CSCI 241 Assignment 8 Part 2

This document only covers the portion of the assignment that evaluates a postfix expression.

1. Make sure you're in the Assign8 directory.

2. Create a file named ***eval.h***. This file should have header guards (use the #ifndef format from lecture).

   In between the header guards, put in the prototype statement for the evaluate function. This function takes a reference to a constant string as its argument and returns an integer.

   This next part is optional but might be helpful for completing the assignment.

   An extra function can make this assignment a little easier to complete. If you choose to write the extra function, the prototype for the function should also be placed in this .h file.

   The extra function can be used to perform an arithmetic operation. This function should take three arguments: an integer that holds the left operand of the arithmetic operation, a character that holds the arithmetic operator, and an integer that holds the right operand of the arithmetic operation. The function should return an integer – the result of the arithmetic operation. The function can be named whatever you like.

3. Create a file named ***eval.cpp***. Place the necessary `#include` statements at the top of this file.

4. Add the code for the function that performs an arithmetic operation. If you are not adding this extra function, jump ahead to step 5.

   This function should take three arguments: an integer that holds the left operand of the arithmetic operation, a character that holds the arithmetic operator, and an integer that holds the right operand of the arithmetic operation.

   The function should return an integer – the result of the arithmetic operation.

   The function will handle 5 possible arithmetic operations (+, -, *, /, ^). Use a cascading decision statement to test the 2nd argument (the character that holds the arithmetic operator) and apply the specific operation to the two operands (the 1st and 3rd arguments). The result should be returned.

   For the division operator, make sure to test for possible division by 0. If division by 0 is possible, display an error message "*** Division by 0 ***" and return 0. Otherwise, perform the normal division.

   For the exponentiation operator (^), you MUST write a loop to perform the calculation that raises the left operand (1st argument) to the right operand (3rd argument) power.

5. In the ***eval.cpp*** file, add code for the evaluate function. This function takes a reference to a constant string that holds the postfix expression as its argument and returns an integer (the result of the arithmetic).

   The goal right now is not to do the calculation. It's just to make sure that the various parts of the postfix string can be accessed individually.

   One way to do this is to use the recommendation from the assignment write-up about using a stringstream object. The stringstream class will take the string argument and turn it into an input stream (similar to cin) that is pre-filled with data (the postfix string). This will allow you to "read" from the stringstream using the input operator (>>). The input operator stops reading when it finds whitespace (this is why the spaces were added in Assignment 7 when the postfix string was being built in the convert function), which means that each individual operand in the postfix string can be processed one at time.

   The recommended code can be found in Part 3.2 eval.cpp of the assignment write-up.

   In the loop from the recommended code, add a cout statement that will display what op is holding.

   Return an integer value. It does not matter what the value is because it will not be used right now. I returned 15 in my code.

6. Compile and execute the code at the command line. Make sure to use input redirection to direct the infix.in file that was copied by the setup command to the program:

   ./inpost < infix.in

   If the evaluate function can break up the string correctly, the output should resemble:

```
  infix: 2
postfix: 2
2
  value: 15

  infix: 2+a
postfix: 2 a +
2
a
+
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
1
a
+
2
^
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
```

```
1
a
+
~
2
^
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
1
a
+
2
^
~
  value: 15

  infix: 0/0
postfix: 0 0 /
0
0
/
  value: 15

  infix: 1/0
postfix: 1 0 /
1
0
/
  value: 15

  infix: 0/1
postfix: 0 1 /
0
1
/
  value: 15

  infix: c^a
postfix: c a ^
c
a
^
  value: 15

  infix: ~c^a
postfix: c ~ a ^
c
~
a
^
  value: 15
```

```
  infix: c^d
postfix: c d ^
c
d
^
  value: 15

  infix: ~c^d
postfix: c ~ d ^
c
~
d
^
  value: 15

  infix: d+1
postfix: d 1 +
d
1
+
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
d
1
2
*
+
  value: 15

  infix: ( d +1) *2
postfix: d 1 + 2 *
d
1
+
2
*
  value: 15

  infix: a-e-a
postfix: a e - a -
a
e
-
a
-
  value: 15

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

