A program must find the product of two floating numbers num1 and num2, which could equally be termed A and B variables. The output of this program is the result of multiplying two floating point numbers. Users enter two float numbers into this program, which stores them into num1 and num2, and then runs. Once entered the numbers will have been multiplied and the program will display the output produced as a result.

**Defining the Float-point**

In defining a floating-point, as the name implies, it is a special type of data that's built into the compiler that allows value definitions with floating decimal points. Floating-point variables can hold a value such as 5928.0 and -4.14, or 1.18954 as a real number. By virtue of its name, a floating-point indicates that the decimal point can float; that is, any number of digits in before and following the decimal point can be supported.

**Float-point Arithmetic with Example**

With floating-point arithmetic, addition, subtraction, multiplication, and division are the arithmetic operations on floating-point numbers. Basically, the operations are performed with algorithms closely replicated to those any given algorithm for sign magnitudes (since the representation is similar) -- for example, only multiplying the number within the same sign. Obviously, division is evident when the numbers are of opposite signs; addition or subtraction in other instances.

Consider the following example of decimal numbers in scientific notation as a great example of multiplication. When multiplied, the mantissa and exponents bear the true value...

0.3 × 10 8

6.8 × 10 -13

\_\_\_\_\_\_\_\_\_\_

2.04 × 10 -4

The answer is:

**Underflow and Overflow in C Floating-points**

Programming in C has taught me that there can be overflows and underflows in floating-points. It is possible to overflow a floating-point number, just like integers which I find fascinating. Exponents can be checked before and during the normalization process for floating-point overflow in a floating-point. However, people learning to program floating-points in C should be aware that infinity can be expressed in a calculation as soon as an overflow occurs. In floating-point, underflow occurs too. The scale becomes ineffective when a number is too small, “near 0,” to be captured.

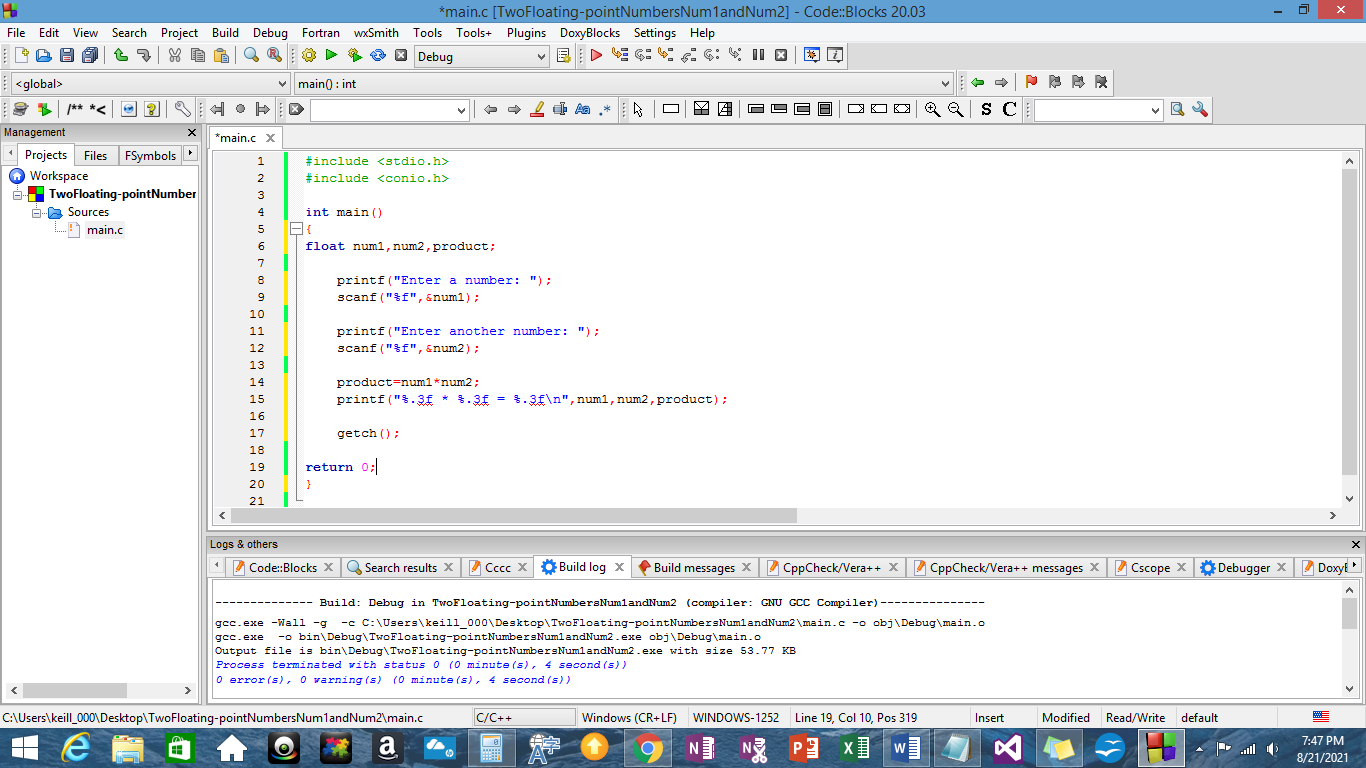
One should definitely have math and code savvy because floating-point operations could be tedious in any language especially, C. The data must be shifted and masked many times before integers in floating-point operations can be used to calculate the result. For the number to be returned to float-point format, further shifting and masking operations has to be executed in the code.

