Assignment #3: C++ Classes

Due Date 1: Friday, October 22, 2021, 5:00 pm
Due Date 2: Friday, October 29, 2021, 5:00 pm
Online Quiz: Monday, November 1, 2021, 5:00 pm

Topics that must have been completed before starting Due Date 1:

1. Software Testing, `produceOutputs` and `runSuite` from A1
2. Object-Oriented Programming: Introduction
3. Object-Oriented Programming: Special class members
4. Object-Oriented Programming: Advanced object uses

Topics that must have been completed before starting Due Date 2:

1. Encapsulation and Introduction to Design Patterns: Invariants and Encapsulation
2. Encapsulation and Introduction to Design Patterns: Design Patterns and Iterators

Learning objectives:

- Object-Oriented programming, Invariants and Encapsulation
- C++ classes, constructors, destructors, and operations
- Dynamic memory allocation for C++ objects and arrays
- Iterator design pattern

- **Questions 1a, 2a, and 3a are due on Due Date 1; questions 1b, 2b, 3b, are due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.**
- 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 that of the latter questions of Assignment 1, so if you did a good job writing your `runSuite` script, that experience will serve you well here.
- Design your test suites with care; they are your primary tool for verifying the correctness of your code. Note that test suite submission `zip` files are restricted to contain a maximum of 40 tests. The size of each input (`.in`) file is also restricted to 300 bytes. This is to encourage you not to combine all of your testing eggs in one basket. There is also a limit for each output file (`.out`), but none of your tests should be able to create such a large output file that you would normally encounter it.
- You must use the standard C++ I/O streaming and memory management (MM) facilities on this assignment; you may **not** use C-style I/O or MM. More concretely, you may `#include` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, 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.

- We will manually check that you follow a reasonable standard of documentation and style, and to verify any assignment requirements that are not automatically enforced by Marmoset. Code to a standard that you would expect from someone else if you had to maintain their code. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/F21/codingguidelines.shtml`
- We have provided some code and sample executables under the appropriate `a3` subdirectories. **These executables have been compiled in the CS student environment and will not run anywhere else.**
- **You may not ask 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.

Note: We suggest creating the directory `~/cs246/f21/a3` and creating all of your assignment solutions in this directory.

Question 1

(20% of DD1; 20% of DD2) In this exercise, 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 a quarter is stored as -1/4 and not 1/-4.

A header file (`rational.h`) containing all the methods and functions to implement has been provided in the `a3/rationals` directory. You should implement the required methods and functions in a file named `rational.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.
- 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 `a3q1.cc`, which you will find in your `a3/rationals` 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 a3q1.cc

1. **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq1.txt`. Zip your suite file, together with the associated `.in`, `.out` into the file `a3q1.zip`.
2. **Due on Due Date 2**: Submit the file `rational.cc` containing the implementation of methods and functions declared in `rational.h`.

Question 2

(40% of DD1; 40% of DD2) In this problem, you will implement a `Polynomial` class to represent and perform operations on single-variable polynomials. We will use the `Rational` class from Q1 to represent the coefficients of the terms in a `Polynomial`. A polynomial can be represented using an array with the value at index `idx` used to store the coefficient of the term with exponent `idx` in the polynomial. For example, $(9/4)x^3 + (-7/3)x + 3/2$ is represented by the array of `Rationals {3/2,-7/3,0/1,9/4}`.

The `degree` of a polynomial is the exponent of the highest non-zero term (3 in the example just discussed). Therefore, an array of size `n+1` is required to store a polynomial of degree `n`. In order to not restrict the `Polynomial` class to a maximum degree, the array used to encode the polynomial is heap-allocated.

You may assume that the coefficients of polynomials (given as input or produced through operations) can always be represented using a `Rational`. Additionally, there is no need to worry about over- and under-flow.

In the file `polynomial.h`, we have provided the type definition of `Polynomial` and signatures for the overloaded input and output operators for the `Polynomial` class. Implement all methods and functions. Place your implementation in `polynomial.cc`. **You are free to use your own implementation from Question 1 of the methods and functions in `rational.h` or use the implementation we have provided in the compiled binary file `rational.o`. Note that your implementation of `polynomial.cc` will be linked to our implementation of `Rational` during testing.**

The default constructor for `Polynomial` should create the `zero` polynomial, i.e. a polynomial whose coefficients are all zero.

The overloaded arithmetic operators work identically to single-variable polynomial arithmetic. For the division operation, two methods are to be implemented; `operator/` should return the quotient after long division and `operator%` should return the remainder.

The input operator reads the input stream till the end of the line and uses the read input to modify an existing `Polynomial` object. The input format is a pair for each non-zero term in the polynomial in decreasing exponent values with no exponent repeated. The first value in each pair is a rational number and follows the input format for `Rational` numbers as specified in Q1. This is the coefficient. The second value is a non-negative integer and represents the exponent of this term. Arbitrary amount of whitespace (excluding newline) is allowed within each pair and between pairs. For example, the input `3/5 5 -2/5 2 1/2 1 3/7 0` represents the polynomial `(3/5)x^5 + (-2/5)x^2 + (1/2)x + (3/7)`.