```
a
-
d
~
1
+
/
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
a
2
^
b
~
2
^
+
5
c
-
*
  value: 15

  infix:   ~ 3 * ~( a + 1) - b / c ^ 2
postfix: 3 ~ a 1 + ~ * b c 2 ^ / -
3
~
a
1
+
~
*
b
c
2
^
/
-
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
246
b
123
/
+
  value: 15

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

```
b
123
/
+
  value: 15

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
246
b
+
123
/
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
a
b
c
d
^
/
+
e
~
f
*
-
  value: 15
```

Note: it might be easier to make a copy of the infix.in file to your account. The copy of the file can be edited to remove some of the infix expressions so there isn't as much data to be processed right away. If you do this, substitute the name of the copy of the file after the < when executing the code.

7. In the ***inpost.cpp*** file, add code to the evaluate function.

   In the loop, write a cascading decision statement that tests:
   - If the operand is an integer literal, display a message that says the operand is an integer literal. (Note: one way to test for the operand being an integer literal is test if the 1st character of the op string is a digit)
   - If the operand is a variable, display a message that says the operand is a variable. (Note: test if the 1st character of op is alphabetic)
   - If the operand is a tilde (~), display a message that says the operand is unary negation
   - Otherwise, display a message that says the operand is a binary operator

   Note: make sure to put the curly braces in for each portion of the cascading decision statement.

8. Compile and execute the code at the command line. Don't forget to use input redirection! The output should now resemble:

```
  infix: 2
postfix: 2
2 is an integer literal
  value: 15

  infix: 2+a
postfix: 2 a +
2 is an integer literal
a is a variable
+ is a binary operator
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
1 is an integer literal
a is a variable
+ is a binary operator
2 is an integer literal
^ is a binary operator
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
1 is an integer literal
a is a variable
+ is a binary operator
~ is unary negation
2 is an integer literal
^ is a binary operator
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
1 is an integer literal
a is a variable
+ is a binary operator
2 is an integer literal
^ is a binary operator
~ is unary negation
  value: 15

  infix: 0/0
postfix: 0 0 /
0 is an integer literal
0 is an integer literal
/ is a binary operator
  value: 15

  infix: 1/0
postfix: 1 0 /
```

```
1 is an integer literal
0 is an integer literal
/ is a binary operator
  value: 15

  infix: 0/1
postfix: 0 1 /
0 is an integer literal
1 is an integer literal
/ is a binary operator
  value: 15

  infix: c^a
postfix: c a ^
c is a variable
a is a variable
^ is a binary operator
  value: 15

  infix: ~c^a
postfix: c ~ a ^
c is a variable
~ is unary negation
a is a variable
^ is a binary operator
  value: 15

  infix: c^d
postfix: c d ^
c is a variable
d is a variable
^ is a binary operator
  value: 15

  infix: ~c^d
postfix: c ~ d ^
c is a variable
~ is unary negation
d is a variable
^ is a binary operator
  value: 15

  infix: d+1
postfix: d 1 +
d is a variable
1 is an integer literal
+ is a binary operator
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
d is a variable
1 is an integer literal
2 is an integer literal
```

```
* is a binary operator
+ is a binary operator
  value: 15

  infix: ( d +1) *2
postfix: d 1 + 2 *
d is a variable
1 is an integer literal
+ is a binary operator
2 is an integer literal
* is a binary operator
  value: 15

  infix: a-e-a
postfix: a e - a -
a is a variable
e is a variable
- is a binary operator
a is a variable
- is a binary operator
  value: 15

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
a is a variable
e is a variable
- is a binary operator
a is a variable
- is a binary operator
d is a variable
~ is unary negation
1 is an integer literal
+ is a binary operator
/ is a binary operator
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
a is a variable
2 is an integer literal
^ is a binary operator
b is a variable
~ is unary negation
2 is an integer literal
^ is a binary operator
+ is a binary operator
5 is an integer literal
c is a variable
- is a binary operator
* is a binary operator
  value: 15

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

```
3 is an integer literal
~ is unary negation
a is a variable
1 is an integer literal
+ is a binary operator
~ is unary negation
* is a binary operator
b is a variable
c is a variable
2 is an integer literal
^ is a binary operator
/ is a binary operator
- is a binary operator
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
246 is an integer literal
b is a variable
123 is an integer literal
/ is a binary operator
+ is a binary operator
  value: 15

