CS246—Assignment 3 (Winter 2018)

Due Date 1: Friday, February 9 5pm
Due Date 2: Monday, February 26, 5pm
Due Date 3: Friday, March 2, 5pm

Questions 1a, 2a, 3a and 4a are due on Due Date 1; Question 1b, 3b and 4b are due on Due Date 2. Question 2b is due on Due Date 3

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may `#include` are `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, and `<utility>`. Marmoset will be programmed to **reject** submissions that violate these restrictions.

Note: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. Moreover, each question asks you to submit a `Makefile` for building your program. For these reasons, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.

Note: There will be a handmarking 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. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/current/AssignmentGuidelines.shtml`

Note: 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. Instead, we will provide compiled executables, suitable for running on `linux.student.cs`, that you can use to check intended behaviour. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

1. 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/a3q1` 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:

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

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.
- The equality and inequality operators, `operator==` and `operator!=`.
- 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/a3q1` 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

(a) **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`, and `.args` files, into the file `a3q1.zip`.

(b) **Due on Due Date 2**: Submit the file `rational.cc` containing the implementation of methods and functions declared in `rational.h`. You must include a `Makefile` that compiles your code, and links it with the provided test harness to create an executable called `a3q1`.

2. The standard Unix tool `make` is used to automate the separate compilation process. When you ask `make` to build a program, it only rebuilds those parts of the program whose source has changed, and any parts of the program that depend on those parts etc. In order to accomplish this, we tell `make` (via a `Makefile`) which parts of the program depend on which other parts of the program. Make then uses the Unix "last modified" timestamps of the files to decide when a file is older than a file it depends on, and thus needs to be rebuilt. In this problem, you will simulate the dependency-tracking functionality of `make`. We provide a test harness (`main.cc`) that accepts the following commands:

- `target: source` — indicates that the file called `target` depends on the file called `source`
- `touch file` — indicates that the file called `file` has been updated. Your program will respond with

  `file updated at time n`

  where `n` is a number whose significance is explained below
- `make file` — indicates that the file called `file` should be rebuilt from the files it depends on. Your program will respond with the names of all targets that must be rebuilt in order to rebuild `file`.

A target should be rebuilt whenever any target it depends on is newer than the target itself. In order to track ages of files, you will maintain a virtual "clock" (just an `int`) that "ticks" every time you issue the `touch` command (successful or not). When a target is rebuilt, its last-modified time should be set to the current clock time. Every target starts with a last-modified time of 0. For example:

```
a: b
touch b
touch b
touch b
```

will produce the output (on stdout)

```
b updated at time 1
b updated at time 2
b updated at time 3
```

It is not permitted to directly update a target that depends on other targets. If you do, your program should print the following message to **stdout**:

```
a: b
touch a
```

(Output:)

```
Cannot update non-leaf object
```

When you issue the `make file` (build) command, the program should rebuild any files within the dependency graph of `file` that are older than the files they depend on. For example:

```
a: b
a: c
b: d
c: e
touch e
make a
```

will produce the output

```
e updated at time 1
Building c
Building a
```

because file `c` depends on `e`, and `a` depends on `c`. Note that `b` is not rebuilt. The order in which the `Building` messages appear is not important.

A file may depend on at most 10 other files. If you attempt to give a file an 11th dependency, issue the message

```
Max dependencies exceeded
```

on **stdout**, but do not end the program. If you give a file the same dependency more than once, this does not count as a new dependency, i.e., if you give a file a dependency that it already has, the request is ignored. For example, if `a` depends on `b`, then adding `b` as a dependency to `a` a second time has no effect. On the other hand, if `b` also depends on `c`, then `c` may still be added to `a` as an additional dependency, even though `a` already indirectly depends on `c`.

There may be at most 20 files in the system. (Note that files are created automatically when you issue the `:` command, but if a file with the same name already exists, you use the existing file, rather than create a new one.) If you create more than 20 files, issue the message

```
Max targets exceeded
```

on **stdout**, but do not end the program.

You may assume: that the dependency graph among the files will not contain any cycles. You may also assume that all `:` commmands appear before all `touch` commands, so that you do not have to worry about updating a leaf that then becomes a non-leaf because you gave it a dependency.

You may not assume: that the program has only one makefile, even though the provided test harness only manipulates a single `Makefile` object.

A Note on Valid Input: This problem specifies that certain actions on the part of the user are not permitted, and tells you to print out specific messages in those cases. Since the behaviour of the program in such cases is well-defined, these cases do constitute valid input, even if they indicate invalid actions on the part of the user.

