Exercise 1: Overflow and Underflow

Compile and run the code, and clearly explain the results.

In the first part of the code, we created 2 32-bit integers, one signed and one unsigned, and both initialized to zero. After decrementing the integers, the signed integer returned -1 and the unsigned returned 4294967295. This is because since the first integer is signed, it is allowed to return negative integers (0-1 = -1). However, the second integer is unsigned, meaning that it cannot be negative. Therefore, when we subtracted one from zero, the unsigned integer returns the maximum value of the 32 bit unsigned integer, which is 4294967295.

For the second part of the code, we created 2 16-bit integers (one signed and one unsigned), and both initialized to the maximum value for its data type (65535). Due to the range of the signed 16-bit integer, even though it is supposed to hold the value 65535, there is an overflow, causing it to equal only -1. For the unsigned integer, it is able to hold up to 65535 without an overflow. When both are incremented by 1, they both equal zero (-1 + 1 = 0, and 65535 + 1 = 0 since that was the maximum range for the unsigned integer and once it is there it must return to 0).

Exercise 2: Dynamic Range and Precision

Compile and run the code, and clearly explain the results.

I made two different versions of the values 1 million, 1, and the sum of both. The difference lies in the data type used. In the first, I used int32_t, which is a variation of the int data type, which handles non decimal numbers. Therefore, when I ran the code, the int32_t gave me the values 1000000, 1, and 1000001. However, the second data type, float, handles decimal values. When I listed the values of 1 million, 1, and 1 million and 1, C++ promoted those numbers to decimal numbers, adding a '.0' at the end. This led to the compiler giving me the results 1000000.0, 1.0, and 1000001.0.

Exercise 3: Integer Division

I have provided skeleton code that shows the results of integer division for every combination of numerator and denominator in the range [1,10]. Compile and run the code, and explain the results.

The result table mimics a division table, in which by choosing a number from the rightmost column to divide with the top row numbers, the corresponding result is the value listed in the table itself. For example, choosing +10 from the right column and dividing it with +2 from the top row, the corresponding table value is +5.

Extend the code to work for all combinations of numerator and denominator in the range [-10,+10]. Compile and run the code, and explain the results.

Similar to the [0,+10] range, the table has simply been extended to work for the range [-10,+10]. Aside from the issues resulted from working in integer instead of float, the rest of the math works fine, such as ensuring that the negative and positive signs are correct for the results.

Note any interesting observations or issues.

The biggest issue of this code is that since it handles only integers, dividing values that result in a value less than 1 produces an output of 0, and dividing values that should result in a decimal number are truncated into a whole number and produces an inaccurate result. For example, 1/9 should produce 0.11, however, since we did not use float (but rather int), C++ simply does not deal with anything past the decimal point, and just gives us an output of zero. Another example, 8/7 should give us 1.14, but we are only given 1 as the answer.

Exercise 4: Floating-Point Division

I have provided skeleton code that shows the results of integer division for every combination of (integer) numerator and denominator in the range [1,10]. Compile and run the code, and explain the results.

The result is the same as Exercise 3, except that this time, we have used float and the values inside the table are mathematically accurate.

Extend the code to work for all combinations of numerator and denominator in the range [-10,+10]. Compile and run the code, and explain the results.

Similar to above and Exercise 3, the range has now been updated to [-10,+10], giving us the negative number possibilities from the positive ones.

Note any interesting observations or issues.

While not exactly an issue, the code seems to not read in a very friendly way, potentially causing some confusion. For example, 1 divided by 2 is 0.5. The code gives us the result +5.000e-01, the e-01 indicates the position/exponent of the +5, which is going to be in the exponent position -1 (and thus 0.5). Therefore, the answer is correct, but it isn't exactly user friendly. An issue that stems from this is that the formatting is all off due to the amount of characters needed for just one answer.

Exercise 5: Building Values From Bits

Explain the resulting outputs.

- We set x equal to 2693408940
- To toggle bit 3, we perform a bitwise XOR operation (1 << 3)
- To toggle bit 3 again, we perform another bitwise XOR operation (1 << 3)
- To clear bit 7, we perform a bitwise AND operation ~(1 << 7)
- To clear bits 16-31 in one operation, we use a for loop and perform a bitwise AND operation ~(1 << i)
- The result is x = 10284