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
246 is an integer literal
b is a variable
123 is an integer literal
/ is a binary operator
+ is a binary operator
  value: 15

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
246 is an integer literal
b is a variable
+ is a binary operator
123 is an integer literal
/ is a binary operator
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
a is a variable
b is a variable
c is a variable
d is a variable
^ is a binary operator
/ is a binary operator
+ is a binary operator
e is a variable
~ is unary negation
f is a variable
```

```
* is a binary operator
- is a binary operator
  value: 15
```

9. In the ***eval.cpp*** file, add code to the evaluate function.

   Remove all the cout statements.

   Add a variable to the function. This should be a mystack object.

   Handle the case of integer literals first.

   The integer literal is currently stored as a string object (the op string from the recommended loop). It must be converted to an actual integer value. This can be done using another stringstream object or by using atoi.

   If using a stringstream object, follow the basic pattern that was used to create op: create a stringstream object put pass op as the argument, use the new stringstream object and >> to put a value into an integer variable.

   If using atoi, the string object op needs to be converted to a C-Style string. This can be done by using the calling the c_str() method for the op object. This method will return a pointer to a C-Style string that contains a null-terminated version of the string object.

   Push the integer value onto the mystack object.

   Add a cout statement that displays the integer value.

10. Compile and execute the code at the command line. Don't forget to use input redirection! The output should now resemble:

```
  infix: 2
postfix: 2
The integer literal is 2
  value: 15

  infix: 2+a
postfix: 2 a +
The integer literal is 2
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
The integer literal is 1
The integer literal is 2
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
The integer literal is 1
```

```
The integer literal is 2
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
The integer literal is 1
The integer literal is 2
  value: 15

  infix: 0/0
postfix: 0 0 /
The integer literal is 0
The integer literal is 0
  value: 15

  infix: 1/0
postfix: 1 0 /
The integer literal is 1
The integer literal is 0
  value: 15

  infix: 0/1
postfix: 0 1 /
The integer literal is 0
The integer literal is 1
  value: 15

  infix: c^a
postfix: c a ^
  value: 15

  infix: ~c^a
postfix: c ~ a ^
  value: 15

  infix: c^d
postfix: c d ^
  value: 15

  infix: ~c^d
postfix: c ~ d ^
  value: 15

  infix: d+1
postfix: d 1 +
The integer literal is 1
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
The integer literal is 1
The integer literal is 2
  value: 15
```

```
  infix: ( d +1) *2
postfix: d 1 + 2 *
The integer literal is 1
The integer literal is 2
  value: 15

  infix: a-e-a
postfix: a e - a -
  value: 15

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
The integer literal is 1
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
The integer literal is 2
The integer literal is 2
The integer literal is 5
  value: 15

  infix:   ~ 3 * ~( a + 1) - b / c ^ 2
postfix: 3 ~ a 1 + ~ * b c 2 ^ / -
The integer literal is 3
The integer literal is 1
The integer literal is 2
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
The integer literal is 246
The integer literal is 123
  value: 15

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
The integer literal is 246
The integer literal is 123
  value: 15

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
The integer literal is 246
The integer literal is 123
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
  value: 15
```

11. In the ***eval.cpp*** file, add code to the evaluate function.

    Handle the case of a variable.

    The variables are all single letters where a = 0, b = 1, c = 2, d = 3, etc… Find the integer value associated with the variable. Part 5 Hints of the assignment write-up shows a simple way to find the integer value.

    Push the integer value onto the mystack object.

    Add a cout statement that displays the integer value.

12. Compile and execute the code at the command line. Don't forget to use input redirection! The output should now resemble:

```
  infix: 2
postfix: 2
The integer literal is 2
  value: 15

  infix: 2+a
postfix: 2 a +
The integer literal is 2
a is equal to 0
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
The integer literal is 1
a is equal to 0
The integer literal is 2
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
The integer literal is 1
a is equal to 0
The integer literal is 2
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
The integer literal is 1
a is equal to 0
The integer literal is 2
  value: 15

  infix: 0/0
postfix: 0 0 /
The integer literal is 0
The integer literal is 0
  value: 15
```

