Assignment #2

Due Date 1: Friday, October 6, 2023, 5:00 pm EST (30% marks)
Due Date 2: Friday, October 20, 2023, 5:00 pm EST (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.**
- On this and subsequent assignments, you will take responsibility for your own testing. This assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. For each question you will be given a compiled executable program that is a program representing a solution to each question. You should use these provided executables to help you write your test cases, as they can show you the resultant output for given inputs. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

  Test suites will be in a format compatible with the `runSuite` script that you used in CS136L. We have also provided binaries for `runSuite` and `produceOutputs` in the `a1` directory compiled in the student environment if you need them.
- See A1 for notes on test suites, undefined behaviour, and using C++-style I/O and memory management.
- You may `import` 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 MM, or libraries other than the ones specified above.
- There will be a hand-marking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. Please code to a standard that you would expect from someone else if you had to maintain their code. **We will not answer Piazza questions about coding style; use your best judgment.** Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/F23/codingguidelines.shtml`
- 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

(30% of DD1; 30% of DD2) For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your program with `g++20m`. 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 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` which 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, populates a `BinaryNum` structure in order with these, and then returns the structure. If a non-whitespace character that is not a one or a zero is encountered before the structure is full, then `readBinaryNum` fills as much of the array as needed, leaving the rest unfilled. If a non one or zero character is encountered, the first offending character should be removed from the input stream. In all circumstances, the field `size` should accurately represent the number of elements actually stored in the array and `capacity` should represent the amount of storage currently allocated to the array.

- Write the function `binaryConcat`, which takes a `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 ones and zeros as are available on `cin`. The behaviour is identical to `readBinaryNum`, except that integers are being added to the end of an existing `BinaryNum`.

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

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

  The function `binaryToDecimal` should return the decimal value equivalent of the binary number represented by the given `BinaryNum` structure. The behaviour of this function is undefined if the decimal 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 const 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 stream `out`, on the same line, separated by characters denoted by `sep`, and followed by a newline. There should be a separator character between each element in the array, this means there should be none before the first element or after the last element.

  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 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 necessitates allocating a new array and copying items over. Your program must not leak memory.

An interface file and test harness are available in the starter files `binarynum.cc` and `a2q1.cc`, which you will find in your `/a2/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 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).

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

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

Question 2

(30% of DD1; 30% of DD2) For this question: use #include. Compile your program with `g++20i`. In this exercise, you will write a C++ class (implemented as a `struct`) that adds support for arithmetic operations of the *extended integer number system* to C++. In mathematics, the extended integer number system results from adding $+\infty$ and $-\infty$ to the integer number system.

In our `ExtendedNum` class, elements of the number system can be integer numbers, $+\infty$, or $-\infty$. To support arithmetic operations in this number system, overload the binary operators $+$, $-$, $*$ and $/$. The operations are described below.

$$
\begin{aligned}
a \pm \infty = \pm\infty + a = \pm\infty, \quad & a \neq \mp\infty \\
a \times (\pm\infty) = \pm\infty \times a = \pm\infty, \quad & a \in (0, +\infty] \\
a \times (\pm\infty) = \pm\infty \times a = \mp\infty, \quad & a \in [-\infty, 0) \\
a/ \pm \infty = 0, \quad & a \in \mathbb{Z} \\
\pm a/0 = \pm\infty, \quad & a \in \mathbb{Z} \\
\pm\infty/a = \pm\infty, \quad & a \in (0, +\infty) \\
\pm\infty/a = \mp\infty, \quad & a \in (-\infty, 0)
\end{aligned}
$$

The results of operations $\infty - \infty$, $0 \times (\pm\infty)$, $0/0$, and $\pm\infty/\pm\infty$ are considered **undefined**. Any arithmetic operation, where at least one of the operands is `undefined`, will produce an `undefined` output.

You must also implement the following:

- A convenience unary (−) operator, which returns the negation of the element (number or infinity). Note, the operator does **not** change the object used in the operation.
- The overloaded input operator as the function:
  `std::istream &operator>>(std::istream &, extendedNum &);`

  The format for reading an `ExtendedNum` element is an `int` value for number, the strings "`-infinity`" and "`+infinity`" for infinities elements, or the string "`undefined`" for an undefined element. Arbitrary amounts of whitespace are permitted before or after any of these terms.
- The overloaded output operator as the function:
  `std::ostream &operator<<(std::ostream &, const extendedNum &);`

  The output is the number value, or the strings "`+infinity`" or "`-infinity`" indicating the infinity element stored in an `ExtendedNum` object or resulting from an operation. Operations with undefined results should output their results as the string "`undefined`".

An interface file (`extendedNum.h`) containing all the methods and functions to implement has been provided in the `a2/q2` directory. Implement the required methods and functions in a file called `extendedNum.cc`. You must define the data members of the class `ExtendedNum` and its constructor.

A test harness is available in the file `a2q2.cc`, which you will find in your `a2/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 `a2q2.cc`.

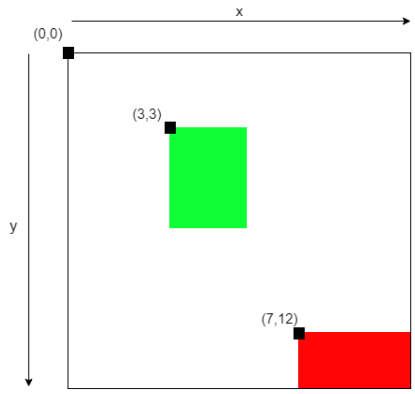
1. **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq2.txt`. Zip your suite file, together with the associated `.in`, and `.out` files (this program does not accept command line arguments, so there will be no `.args` files), into the file `a2q2.zip`.

2. **Due on Due Date 2**: Submit the file `extendedNum.cc` containing the implementation of operators declared in `extendedNum.h`.

Question 3

(40% of DD1;40% of DD2) For this question: use #include. Compile your program with `g++20i`. 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 displays such as a Cathode Ray Tube monitor uses, a `Canvas` is defined such that its upper-left corner has the coordinates (0,0).[1] 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 dimensions. Rectangle indices j are used to uniquely identify each rectangle within the canvas. The indices start at 0, and if a canvas rectangle is removed, the indices of the rectangles after the one removed need to be decreased by 1. The removal of a rectangle also "shrinks" the `Canvas` to a size that just accommodates the remaining rectangles.

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 public interfaces**; however, you may add private instance variables, private helper methods, and comments. You will also need to fill in the necessary private 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, `canvas`, for the output format of the `Canvas`, `Rectangle`

[1] Ideally we'd prevent points from having negative values for their x- and y-coordinates, but that is not a requirement of this question.

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.**

1. **Due on Due Date 1:** Design the test suite `suiteq3.txt` for this program and zip the suite into `a2q3a.zip` (this program does not accept command line arguments, so there will be no .args files).
2. **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.**