



San Francisco Bay University

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

Due day: 3/22/2024

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)

1. Correct the errors in the following snippets and explain

- a. Assume the following prototype is declared in class *Time*:

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

→ `~Time();` , #prototype for destructor in 'Time'

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

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

→ `Employee(string, string);` #making the constructor for 'Employee' by removing int return type.

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

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

```

        static int count;
    }; // end class Example

```

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:



```

1 #include <cmath> //cmath function importing all the math functions
2 #include <iostream>
3
4 class Rational {
5 private:
6     int numerator;
7     int denominator;
8
9     int greatest_common_factor(int a, int b) {
10         if (b == 0) {
11             return a;
12         }
13         return greatest_common_factor(b, a % b);
14     }
15
16     // Function to reduce the fraction
17     void reduce_fraction() {
18         int c_d =
19             greatest_common_factor(std::abs(numerator),
20 std::abs(denominator));
21         numerator /= c_d;
22         denominator /= c_d;
23     }
24 public:
25     // Constructor with default values and reduction
26     Rational(int num = 0, int denom = 1) : numerator(num),
27         denominator(denom) {
28         reduce_fraction();
29     }

```

```

public:
    // Constructor with default values and reduction
    Rational(int num = 0, int denom = 1) : numerator(num),
    denominator(denom) {
        reduce_fraction();
    }

```

- a. Adding two *Rational* numbers. The result should be stored in reduced form.

```

// add two Rational numbers
Rational add(const Rational &other) const {
    int new_num = numerator * other.denominator + other.numerator *
denominator;
    int new_den = denominator * other.denominator;
    return Rational(new_num, new_den);
}

```

- b. Subtracting two *Rational* numbers. The result should be stored in reduced form.

```

// subtract two Rational numbers
Rational subtract(const Rational &other) const {
    int new_num = numerator * other.denominator - other.numerator *
denominator;
    int new_den = denominator * other.denominator;
    return Rational(new_num, new_den);
}

```

- c. Multiplying two *Rational* numbers. The result should be stored in reduced form.

```

// multiply two Rational numbers
Rational multiply(const Rational &other) const {
    int new_num = numerator * other.numerator;
    int new_den = denominator * other.denominator;
    return Rational(new_num, new_den);
}

```

- d. Dividing two *Rational* numbers. The result should be stored in reduced form.

```

// divide two Rational numbers with Errir by zero division as well
Rational divide(const Rational &other) const {
    if (other.numerator == 0) {
        std::cerr << "Error: Division by zero!\n";
        return Rational();
    }
    int new_num = numerator * other.denominator;
    int new_den = denominator * other.numerator;
    return Rational(new_num, new_den);
}

```

- e. Printing *Rational* numbers in the form a/b , where a is the numerator and b is the denominator.

```
// Rational number to fraction
void print_frac() const { std::cout << numerator << "/" <<
denominator; }
```

- f. Printing *Rational* numbers in floating-point format.

```
// rational num to floatig point
void print_float() const {
    double result = static_cast<double>(numerator) / denominator;
    std::cout << result;
}
```

```
int main() {
    Rational f1(2, 4);
    Rational f2(3, 5);

    Rational r_a = f1.add(f2);
    Rational r_s = f1.subtract(f2);
    Rational r_m = f1.multiply(f2);
    Rational r_d = f1.divide(f2);

    // Results
    std::cout << "Addition Result: ";
    r_a.print_frac();
    std::cout << "\nSubtraction Result: ";
    r_s.print_frac();
    std::cout << "\nMultiplication Result: ";
    r_m.print_frac();
    std::cout << "\nDivision Result: ";
    r_d.print_frac();

    std::cout << "\n\nFloating-Point Representation of 1/2: ";
    f1.print_float();
    std::cout << "\nFloating-Point Representation of 3/5: ";
    f2.print_float();

    return 0;
}
```

Output:

```
~/SwekchhaHamalhw4CS360L$ ls
main main.cpp Makefile replit.nix
~/SwekchhaHamalhw4CS360L$ g++ main.cpp -o res1
~/SwekchhaHamalhw4CS360L$ ./res1
Addition Result: 11/10
Subtraction Result: -1/10
Multiplication Result: 3/10
Division Result: 5/6

Floating-Point Representation of 1/2: 0.5
Floating-Point Representation of 3/5: 0.6~/Swekcl
```

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*.