```
  infix: 1/0
postfix: 1 0 /
The integer literal is 1
The integer literal is 0
  value: 15

  infix: 0/1
postfix: 0 1 /
The integer literal is 0
The integer literal is 1
  value: 15

  infix: c^a
postfix: c a ^
c is equal to 2
a is equal to 0
  value: 15

  infix: ~c^a
postfix: c ~ a ^
c is equal to 2
a is equal to 0
  value: 15

  infix: c^d
postfix: c d ^
c is equal to 2
d is equal to 3
  value: 15

  infix: ~c^d
postfix: c ~ d ^
c is equal to 2
d is equal to 3
  value: 15

  infix: d+1
postfix: d 1 +
d is equal to 3
The integer literal is 1
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
d is equal to 3
The integer literal is 1
The integer literal is 2
  value: 15

  infix: ( d +1) *2
postfix: d 1 + 2 *
d is equal to 3
The integer literal is 1
```

```
The integer literal is 2
  value: 15

  infix: a-e-a
postfix: a e - a -
a is equal to 0
e is equal to 4
a is equal to 0
  value: 15

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
a is equal to 0
e is equal to 4
a is equal to 0
d is equal to 3
The integer literal is 1
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
a is equal to 0
The integer literal is 2
b is equal to 1
The integer literal is 2
The integer literal is 5
c is equal to 2
  value: 15

  infix:   ~ 3 * ~( a + 1) - b / c ^ 2
postfix: 3 ~ a 1 + ~ * b c 2 ^ / -
The integer literal is 3
a is equal to 0
The integer literal is 1
b is equal to 1
c is equal to 2
The integer literal is 2
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

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

```
postfix: 246 b + 123 /
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
a is equal to 0
b is equal to 1
c is equal to 2
d is equal to 3
e is equal to 4
f is equal to 5
  value: 15
```

13. In the ***eval.cpp*** file, add code to the evaluate function.

    Handle the case of unary negation (the ~ operator).

    The top value on the mystack object needs to be removed from the stack. Remember that this is a two-step process because the value that is removed is needed to be able to evaluate the expression. Make sure to use the top and pop methods to remove the integer.

    Push the negated value of the integer that was at the top of the stack onto the mystack object.

    Add a cout statement that display the negated value of the integer that was at the top of the stack.

14. Compile and execute the code at the command line. Don't forget to use input redirection! The output should now resemble:

```
  infix: 2
postfix: 2
The integer literal is 2
  value: 15

  infix: 2+a
postfix: 2 a +
The integer literal is 2
a is equal to 0
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
The integer literal is 1
a is equal to 0
The integer literal is 2
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
```

```
The integer literal is 1
a is equal to 0
unary negation results in 0
The integer literal is 2
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
The integer literal is 1
a is equal to 0
The integer literal is 2
unary negation results in -2
  value: 15

  infix: 0/0
postfix: 0 0 /
The integer literal is 0
The integer literal is 0
  value: 15

  infix: 1/0
postfix: 1 0 /
The integer literal is 1
The integer literal is 0
  value: 15

  infix: 0/1
postfix: 0 1 /
The integer literal is 0
The integer literal is 1
  value: 15

  infix: c^a
postfix: c a ^
c is equal to 2
a is equal to 0
  value: 15

  infix: ~c^a
postfix: c ~ a ^
c is equal to 2
unary negation results in -2
a is equal to 0
  value: 15

  infix: c^d
postfix: c d ^
c is equal to 2
d is equal to 3
  value: 15

  infix: ~c^d
postfix: c ~ d ^
c is equal to 2
```

```
unary negation results in -2
d is equal to 3
  value: 15

  infix: d+1
postfix: d 1 +
d is equal to 3
The integer literal is 1
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
d is equal to 3
The integer literal is 1
The integer literal is 2
  value: 15

  infix: ( d +1) *2
postfix: d 1 + 2 *
d is equal to 3
The integer literal is 1
The integer literal is 2
  value: 15

  infix: a-e-a
postfix: a e - a -
a is equal to 0
e is equal to 4
a is equal to 0
  value: 15

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
a is equal to 0
e is equal to 4
a is equal to 0
d is equal to 3
unary negation results in -3
The integer literal is 1
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
a is equal to 0
The integer literal is 2
b is equal to 1
unary negation results in -1
The integer literal is 2
The integer literal is 5
c is equal to 2
  value: 15

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

