San Francisco Bay University

CS360L - Programming in C and C++ Lab
Lab Assignment #4

Due day: 3/22/2024

Tasmita Tanjim Tanha
Id: 19723
REPLIT LINK: https://replit.com/@TASMITA-TANJIMT/Lab42
GITHUB LINK: https://github.com/tasmita0131/Lab4_2_tasmita_tanjim_19723

1. Correct the errors in the following snippets and explain
   a. Assume the following prototype is declared in class *Time*:

```
void ~Time( int );
```

   b. Assume the following prototype is declared in class *Employee*:

```
int Employee( string, string );
```

   c. The following is a definition of class *Example*:

```
class Example{
  public:
    Example( int y = 10 ): data( y ){
    // empty body
    } // end Example constructor
    int getIncrementedData() const{
      return ++data;
    } // end function getIncrementedData
    static int getCount(){
      cout << "Data is " << data << endl;
      return count;
    } // end function getCount
  private:
    int data;
    static int count;
}; // end class Example
```

CORRECTION WITH EXPLANATION:

a.

Incorrect Destructor Declaration in Class 'Time' because Destructors do not take parameters and do not return a value, so specifying void is incorrect. The corrected one should have been **~Time().** The most accurate method of declaring a destructor is without any return type and parameters.

b.

There was an incorrect Constructor Declaration in Class ‘Employee’. It should have been “ int Employee(string, string) ” because constructors do not have a return type, not even int. However, there is not a syntax error, it's just unclear.

Corrected: **Employee(string firstName, string lastName);**

c.

There are Issues in the ‘Class’ Example. Problems are given below:

- Member functions declared as const should not modify any data members.
- Static member functions cannot access non-static data members directly.

A corrected version of the code:

```
class Example {
public:
   Example(int y = 10): data(y) {
   }

   int getIncrementedData() const {
      return data + 1;
   }

   static int getCount() {
      cout << "Count is " << count << endl;
      return count;
   }

private:
   int data;
   static int count;
};
```

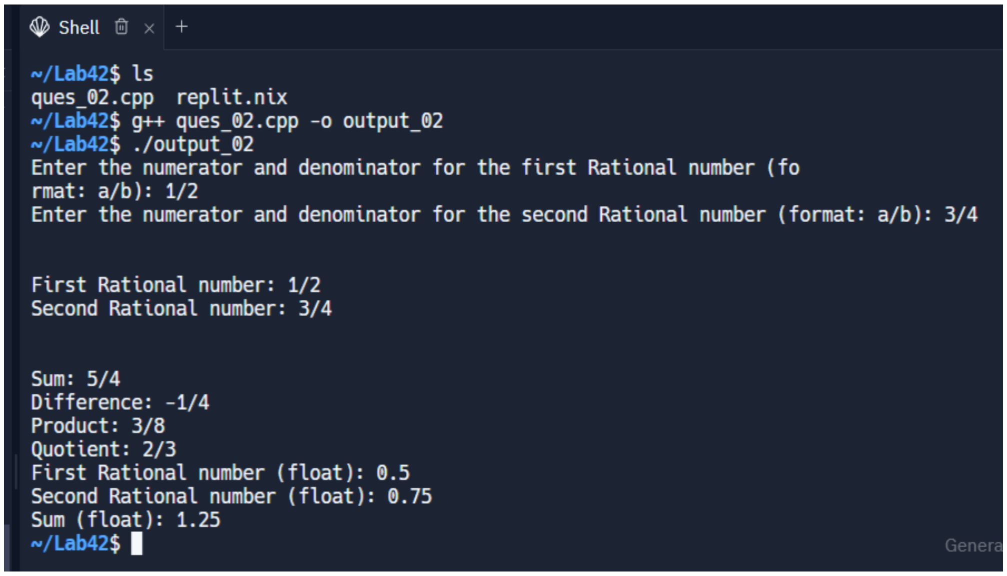
2. Generate a class called *Rational* to perform arithmetic with fractions. Write a program to test your class. Use integer variables to represent the private data of the class--the *numerator* and the *denominator*. Provide a constructor that enables an object of this class to be initialized when it’s declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, $\frac{2}{4}$, the fraction would be stored in the object as *1* in the numerator and *2* in the denominator. Provide public member functions that perform each of the following tasks:

a. Adding two *Rational* numbers. The result should be stored in reduced form.
b. Subtracting two *Rational* numbers. The result should be stored in reduced form.
c. Multiplying two *Rational* numbers. The result should be stored in reduced form.
d. Dividing two *Rational* numbers. The result should be stored in reduced form.
e. Printing *Rational* numbers in the form *a/b*, where *a* is the numerator and *b* is the denominator.
f. Printing *Rational* numbers in floating-point format.

