Assignment 8

In this assignment, you will write a new version of the `mystack` class as well as a function called `evaluate()` that will evaluate a postfix expression using a stack.

You will not write a `main()` routine. One will be supplied for you and it will call your `convert()` and `evaluate()` functions.

1. Initial Setup

---

1. Log in to Unix.
2. Run the `setup` script for Assignment 8 by typing:

```
setup 8
```

2. Files We Give You

When you run the setup command, you will get five files: the usual `makefile`; `main.cpp`, which is the main routine that calls your functions; a sample input file called `infix.in`; the correct output generated by that input file, `postfix.key`; and a separate driver program called `stack_test.cpp` that can be used to perform unit testing for the member functions of your stack class.

Note that the files `main.cpp` and `stack_test.cpp` are not the same as the versions of those files used for Assignment 7.

Once again, use `diff` to verify that your output matches that found in `postfix.key`.

To build the program for this assignment, all you need to do is type:

```
make
```

You can run the assignment program with the included input file by typing:

```
./inpost < infix.in
```

To perform unit testing on the functions of your stack class (including those not called by the assignment program), you can run the following commands to build and then run the unit testing program:

```
make stack_test
./stack_test
```

The `stack_test` program will call the various member functions of your stack class and report whether the result was a success or a failure.

To receive full credit for this assignment, your stack class must pass all of the `stack_test` unit tests **and** you must also have the correct output for the assignment.

Running `make clean` will clean up both the `inpost` and `stack_test` executable files.

3. Files You Must Write

You will write the following files:

- `mystack.h` - contains the class definition for the `mystack` class.
- `mystack.cpp` - contains the definitions for member functions of the `mystack` class.
- `inpost.cpp` - contains your `convert()` function.
- `inpost.h` - contains the function prototype for `convert()` so that the `main()` can call it.
- `eval.cpp` - contains your `evaluate()` function.
- `eval.h` - contains the function prototype for `evaluate()` so that the `main()` can call it.

`inpost.cpp` and `inpost.h` are unchanged from Assignment 7. Each of the other files (with the exception of `eval.h`) is described in more detail below. All header files should contain header guards to prevent them from being included multiple times in the same source file.

3.1. The `mystack` class

This new version of the `mystack` class represents a stack of integers implemented as a linked list. Like the other classes we've written this semester, this class should be implemented as two separate files.

The class definition should be placed in a header file called `mystack.h`. Include header guards to make it impossible to `#include` it more than once in the same source code file.

If you look back at the algorithm used to convert infix to postfix for Assignment 7, you will note that you never use the stack to store operands (variables or constants). You only use the stack to store operators or `'('`, which are always single characters. A character is just a number, so it can be saved in a stack designed to hold only integers. That means that this new version of the `mystack` class will still work with your `convert()` function from Assignment 7.

Data Members

The new version of the `mystack` class should contain the following `private` data members:

- a pointer to a `node`, which will either point to the first node in the linked list or be `nullptr`. I'll refer to this data member as the *stack top pointer*.
- a `size_t` variable used to track the number of values current stored in the stack's linked list. I'll refer to this data member as the *stack size*.

In addition to the data members described above, your class definition will need public prototypes for the member functions described below. Note that most of the prototypes are either identical or very similar to the ones that you wrote for Assignment 7; only the internal logic of the function definitions will change.

Member Functions

The definitions for the member functions of the class should be placed in a separate source code file called `mystack.cpp`. Make sure to `#include "mystack.h"` at the top of this file.

The `mystack` class should have the following member functionss (most of which are quite small):

- `mystack::mystack()`

  This "default" constructor for the `mystack` class should initialize a new `mystack` object to an empty stack. When the function ends:

  - The stack size for the new object should be 0.
  - The stack top pointer should be `nullptr`.

- `mystack::mystack(const mystack& x)`

  This "copy constructor" for the `mystack` class should initialize a new `mystack` object to the same string as the existing `mystack` object `x`. When the function ends:

  - The stack size for the new object should be equal to the stack size of the object `x`.
  - If the stack size is 0, the stack top pointer for the new object should be `nullptr`. Otherwise, the stack top pointer should point to the first node of a singly-linked list of nodes. The nodes of the linked list should contain the same values (in the same order) as the linked list of the object `x`.

- `mystack::~mystack()`

  The destructor can simply call the `clear()` member function.

- `mystack& mystack::operator=(const mystack& x)`

  This overloaded copy assignment operator should assign one `mystack` object (the object `x`) to another (the object that called the member function, which is pointed to by `this`). The state of the data members when the function ends should be same as described above for the copy constructor.

- `size_t mystack::size() const`

  This member function should return the stack size.

- `bool mystack::empty() const`

  This member function should return `true` if the stack size is 0. Otherwise, it should return `false`.

- `void mystack::clear()`

  This member function should delete all of the nodes in the stack's linked list and set the stack size back to 0. An easy way to accomplish both things is to repeatedly call the `pop()` member function as long as the stack is not empty.

- `const int& mystack::top() const`

  This member function should return the `value` in the top node of the stack (i.e., the first node in the linked list pointed to by the stack top pointer). You may assume that this function will not be called if the stack is empty.

- `void mystack::push(int value)`

  This member function should push the integer `value` onto the top of the stack.

