**CS 405 Module One Numeric Overflow – Summary**

For this project, the goal was to create a C++ program that can detect when numbers become too big (overflow) or too small (underflow) for their data type. Overflow happens when a number exceeds the maximum value a data type can hold, and underflow occurs when it goes below the minimum value. To handle this, I added some logic to two functions, add\_numbers() and subtract\_numbers(), to catch these situations and stop the program from making errors.

Screenshot of successful build:

A computer screen shot of a program

Description automatically generated

Screenshot of successful run process:

A screenshot of a computer program

Description automatically generated

**Changes Made**

* **Detecting Overflow in add\_numbers()**:  
  The add\_numbers() function starts with a number and keeps adding an increment to it. However, before each addition, the code checks if adding the next number will go past the data type's limit. This is done using std::numeric\_limits<T>::max(), which gives us the maximum value for each type (like int or float).

If overflow is about to happen, the program prints **"Overflow detected!"** and returns the max value instead of continuing.

**Code Used**:

if (std::numeric\_limits<T>::max() - result < increment) {

std::cout << "Overflow detected!" << std::endl;

return std::numeric\_limits<T>::max();

}

**Detecting Underflow in subtract\_numbers()**:

In a similar way, subtract\_numbers() checks if subtracting a value will make the number drop below the minimum value allowed for its data type, using std::numeric\_limits<T>::min().

If underflow is about to happen, it prints **"Underflow detected!"** and returns the minimum value instead.

**Code Used**:

if (result - std::numeric\_limits<T>::min() < decrement) {

std::cout << "Underflow detected!" << std::endl;

return std::numeric\_limits<T>::min();

}

**Handling Results and Letting the User Know**:

Both the add\_numbers() and subtract\_numbers() functions return either the correct result or the max/min value if overflow or underflow happens. Then, in the test\_overflow() and test\_underflow() functions, I added logic to print messages to the user, telling them whether the addition or subtraction succeeded or failed.

**Code Used**:

if (result == std::numeric\_limits<T>::max()) {

std::cout << "\tOverflow occurred!" << std::endl;

}

else {

std::cout << +result << std::endl;

}

**Testing Different Data Types**:

I also tested the program on different data types, including char, int, float, double, and more. For each type, I tested normal operations that stay within the limits and forced operations that cause overflow and underflow.

For example, when testing with int, I made sure that adding too many large numbers would trigger overflow, and the same for subtraction to test underflow.

**Code Used for Testing Overflow with int**:

test\_overflow<int>();

**Results and Console Output**:

After each test, the program prints the results to the console. It shows either the correct result when there’s no overflow or underflow, or prints **"Overflow detected!"** or **"Underflow detected!"** when something goes wrong.

**Output for an int Overflow**:

Overflow Test of Type = int

Adding Numbers Without Overflow (0, 429496729, 5) = 2147483645

Adding Numbers With Overflow (0, 429496729, 6) = Overflow detected!

Overflow occurred!

**Challenges:**

One of the challenges I faced was making sure the program worked for different data types like int, float, and char. Each type has a different range of values it can handle. I solved this by using std::numeric\_limits<T>, which automatically adjusts based on the data type being used, allowing the same logic to work for all of them.

**Output Screenshots:**

**Screenshot 1: Overflow Tests**

This shows the program detecting overflow across different data types:

* char, wchar\_t, short, int, long, and \_\_int64: The program adds numbers successfully up to 5 iterations, but when it tries a 6th, it detects overflow.
* unsigned char, unsigned short, unsigned int, unsigned long, and unsigned \_\_int64: Similar to the signed types, overflow is detected when the numbers exceed the maximum value.
* float, double, and long double: For floating-point numbers, the program detects overflow once the values get too large.

A screenshot of a computer program

Description automatically generated

**Screenshot 2: Underflow Tests Begin**

This starts the **underflow tests**:

* **char, wchar\_t, short, int, long, and \_\_int64**: The program subtracts values successfully, but detects underflow when an extra iteration pushes the number below the minimum limit.

A screenshot of a computer program

Description automatically generated

**Screenshot 3: More Underflow Tests**

Continues with **underflow** on unsigned types and floating-point numbers:

* **unsigned char, unsigned short, unsigned int, unsigned long**: Underflow is detected when subtracting values causes the number to drop below zero (unsigned types can't hold negative values).
* **float, double, and long double**: The program detects underflow when the values drop below the minimum for floating-point types.

A screenshot of a computer program

Description automatically generated

**Screenshot 4: Final Underflow Tests and Completion**

Shows the final underflow tests and program completion:

* **unsigned long, unsigned \_\_int64**: Same as other unsigned types, underflow is detected when values go below zero.
* **float, double, and long double**: Underflow is detected in floating-point types.
* The program successfully completes all overflow and underflow tests, printing that all tests are complete.

A screenshot of a computer program

Description automatically generated