Code after fulfilling all the given requirements:

```
#include <iostream>
using namespace std;

class Rational {
private:
    int numerator;
    int denominator;

    void reduce() {
        int gcd = greatestCommonDivisor(abs(numerator), abs(denominator)); // Ensure
positive GCD.
        numerator /= gcd;
        denominator /= gcd;
    }

    static int greatestCommonDivisor(int a, int b) {
        // Recursive function to find GCD using Euclid's algorithm.
        return b == 0 ? a : greatestCommonDivisor(b, a % b);
    }

public:
    // Constructor with default values.
    Rational(int num = 0, int denom = 1) : numerator(num), denominator(denom) {
        if (denominator == 0) {
            cout << "Invalid fraction: Denominator cannot be 0. Defaulting to 0/1." << endl;
            numerator = 0;
            denominator = 1;
        } else {
            reduce(); // Reduce the fraction upon initialization.
        }
    }

    // Methods for arithmetic operations
    Rational add(const Rational& rhs) const {
        return Rational(numerator * rhs.denominator + rhs.numerator * denominator,
denominator * rhs.denominator);
    }

    Rational subtract(const Rational& rhs) const {
        return Rational(numerator * rhs.denominator - rhs.numerator * denominator,
denominator * rhs.denominator);
    }

    Rational multiply(const Rational& rhs) const {
        return Rational(numerator * rhs.numerator, denominator * rhs.denominator);
    }

    Rational divide(const Rational& rhs) const {
        return Rational(numerator * rhs.denominator, denominator * rhs.numerator);
    }

    // Display functions
    void printFraction() const {
        cout << numerator << "/" << denominator << endl;
    }

    void printFloat() const {
        cout << static_cast<double>(numerator) / denominator << endl;
    }
};



int main() {
    int num1, denom1, num2, denom2;
    char slash; // For input format

    // User input for the first Rational number
    cout << "Enter the numerator and denominator for the first Rational number (format: a/b):
";
    cin >> num1 >> slash >> denom1;

    // User input for the second Rational number
    cout << "Enter the numerator and denominator for the second Rational number (format:
a/b): ";
    cin >> num2 >> slash >> denom2;

    Rational frac1(num1, denom1);
    Rational frac2(num2, denom2);

    // Display entered Rational numbers
    cout << "First Rational number: ";
    frac1.printFraction();
    cout << "Second Rational number: ";
    frac2.printFraction();

    // Performing operations
    Rational sum = frac1.add(frac2);
    Rational difference = frac1.subtract(frac2);
    Rational product = frac1.multiply(frac2);
    Rational quotient = frac1.divide(frac2);

    // Displaying results
    cout << "Sum: ";
    sum.printFraction();
    cout << "Difference: ";
    difference.printFraction();
    cout << "Product: ";
    product.printFraction();
    cout << "Quotient: ";
    quotient.printFraction();

    // Floating-point representation
    cout << "First Rational number (float): ";
    frac1.printFloat();
    cout << "Second Rational number (float): ";
    frac2.printFloat();
    cout << "Sum (float): ";
    sum.printFloat();
    return 0;
}
```

OUTPUT:



```
~/Lab42$ ls
ques_02.cpp  replit.nix
~/Lab42$ g++ ques_02.cpp -o output_02
~/Lab42$ ./output_02
Enter the numerator and denominator for the first Rational number (fo
rmat: a/b): 1/2
Enter the numerator and denominator for the second Rational number (format: a/b): 3/4


First Rational number: 1/2
Second Rational number: 3/4


Sum: 5/4
Difference: -1/4
Product: 3/8
Quotient: 2/3
First Rational number (float): 0.5
Second Rational number (float): 0.75
Sum (float): 1.25
~/Lab42$
```

3. Create a class HugeInteger that uses a 40-element array of digits to store integers as large as 40 digits each. Provide member functions input, output, add and subtract.For comparing HugeInteger objects, provide functions isEqualTo, isNotEqualTo, isGreaterThan, isLessThan, isGreaterThanOrEqualTo and isLessThanOrEqualTo--each of these is a "predicate " function that simply returns true if the relationship holds between the two HugeIntegers and returns false if the relationship does not hold. Also, provide a predicate function isZero. After that, provide member functions multiply, divide and modulus.

<u>CODE: </u>There are 3 files for this code: HugeInteger.h file, HugeInteger.cpp file, main.cpp file. All of these you can find inside the ques_03 folder.

