Assignment #2

Due Date 1: Friday, October 4, 2024, 5:00 pm EDT (30% marks)
Due Date 2: Friday, October 11, 2024, 5:00 pm EDT (70% marks)

Learning objectives:

- Dynamic memory allocation for C++ objects and arrays
- C++ classes, constructors, destructors, and operations
- Object-Oriented programming, invariants and encapsulation

- **Questions 1a, 2a, and 3a are due on Due Date 1; questions 1b, 2b, 3b, are due on Due Date 2.**
- See A1 for notes on: test suites, undefined behaviour, test partners, using C++-style I/O and memory management, hand-marking,
- In all cases, your test suites should be testing the functionality that you are responsible for. **Do not submit test cases whose sole purpose is to verify the behaviour of the test harness.** There is no value in that, and there are no marks allocated for that.
- You may `import`/`include` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, and `string`. Marmoset will be setup to **reject** submissions that use C-style I/O or memory management, or libraries other than the ones specified above.
- We have provided some code and sample executables under the appropriate `a2` subdirectories. **These executables have been compiled in the CS student environment and will not run anywhere else.**
- **Before asking on Piazza, see if your question can be answered by the sample executables we provide. You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do.** A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

Question 1

(25% of DD1; 25% of DD2) For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your program with `g++20`.

We typically use arrays to store collections of items (say, booleans). We can allow for limited growth of a collection by allocating more space than typically needed, and then keeping track of how much space was actually used. We can allow for unlimited growth of the array by allocating the array on the heap and resizing as necessary. The following structure simulates *non-negative* binary numbers by encapsulating a partially-filled array:

```
struct BinaryNum {
    int size;         // number of elements the array currently holds
    int capacity;     // number of elements the array could hold,
                      // given current memory allocation to contents
    bool *contents;   // (pointer to) heap-allocated array of bools
};
```

- Write the function `readBinaryNum` that returns a `BinaryNum` structure, and whose signature is as follows:

```
BinaryNum readBinaryNum();
```

The function `readBinaryNum` consumes as many ones and zeroes from `cin` as are available, populating a `BinaryNum` structure in order with these, and then returns the structure. (You should make no assumptions about whether or not the input is on the same line as the read command. There must be at least 1 digit read.) If a character other than a one or a zero is encountered, this marks the end of the input sequence and this character is ignored/thrown away. If the terminating character is encountered before the array is full, then `readBinaryNum` stores as many values into the array as it successfully read before reading this character, leaving the rest of the array "unfilled".

In all circumstances, the field `size` should accurately represent the number of elements actually stored in the array and `capacity` should represent the total amount of storage currently allocated to the array.

- Write the function `binaryConcat` that takes an existing `BinaryNum` structure by reference whose signature is as follows:

```
void binaryConcat(BinaryNum &binNum);
```

The function `binaryConcat` concatenates on to the end of the structure passed in as many booleans as are available on `cin`. The behaviour is identical to `readBinaryNum`, except that 0 or more boolean values are being added to the end of an existing `BinaryNum`.

- Write the function `binaryToDecimal`, which takes a `BinaryNum` structure by reference to a constant and returns an integer. The signature is as follows:

```
int binaryToDecimal(const BinaryNum &binNum);
```

The function `binaryToDecimal` should return the integer value equivalent of the binary number represented by the given `BinaryNum` structure. The behaviour of this function is *undefined* if the integer equivalent of the binary number exceeds the maximum value that can be stored within an `int`.

- Write the function `printBinaryNum`, which takes a `BinaryNum` structure by reference to a constant and a separator character, and whose signature is as follows:

```
void printBinaryNum(std::ostream &out, const BinaryNum &binNum, char sep);
```

The function `printBinaryNum` prints the contents of `binNum` (as many elements as are actually present) to the given output stream `out`, on the same line, where each digit is separated by the `sep` character, and followed by a newline. **There should be a separator character only between each element in the array.** This means there should be none before the first element or after the last element. **Note:** Because of the way the test harness is set up, the separator character cannot be a whitespace (space, tab or newline) character.

It is not valid to perform any operations on a `BinaryNum` that has not first been read, because its fields may not be properly set. You should not test this.

For memory allocation, you *must* follow this allocation scheme: every `BinaryNum` structure begins with a capacity of 0. The first time data is stored in a `BinaryNum` structure, it is given a capacity of 4 and space is allocated accordingly. If at any point, this capacity proves to be not enough, you must double the capacity (so capacities will go from 4 to 8 to 16 to 32 ...). Note that there is no `realloc` in C++, so doubling the size of an array requires allocating a new array and copying items over. Your program must not leak memory.

An interface file (`binarynum.cc`) containing all the functions to implement has been provided in the `a2/codeForStudents/q1` directory. Implement the required functions in a file called `binarynum-impl.cc`.

A test harness is available in the file `a2q1.cc`, which you will find in your `a2/codeForStudents/q1` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** A sample test case that can be run using the test harness and the provided sample executable is also provided. Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use). You should not modify `binarynum.cc` or `a2q1.cc`.

a) **Due on Due Date 1:** Design a test suite for this program, using the `main` function provided in the test harness. Call your suite file `suiteq1.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q1a.zip`. (This program does not accept command line arguments, so there will be no `.args` files.)

b) **Due on Due Date 2:** Submit the file `binarynum-impl.cc` containing the implementation of functions declared in `binarynum.cc`.

Question 2

(35% of DD1; 35% of DD2) For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your program with `g++20`.