```
The integer literal is 3
unary negation results in -3
a is equal to 0
The integer literal is 1
unary negation results in -1
b is equal to 1
c is equal to 2
The integer literal is 2
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
The integer literal is 246
b is equal to 1
The integer literal is 123
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
a is equal to 0
b is equal to 1
c is equal to 2
d is equal to 3
e is equal to 4
unary negation results in -4
f is equal to 5
  value: 15
```

15. In the ***eval.cpp*** file, add code to the evaluate function.

    Handle the case of a binary operator (+, -, *, /, ^).

    Remove the integer that is at the top of the mystack object. Remember this is a two-step process. **This is the right operand for the binary operator**.

    Remove the integer that is at the top of the mystack object. Remember this is a two-step process. **This is the left operand for the binary operator**.

    Use the function coded in step 4 to perform the arithmetic operation OR write the code to perform the specific arithmetic operation here. No matter how the arithmetic operation is performed make sure the result of the operation is saved in an integer variable.

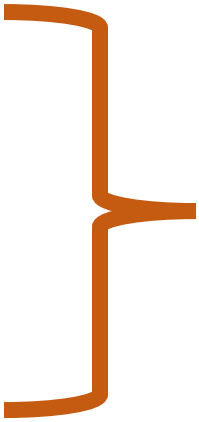
    Push the result of the operation onto the mystack object.

    Add a cout statement to display the operation that is performed and its result.

16. Compile and execute the code at the command line. Don't forget to use input redirection! Make sure to check that the division by 0 error appears in a couple of the expressions. The output should now resemble:

```
  infix: 2
postfix: 2
The integer literal is 2
  value: 15

  infix: 2+a
postfix: 2 a +
The integer literal is 2
a is equal to 0
2 + 0 = 2
  value: 15

  infix: (1+a) ^2
postfix: 1 a + 2 ^
The integer literal is 1
a is equal to 0
1 + 0 = 1
The integer literal is 2
1 ^ 2 = 1
  value: 15

  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
The integer literal is 1
a is equal to 0
1 + 0 = 1
unary negation results in -1
The integer literal is 2
-1 ^ 2 = 1
```

```
  value: 15

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
The integer literal is 1
a is equal to 0
1 + 0 = 1
The integer literal is 2
1 ^ 2 = 1
unary negation results in -1
  value: 15

  infix: 0/0
postfix: 0 0 /
The integer literal is 0
The integer literal is 0
*** Division by 0 ***
0 / 0 = 0
  value: 15

  infix: 1/0
postfix: 1 0 /
The integer literal is 1
The integer literal is 0
*** Division by 0 ***
1 / 0 = 0
  value: 15

  infix: 0/1
postfix: 0 1 /
The integer literal is 0
The integer literal is 1
0 / 1 = 0
  value: 15

  infix: c^a
postfix: c a ^
c is equal to 2
a is equal to 0
2 ^ 0 = 1
  value: 15

  infix: ~c^a
postfix: c ~ a ^
c is equal to 2
unary negation results in -2
a is equal to 0
-2 ^ 0 = 1
  value: 15

  infix: c^d
postfix: c d ^
c is equal to 2
d is equal to 3
```



Division by 0 Error

```
2 ^ 3 = 8
  value: 15

  infix: ~c^d
postfix: c ~ d ^
c is equal to 2
unary negation results in -2
d is equal to 3
-2 ^ 3 = -8
  value: 15

  infix: d+1
postfix: d 1 +
d is equal to 3
The integer literal is 1
3 + 1 = 4
  value: 15

  infix:   d +1 *2
postfix: d 1 2 * +
d is equal to 3
The integer literal is 1
The integer literal is 2
1 * 2 = 2
3 + 2 = 5
  value: 15

  infix: ( d +1) *2
postfix: d 1 + 2 *
d is equal to 3
The integer literal is 1
3 + 1 = 4
The integer literal is 2
4 * 2 = 8
  value: 15

  infix: a-e-a
postfix: a e - a -
a is equal to 0
e is equal to 4
0 - 4 = -4
a is equal to 0
-4 - 0 = -4
  value: 15

  infix: (a-e-a)/(  ~d + 1)
postfix: a e - a - d ~ 1 + /
a is equal to 0
e is equal to 4
0 - 4 = -4
a is equal to 0
-4 - 0 = -4
d is equal to 3
unary negation results in -3
```