- `void mystack::pop()`

  This member function should pop the top item off of the stack and delete the node that contained it. You may assume that this function will not be called if the stack is empty.

You are welcome to write additional private member functions (such as a function to copy the linked list that can be called by both the copy constructor and the copy assignment operator).

Important Point

Once again, many of the member functions of this version of the `mystack` class will not be used in Assignment 8. However, you are still required to write them, and you should expect to lose points if they do not work correctly. Thoroughly testing the member functions of this class to make sure that they work is **your responsibility**.

3.2. `eval.cpp`

This file should contain a definition for the following function:

```
int evaluate(const string& postfix)
```

This function evaluates the postfix expression passed to it (which will be the result of your `convert()` function) and returns the calculated value. You may assume the following:

- `postfix` is a valid expression with no leading whitespace.
- All operators/operands have at least one space between them.
- `postfix` may contain any of the operators described in Assignment 7, single character lower case variables, or constants.
- All constants are integers.

- All exponents are >= 0.

In evaluating the expression you must assign the following values to any variables you encounter: `a = 0`, `b = 1`, `c = 2`, etc. See **Hints** below for an easy way to do that.

When performing exponentiation, you must calculate the value with your own code, i.e., you must write your own loop to calculate the value. When performing division, if you encounter a division by 0 error (i.e., if the divisor is 0), do not attempt to divide by 0. Instead, print an error message `"*** Division by 0 ***"` and "push" the result 0 onto the stack.

To process the postfix string and break it into operators/operands, you may find it useful to use the standard library class `stringstream` to process `postfix`. You can create an object of the `stringstream` class from a C++ string, and then use the same operators and member functions to read input from the `stringstream` that you can use with any other input stream (such as `cin` or an input file stream variable).

If you do use the `stringstream` class, your code for `evaluate()` will look something like this:

```
#include <string>
#include <sstream>
...

using std::string;
using std::stringstream;
...

int evaluate(const string& postfix)
{
    string op;
    stringstream ss(postfix);     // Create a stringstream object from the postfix
string.

    // You can now read from the stringstream as if it were standard input. The end
of the
    // string will be treated as the end of input.
    while (ss >> op)
    {
        // op is a C++ string containing the next operator/operand in the postfix
expression.

        ...
    }

    ...
}
```

Postfix evaluation algorithm

Here is a description of the logic for evaluating a postfix expression using a stack.

- Let `eval_stack` be a stack used to hold operands during evaluation.
- Let `postfix` be a string containing the postfix expression tokens.

To evaluate the postfix expression:

Scan the `postfix` string from left to right, extracting and processing one token (operator/operand) at a time:

- If the token is an integer literal, push it on the `eval_stack`.
- If the token is a variable, calculate the value of the variable and push that value on the `eval_stack`.
- If the token is a `'~'` operator, get the top item from the `eval_stack`, pop the stack, apply the operator, and push the result on the `eval_stack`.
- If the token is any other operator, you will need to obtain the right operand by getting the top item of the `eval_stack` and then popping the stack. Repeat those two steps to get the left operand. Perform the arithmetic specified by the operator with the left and right operands and then push the result on to the `eval_stack`.

When you reach the end of the postfix string, the final result of evaluating the postfix expression will be the top (and only) item on the `eval_stack`.

4. Output

The only output generated by your files is the `"*** Division by 0 ***"` message printed in `eval.cpp`. The remaining output from this program is generated by the main routine supplied for you in `inposteval_main.cpp`.

The following is a sample of what the given `main()` function will output when you run your finished program. It is not the complete output.

```
  infix: 1/0
postfix: 1 0 /
*** Division by 0 ***
  value: 0

  infix: ( d +1) *2
postfix: d 1 + 2 *
```

```
  value: 8

  infix: a-e-a
postfix: a e - a -
  value: -4

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
  value: 2

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
  value: 3

  infix:   ~ 3 * ~( a + 1) - b / c ^ 2
postfix: 3 ~ a 1 + ~ * b c 2 ^ / -
  value: 3

  infix: 246 + b /123
postfix: 246 b 123 / +
  value: 246

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
  value: 246
```

5. Hints

- You may find it convenient to calculate the value of each variable (`a` through `z`) simply by using the expression `var - 'a'`, which assumes that the single letter variable name is stored in your program's variable

```
char var;
```

- You will need to use a stack in both your `convert()` function and in your `evaluate()` function. If you look at the algorithm to convert infix to postfix you will note that you never use the stack to store operands (variables or constants). You only use the stack to store operators or `'('`, which are always single characters. Recall that a character is just a number that can be saved in a stack designed to hold only integers. Since the `eval()` function only stores integers in the stack, and since you can store characters as integers in the stack in your `convert()` function, the new version of `mystack` will work for both functions.

- Implement the stack as a linked list as shown in class and described in the notes on linked stacks on the course web site. Your nodes will look like this:

```
struct node
{
    node* next;
    int value;

    node(int value, node* next = nullptr)
    {
        this->value = value;
        this->next = next
    }
};
```

- You may find it easier to write the new `mystack` class first, test it with your existing `convert()` function, convince yourself that it works correctly, and then start `evaluate()`. You might want to modify the main routine by commenting out the call to `evaluate()` until you are ready to write it.