**Programming the Floating-points Num1 and Num2**

As you will notice in my images below, for multiplying two floating-point numbers using the submitted sources program, I enter two numbers into the keyboard, which are scanned using scanf() and stored in variables num1 and num2. By arithmetic operation of \* the variables num1 and num2 are multiplied together, with the result kept in the product variable. In storing floating-point numbers, the sign, mantissa, and exponent are stored individually within separate fields (Bronson, 2007).

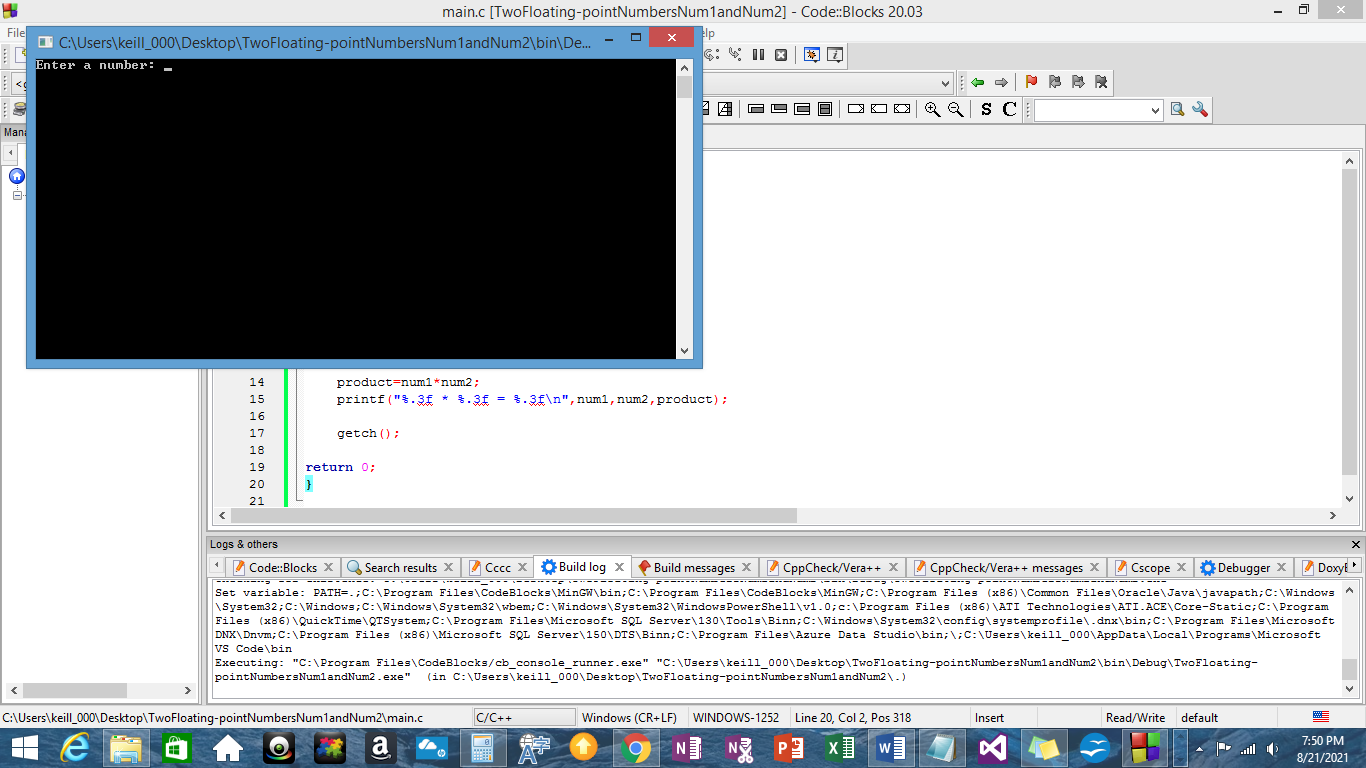
**Image 1**

* I entered my code for floating-point num1 and num2 storage. I then ran the program.



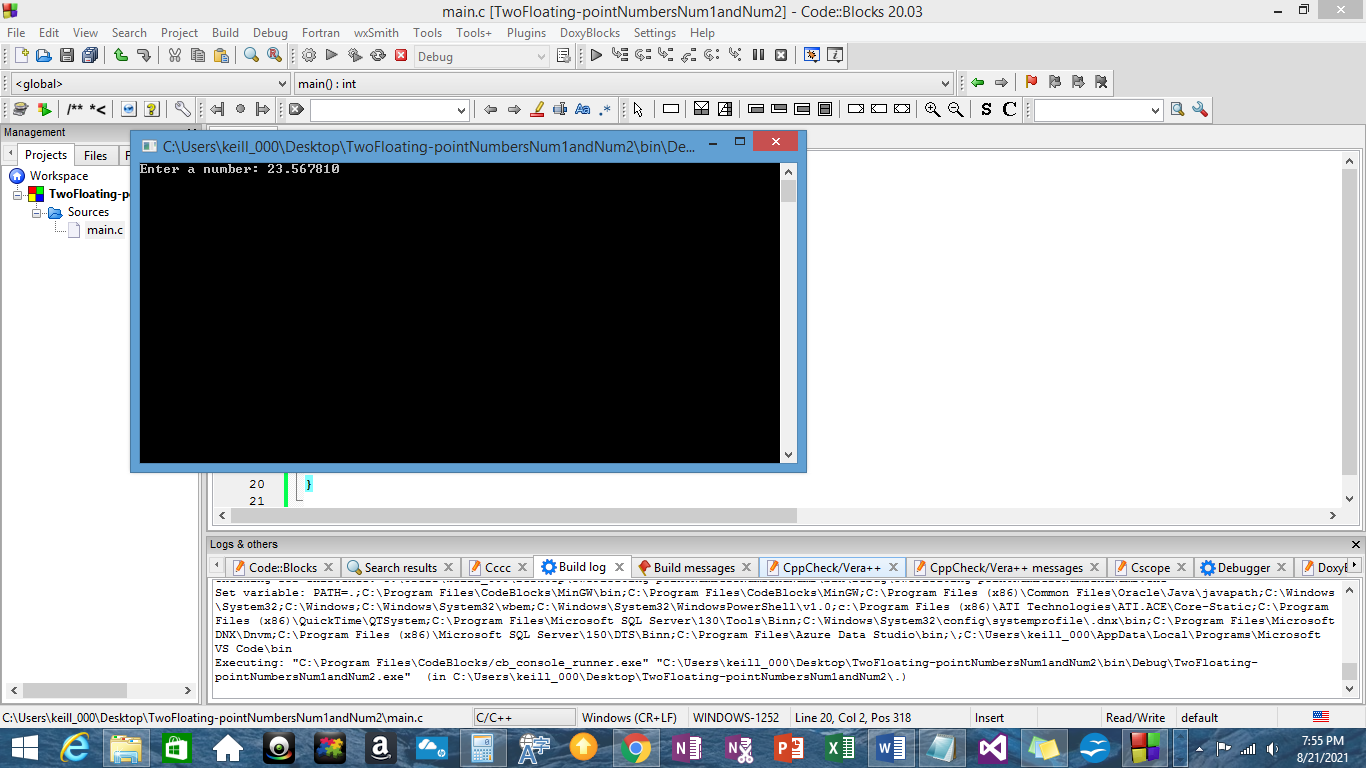
**Image 2**

* I am prompted to keyboard enter a number into the console.