<u>HugeInteger.h:</u>

```
#ifndef HUGEINTEGER_H
#define HUGEINTEGER_H
#include <string>

class HugeInteger {
private:
  int digits[40];

public:
  HugeInteger();
  void input(const std::string &number);
  void output() const;
  HugeInteger add(const HugeInteger &other) const;
  HugeInteger subtract(const HugeInteger &other) const;
  bool isEqualTo(const HugeInteger &other) const;
  bool isNotEqualTo(const HugeInteger &other) const;
  bool isGreaterThan(const HugeInteger &other) const;
  bool isLessThan(const HugeInteger &other) const;
  bool isGreaterThanOrEqualTo(const HugeInteger &other) const;
  bool isLessThanOrEqualTo(const HugeInteger &other) const;
  bool isZero() const;
};

#endif
```

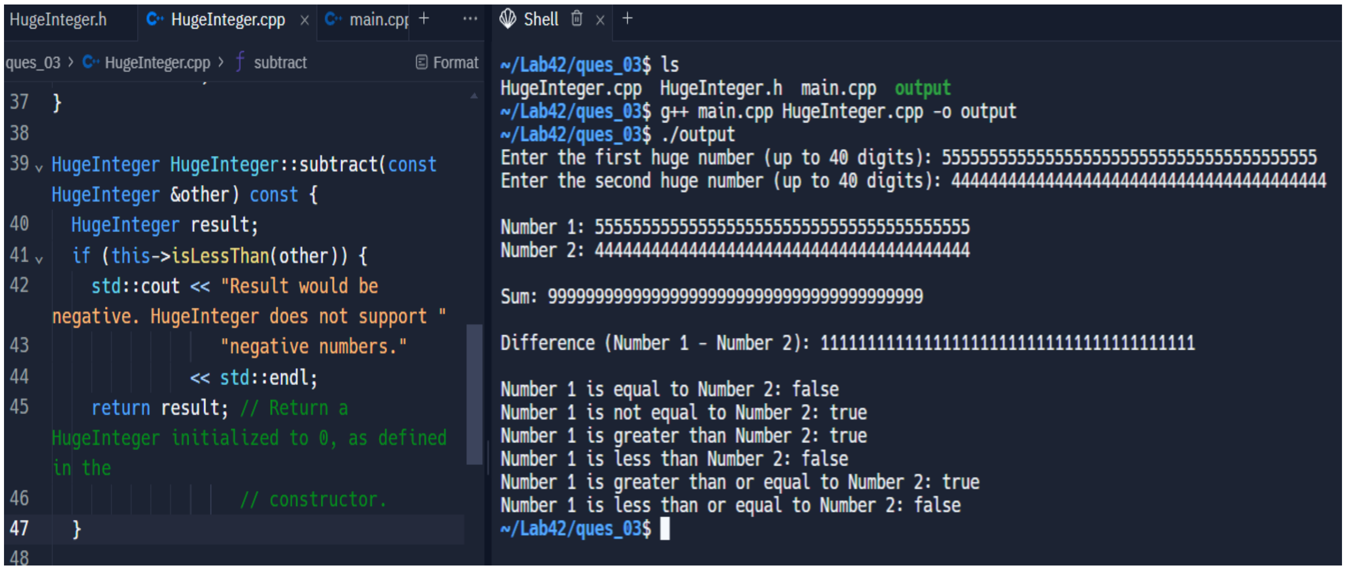
<u>HugeInteger.cpp:</u>

```
#include "HugeInteger.h"
#include <cstring>  // For memset
#include <iostream> // For output

HugeInteger::HugeInteger() { memset(digits, 0, sizeof(digits)); }

void HugeInteger::input(const std::string &number) {
  int length = number.size();
  int start = 40 - length;
  for (int i = 0; i < length; ++i) {
    digits[start + i] = number[i] - '0';
  }
}

void HugeInteger::output() const {
  bool foundNonZero = false;
  for (int i = 0; i < 40; ++i) {
    if (digits[i] != 0)
      foundNonZero = true;
    if (foundNonZero)
      std::cout << digits[i];
  }
  if (!foundNonZero)
    std::cout << '0'; // Handles case where all digits are zero
  std::cout << '\n';
}

HugeInteger HugeInteger::add(const HugeInteger &other) const {
  HugeInteger result;
  int carry = 0;
  for (int i = 39; i >= 0; --i) {
    int sum = digits[i] + other.digits[i] + carry;
    result.digits[i] = sum % 10;
    carry = sum / 10;
  }
  return result;
}

HugeInteger HugeInteger::subtract(const HugeInteger &other) const {
  HugeInteger result;
  if (this->isLessThan(other)) {
    std::cout << "Result would be negative. HugeInteger does not support "
                 "negative numbers."
              << std::endl;
    return result; // Return a HugeInteger initialized to 0, as defined in the
                   // constructor.
  }

  int borrow = 0;
  for (int i = 39; i >= 0; --i) {
    int diff = digits[i] - other.digits[i] - borrow;
    if (diff < 0) {
      diff += 10;
      borrow = 1;
    } else {
      borrow = 0;
    }
    result.digits[i] = diff;
  }
  return result;
}

bool HugeInteger::isEqualTo(const HugeInteger &other) const {
  for (int i = 0; i < 40; ++i) {
    if (digits[i] != other.digits[i])
      return false;
  }
  return true;
}

bool HugeInteger::isNotEqualTo(const HugeInteger &other) const {
  return !isEqualTo(other);
}

bool HugeInteger::isGreaterThan(const HugeInteger &other) const {
  for (int i = 0; i < 40; ++i) {
    if (digits[i] > other.digits[i])
      return true;
    else if (digits[i] < other.digits[i])
      return false;
  }
  return false;
}

bool HugeInteger::isLessThan(const HugeInteger &other) const {
  return !isGreaterThanOrEqualTo(other);
}

bool HugeInteger::isGreaterThanOrEqualTo(const HugeInteger &other) const {
  return isGreaterThan(other) || isEqualTo(other);
}

bool HugeInteger::isLessThanOrEqualTo(const HugeInteger &other) const {
  return !isGreaterThan(other);
}

bool HugeInteger::isZero() const {
  for (int i = 0; i < 40; ++i) {
    if (digits[i] != 0)
      return false;
  }
  return true;
}
```

