



## San Francisco Bay University

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

Due day: 4/13/2024

#### Instruction:

1. Push the answer sheets/source code to Github
2. Please follow the code style rule like programs on handout.
3. Overdue lab assignment submission can't be accepted.
4. Take academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)

REPLIT LINK: <https://replit.com/@TASMITA-TANJIMT/CSLAB01>

GUTHUB LINK: <https://github.com/tasmita0131/CSLAB01>

1. Consider 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.

#### CODE:

##### Complex.h file:

```
#ifndef COMPLEX_H
#define COMPLEX_H

#include <iostream>

using namespace std;

class Complex {
public:
    explicit Complex(double = 0.0, double = 0.0); // constructor
    Complex operator+(const Complex &) const; // addition
    Complex operator-(const Complex &) const; // subtraction
```

```

Complex operator*(const Complex &) const; // multiplication
bool operator==(const Complex &) const; // equal to
bool operator!=(const Complex &) const; // not equal to

// Overloaded I/O operators as friends
friend ostream &operator<<(ostream &, const Complex &);
friend istream &operator>>(istream &, Complex &);

private:
    double real; // real part
    double imaginary; // imaginary part
}; // end class Complex

#endif

Complex.cpp file:

#include "complex.h"

// Constructor
Complex::Complex(double realPart, double imaginaryPart)
    : real(realPart), imaginary(imaginaryPart) {
    // empty body
}

// Addition operator
Complex Complex::operator+(const Complex &operand2) const {
    return Complex(real + operand2.real, imaginary + operand2.imaginary);
}

// Subtraction operator
Complex Complex::operator-(const Complex &operand2) const {
    return Complex(real - operand2.real, imaginary - operand2.imaginary);
}

// Multiplication operator
Complex Complex::operator*(const Complex &operand2) const {
    return Complex(real * operand2.real - imaginary * operand2.imaginary,
        real * operand2.imaginary + imaginary * operand2.real);
}

// Equality operator
bool Complex::operator==(const Complex &operand2) const {
    return real == operand2.real && imaginary == operand2.imaginary;
}

// Inequality operator

```

```

bool Complex::operator!=(const Complex &operand2) const {
    return !(*this == operand2);
}

// Overloaded output operator
ostream &operator<<(ostream &output, const Complex &number) {
    output << '(' << number.real << ", " << number.imaginary << ')';
    return output;
}

// Overloaded input operator
istream &operator>>(istream &input, Complex &number) {
    char ignoreChar;
    // Assume the input is in the form (a, b)
    input >> ignoreChar >> number.real >> ignoreChar >> number.imaginary >>
        ignoreChar;
    return input;
}

```

### **Main.cpp file:**

```

#include "complex.h"
#include <iostream>

using namespace std;

int main() {
    Complex x;
    Complex y(4.3, 8.2);
    Complex z(3.3, 1.1);

    cout << "Enter a complex number in the form (a, b): ";
    cin >> x; // using the overloaded >> operator
    cout << "x: " << x << "\ny: " << y << "\nz: " << z << endl;

    x = y + z;
    cout << "\nx = y + z:\n" << x << " = " << y << " + " << z << endl;

    x = y - z;
    cout << "\nx = y - z:\n" << x << " = " << y << " - " << z << endl;

    // Demonstrate multiplication
    Complex w;
    w = y * z;
    cout << "\nw = y * z:\n" << w << " = " << y << " * " << z << endl;

    // Demonstrate equality and inequality