The output operator prints polynomials as the addition of terms in decreasing exponent order. Each term is output as `(a/b)x^n` and subject to the following additional requirements and exceptions:

- Terms with zero coefficients are not printed.
- Coefficients are printed using the output format for `Rational` numbers as described in Q1.
- A term whose exponent is 1 is printed as `(a/b)x` and a term whose exponent is 0 is printed as `(a/b)`.

An example of the output produced by the output operator is shown above during the discussion of the input operator.

A test harness is available in the file `a3q2.cc`, which you will find in your `a3/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 a3q2.cc

(a) **Due On Due Date 1**: Design a test suite for this program. The suite should be named `suiteq2.txt` and zip the suite into a zip file named `a3q2a.zip`.

(b) **Due On Due Date 2**: Full implementation of the Polynomial class in C++. Your zip archive should contain at minimum the unchanged file `a3q2.cc`, `polynomial.h`, `polynomial.cc` and your Makefile. Your Makefile must create an executable named `a3q2`. Note that the executable name is case-sensitive. Any additional classes (if created) must each reside in their own `.h` and `.cc` files. Name the zip file `a3q2b.zip`

Question 3

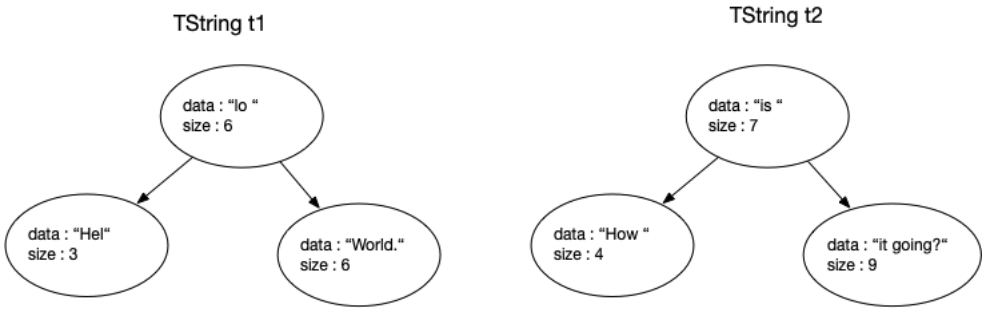
(40% of DD1;40% of DD2) In C, it is common to think of strings as arrays of characters. However, it can also be beneficial to consider a tree-like structure for strings, especially when we are working with very large strings that we want to modify often and at arbitrary locations (not just the end). An example of where such a representation is useful is text editors. Using a tree representation has the advantage that we can concatenate strings lazily rather than dealing with common inefficiencies with resizing arrays and inserting at arbitrary locations within an array.

In this question, you will implement such a tree-like structure. **This means that you are not allowed to use string concatenation in your solution as it nullifies the usefulness of this structure.**

Consider the following skeleton of a class definition for a tree-based string `TString` class provided in files `tstring.h` (some methods and documentation is omitted here for brevity and can be found in the header files):

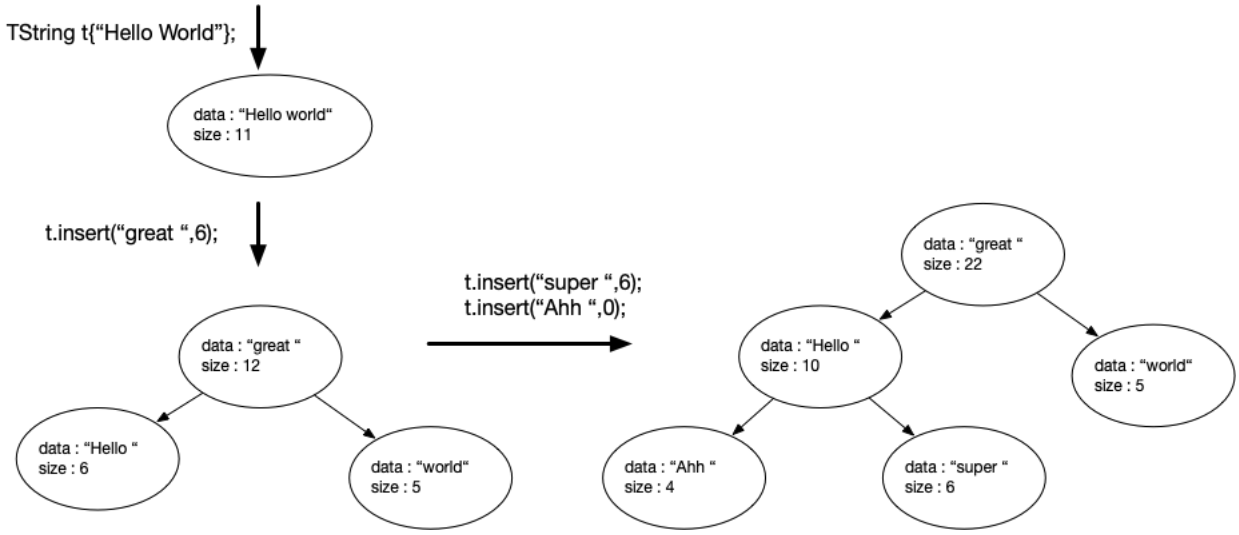
```
class TString {
    struct TNode {
      std::string data;
      TNode *left, *right;
      size_t size; // represents the size of the string represented by this node's
                   // left subtree, plus data's length
    };
    TNode *root;  // root of the tree
  public:
    TString(const std::string &);
    TString(const TString &);
    TString operator+(const TString &) const;
    void insert(const std::string &, const size_t);
    char &operator[](const int);
    friend std::ostream& operator<<(std::ostream&, const TString&)
    class TStringIter {
        TStringIter(TNode*, int);
      public:
        char &operator*();
        TStringIter &operator++();
        bool operator!=(const TStringIter &);
        friend class TString;
    };
    TStringIter begin();
    TStringIter end();
};
```