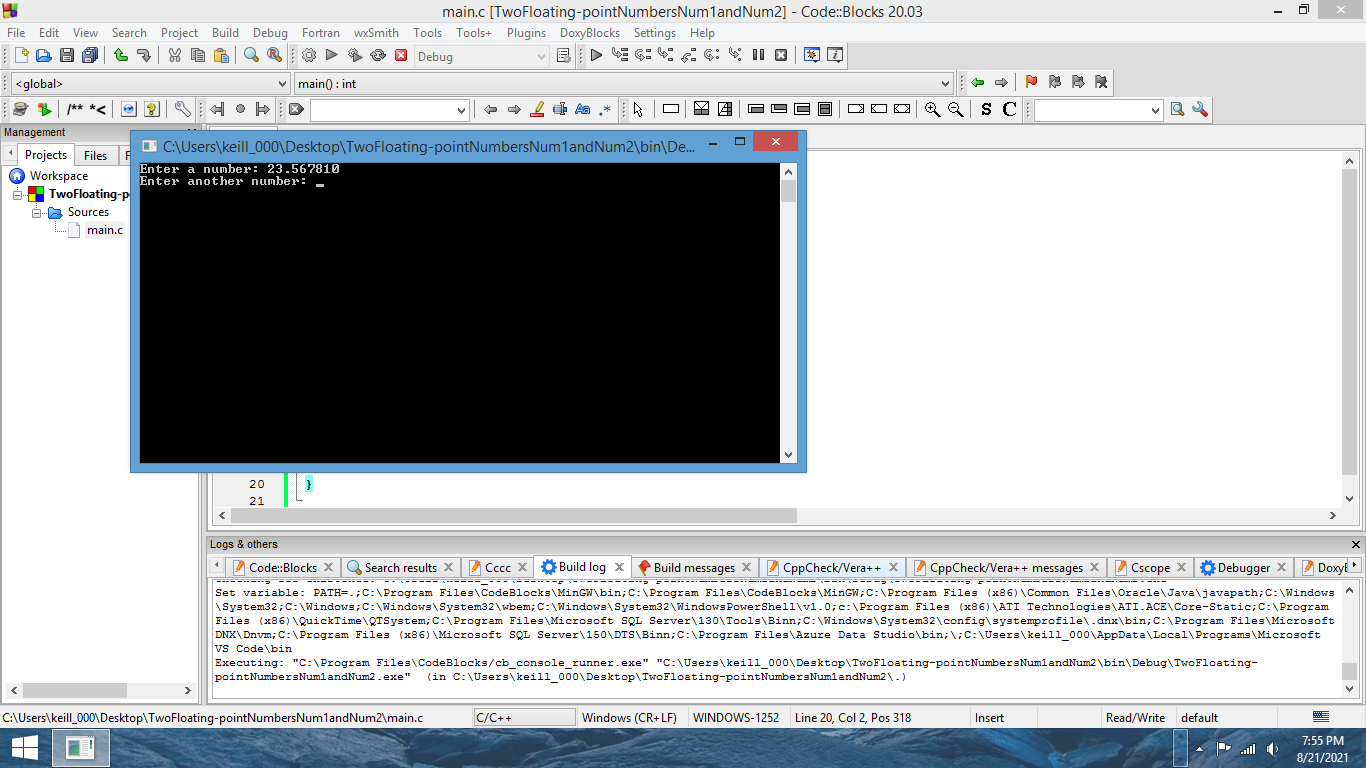
**Image 3**

* I enter the decimal number “23.567810” into the console. I pressed enter after.