```

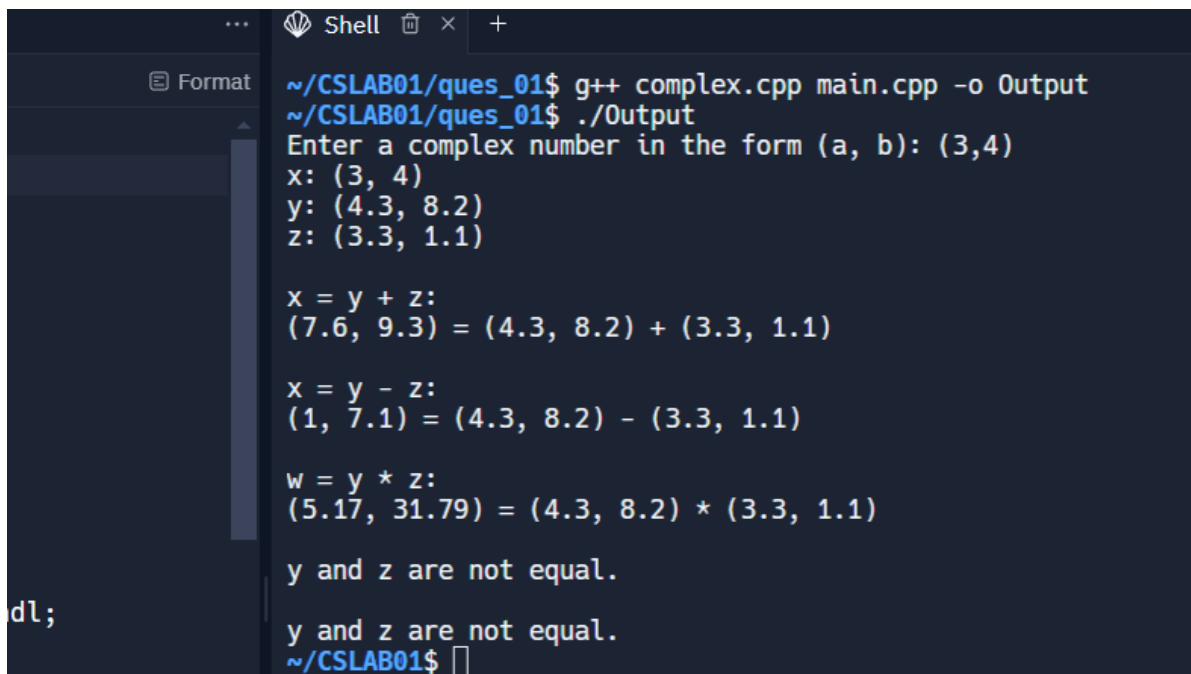
```

if (y == z)
    cout << "\ny and z are equal.\n";
else
    cout << "\ny and z are not equal.\n";

if (y != z)
    cout << "\ny and z are not equal.\n";
else
    cout << "\ny and z are equal.\n";

return 0;
}

```



The screenshot shows a terminal window with the following content:

```

~/CSLAB01/ques_01$ g++ complex.cpp main.cpp -o Output
~/CSLAB01/ques_01$ ./Output
Enter a complex number in the form (a, b): (3,4)
x: (3, 4)
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)

w = y * z:
(5.17, 31.79) = (4.3, 8.2) * (3.3, 1.1)

y and z are not equal.

y and z are not equal.
~/CSLAB01$

```

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.  
How the HugeInt Class Operates:

The HugeInt class is designed to manage very large integers by storing each digit of the integer in an array. This allows the class to handle operations on integers that far exceed the size limitations of standard integer types in C++. Here's how it operates:

Data Structure:

It uses an `std::array<short, 30>` to store the digits of the large integer. Each element of the array represents a single digit. This array can store integers up to 30 digits long.

Constructors:

`HugeInt(long = 0)`: Converts a long integer into a HugeInt object by placing each digit of the integer into the integer array from right to left.

`HugeInt(const std::string &)`: Converts a string that represents a large integer into a HugeInt object by placing each digit of the string into the integer array from right to left, provided that the characters are digits.

Addition Operators:

`operator+(const HugeInt&)`: Implements addition of two HugeInt objects using standard digit-by-digit addition with carry.

`operator+(int)`: Facilitates adding an int to a HugeInt by first converting the int to a HugeInt.

`operator+(const std::string&)`: Enables the addition of a string representation of an integer to a HugeInt by converting the string to a HugeInt.

Output Operator:

`friend std::ostream& operator<<(...)`: Overloads the output stream operator to print HugeInt objects. It skips leading zeros and prints the digits.

**b. What restrictions does the class have?**

**Restrictions of the HugeInt Class:**

Fixed Length:

The class can only handle numbers up to 30 digits long. Numbers longer than this will not be stored correctly.

Non-Negative Integers:

The current implementation does not support negative numbers. All operations are assumed to be on non-negative integers.

Error Handling:

The class does not handle errors related to non-numeric input in its string constructor robustly. It assumes all characters in the input string are digits.

Limited Functionality:

Currently, it only supports addition and not subtraction, multiplication, division, or any comparison operations.

**c. Overload the \* multiplication operator.**

```
// Multiplication operator; HugeInt * HugeInt
HugeInt HugeInt::operator*(const HugeInt& op2) const {
    HugeInt temp;
    for (int i = 0; i < digits; ++i) {
        if (integer[i] == 0) continue;
        int carry = 0;
        for (int j = 0; j < digits - i; ++j) {
            if (op2.integer[j] == 0) continue;
            int product = integer[i] * op2.integer[j] + temp.integer[i + j] + carry;
            temp.integer[i + j] = product % 10;
            carry = product / 10;
        }
    }
    return temp;
}
```

**d. Overload the / division operator.**

```
// Division operator; HugeInt / int
HugeInt HugeInt::operator/(int divisor) const {
    HugeInt result;
    int idx = 0;
    int temp = integer[digits - 1];
    while (temp < divisor && idx < digits - 1) {
        temp = temp * 10 + integer[digits - 2 - idx++];
    }
    while (idx < digits) {
        result.integer[digits - 1 - idx] = temp / divisor;
        temp = (temp % divisor) * 10 + integer[digits - 2 - idx++];
    }
    return result;
}
```