Implementation notes

- We will provide skeleton classes in `.h` files that will help you to structure your solution. You may add fields and methods to these, as you deem necessary.
- Do not modify the provided test harness, `main.cc`.
- For your own testing, because the order in which the `Building` messages occurs may be hard to predict, you may wish to modify your `runSuite` script to sort the outputs before comparing them. This does not provide perfect certainty of correctness, but it is probably close enough.

Deliverables

(a) **Due on Due Date 1**: Design a test suite for this program (call the suite file `suiteq2.txt` and zip the suite into `a3q2a.zip`)

(b) **Due on Due Date 3**: Complete the program. Put all of your `.h` and `.cc` files, and your `Makefile`, into `a3q2b.zip`. Your `Makefile` must build an executable called `a3q2`.

3. 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 zero parameter constructor for `Polynomial` should create the `zero` polynomial i.e. a polynomial with no non-zero coefficient. There is also a constructor that takes a `Rational` and an exponent, and creates the corresponding monomial.

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 `a3q3.cc`, which you will find in your `a3/a3q3` 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 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 Polynomial class in C++. Your zip archive should contain at minimum the unchanged file `a3q3.cc`, `polynomial.h`, `polynomial.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`

4. Consider a shared document (e.g., wiki article, Piazza post, Google document, email thread, software under version control) being edited by several users. Each time someone edits the document (e.g., responds to the thread, revises the software, etc.) the result is a new version of the document. Some edits are made to the latest version; other edits are made to older versions of the document. When an older version of the document is edited, we obtain a new "latest" version (in addition to the one we already had)—the document has "branched".

   Edits of this kind can be modelled as a collection of linked lists with a common tail. For simplicity, the linked lists in this problem will contain simply ints, but in reality they could contain document edits, email replies, etc. An example of such a collection of lists is pictured below:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 0
   /                    /
10-     13 -> 12 -> 11-
             /
         14-

heads:  1 10 13 14
```

   In the diagram above, 0 is the original document. 7 is an edit to 0; 6 is an edit to 7; 5 and 11 are both edits to 6; 4 is an edit to 5; 12 is an edit to 11; etc. This example has four "latest" versions: 1, 10, 13, and 14.

   In this problem, you will simulate a sequence of edits to a shared document. Your data structure will be a collection of linked list nodes, where multiple nodes could be pointing to the same "next" node. Along with these nodes, you will have an array (of fixed size 10) of "head" pointers, so that we can track the full collection of "latest" documents.

   One of the biggest challenges with this kind of data structure is proper deallocation. If two nodes point to the same "next" node, their destructors can't both delete that node; that would result in a "double free", which leads to undefined behaviour.

   To solve this problem, we will encode a notion of *ownership* into our nodes. A node that is attached to a head node becomes the head node for that branch, and owns it successor. A node attached to any other node becomes a head node for a new branch, but does not own its successor (because another node already does). When the data structure is deallocated, a node should only delete its successor if it owns its successor. Your `Node` class will, therefore, contain a boolean flag called `ownsSuccessor` that is set accordingly.

   As stated above, your solution will support up to 10 distinct head nodes. Any attempt to create more than 10 distinct heads constitutes undefined behaviour and must not be tested.

   We will provide a test harness that you can use to test your solution. The test harness recognizes the following commands:

   - `print n` (where `n` ranges from `0` to `9`) prints the list starting from the `n`th head. For example (assuming that 13 is head number 2):

```
print 2
13 12 11 6 7 0
```

- `attach n m x` (where `n` ranges from `0` to `9`). Attach a new node with label number `x` such that its `next` field points at the `m`th item (starting at 0) of the list headed at list `n`. For example, under the same assumptions as above:

  ```
  attach 2 3 17   (this would create a new list head, head number 4)
  print 4
  17 6 7 0
  ```

  It is undefined behaviour if `m` is too large to denote a valid list item.

- `mutate n m x` (where `n` ranges from `0` to `9`). Mutate the `m`th item of the `n`th list, such that it now stores `x`. We wouldn't normally see this in a versioning system (i.e., we don't directly mutate old versions), but this command helps us to see that the tails of the lists are indeed shared. For example:

  ```
  mutate 0 2 55
  print 1
  10 2 55 4 5 6 7 0   (assuming 10 is head number 1)
  ```

The data structure will start out with a single node containing 0, and this will be the only head, until edits are made. Note that nodes do not have to contain distinct labels, and that the first node can be mutated to hold something other than 0.

(a) **Due on Due Date 1**: Design the test suite `suiteq4.txt` for this program and zip the suite into `a3q4a.zip`.

(b) **Due on Due Date 2**: Implement this in C++ and place your `Makefile`, `a3q4.cc` and all `.h` and `.cc` files that make up your program in the zip file, `a3q4b.zip`. Your `Makefile` must build an executable called `a3q4`.