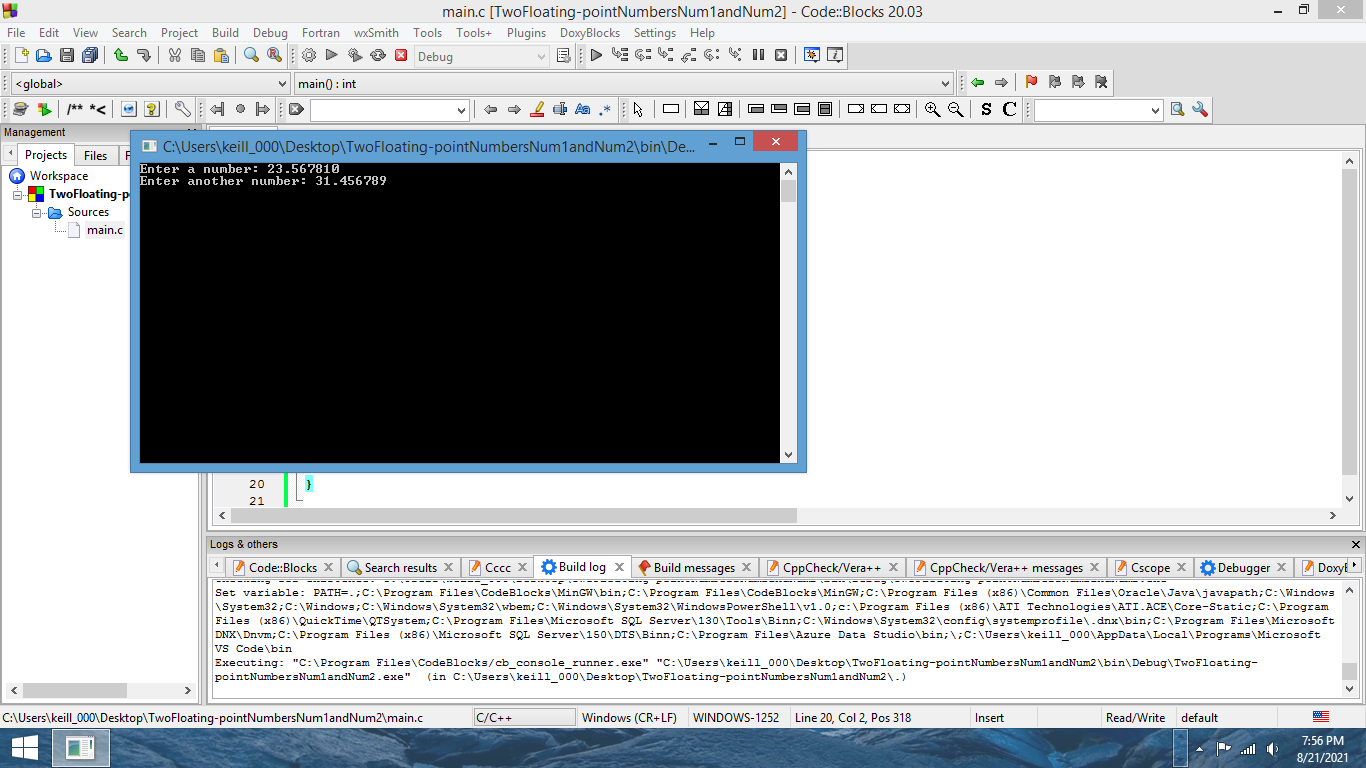
**Image 4**

* I am prompted to enter another number in with the previous number “23.567810” after pressing enter.



