CS246—Assignment 3 (Spring 2017)

Due Date 1: Monday, June 12, 5pm
Due Date 2: Monday, June 19, 5pm

Questions 1a, 2a and 3a are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

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, some questions also ask 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: Questions on this assignment will be hand-marked to ensure that you are writing high-quality code, and to ensure that your solutions employ the programming techniques mandated by each question.
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) to simulate a simple hand-held calculator. A calculator has a limited amount of memory, suitable for holding the following values:

   - the current contents of the display;
   - the current computed result (with which the displayed value will eventually be combined);
   - the contents of memory (to emulate the M+/MR/MC functionality on calculators);
   - the most recently pressed operator key (a character: +, -, *, or /);
   - whether or not the most recent operation produced an error.

   You will use the provided `calc.h` file as the basis for your class. **You are not allowed to change `calc.h`.**

   Note on M+/MR/MC: Simple hand-held calculators often give you one int of memory to store a value for later use. The memory initially contains the value 0. Pressing M+ adds the displayed value to the contents of memory, and stores the result in memory (at which point an M customarily appears on the screen). Pressing MR copies the contents of memory to the display. Pressing MC clears the contents of the memory (sets it to 0).

You are to implement the following methods:

- Constructor and copy constructor
- `void digit(int digit)` — adds `digit` (which must be between 0 and 9, inclusive) to the end of the display value.
- `void op(char operator)` — sets the operator field. If the operator field previously contained an operator, it combines display and result with that operator, and stores the result. If not, display is copied to result. In either case, display is cleared (reset to 0).
- `void equals()` — combines result and display using operator, and stores the result in result and display. Clears the operator field.
- `void memPlus(), void memClear(), void memRecall()` — behave like M+, MC and MR as described above
- `bool isError()` — answers true if the error flag is set (because of division by 0).
- `void allClear()` — resets all memory to 0, clears the operator, and resets the error flag.

You are also to implement the output operator, which works as follows:

- if memory contains 0, just print out the value of display;
- if memory is non-zero print out

  `value-of-display M: value-of-memory`

- if the error flag is set, append `E` (a space and an E) to the above.

We have provided the file `main.cc` which you can use to test your calculator class.

A sample session follows:

```
456-123+456=
789
1+1=
7892
AC
1+1=
2
M+
=
2 M: 2
/0
=
0 M: 2 E
AC
=
0
```

Note the second calculation: because the display already contains 789, the 1+1 actually appends the 1 to 789, and requests 7891+1 (7892). This was done for reasons of simplicity.

(a) **Due on Due Date 1:** Design a test suite for this program (call the suite file `suiteq1.txt` and zip the suite into `a3q1a.zip`).

(b) **Due on Due Date 2:** calc.cc

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 valid to directly update a target that depends on other targets. If you do, your program should issue an error message on stdout, as illustrated below:

   ```
   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 error 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 error 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.

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 2**: Complete the program. Put all of your `.h` and `.cc` files, and your `Makefile`, into `a3q2b.zip`. Your Makefile must create an executable named `a3q2`. Note that the executable name is case-sensitive.

3. In this problem, you will implement a `Polynomial` class to represent and perform operations on single variable polynomials. A polynomial can be represented using an integer array with the value at index `idx` used to store the coefficient of the term with exponent `idx` in the polynomial. For example, $9x^3 + 7x + 3$ is represented by the array `{3,7,0,9,...}`. The `max_exp` for this polynomial is 3 as it is the highest non-zero coefficient. Therefore, if an array `coeffs` represents a polynomial, it is guaranteed that `coeffs[max_exp]` is not 0.

   You may assume that the coefficients of polynomials (given as input or produced through operations) can always be represented using a C++ `int` i.e. you need not worry about over and under flow.

   Since the `max_exp` of a `Polynomial` object can change due to operations performed on the polynomial (e.g. multiplication of one polynomial with another), the integer array used to represent the polynomial should be allocated on the heap so that it can be resized accordingly.

   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 the methods and functions. Place your implementation in `polynomial.cc`.

   The constructor `Polynomial(const string &str)` constructs a polynomial from a string. The input operator reads in a string and modifies an existing Polynomial. For both, the string must be in a format specified by the following regular expression:

   ```
   -?  +<term>( +<operator> +<term>)*
   ```

   Where

   - `<operator>` is either `+` or `-`, and
   - `<term>` is `<coefficient><exponent>`, where: `<coefficient>` is a **non-negative** integer and `<exponent>` is empty string if the exponent of this term is 0, `x` if the exponent of this term is 1, and the string `x^<exponent_value>` for exponent values larger than 1.

   Note that any of these can be separated by arbitrary non-zero number of spaces (but not newlines). The following are all valid strings:

   - `0`
   - `1 + x`
   - `1 + 2x^2`
   - `- 1        + 2x^5 + 3x^2`

   The output operator prints polynomials in a similar format as discussed above with the exception that exponents are always printed in increasing order and only non-zero coefficient terms are printed (except the polynomial 0).

   A test harness has been provided in `main.cc`. **Note that this test harness is quite complicated. Make sure you read and understand this code before you start writing your test suite.** Do not make changes to this file. As usual, we may use a different test harness to evaluate the code you submit on Due Date 2.

   (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.

   (b) **Due On Due Date 2**: Full implementation of the Polynomial class in C++. Your zip archive should contain at minimum the files `main.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.