San Francisco Bay University

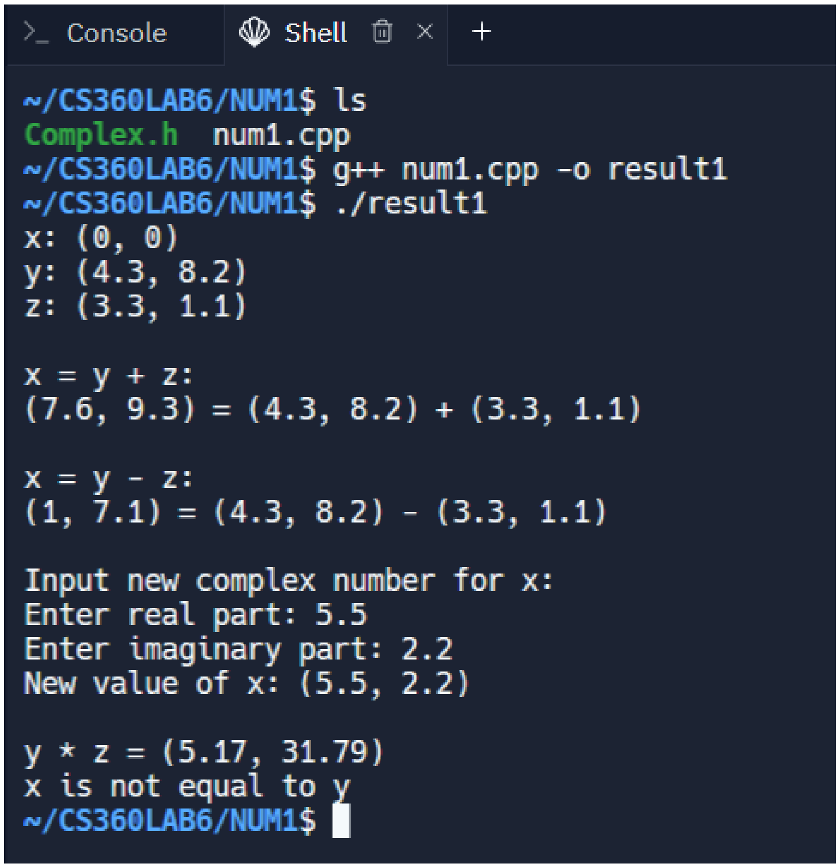
CS360L - Programming in C and C++ Lab
Lab Assignment #6

Due day: 4/13/2024

Kritika Regmi
ID: 19702

1. Consider the class *Complex* shown as follows. The class enables operations on so-called *complex numbers*. These are numbers of the form *realPart* + *imaginaryPart***i*, where *i* has the value $\sqrt{-1}$
   a. Modify the class to enable input and output of complex numbers via overloaded >> and << operators, respectively (you should remove the print function from the class).
   b. Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.
   c. Overload the == and != operators to allow comparisons of complex numbers.

ANS: **<u>OUTPUT:</u>**



```
~/CS360LAB6/NUM1$ ls
Complex.h  num1.cpp
~/CS360LAB6/NUM1$ g++ num1.cpp -o result1
~/CS360LAB6/NUM1$ ./result1
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)

Input new complex number for x:
Enter real part: 5.5
Enter imaginary part: 2.2
New value of x: (5.5, 2.2)

y * z = (5.17, 31.79)
x is not equal to y
~/CS360LAB6/NUM1$ 
```

2. A machine with *32-bit* integers can represent integers in the range of approximately -*2* billion to +*2* billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. Consider class *HugeInt* in the following program. Study the class carefully, then answer the following:

