Arithmetic Expressions in C

Arithmetic Expressions consist of numeric literals, arithmetic operators, and numeric variables. They simplify to a single value, when evaluated. Here is an example of an arithmetic expression with no variables:

3.14*10*10

This expression evaluates to 314, the approximate area of a circle with radius 10. Similarly, the expression

3.14*radius*radius

would also evaluate to 314, if the variable radius stored the value 10.

You should be fairly familiar with the operators + , - , * and /.

Here are a couple expressions for you all to evaluate that use these operators:

Expression	Value
3 + 7 - 12	
6*4/8	
10*(12 - 4)	

Notice the parentheses in the last expression helps dictate which order to evaluate the expression. For the first two expressions, you simply evaluate the expressions from left to right.

But, the computer doesn't ALWAYS evaluate expressions from left to right. Consider the following expression:

3 + 4*5

If evaluated from left to right, this would equal (3+4)*5 = 35

BUT, multiplication and division have a higher order of precedence than addition and subtraction. What this means is that in an arithmetic expression, you should first run through it left to right, only performing the multiplications and divisions. After doing this, process the expression again from left to right, doing all the additions and subtractions. So, 3+4*5 first evaluates to 3+20 which then evaluates to 23.

Consider this expression:

3 + 4*5 - 6/3*4/8 + 2*6 - 4*3*2

First go through and do all the multiplications and divisions:

3 + 20 - 1 + 12 - 24

Now, do all the additions and subtractions, left to right:

10

If you do NOT want an expression to be evaluated in this manner, you can simply add parentheses (which have the highest precedence) to signify which computations should be done first. (This is how we compute the subtraction first in 10*(12 - 4).)

So, for right now, our precedence chart has three levels: parentheses first, followed by multiplication and division, followed by addition and subtraction.

Integer Division

The one operation that may not work exactly as you might imagine in C is division. When two integers are divided, the C compiler will always make the answer evaluate to another integer. In particular, if the division has a leftover remainder or fraction, this is simply discarded. For example:

13/4 evaluates to 3
19/3 evaluates to 6 but

Similarly if you have an expression with integer variables part of a division, this evaluates to an integer as well. For example, in this segment of code, y gets set to 2.

```
int x = 8;
int y = x/3;
```

However, if we did the following,

```
double x = 8;
double y = x/3;
```

y would equal 2.66666666 (approximately).

The way C decides whether it will do an integer division (as in the first example), or a real number division (as in the second example), is based on the TYPE of the operands. If both operands are ints, an integer division is done. If either operand is a float or a double, then a real division is done. The compiler will treat constants without the decimal point as integers, and constants with the decimal point as a float. Thus, the expressions 13/4 and 13/4.0 will evaluate to 3 and 3.25 respectively.

The mod operator (%)

The one new operator (for those of you who have never programmed), is the mod operator, which is denoted by the percent sign(%). This operator is ONLY defined for integer operands. It is defined as follows:

a%b evaluates to the remainder of a divided by b. For example,

12%5 = 2
19%6 = 1
14%7 = 0
19%200 = 19

The precendence of the mod operator is the same as the precedence of multiplication and division.

For practice, try evaluating these expressions:

Expression	Value
3 + 10*(16%7) + 2/4	
3.0/6 + 18/(15%4+2)	
24/(1 + 2%3 + 4/5 + 6 + 31%8)	

(Note: The use of the % sign here is different than when it is used to denote a code for a printf statement, such as %d.)

Initialization of variables

In the previous program example, we first declared our variables and then initialized them. However, these two steps can be done at once. Thus the lines:

```
int feet_in_mile, yards_in_mile;
int feet_in_yard;
yards_in_mile = 1760;
feet_in_yard = 3;
```

can be replaced by

```
int feet_in_mile, yards_in_mile=1760;
int feet_in_yard=3;
```

Generally, it is a good practice to initialize variables (when their initial value is known.) The reason is that before you initialize a variable, any random value could be stored in it. By initializing the variable, you know definitively what the variable will evaluate to at any point in your program.

Use of #define and #include

The # sign indicates a preprocessing directive. This means that sometime is done BEFORE the compiler runs. When we do a #include, this tells the compiler to include the given file before compilation. If the file is part of the C library (like stdio.h is), then the format is as follows:

#include <filename>

Other common include files in the C library are: math.h, string.h, and stdlib.h.

If however, you want to include a file that isn't in C's standard library, you must do it as follows:

#include "filename"

We won't be using this type of include till near the end of the course.

What a #define does is replace a given variable name with a value, before a program is compiled. Thus, we could change the beginning of mile conversion program to read as follows:

```
#include <stdio.h>

#define YARDS_IN_MILE  1760
#define FEET_IN_YARD   3
```

Note that I have changed the capitalization of the variables because the standard is to have all constant names be in CAPS. What this does is replace each instance of YARDS_IN_MILE with the value 1760, and replace each instance of FEET_IN_YARD with 3 before the compiler is invoked.

Use of printf

The most simple use of printf only prints out a string literal. A string literal is a string of characters and is denoted by the characters inside of double quotes.

However, it is useful to print out the values stored in variables. But, you can't put variable names inside of a string literal, because then the variable name itself would print out:

```
int x = 5;
printf("the value is x\n");
```

will print out `the value is x` not `the value is 5.`

To deal with this, C uses conversion characters inside of the string literal. These are denoted by a percent sign followed by a letter. We will most commonly be using %d, %f, %lf, %c and %s. These simply signify to print out a certain type of variable, but not what that variable is. To clarify this point, you must list all the necessary variables separated by commas after the string literal. Consider the following example:

```
int x = 7;
float y = 3.1;
printf("x = %d, y = %f", x, y);
```

This will print out:

```
x = 7, y = 3.100000
```

As you can see from this example, you list the variables in the corresponding order in which they will appear in the print statement. *If you use the wrong code, the output is unpredictable!!!*

Formatting output spacing

One other modification that can be added to the output of variables is a field width. If you want a certain variable to print out in a given number of spaces, then you can place that in the conversion code. Consider the following code:

```
char first = 'A';
printf("%c%3c%3c\n",first,'R', 'G');
```

This will print out:

```
A  R  G
```

The 3 before the second c specifies to allocate three spaces to print out the second character. The printout defaults to printing the character out right-justified.

While this may not seem useful, the same type of formatting is useful for floats. For floats, you can specify the number of digits before and after the decimal point:

```
float y = 3.12;
printf("y = %1.2f", y);
```

will print out

```
y = 3.12
```

instead of

```
y = 3.120000
```

The first number in the percent code represents the total length of the field while the second number represents the number of decimals. In this particular example, since 1 is less

than the minimum field width of 4 characters, the number is printed with 4 characters.

Something like

```
float y = 3.12;
printf("y = %9.2f", y);
```

will print out

```
y =       3.12
```

where there are an extra 5 spaces in front of the 3.12, since it's printing out a right justified result in a field of 9 characters.

You should experiment on your own to find out what happens with various conversion codes and when the conversion code doesn't match the actual number of digits in the float variable being printed.