**

Here is a very fundamental rule for you. It is only 5 simple words, but they are VERY IMPORTANT. Here they are: ***Integer math yields integer results*.**

Now, you probably need to know exactly what that means. (I don't blame you.) What I mean by that statement is if you have a calculation where all of the values being used in the calculation are integer, then the result is of type integer. If however, at least one of the values in the calculation is a floating point number, then the result is of type float.

An example will help explain this further:

```
int  sum;
int  num1 = 10;
int  num2 = 4;

sum = num1 + num2;  /* addition */
```

In this case, the value of variable **sum** after the addition is **14**. Notice both **num1** and **num2** are of type **int**, so, the result of this addition is of type **int**. Good thing,

because we are trying to stuff the result of the addition into a variable of type **int** (variable sum)!

Let's try another calculation using the above variable declarations:

```
sum = num1 / sum2; /* division */
```

In this case, the answer will still be of type **int** because "integer math yields integer results", and *both* **num1** and **num2** are integers. So, the result of integer division is also of type **int**. The value of **sum** in this case would be 2. (10/4 = 2).

What about:

```
sum = num1 % num2;   /* modulus or remainder */
```

As you probably guessed. The remainder of 10 / 4 is also **2**. So, the result of the math operation above would be 2.
If we modified the data type of **num2** to be **float**, and had the following code segment:

```
int    sum;
int    num1 = 10;
float  num2 = 4.0;

sum = num1 + num2;
```

Well, here we have a problem. Now, variable **num2** is no longer an integer. So, the result of this addition is of type **float**. We are trying to stuff a floating point result (which would be 14.0) into integer variable **sum**. Some compilers would handle this okay (they would chop off the .0, and stuff the number 14 into variable **sum**). But not all! So, you want to keep your code as portable as possible by NOT doing this sort of thing.

Recall another very important rule: *The data type of the data on the right side of the assignment operator should be the same as the data type of the variable on the left of the assignment operator.*

In the above (incorrect) example, if we want to resolve the problem, we have 2 choices:

1. Modify the data type of variable **sum** to be of type **float**. Then the data type on both sides of the assignment statement would be the same. (Recall, only 1 of the variables in a mathematical expression needs to be **float** for the calculation to yield a **float** result.)

2. If you wanted to keep **sum** as type **int**, and you wanted to keep **num2** as type **float**, you can do something quite "magical" in the programming world. It is called "*type casting*". (This sounds like something right out of a Harry Potter book!) Actually, it's pretty straightforward, and VERY handy for ensuring that your data types match. What you do is you "temporarily" modify the data type of the variable in question, to be the data type you want it to be. This type-casting has no effect on the original data type of the variable. How do you do this you ask? Well like this. I will type-cast variable **num2** to be of type **int** just during the addition, so that both sides of the assignment operator will be of type **int**:

sum = num1 + (int)num2;

Do you see the data type **int** in parenthesis right in the addition expression?!?!? That is how we typecast. I have told the C compiler to *temporarily* make variable **num2** to be of type **int**. Only in this statement will it be of type **int**. If this program segment had additional lines of code using variable **num2** following this statement, it would be considered its original data type, which is of type **float**. You can typecast any variable to be of any data type you choose.

A good programming practice: *put a space between the operators and variables.* Notice, the two statements below, which one do you think is more readable?

1. **sum=num1+num2;**
2. **sum = num1 + num2;**

The more readable statement is #2 because of the spaces surrounding the operators and variables. Very simple to do, yet very effective!

One last point in this section, there is an "*order of precedence*" when working with mathematical operators in C. It follows the same precedence as in regular math. That is, if you have an expression with more than one math operator, C processes the expression from left to right, and will do multiplication and division first, then addition and subtraction.

Example:

int result;

result = 3 + 4 * 2;

In this case, **result** would equal **11**, because the multiplication has a higher precedence than the addition. If you in fact wanted to do the addition first, then you must use paranthesis, which has a higher priority than the multiplication (and division). It would look like:

result = (3 + 4) * 2;

Now, **result** would equal **14**, the desired value (if you wanted to add first).

My advice to you (another readability issue), is to *use parenthesis* if you have more than 1 operator because it is *more clear*. For example, even if you wanted to perform the multiplication above first, it is much more clear that you intend to do the multiplication first if you type in your code as follows:

result = 3 + (4 * 2);

In this case the parenthesis are not necessary, since C will do the multiplication first anyway, however, you must agree, that by adding the parenthesis, it is a heck of a lot more clear that the multiplication is being done first.

Oh yes; one last thing (really). **Do not** attempt to **divide by the number zero**! What happens to your program if you try varies from machine to machine. Some will give you a compiler error (if the compiler detects it). Other machines will compile and link fine, and then give you a fatal error during runtime. Some machines will hang during runtime, and you will have to reboot. This can be a very time-consuming process. You should simply try to avoid it.

Please review and step through program 3.2 on page 31.

This will reinforce the concepts discussed in this section.

That's all for basic math in C. We will use basic math throughout our entire course. Again, if you have any questions regarding this segment, please post them in the discussion area for this week.

That’s all for lecture notes this week. And now, the real fun begins. It’s time to start working on your first programming assignment!

Next week we will be discussing 2 topics that the textbook chooses to discuss in later sections of the textbook. I feel however, that it is a good point in the course to review both of these topics. The topics discussed next are: "Program input" (prompting the user), and the "Standard C Library".