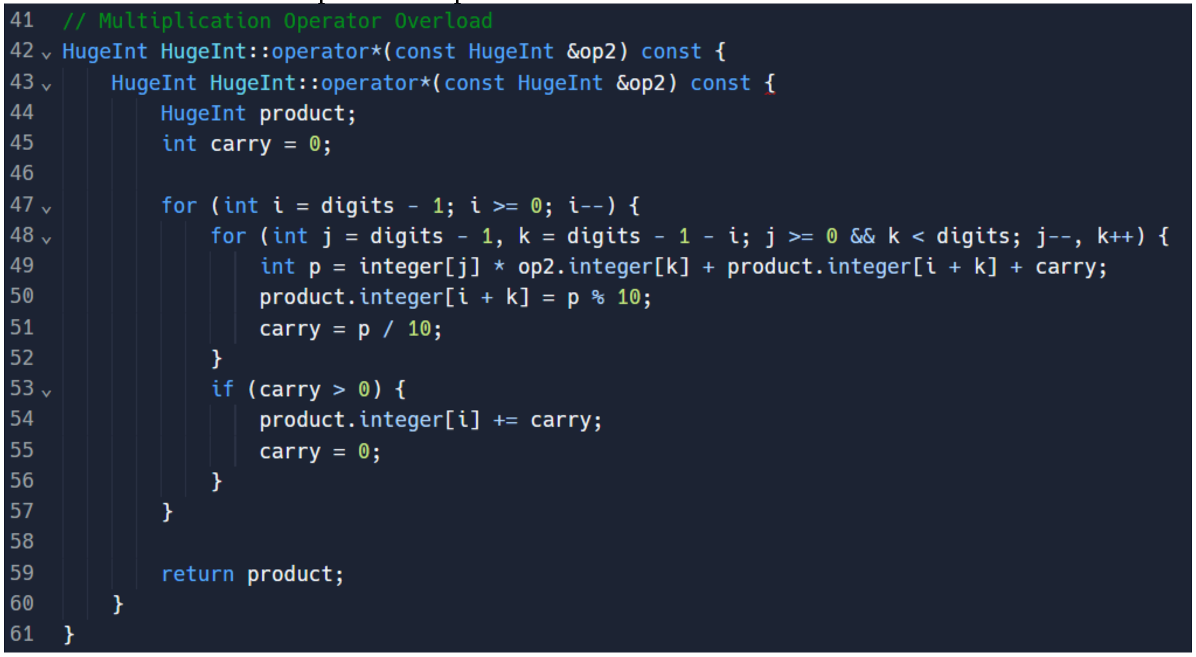
   a. Describe precisely how it operates.

   ANS: The HugeInt class has two **constructors**: one that takes a long integer and another that accepts a string. Both constructors initialize an array of 30 elements, where each element is a short integer representing a single digit of the number. The digits are stored in reverse order.
   The operator+ is overloaded to handle three types of addition: HugeInt with HugeInt, HugeInt with an int, and HugeInt with a string. In each case, the addition involves iterating over the digits from least to most significant, managing carryovers when the sum of digits exceeds 9.
   The operator<< is overloaded to facilitate the output of HugeInt objects. It skips leading zeros and prints the remaining digits.

   b. What restrictions does the class have?

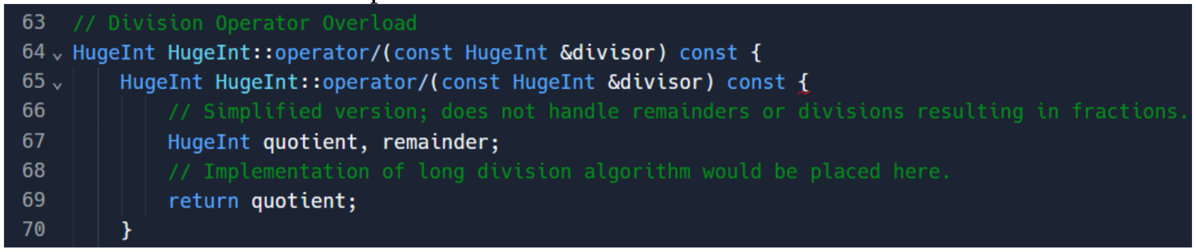
   ANS: The class can handle numbers up to 30 digits only. Any larger number will not be properly represented. There is no error handling for incorrect inputs or overflow beyond the 30 digits.
   Currently, the class only supports addition. It lacks functionality for subtraction, multiplication, division, and more complex mathematical or bitwise operations.

   c. Overload the * multiplication operator.