Main.cpp:

```
#include "HugeInteger.h"
#include <iostream>

int main() {
   HugeInteger num1;
   HugeInteger num2;

   // Input two HugeIntegers
   std::cout << "Enter the first huge number (up to 40 digits): ";
   std::string number1;
   std::cin >> number1;
   num1.input(number1);

   std::cout << "Enter the second huge number (up to 40 digits): ";
   std::string number2;
   std::cin >> number2;
   num2.input(number2);

   std::cout<<'\n';
   // Output the HugeIntegers
   std::cout << "Number 1: ";
   num1.output();
   std::cout << "Number 2: ";
   num2.output();
   std::cout << '\n';

   // Add the two HugeIntegers and output the result
   HugeInteger sum = num1.add(num2);
   std::cout << "Sum: ";
   sum.output();
   std::cout << '\n';

   // Subtract the two HugeIntegers and output the result
   HugeInteger difference = num1.subtract(num2);
   std::cout << "Difference (Number 1 - Number 2): ";
   difference.output();
   std::cout << '\n';

   // Compare the two HugeIntegers
   std::cout << "Number 1 is equal to Number 2: " << std::boolalpha <<
num1.isEqualTo(num2) << '\n';
   std::cout << "Number 1 is not equal to Number 2: " << num1.isNotEqualTo(num2) << '\n';
   std::cout << "Number 1 is greater than Number 2: " << num1.isGreaterThan(num2) <<
'\n';
   std::cout << "Number 1 is less than Number 2: " << num1.isLessThan(num2) << '\n';
```

```
    std::cout << "Number 1 is greater than or equal to Number 2: " <<
num1.isGreaterThanOrEqualTo(num2) << '\n';
    std::cout << "Number 1 is less than or equal to Number 2: " <<
num1.isLessThanOrEqualTo(num2) << '\n';

    return 0;
}
```



```
HugeInteger HugeInteger::subtract(const
HugeInteger &other) const {
  HugeInteger result;
  if (this->isLessThan(other)) {
    std::cout << "Result would be
negative. HugeInteger does not support "
                  "negative numbers."
                << std::endl;
    return result; // Return a
HugeInteger initialized to 0, as defined
in the
                      // constructor.
  }
```