**e. Overload all the relational and equality operators.**

```
// Equality operator
bool HugeInt::operator==(const HugeInt& rhs) const {
    for (int i = 0; i < digits; ++i) {
        if (integer[i] != rhs.integer[i])
```

```

    return false;
}
    return true;
}

```

### **ENTIRE CODE:**

```

#ifndef HUGEINT_H
#define HUGEINT_H

#include <array>
#include <iostream>
#include <string>
class HugeInt {
    friend std::ostream& operator<<(std::ostream&, const HugeInt&);
    friend std::istream& operator>>(std::istream&, HugeInt&);
public:
    static const int digits = 30; // maximum digits in a HugeInt
    HugeInt(long = 0); // conversion/default constructor
    HugeInt(const std::string&); // conversion constructor from string
    // Arithmetic operators
    HugeInt operator+(const HugeInt&) const;
    HugeInt operator+(int) const;
    HugeInt operator+(const std::string&) const;
    HugeInt operator*(const HugeInt&) const;
    HugeInt operator/(int) const;
    // Relational and equality operators
    bool operator==(const HugeInt&) const;
    bool operator!=(const HugeInt&) const;
    bool operator<(const HugeInt&) const;
    bool operator<=(const HugeInt&) const;
    bool operator>(const HugeInt&) const;
    bool operator>=(const HugeInt&) const;
private:
    std::array<short, digits> integer; // stores the digits of the number
};
#endif
#include "Hugeint.h"
#include <cctype>
#include <cmath>
using namespace std;
// Constructor from long
HugeInt::HugeInt(long value) {
    integer.fill(0);
    for (int i = digits - 1; value != 0 && i >= 0; --i) {
        integer[i] = value % 10;
        value /= 10;
    }
}
// Constructor from string

```

```

HugeInt::HugeInt(const string& number) {
    integer.fill(0);
    int length = number.size();
    for (int i = digits - length, j = 0; i < digits; ++i, ++j) {
        if (isdigit(number[j]))
            integer[i] = number[j] - '0';
    }
}

// Addition operator: HugeInt + HugeInt
HugeInt HugeInt::operator+(const HugeInt& op2) const {
    HugeInt temp;
    int carry = 0;
    for (int i = digits - 1; i >= 0; --i) {
        int sum = integer[i] + op2.integer[i] + carry;
        temp.integer[i] = sum % 10;
        carry = sum / 10;
    }
    return temp;
}

// Addition operator: HugeInt + int
HugeInt HugeInt::operator+(int op2) const {
    return *this + HugeInt(op2);
}

// Addition operator: HugeInt + string
HugeInt HugeInt::operator+(const string& op2) const {
    return *this + HugeInt(op2);
}

// Multiplication operator
HugeInt HugeInt::operator*(const HugeInt& op2) const {
    HugeInt result;
    for (int i = 0; i < digits; ++i) {
        if (integer[i] == 0) continue;
        int carry = 0;
        for (int j = 0, k = digits - 1 - i; j < digits - i; ++j, ++k) {
            int product = integer[digits - 1 - i] * op2.integer[digits - 1 - j] + result.integer[k] + carry;
            result.integer[k] = product % 10;
            carry = product / 10;
        }
    }
    return result;
}

// Division operator
HugeInt HugeInt::operator/(int divisor) const {
    HugeInt result;
    int idx = 0;
    int remainder = 0;
    for (int i = 0; i < digits; ++i) {
        int current = remainder * 10 + integer[i];
        result.integer[i] = current / divisor;
        remainder = current % divisor;
    }
    return result;
}

```



```
// Relational and equality operators
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);
}
// Stream insertion operator
ostream& operator<<(ostream& out, const HugeInt& num) {
    int i = 0;
    while (i < HugeInt::digits && num.integer[i] == 0) ++i;
    if (i == HugeInt::digits)
        out << 0;
    else
        for (; i < HugeInt::digits; ++i)
            out << num.integer[i];
    return out;
}
// Stream extraction operator
istream& operator>>(istream& in, HugeInt& num) {
    string input;
    in >> input;
    num = HugeInt(input);
    return in;
}
#include <iostream>
// #include "Hugeint.h"
using namespace std;
int main() {
    HugeInt n1(7654321);
    HugeInt n2(7891234);
    HugeInt n3("99999999999999999999999999999999");
    HugeInt n4("1");
    HugeInt result;
    cout << "n1 is " << n1 << "\nn2 is " << n2
    << "\nn3 is " << n3 << "\nn4 is " << n4 << "\n\n";
    result = n1 + n2;
    cout << n1 << " + " << n2 << " = " << result << "\n\n";
    result = n3 + n4;
    cout << n3 << " + " << n4 << " = " << result << "\n\n";
```

```
geInt&);
);
geInt
from string
```

Generate