In this question you will write a C++ class (implemented as a `struct`) that adds support for *rational numbers* to C++. In mathematics, a rational number is any number that can be expressed as a fraction `n/d` of two integers, a numerator `n` and a non-zero denominator `d`. Since `d` could be equal to 1, every integer is also a rational number. In our `Rational` class, rational numbers are always stored in their most simplified form e.g. 4/8 is stored as 1/2, 18/8 as 9/4. Additionally, negative rational numbers are stored with a negative numerator and a positive denominator e.g. negative one-quarter is stored as -1/4 and not 1/-4.

An interface file (`rational.cc`) containing all the methods and functions to implement has been provided in the `a2/codeForStudents/q2` directory. Implement the required methods and functions in a file called `rational-impl.cc`.

Implement a two-parameter constructor with the following signature:

```
Rational(int num = 0, int den = 1);
```

To support arithmetic for rational numbers, overload the binary operators +, −, ∗ and / to operate on two rational numbers. In case you have forgotten how these operations work, here is a refresher:

$\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ $\frac{a}{b} - \frac{c}{d} = \frac{ad-bc}{bd}$ $\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$ $\frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$

Also, implement the following:

- A convenience unary (`-`) operator, which negates the rational number.
- Convenience `+=`, `-=` operators where `a += b` has the effect of setting `a` to `a + b` (similarly for `-=`)
- The helper method `simplify()`, which can be used to update a rational number to its simplest form.
- Accessor methods `getNumerator()` and `getDenominator()` that return the numerator and denominator of the rational number, respectively.
- Method `isZero()` returns true if the rational number is zero.
- Implement the overloaded input operator for rational numbers as the function:
  `std::istream &operator>>(std::istream &, Rational &);`

  The format for reading rational numbers is: an int-value-for-numerator followed by the / character followed by an int-value-for-denominator. Arbitrary amounts of whitespace are permitted before or in between any of these terms. A denominator must be provided for all values even if the denominator is 1 (this includes the rational number 0) e.g. the rational number 5 must be input as `5/1`.
- Implement the overloaded output operator for rational numbers as the function:
  `std::ostream &operator<<(std::ostream &, const Rational &);`

  The output format is the numerator followed by the / character and then the denominator without any whitespace. Rational numbers that have a denominator of 1 are printed as integers e.g. 17/1 is printed as 17.

A test harness is available in the file `a2q2.cc`, which you will find in your `a2/codeForStudents/q2` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use). You should not modify `rational.cc` or `a2q2.cc`.

a) **Due on Due Date 1:** Design a test suite for this program. Call your suite file `suiteq2.txt`. Zip your suite file, together with the associated `.in`, and `.out` files (this program does not accept command line arguments, so there will be no `.args` files), into the file `a2q2a.zip`.

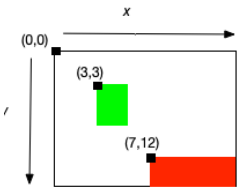
b) **Due on Due Date 2:** Submit the file `rational-impl.cc` containing the implementation of methods and functions declared in `rational.cc`.

Question 3

(40% of DD1; 40% of DD2) For this question: use `#includes`. Compile your program with `g++20`.

In this question you will implement the `Canvas` class where each `Canvas` holds zero or more `Rectangle` objects. While you are allowed to add, remove, and manipulate rectangles within a given canvas, and perform some simple operations upon the canvas itself, the key idea being tested is that you have correctly implemented the move and copy semantics for your structures to make *deep* copies.

Following the form of classic raster screen display such as a Cathode Ray Tube monitor uses, a `Canvas` is defined such that its upper-left corner has the coordinates (0,0). X-coordinates increase towards the right (there is no negative value). Y-coordinates increase downwards (there is no negative value).



A `Canvas` starts initially empty, with the height and width both integers with the value 0. When a `Rectangle` is added to a `Canvas`, the `Canvas` "stretches" to accommodate the new `Rectangle` (if necessary) by calculating its new dimensions based upon the `Point` of the `Rectangle` and its **non-zero dimensions**. Rectangle indices j are used to uniquely identify each rectangle within the canvas. The indices start at 0, and if a canvas's rectangle is removed, the indices of the rectangles after the one removed need to be decreased by 1. (Hint: this may not require much extra work, depending upon your implementation of how rectangles are stored.) **Note:** operations on invalid rectangle indices are invalid test cases, as are scaling rectangle dimensions by values ≤ 0 or translating a rectangle/point such that one or both of its coordinates are < 0.

You are given some header files that define the interfaces for the `Point`, `Rectangle`, and `Canvas` classes, named `point.h`, `rectangle.h`, and `canvas.h` respectively. **Read these files for the required definitions of your classes, their methods and the I/O format. You may not change the interfaces**; however, you may add instance variables, helper methods, and comments. You will also need to fill in the necessary declarations for the information held in these classes.

You have also been provided with a simple test harness, `a2q3.cc`, a sample input file, `sample.in`, and the corresponding output, `sample.out`, produced by the sample executable, `a2q3`, for the output format of the `Canvas` `Rectangle` and `Point` objects. Read these files carefully to understand how to interact with the test harness and the sample executable. **Do not change the test harness.**

a) **Due on Due Date 1:** Design the test suite `suiteq3.txt` for this program and zip all `.in`, `.out` and `suiteq3.txt` files into `a2q3a.zip`. (This program does not accept command line arguments, so there will be no `.args` files.)

b) **Due on Due Date 2:** Implement this in C++ and place your `Makefile`, `a2q3.cc` and all `.h` and `.cc` files that make up your program in the ZIP file, `a2q3b.zip`. Your Makefile must create an executable named `canvas`. Note that the executable name is case-sensitive.