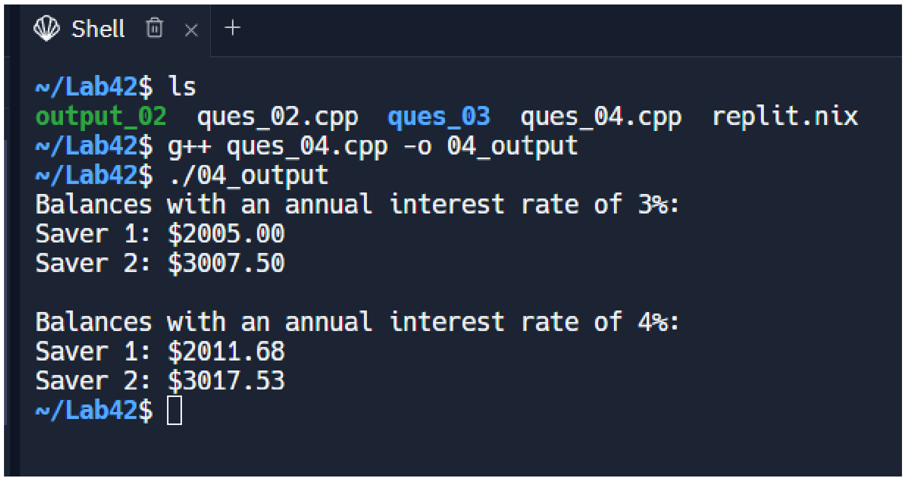
```
~/Lab42/ques_03$ ls
HugeInteger.cpp  HugeInteger.h  main.cpp  output
~/Lab42/ques_03$ g++ main.cpp HugeInteger.cpp -o output
~/Lab42/ques_03$ ./output
Enter the first huge number (up to 40 digits): 5555555555555555555555555555555555555555
Enter the second huge number (up to 40 digits): 4444444444444444444444444444444444444444

Number 1: 5555555555555555555555555555555555555555
Number 2: 4444444444444444444444444444444444444444

Sum: 9999999999999999999999999999999999999999

Difference (Number 1 - Number 2): 1111111111111111111111111111111111111111

Number 1 is equal to Number 2: false
Number 1 is not equal to Number 2: true
Number 1 is greater than Number 2: true
Number 1 is less than Number 2: false
Number 1 is greater than or equal to Number 2: true
Number 1 is less than or equal to Number 2: false
~/Lab42/ques_03$
```

4. Create a SavingsAccount class. Use a static data member annual-InterestRate to store the annual interest rate for each of the savers. Each member of the class contains a private data member savingsBalance indicating the amount the saver currently has on deposit. Provide member function calculateMonthlyInterest that calculates the monthly interest by multiplying the savingsBalance by annualInterestRate divided by 12; this interest should be added to savingsBalance. Provide a static member function modifyInterestRate that sets the static annualInterestRate to a new value. Write a driver program to test class SavingsAccount. Instantiate two different objects of class SavingsAccount, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set the annualInterestRate to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set the annualInterestRate to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

<u>CODE:</u>

```
#include <iomanip>
#include <iostream>
```

```
using namespace std;

class SavingsAccount {
private:
  double savingsBalance;

public:
  static double annualInterestRate;

  SavingsAccount(double balance) : savingsBalance(balance) {}

  void calculateMonthlyInterest() {
    double monthlyInterest = (savingsBalance * annualInterestRate) / 12;
    savingsBalance += monthlyInterest;
  }

  static void modifyInterestRate(double newInterestRate) {
    annualInterestRate =
        newInterestRate / 100.0; // Convert percentage to decimal
  }

  double getBalance() const { return savingsBalance; }
};

// Initialize static member
double SavingsAccount::annualInterestRate = 0.0;

int main() {
  SavingsAccount saver1(2000.00);
  SavingsAccount saver2(3000.00);

  // Set precision for dollar output
  cout << fixed << setprecision(2);

  // Initial interest rate: 3%
  SavingsAccount::modifyInterestRate(3);
  saver1.calculateMonthlyInterest();
  saver2.calculateMonthlyInterest();

  cout << "Balances with an annual interest rate of 3%:" << endl;
  cout << "Saver 1: $" << saver1.getBalance() << endl;
  cout << "Saver 2: $" << saver2.getBalance() << endl;

  // Change interest rate to 4%
  SavingsAccount::modifyInterestRate(4);
  saver1.calculateMonthlyInterest();
  saver2.calculateMonthlyInterest();
  cout << "\nBalances with an annual interest rate of 4%:" << endl;
  cout << "Saver 1: $" << saver1.getBalance() << endl;
  cout << "Saver 2: $" << saver2.getBalance() << endl;

  return 0;
}
```



```
~/Lab42$ ls
output_02  ques_02.cpp  ques_03  ques_04.cpp  replit.nix
~/Lab42$ g++ ques_04.cpp -o 04_output
~/Lab42$ ./04_output
Balances with an annual interest rate of 3%:
Saver 1: $2005.00
Saver 2: $3007.50

Balances with an annual interest rate of 4%:
Saver 1: $2011.68
Saver 2: $3017.53
~/Lab42$
```