**Reflection of Workshop 3**

1. The code didn’t have any **syntactic** errors. It meant that I could run the program and the compiler didn’t throw an error. All the bugs were **semantic!** So I had to inspect the code line by line to find the errors. Of course, I had to run the code first to know that there is a problem in the code. When I ran it, it exited with a non-zero number and crashed. Took me a while to inspect and find out that the definition and implementation of the factorial function was invalid. The condition always came out true so it kept executing until running out of stack memory. Then after fixing that, I found out that reduceFactorial had a meaningless return value. (n/n) which always returned 1 for any non-zero numbers and zero for 0. To find the bugs I used the Visual Studio debugger. This approach allowed me to observe the behavior of the program. It made it easier to identify where the code deviated from its expected behavior. Compared to other techniques such as code review or adding print statements, using a debugger provides a more comprehensive and immediate understanding of the program's state and flow. It is particularly effective for detecting logical errors and understanding complex recursions or pointer manipulations. That’s why this is the most useful debugging technique compared to others.
2. **A)** **Largest integer and double values I can store-**

**Largest Integer**: in a 32-bit environment 2,147,483,647 (2^31 - 1).

**Largest Double**: The largest value that can be stored in a double

(IEEE 754 double- precision) is approximately 1.7976931348623157 ×

10^308.

**B)** **The reason for a limit on the maximum value I can store in a variable:**

There is a limit on the maximum value i can store in a variable because the

data type fixes the size of the variable. For integers, the limit is determined by

the number of bits allocated to store the value. For instance, a 32-bit integer can

hold 2^32 distinct values ranging from -2,147,483,648 to 2,147,483,647. For floating-

point numbers, the limit is defined by the IEEE 754 standard, which specifies how the

bits are used to represent the mantissa, exponent, and sign.

**C)** **what happens if I exceed the maximum value an integer can hold:**

An overflow occurs if i exceed the maximum value an integer can hold. In the case

of a 32-bit signed integer, exceeding 2,147,483,647 causes it to wrap around to the

minimum value of -2,147,483,648. This happens because integers are stored in a fixed

number of bits, and adding a value that exceeds the maximum representable value

causes the most significant bit to be incorrectly set, resulting in a wrap-around effect.

This caused the program to print weird values.

**D) The format for the storage of a floating-point variable :**

Floating-point variables are stored using the IEEE 754 standard, which divides the

storage into three parts: the sign bit, the exponent, and the mantissa (or significand).

For a 64-bit double, there is 1 bit for the sign, 11 bits for the exponent, and 52 bits

for the mantissa. This differs from integers, which are stored as a straightforward

binary representation of the number, with all bits directly contributing to the value of

the number. The floating-point format allows for a much wider range of values by

representing numbers in scientific notation, but it also introduces precision

limitations due to the finite number of bits used for the mantissa.

1. **Default stack size:** The default stack size for a program in Visual Studio is 1MB (MegaByte).

**Default Heap size:** The default heap size is much larger, starting at 1 MB but can grow as

needed, typically limited by the system's virtual memory.

I did not hit **the limits** during my code execution, but hitting the **stack or heap** limit is possible in other scenarios with **large recursive functions** or **extensive dynamic memory allocations**. Increasing the memory allocated to the program could fix the problem if the limit is the only issue causing the failure. However, it's important to be cautious with stack size as excessive recursion or deep function calls can lead to stack overflow, and simply increasing the stack size may only delay the issue without addressing the root cause.

Limits on stack and heap sizes are in place to prevent programs from **consuming excessive amounts of memory**, which could impact system performance or stability. They ensure that individual programs do not interfere with each other or the operating system by **monopolizing memory resources**.