**Numeric Overflow Coding**

Justin Starr

Department of STEM

CS 405 – Secure Coding

Professor Mike Alesso

July 7, 2024

**Numeric Overflow Coding**

Screenshot showing overflow tests:

A screenshot of a computer program

Description automatically generated

Screenshot showing underflow tests:

A screenshot of a computer program

Description automatically generated

**Process Summary**

The approach that was taken for completing the exercise started by evaluating the provided source code and understanding what needed to be modified to have the program execute as intended. Before I made any changes to the code, I ran the program to see how it executed and what the initial output was. This helped me to understand when an overflow or underflow was happening. Overflows occurred when in the add\_numbers function, the value of result after incrementing the number of steps passed in, was wrapped. This means that while incrementing, the value of result exceeded the maximum numerical value for the particular type of result, and the final value of result was then wrapped making the value of result less than the previous value of result during iteration. The same thing happened similarly in the subtract\_numbers function, however, the wrapping occurred when decrementing, meaning that the value of result exceeded the minimum numerical value for result’s type. When decrementing result, an underflow occurred when the value of result was wrapped, and its value was greater than the previous value of result.

When implementing code to detect when an overflow/underflow would occur, I started in the add\_numbers function and created a variable, nextResult, that would be used to calculate what the next value of result would be during each step of iteration. For each iteration or step, we calculate the value of nextResult by adding the value of increment. If nextResult’s value was less than the current value of result, we know an overflow occurred. If an overflow occurs, the result is thrown back to the test\_overflow function for error handling, and the overflow is prevented from happening. If no overflow occurs, the next value of result is calculated, and the process repeats until the number of steps has been completed, where result’s value is then returned if no overflow occurs. The same applied to the subtract\_numbers function, however the difference with this function is that we are detecting underflows, which occur when the value of nextResult is greater than the current value of result. Like the add\_numbers function, a nextResult variable was used to calculate the next value of result after decrementing during each iteration. If nextResult’s value during iteration was greater than result’s current value, we know an underflow occurred. If an underflow occurs, result is thrown back to the test\_underflow function for error handling, and the underflow is prevented from happening. If not, the next value of result is calculated, and the process repeats until the number of steps has been completed, and then result’s value is returned if no underflow occurs.

Once I had implemented my add\_numbers and subtract\_numbers functions correctly, in both the test\_overflow and test\_underflow functions, I implemented try/catch blocks to handle the errors when either an overflow or underflow occurred. Each of the methods contained two try/catch blocks, one for testing without overflow and underflow, and one for testing with overflow and underflow. In test\_overflow, the first try/catch block tries to calculate the value of result, without overflow, by calling add\_numbers and passing the value of start, increment, and steps. We expect this first add\_numbers call to have no errors thrown because the passed-in number of steps will cause no numerical overflow. However, the catch is in place to handle any overflow errors that could potentially be encountered. If result is not thrown in add\_numbers, the returned value of result is displayed to the user. The second try/catch block tries to calculate the value of result, with the possibility of overflow occurring because the passed-in number of steps to the add\_numbers function is increased by one. This time, for almost every variable type we can expect an error to be thrown back to the test\_overflow function because the value of increment is set to the type of variables maximum value divided by 5. Since the passed-in number of steps is 6 (steps + 1 = 6), we know that an overflow will occur in almost every case. When the overflows occur, the catch receives the thrown result and displays an error message to the user. The test\_underflow function was implemented in the same manner as the test\_overflow function, incorporating the use of try/catch blocks to handle errors when underflows occur, otherwise outputting the value of result each time subtract\_numbers is called for both with and without underflows. Also, in all of the try/catch blocks, I was able to use the “(…)” syntax in the catch because result’s type changes each time test\_overflow and test\_underflow are called. Therefore, anytime result was thrown, the test functions would know when either an overflow or underflow occurred.

The biggest issue I encountered while working on this activity was that I had never worked with C++ templates before, so I had to learn about how templates work and how they can operate with any data type. Once I understood this, I realized how the functions could work the same for any data type that was used and that all I really needed to do was incorporate logic for determining when either an overflow or underflow happened and how to handle these errors. Following the instructions provided in the commented code was beneficial to ensuring my code was functioning properly. Also, while testing my code, I incorporated many console output statements to see what the values of result and nextResult were so that I could ensure that overflow and underflow error handling was happening when it should. I then removed these statements once I had successfully debugged my code.