The screenshot shows a C++ IDE with a file explorer on the left, a code editor in the center, and a console on the right. The code editor displays the implementation of the *HugeInteger* class in *second.cpp*. The class has a private member *digits* of type *int* with a size of 40, and a public member *size* of type *int*. The *HugeInteger* constructor initializes the *digits* array to zero. The *input* function takes a string and stores its digits in the *digits* array. The *output* function prints the digits of the *HugeInteger* object. The *add* function is shown, which takes another *HugeInteger* object and returns a new *HugeInteger* object. The console shows the output of the program, which matches the output shown in the first block.

```
~/SwekchhaHamalhw4CS360L$ ls
main main.cpp Makefile replit.nix
~/SwekchhaHamalhw4CS360L$ g++ main.cpp -o res1
~/SwekchhaHamalhw4CS360L$ ./res1
Addition Result: 11/10
Subtraction Result: -1/10
Multiplication Result: 3/10
Division Result: 5/6

Floating-Point Representation of 1/2: 0.5
Floating-Point Representation of 3/5: 0.6~/Swekcl
```

```
HugeInteger add(const HugeInteger &other) const {
    HugeInteger result;
    int carry = 0;
    int maxSize = max(size, other.size);

    for (int i = 0; i < maxSize || carry; i++) {
        int sum = carry;
        if (i < size)
            sum += digits[i];
        if (i < other.size)
            sum += other.digits[i];

        result.digits[i] = sum % 10;
        carry = sum / 10;
        result.size++;
    }

    return result;
}

HugeInteger subtract(const HugeInteger &other) const {
    HugeInteger result;
    int borrow = 0;
    for (int i = 0; i < size; i++) {
        int diff = digits[i] - borrow;
        if (i < other.size)
            diff -= other.digits[i];
        if (diff < 0) {
```

```

        if (diff < 0) {
            diff += 10;
            borrow = 1;
        } else {
            borrow = 0;
        }
        result.digits[i] = diff;
        if (result.size == 0 && diff != 0) {
            result.size = i + 1;
        }
    }

    return result;
}

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

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

```

```
bool isGreaterThan(const HugeInteger &other) const {
    if (size > other.size)
        return true;
    if (size < other.size)
        return false;
    for (int i = size - 1; i >= 0; i--) {
        if (digits[i] > other.digits[i])
            return true;
        if (digits[i] < other.digits[i])
            return false;
    }
    return false;
}

bool isLessThan(const HugeInteger &other) const {
    return !isGreaterThan(other) && !isEqualTo(other);
}

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

bool isLessThanOrEqualTo(const HugeInteger &other) const {
    return isLessThan(other) || isEqualTo(other);
}
```



```

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

bool isLessThanOrEqualTo(const HugeInteger &other) const {
    return isLessThan(other) || isEqualTo(other);
}

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

HugeInteger multiply(const HugeInteger &other) const {
    HugeInteger result;
    for (int i = 0; i < size; i++) {
        int carry = 0;
        for (int j = 0; j < other.size || carry; j++) {
            int mul = result.digits[i + j] +
                digits[i] * (j < other.size ? other.digits[j] : 0) +
carry;
            result.digits[i + j] = mul % 10;
            carry = mul / 10;
            if (i + j + 1 > result.size && (carry > 0 || mul > 0)) {

```

```

        result.digits[i + j] = mul % 10;
        carry = mul / 10;
        if (i + j + 1 > result.size && (carry > 0 || mul > 0)) {
            result.size = i + j + 1;
        }
    }
}
return result;
}

HugeInteger divide(const HugeInteger &other) const {
    HugeInteger quotient;
    HugeInteger remainder = *this;

    if (other.isZero()) {
        cerr << "Error: Division by zero\n";
        return quotient;
    }

    while (remainder.isGreaterThanOrEqualTo(other)) {
        HugeInteger temp = other;
        HugeInteger count;
        count.input("1");

        while (temp.multiply(count).isLessThanOrEqualTo(remainder)) {
            count = count.add(HugeInteger("1"));
        }
    }
}