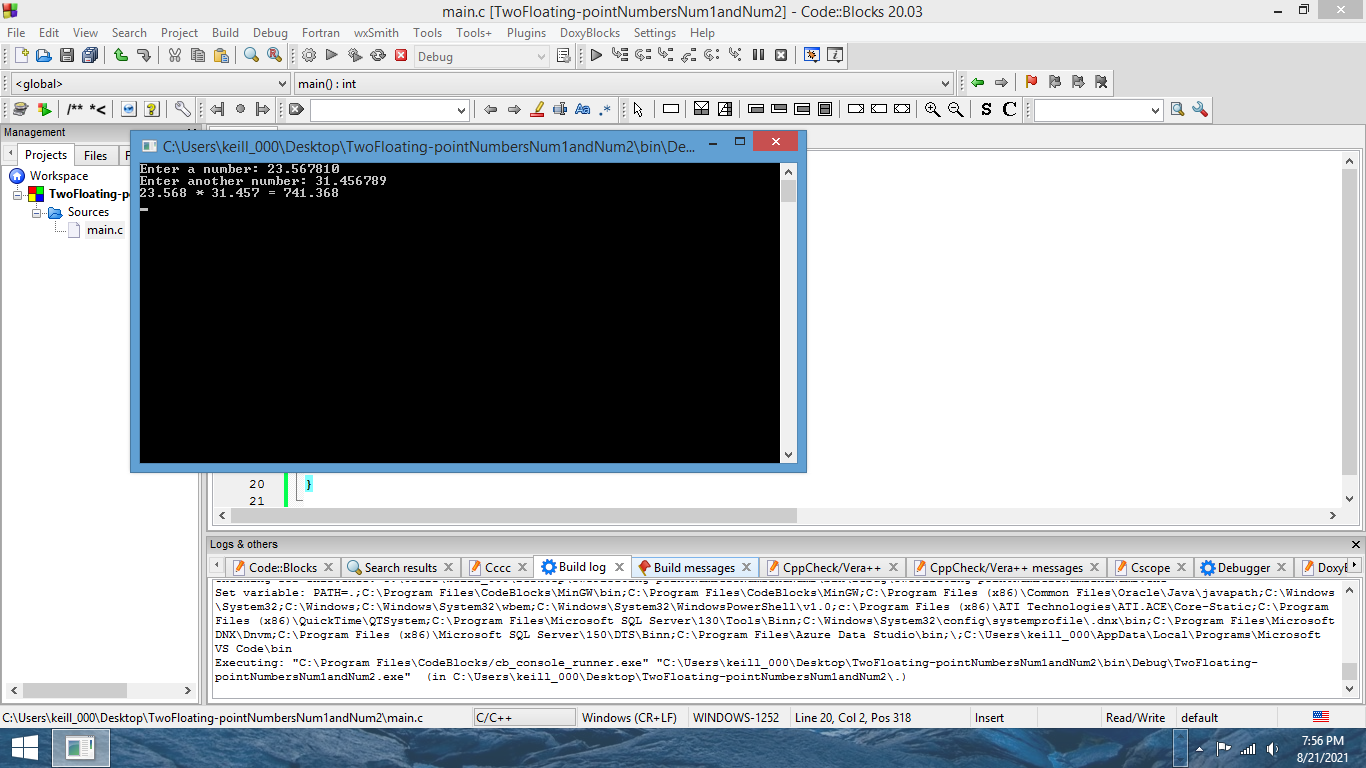
**Image 5**

* I keyboard entered into the console the number “31.456789.” I pressed enter.



**Image 6**

* After pressing enter, I received input as a result of multiplication; 23.568 \* 31.457 = 741.368 for the final result.



**Floating-point Number Precision**

In multiple precision arithmetic, exact rounding is necessary. The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right (Shanthi, n.d.). Higher precision can be achieved in two ways. Assembler programs can code routines that manipulate floating-point numbers by using an array of words containing a large significance.

The second method uses an array of floating-point numbers as a representation of higher precision floating-point numbers, adding the elements of the array in infinite precision to recover the high precision floating-point number. As a result of C, it is a semi-portable lower-level language when using floating-point numbers, but it requires rounded arithmetic to be correctly implemented.

**Conclusion**

The fixed point representation in floating-points cannot be used for very small or very large number representations. There will be a loss of accuracy in programming. Due to this, you have to render binary points in floating-point representations. Decimal numbers can be written in scientific notation form but it can also represent the fixed point number for example, -1.12345. Another part of this fraction represents the exponent value, and indicates that the actual binary point position is 9 positions to the right (left) of the indicated binary point.

The binary point implicitly resides immediately to the left of mantissa bit 22, and a leading one is always assumed (Bronson, 2007). Floating-point representations are those in which the binary point can be moved to any position and the exponent value is adjusted accordingly. Conventionally, you place a floating-point directly after the first significant digit non-zero (normalized representation). Signs, digits and the signed exponent are the representations in place of the explicit base.

**Bibliography**

Bronson, G.J. (2007). *A First Book of ANSI C, 4th ed.* Boston: Thomson Course Technology

Shanthi, A.P. (n.d.). *Floating Point Arithmetic Unit.* Computer Architecture. Ch.7. Retrieved August 21, 2021, https://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html