HW3

- Due Date: Tuesday, September 25, 11:59pm
- Directory name in your github repository for this homework (case sensitive): `hw3`
- Provide a `README.md` file
- Place your answers to problem 2 and 3 in a file named `hw3.txt`
- You should provide a `Makefile` to compile and run the code for your tests/programs in problem 5.
- Warning: the last problem is hard! Start early!

Skeleton Code

Some skeleton code has been provided for you in the `hw3` folder of 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-fall2018/homework-r
```

Problem 1 (Review Material)

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

Problem 2 (Queues, Stacks, Runtime, 10%)

It is possible to implement a queue using two stacks `stack1` and `stack2`, by implementing the functions in the following manner:

- `enqueue (x)`: push `x` on `stack1`.
- `dequeue ()`: if `stack2` is not empty, then pop from it. Otherwise, pop the entire contents of `stack1`, pushing each one onto `stack2`. Now pop from `stack2`.

As always, explain your answers thoroughly.

Part (a)

Starting with an empty queue, indicate the contents of `stack1` and `stack2` after all of the following operations have occurred. Clearly indicate the top and bottom of each stack:

- enqueue 1
- enqueue 2
- dequeue
- enqueue 3
- enqueue 4
- dequeue
- enqueue 5
- enqueue 6

Part (b)

What is the worst-case runtime of `enqueue(x)` and `dequeue()`, assuming the underlying `stack` structure attains constant time for all operations?

Part (c)

Suppose the underlying `stack` structure is poorly implemented, with `push(x)` taking Θ(1) time, and `pop()` taking Θ(`n`) time, where `n` is the number of elements in the stack. Now what is the worst-case runtime of `enqueue(x)` and `dequeue()`?

Problem 3 (Runtime Analysis, 20%)

In Big-Θ notation, analyze the running time of the following pieces of code/pseudo-code. Describe it as a function of the input size (here, `n`). You should **always** explain your work when solving mathematics problems.

Part (a)

```
for (int i = 0; i < n; i ++)
    if (A[i] == 0) {
       for (int j = 1; j < n; j *= 2)
          { /* do something that takes O(1) time */ }
    }
```

Part (b)

```
int tally=0;
for (int i = 1; i < n; i ++)
{
   for (int j = i; j < n; j ++)
   {
        if (j % i == 0)
        {
            for (int k = 1; k < n; k *= 2)
            {
                tally++;
            }
        }
   }
}
```

Part (c)

```
int *a = new int [10];
int size = 10;
for (int i = 0; i < n; i ++)
{
   if (i == size)
   {
       int newsize = size+10;
       int *b = new int [newsize];
       for (int j = 0; j < size; j ++) b[j] = a[j];
       delete [] a;
       a = b;
       size = newsize;
   }
   a[i] = sqrt(i);
}
```

Part (d)

Notice that this code is very similar to what happens if you keep inserting into a vector.

```
int *a = new int [10];
int size = 10;
for (int i = 0; i < n; i ++)
{
   if (i == size)
   {
       int newsize = 2*size;
       int *b = new int [newsize];
       for (int j = 0; j < size; j ++) b[j] = a
       delete [] a;
       a = b;
       size = newsize;
   }
   a[i] = sqrt(i);
}
```

Problem 4 (Stacks, 15%)

Use your `CircularListInt` class from hw2 to create a Stack data structure for variables of type `int`. Alternatively, you can use the STL `list<int>` if you did not finish hw2, at cost of 3 points. Download and use the provided stackint.h as is. Notice the stack has a CircularListInt as a data member (which you'll need to change to `list<int>` if you use that class instead, but don't make any other changes). 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, 55%)

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 (the executable should be named `parser`) 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 expre
   namely a positive integer.
2. If Y1, Y2, ..., Yk are simple arithmetic expressi
   then the following are simple arithmetic expressions for `k>1`:
   - <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,
must **not** use recursion to solve this problem. Instea
stack on which you push pieces of formula. **Use you**
`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.

Submission Link

You can submit your homework here. Please make sure you have read and understood the submission instructions.

WAIT! You aren't done yet. Complete the last section below to ensure you've committed all your code.

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-fall2018/hw-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.