```
The integer literal is 1
-3 + 1 = -2
-4 / -2 = 2
  value: 15

  infix: (a^2 + ~b ^ 2) * (5 - c)
postfix: a 2 ^ b ~ 2 ^ + 5 c - *
a is equal to 0
The integer literal is 2
0 ^ 2 = 0
b is equal to 1
unary negation results in -1
The integer literal is 2
-1 ^ 2 = 1
0 + 1 = 1
The integer literal is 5
c is equal to 2
5 - 2 = 3
1 * 3 = 3
  value: 15

  infix:   ~ 3 * ~( a + 1) - b / c ^ 2
postfix: 3 ~ a 1 + ~ * b c 2 ^ / -
The integer literal is 3
unary negation results in -3
a is equal to 0
The integer literal is 1
0 + 1 = 1
unary negation results in -1
-3 * -1 = 3
b is equal to 1
c is equal to 2
The integer literal is 2
2 ^ 2 = 4
1 / 4 = 0
3 - 0 = 3
  value: 15

  infix: 246 + b /123
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
1 / 123 = 0
246 + 0 = 246
  value: 15

  infix: (   246 + ((  b /123)   )   )
postfix: 246 b 123 / +
The integer literal is 246
b is equal to 1
The integer literal is 123
1 / 123 = 0
246 + 0 = 246
```

```
  value: 15

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
The integer literal is 246
b is equal to 1
246 + 1 = 247
The integer literal is 123
247 / 123 = 2
  value: 15

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
a is equal to 0
b is equal to 1
c is equal to 2
d is equal to 3
2 ^ 3 = 8
1 / 8 = 0
0 + 0 = 0
e is equal to 4
unary negation results in -4
f is equal to 5
-4 * 5 = -20
0 - -20 = 20
  value: 15
```

17. In the ***eval.cpp*** file, remove all the cout statements that have been added in each part of the cascading decision statement.

18. Finally, add the last bit of code to the evaluate function.

    Remove the integer that is at the top of the mystack object. Still a two-step process. **This is the result of evaluating the postfix expression**.

    Return the result of the postfix expression.

19. Compile and execute the code at the command line. Don't forget to use input redirection! The output should be correctly formatted now and it should display the correct result for each expression.

```
  infix: 2
postfix: 2
  value: 2

  infix: 2+a
postfix: 2 a +
  value: 2

  infix: (1+a) ^2
postfix: 1 a + 2 ^
  value: 1
```

```
  infix: ~(1+a) ^2
postfix: 1 a + ~ 2 ^
  value: 1

  infix: ~((1+a)^2)
postfix: 1 a + 2 ^ ~
  value: -1

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

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

  infix: 0/1
postfix: 0 1 /
  value: 0

  infix: c^a
postfix: c a ^
  value: 1

  infix: ~c^a
postfix: c ~ a ^
  value: 1

  infix: c^d
postfix: c d ^
  value: 8

  infix: ~c^d
postfix: c ~ d ^
  value: -8

  infix: d+1
postfix: d 1 +
  value: 4

  infix:   d +1 *2
postfix: d 1 2 * +
  value: 5

  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

  infix: (  ( 246 +  b)  /123)
postfix: 246 b + 123 /
  value: 2

  infix: a+b/c^d-~e*f
postfix: a b c d ^ / + e ~ f * -
  value: 20
```

20. Professor McMahon included a link to the expected output as one of the files that was copied to your account by the setup command. The output from this file (postfix.key) can be electronically compared to the output from your program by executing the code and redirecting the output to an output file and then using the diff command to compare the two output files.

    ./inpost < infix.in > output.txt

    diff output.txt postfix.key

    You can use any name for the output file that you want (just make sure that it's not the same as any of the files that already exist).

    If the output files are the same, the diff command will not produce any output.