CSCI 104 - Spring 2016 Data Structures and Object Oriented Design

Data Structures and Object Oriented Design

HW3

- Directory name for this homework (case sensitive): `hw3`
  - This directory should be in your `hw_username` repository
  - This directory needs its own `README.md` file
  - You should provide a `Makefile` to compile and run the code for your tests/programs in problems 3, 4, and 5. See instructions in each problem for specific rules.

Skeleton Code

Some skeleton code has been provided for you in the `hw3` folder and has been pushed to the Github repository `homework-resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

```
$ git clone git@github.com:usc-csci104-spring2016/homework-resources
```

Problem 1 (Review Material)

Carefully review Array Lists, Queues, Stacks, Amortized Runtime, Operator Overloading.

Problem 2 (Amortized Analysis, 15%)

Consider the following function:

```
void someclass::somefunc() {
  if (this->n == this->max) {
    bar();
    this->max *= 2;
  } else {
    foo();
  }
  (this->n)++;
}
```

Assume that when someclass is created, `n=0` and `max=10`.

Part (a)

What is the worst-case runtime for somefunc, if bar takes Θ(n^2) time and foo takes Θ(log n) time?

Part (b)

What is the amortized runtime for somefunc, if bar takes Θ(n^2) time and foo takes Θ(log n) time?

Part (c)

What is the amortized runtime for somefunc, if bar takes Θ(n^2) time and foo takes Θ(n log n) time?

Part (d)

Suppose there is another function:

```
void someclass::anotherfunc() {
  if (this->n > 0) (this->n)--;
  if (this->n <= (this->max)/2) {
    bar();
    this->max /= 2;
  } else {
    foo();
  }
}
```

Assume that bar takes Θ(n^2) time and foo takes Θ(log n) time. What is the worst-case sequence of calls to somefunc and anotherfunc? What would be the amortized runtime per function call?

Problem 3 (Copy Constructors and Operator Overloading, 25%)

Copy your `LListInt` class (.h and .cpp files) from your `hw2` folder to your `hw3` folder. Add the following `public` member functions to your .h file and implement them in your .cpp file:

```
  /**
   * Copy constructor (deep copy)
   */
  LListInt(const LListInt& other);

  /**
   * Assignment Operator (deep copy)
   */
```

```
  LListInt& operator=(const LListInt& other);

  /**
   * Concatenation Operator (other should be appended to the end of this)
   */
  LListInt operator+(const LListInt& other) const;

  /**
   * Access Operator
   */
  int const & operator[](int position) const;
```

When you have completed the above functions, you should write a Google Test-based unit test program named llisttest.cpp (using your knowledge of unit-testing) that ensures each of the above member functions work and also use valgrind to check that there are no memory leaks. Add a rule llisttest to your Makefile to compile all the needed code and your test program. We should be able to compile your program by simply typing make llisttest.

Problem 4 (Stacks, 10%)

Use your `LListInt` from Problem 2 to create a Stack data structure for variables of type `int`. Download and use the provided stackint.h as is (do NOT change it). Notice the stack has a LListInt as a data member. This is called **composition**, where we compose/build one class from another, already available class. Essentially the member functions of the `StackInt` class that you write should really just be wrappers around calls to the underlying linked list.

You should think **carefully** about efficiency. **All operations (other than possibly the destructor) should run in O(1)**

Problem 5 (Simple Arithmetic Parser and Evaluator, 50%)

Simple arithmetic expressions consist of integers, the operators PLUS (+), MULTIPLY (*), SHIFTLEFT (<), and SHIFTRIGHT (>), along with parentheses to specify a desired order of operations. The SHIFTLEFT operator indicates you should double the integer immediately following the operator. The SHIFTRIGHT operator indicates you should divide the integer by 2 (rounding down).

Your task is to write a program that will read simple arithmetic expressions from a file, and evaluate and show the output of the given arithmetic expressions.

Simple Arithmetic Expressions are defined formally as follows:

1. Any string of digits is a simple arithmetic expression, namely a positive integer.

2. If Y1, Y2, ..., Yk are simple arithmetic expressions then the following are simple arithmetic expressions:
   - <Y1
   - >Y1
   - (Y1+Y2+Y3+...+Yk)
   - (Y1*Y2*Y3*...*Yk)

Notice that our format rules out the expression 12+23, since it is missing the parentheses. It also rules out (12+34*123) which would have to instead be written (12+(34*123)), so you never have to worry about precedence. This should make your parsing task significantly easier. Whitespace may occur in arbitrary places in arithmetic expressions, but never in the middle of an integer. Each expression will be on a single line.

Examples (the first three are valid, the other three are not):

```
(<<14 *(>>123+333 )) // evaluates to 20328
<>(2 * 1* ( >500000000 + <<0)) // evaluates to 500000000
<>(1 * >3 * 3) // evaluates to 2
((<123*234)    // missing parenthesis
(1337*9001+42)   // mixing operators
(*1138*3720)    // extra *
```

Your program should take the filename in which the formulas are stored as an input parameter. For each expression, your program should output to `cout`, one per line, one of the options:

- `Malformed` if the formula was malformed (did not meet our definition of a formula) and then continue to the next expression.
- An integer equal to the evaluation of the expression, if the expression was well-formed.

Each expression will be on a single line by itself so that you can use getline() and then parse the whole line of text that is returned to you. If you read a blank line, just ignore it and go on to the next. The numbers will always fit into `int` types, but as you can see from the example, they can be pretty large.

While this may be contrary to your expectation of us, you must **not** use recursion to solve this problem. Instead keep a stack on which you push pieces of formula. **Use your `StackInt` class** from Problem 4 for this purpose. Push open parenthesis '(', integers, and operators onto the stack. When you encounter a closing parenthesis ')', pop things from the stack and evaluate them until you pop the open parenthesis '('. Now --- assuming everything was correctly formatted --- compute the value of the expression in parentheses, and push it onto the stack as an integer. When you reach the end of the string, assuming that everything was correctly formatted (otherwise, report an error), your stack should contain exactly one integer, which you can output.

In order to be able to push all those different things (parentheses, operators, and integers) onto the stack, you will need to represent each item with an integral value. It is your choice how to do this mapping. One option is to store special characters (parentheses and operators) as special numbers that you reserve specifically for these purposes. It might make your code more readable to define the mapping of special characters to integers by declaring them as const ints as in:

```
const int OPEN_PAREN = -1;
```

That way, your code can use `OPEN_PAREN` wherever you want to check for that value. Remember that all numbers you are given will be positive, so you can use negative integers for your const values.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your `hw3` directory to your `hw_usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2016/hw_usc-username.git`
5. Go into your hw3 folder `$ cd hw_username/hw3`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.