```

```

        while (temp.multiply(count).isLessThanOrEqualTo(remainder)) {
            count = count.add(HugeInteger("1"));
        }

        count = count.subtract(HugeInteger("1"));
        quotient = quotient.add(count);
        remainder = remainder.subtract(temp.multiply(count));
    }

    return quotient;
}

HugeInteger modulus(const HugeInteger &other) const {
    HugeInteger remainder = *this;

    if (other.isZero()) {
        cerr << "Error: Division by zero\n";
        return remainder;
    }

    while (remainder.isGreaterThanOrEqualTo(other)) {
        HugeInteger temp = other;
        HugeInteger count;
        count.input("1");

        while (temp.multiply(count).isLessThanOrEqualTo(remainder)) {
            count = count.add(HugeInteger("1"));

```

```
int main() {  
    HugeInteger num1("123456789012345678901234890");  
    HugeInteger num2("987654321098765438876543210");  
  
    cout << "num1: ";  
    num1.output();  
  
    cout << "num2: ";  
    num2.output();  
  
    HugeInteger sum = num1.add(num2);  
    cout << "Sum: ";  
    sum.output();  
  
    HugeInteger difference = num1.subtract(num2);  
    cout << "Difference: ";  
    difference.output();  
  
    HugeInteger product = num1.multiply(num2);  
    cout << "Product: ";  
    product.output();  
  
    HugeInteger quotient = num1.divide(num2);  
    cout << "Quotient: ";  
    quotient.output();  
  
    HugeInteger remainder = num1.modulus(num2);  
    cout << "Remainder: ";
```

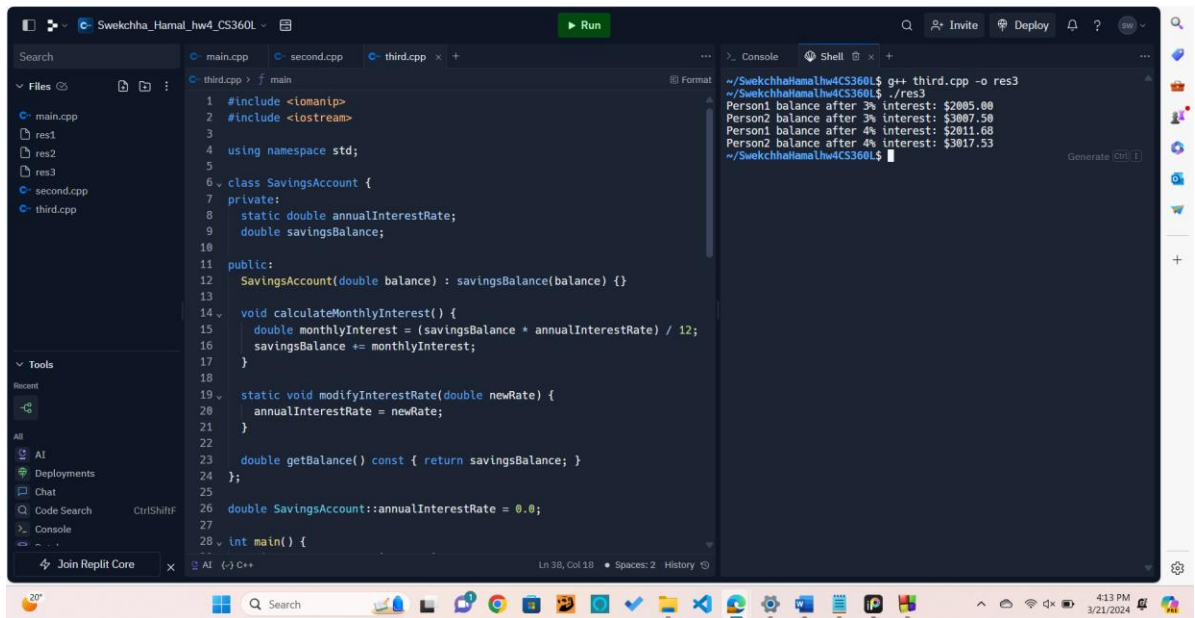
Output:

```

main.cpp Makefile replit.nix res1 second.cpp
~/SwekchhaHamaIhw4CS360L$ g++ second.cpp -o res2
~/SwekchhaHamaIhw4CS360L$ ./res2
num1: 123456789012345678901234890
num2: 987654321098765438876543210
Sum: 111111111011111117777778100
Difference: 80
Product: 40587228302-3-2-8-8-3-7-1-4-5-8-2-60-4-2-8-4-3-7-1-9-1-2-5
0-9-2-4-1-3-3-2-5-10-5-3-1-5-8-8-3-5-4-3-7-4-5-10-8-4-2-5-4-2-5-8-3
-8-5-3-2-8-8-7-4-1-5-7-7-5-5-1-7-8-9-6-2-6-7-6-2-6-6-4-5-5-7-2-9-6-
1-6-5-3-8-6-1-5-2-1-4-4-6-3-7-2-3-2-4-3-4-9-5-5-90-3-6-9-30-5-1-7-3
-5-6-4-7-5-5-4-7-5-9-6-1-2-3-3-4-30-1-1-7-4-3-6-4-2-5-3-2-1-5-4-2-9
-2-40-1-7-4-8-6-9-8-1-9-4-40-2-6-8-90-6-8-60-4-3-3-9-9-2-6-4-6-9-20
0-2-5-4-9-5-8-20-4-3-6-1-8-6-1-4-8-4-2-4-5-4-7-30-5-1-70-7-70-8-2-5
-3-7-9-5-4-1-8-3-6-5-4-1-4-30-5-3-7-4-3-7-9-3-10-2-4-4-6-1-9-5-2-7-
4-8-7-10-2-6-5-3-6-7-2-7-8-9-2-8-4-6-8-6-1-1-2-7-3-902-1-2-5-5-2-54
8-1-4-6-8-4-2-7-6-4-5-2-9180-3-8-6-1-3-20-4-2-4-2-790466-1-471-4-6-
81-3-4-7-5-5-437-9-1-6069-5-166-1-7-1-4-3-4520-4-3-7-8-1-7-9-1-9-1-
3-4-5-7-9-7-60-3-7-3-7-10-1-2-1-9-2-9-9-6-3-7-4-3-7-7-1-1-4-80-1-8-
9-5-2-1-5-9-4-6-10-4-7-9-7-1-4-5-4-1-9-3-40-6-4-80-2-3-6-5-6-522643
94106755131796061576289865721723665449523444596900
Quotient:
Remainder: 123456789012345678901234890