A `TString` object encodes a string by representing parts of the string as `TNode` objects. The string representation can be obtained via an in-order traversal of the `TString` object. For example, the in-order traversal of `TString t1` (shown below) would print `Hello World.` and the in-order traversal of `TString t2` (shown below) would print `How is it going?` Notice how each `TNode` object always maintains the invariant that `size` is the sum of the `size` of the left subtree and the length of the `data` field.

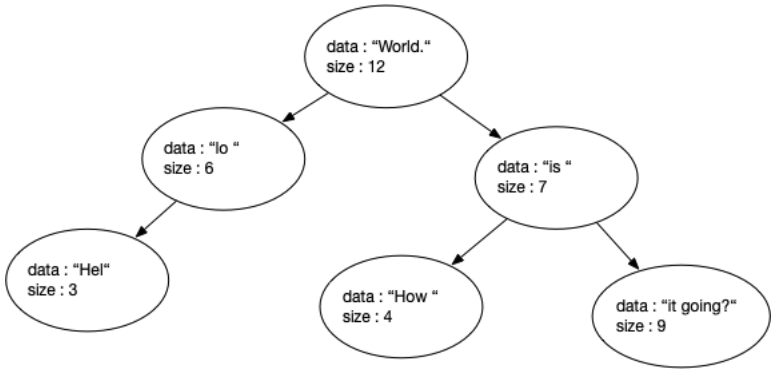


- The constructor parameterized with a `std::string` constructs a new `TString` object with a single node that stores the given string.
- Your copy constructor must ensure that all copies of a `TString` object are deep copies.
- The `TString::insert` method allows for the insertion of a string into an existing `TString` object by specifying an `index` at which to insert the new string. The `index` refers to the overall string represented by this `TString` object. Follow the procedure:
  - Determine the **deepest** node `N` in the `TString` object such that inserting at `i` requires inserting a new `TNode` either immediately before, immediately after, or somewhere within the data of `N`.
  - To insert before, add a new `TNode` as a new left child of `N`, updating the rest of the tree as needed.
  - To insert after, add a new `TNode` as a new right child of `N`, updating the rest of the tree as needed.
  - To add somewhere within the data at node `N`, i.e., index `i` falls within the data of `N`, the data at node `N` must be split into two new nodes that become N's new left and right children (along with any necessary updates). Set N's data to the string being inserted.

  The following sequence of insertions shows the procedure pictorially:



- The concatenation operation (`operator+`, `t1 + t2`) concatenates two `TString` objects by following the procedure: take the right-most child of `t1` and make it the root of the new `TString` object. The right child of this new object is `t2`. The left child is a copy of `t1` without its right-most child. The following diagram shows this pictorially for the objects `t1` and `t2` from the diagram above:



- The index operation (`operator[]`) indexes the `TString` and returns a reference to the appropriate character at that index in the string allowing the user to modify the contents of the `TString`.
- The output operator (`operator<<`) prints out the string represented by the given `TString`.
- The iterator class (`TStringIter`) implements the three operations necessary for an iterator as described in the notes, allowing for iteration over the characters of a `TString`.
- The `begin()` and `end()` methods return `TStringIter` objects for the beginning and end of the given `TString` respectively in order to implement the iterator pattern and allow for range-based for loops to be executed on `TString` objects.

A test harness is available in the file `a3q3.cc`, which you will find in your `codeForStudents/q3` directory. For your convenience, we have implemented a way to pretty-print the tree structure representation of a `TString` at any time. The 'd' option in the test harness can be used to invoke the pretty-printer on any `TString`. **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 a3q3.cc.

(a) **Due On Due Date 1**: Design a test suite for this program. The suite should be named `suiteq3.txt` and zip the suite into a zip file named `a3q3a.zip`.

(b) **Due On Due Date 2**: Full implementation of the `TString` class in C++. Your zip archive should contain at minimum `tstring.h`, `tstring.cc`, the unchanged file `a3q3.cc`, and your Makefile. Your Makefile must create an executable named `a3q3`. Note that the executable name is case-sensitive. Any additional classes (if created) must each reside in their own `.h` and `.cc` files. Name the zip file `a3q3b.zip`