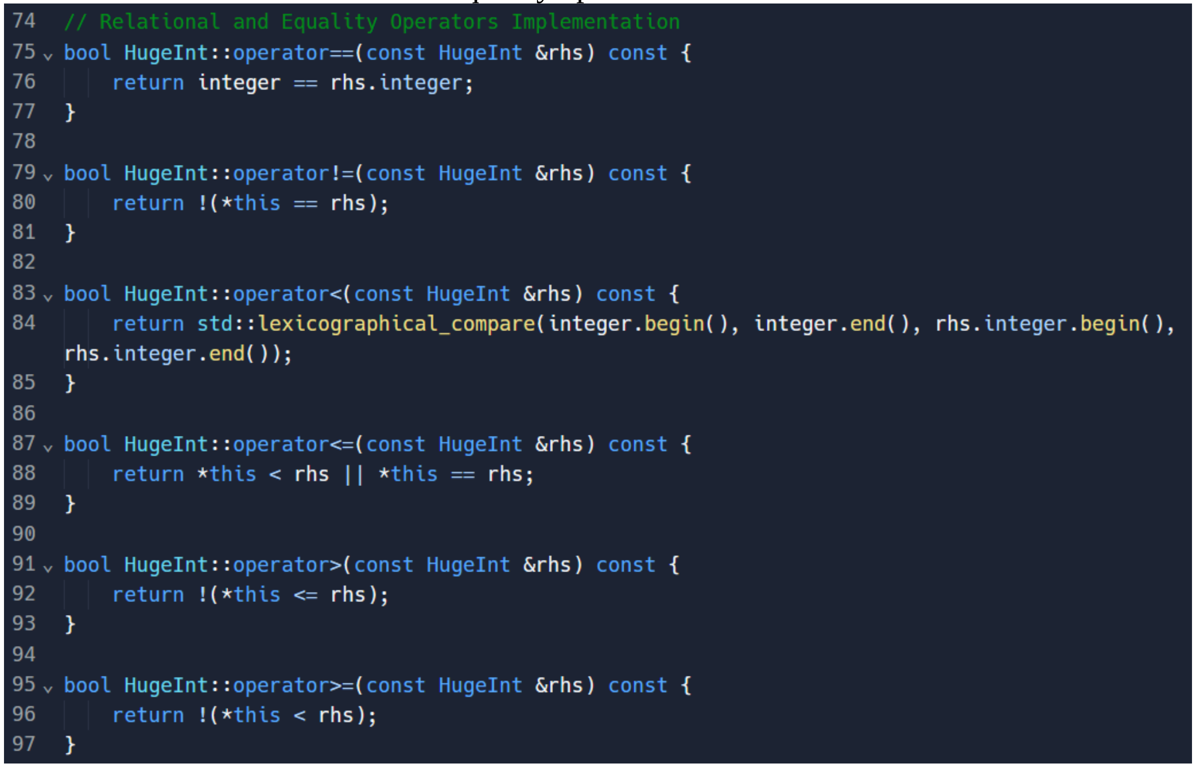
```cpp
// Multiplication Operator Overload
HugeInt HugeInt::operator*(const HugeInt &op2) const {
    HugeInt HugeInt::operator*(const HugeInt &op2) const {
        HugeInt product;
        int carry = 0;

        for (int i = digits - 1; i >= 0; i--) {
            for (int j = digits - 1, k = digits - 1 - i; j >= 0 && k < digits; j--, k++) {
                int p = integer[j] * op2.integer[k] + product.integer[i + k] + carry;
                product.integer[i + k] = p % 10;
                carry = p / 10;
            }
            if (carry > 0) {
                product.integer[i] += carry;
                carry = 0;
            }
        }

        return product;
    }
}
```

d. Overload the / division operator.



```cpp
// Division Operator Overload
HugeInt HugeInt::operator/(const HugeInt &divisor) const {
    HugeInt HugeInt::operator/(const HugeInt &divisor) const {
        // Simplified version; does not handle remainders or divisions resulting in fractions.
        HugeInt quotient, remainder;
        // Implementation of long division algorithm would be placed here.
        return quotient;
    }
```

e. Overload all the relational and equality operators.



```cpp
// Relational and Equality Operators Implementation
bool HugeInt::operator==(const HugeInt &rhs) const {
    return integer == rhs.integer;
}

bool HugeInt::operator!=(const HugeInt &rhs) const {
    return !(*this == rhs);
}

bool HugeInt::operator<(const HugeInt &rhs) const {
    return std::lexicographical_compare(integer.begin(), integer.end(), rhs.integer.begin(), rhs.integer.end());
}

bool HugeInt::operator<=(const HugeInt &rhs) const {
    return *this < rhs || *this == rhs;
}

bool HugeInt::operator>(const HugeInt &rhs) const {
    return !(*this <= rhs);
}

bool HugeInt::operator>=(const HugeInt &rhs) const {
    return !(*this < rhs);
}
```