```

4. Create a *SavingsAccount* class. Use a *static* data member *annualInterestRate* 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.



```
1 #include <iomanip>
2 #include <iostream>
3
4 using namespace std;
5
6 class SavingsAccount {
7 private:
8     static double annualInterestRate;
9     double savingsBalance;
10
11 public:
12     SavingsAccount(double balance) : savingsBalance(balance) {}
13
14     void calculateMonthlyInterest() {
15         double monthlyInterest = (savingsBalance * annualInterestRate) / 12;
16         savingsBalance += monthlyInterest;
17     }
18
19     static void modifyInterestRate(double newRate) {
20         annualInterestRate = newRate;
21     }
22
23     double getBalance() const { return savingsBalance; }
24 };
25
26 double SavingsAccount::annualInterestRate = 0.0;
27
28 int main() {
```

```
~/SwkchhaHamaIhw4CS360L$ g++ third.cpp -o res3
~/SwkchhaHamaIhw4CS360L$ ./res3
Person1 balance after 3% interest: $2005.00
Person2 balance after 3% interest: $3007.50
Person1 balance after 4% interest: $2011.68
Person2 balance after 4% interest: $3017.53
~/SwkchhaHamaIhw4CS360L$
```

```
double SavingsAccount::annualInterestRate = 0.0;

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

    SavingsAccount::modifyInterestRate(0.03);
    saver1.calculateMonthlyInterest();
    saver2.calculateMonthlyInterest();

    cout << fixed << setprecision(2);
    cout << "Person1 balance after 3% interest: $" << saver1.getBalance()
    << endl;
    cout << "Person2 balance after 3% interest: $" << saver2.getBalance()
    << endl;

    SavingsAccount::modifyInterestRate(0.04);

    saver1.calculateMonthlyInterest();
    saver2.calculateMonthlyInterest();

    cout << "Person1 balance after 4% interest: $" << saver1.getBalance()
    << endl;
    cout << "Person2 balance after 4% interest: $" << saver2.getBalance()
    << endl;

    return 0;
}
```