makefile

```
############################################################
# Program:
#     Week 02, STACK
#     Brother Ercanbrack, CS235
# Author:
#     Yurii Vasiuk
# Summary:
#     Custom made stack and its application                    82/100
# Time:
#     10 hours
############################################################

############################################################
# The main rule
############################################################
a.out: stack.h week02.o infix.o
       g++ -o a.out week02.o infix.o
       tar -cf week02.tar *.h *.cpp makefile

############################################################
# The individual components
#      week02.o       : the driver program
#      infix.o        : the logic for the infix --> postfix program
############################################################
week02.o: stack.h infix.h week02.cpp
       g++ -c week02.cpp

infix.o: stack.h infix.h infix.cpp
       g++ -c infix.cpp
```

Commented [ES1]: Late -10; Infix-To-Postfix formatting issues, and expressions with "(" is incorrect. -8

infix.h

```
/************************************************************************
* Header:
*    INFIX
* Summary:
*    This will contain just the prototype for the convertInfixToPostfix()
*    function
* Author
*    Yurii Vasiuk
************************************************************************/

#ifndef INFIX_H
#define INFIX_H

/*****************************************************
 * TEST INFIX TO POSTFIX
 * Prompt the user for infix text and display the
 * equivalent postfix expression
 *****************************************************/
void testInfixToPostfix();

/*****************************************************
 * TEST INFIX TO ASSEMBLY
 * Prompt the user for infix text and display the
 * resulting assembly instructions
 *****************************************************/
void testInfixToAssembly();

#endif // INFIX_H
```

stack.h

```
/************************************************************************
* Header:
```

```
*    Stack
* Summary:
*    Custom made Stack, analofous to the STD stack.
*
* Author
*    Br. Helfrich, Yurii Vasiuk
************************************************************************/

#ifndef STACK_H
#define STACK_H

#include <cassert>

/*********************************************
* STACK
* A class that holds stuff
*********************************************/
template <typename T>
class Stack
{
public:
   // default constructor : empty and kinda useless
   Stack() : _top(-1), _capacity(0), _data(0x00000000) {} //done

   // copy constructor : copy it
   Stack(const Stack & rhs) throw (const char *); //done

   // non-default constructor : pre-allocate
   Stack(int capacity) throw (const char *); //done

   // destructor : free everything
   ~Stack()        { if (_capacity) delete[] _data; } //done

   // copy data from one stack to another
   Stack <T> & operator=(const Stack <T> & rhs) throw (const char *); //done

   // is the stack currently empty
   bool empty() const  { return _top == -1; } //done

   // how many items are currently in the stack?
   int size() const    { return _top + 1; } //done

   // return the current capasity of the stack
   int capacity() const { return _capacity; } //done

   // add an item to the stack
   void push(const T & t) throw (const char *); //done

   // remove an item from the end of the stack
   void pop() throw (const char *); //done

   // return the item that is at the end of the stack
   T & top() throw (const char *);

private:
   T * _data;          // dynamically allocated array of T
   int _top;      // how many items are currently in the Stack?
   int _capacity;      // how many items can I put on the Stack before full?
};


/*****************************************
* STACK :: COPY CONSTRUCTOR
*****************************************/
template <typename T>
Stack <T> :: Stack(const Stack <T> & rhs) throw (const char *)
{
   assert(rhs._capacity >= -1);

   // do nothing if there is nothing to do
   if (rhs._capacity == 0)
   {
      _capacity = 0;
      _top = -1;
      _data = 0x00000000;
      return;
   }

   // attempt to allocate
```

```
   try
   {
      _data = new T[rhs._capacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }

   // copy over the stuff
   assert(rhs._top >= -1 && rhs._top < rhs._capacity);
   _capacity = rhs._capacity;
   _top = rhs._top;
   for (int i = 0; i <= _top; i++)
      _data[i] = rhs._data[i];
}

/*******************************************
* STACK : NON-DEFAULT CONSTRUCTOR
* Preallocate the stack to "capacity"
*******************************************/
template <typename T>
Stack <T> :: Stack(int capacity) throw (const char *)
{
   assert(capacity >= 0);

   // do nothing if there is nothing to do
   if (capacity == 0)
   {
      _capacity = 0;
      _top = -1;
      _data = 0x00000000;
      return;
   }

   // assign capacity and numItems
   _capacity = capacity;
   _top = -1;
   // allocate
   try
   {
      _data = new T[capacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }

}

/*****************************************
* STACK :: ASSIGNMENT OPERATOR
* Copy items from one stack to another.  Return the new stack by reference!!
*****************************************/
template <typename T>
Stack <T> & Stack <T> :: operator=(const Stack <T> & rhs) throw (const char *)
{
   assert(rhs._capacity >= 0);

   // attempt to allocate
   if (_capacity < rhs._capacity)
   {
      try
      {
         delete[] _data;  // prevents memory leak!!!
         _data = new T[rhs._capacity];
      }
      catch (std::bad_alloc)
      {
         throw "ERROR: Unable to allocate buffer";
      }
   }

   // copy over the stuff
   assert(rhs._top >= -1 && rhs._top < rhs._capacity);
   _capacity = rhs._capacity;
   _top = rhs._top;
   for (int i = 0; i <= _top; i++)
      _data[i] = rhs._data[i];
```

```
   // the rest needs to be filled with the default value for T
   for (int i = (_top + 1); i < _capacity; i++)
      _data[i] = T();

   return *this;
}

/***********************************************
* STACK :: PUSH
* Push an item on the top of the stack
***********************************************/
template <typename T>
void Stack <T> :: push(const T & t) throw (const char *)
{
   if (_capacity == 0) // case 1 (0 capacity)
   {
      _capacity = 1;
      _top = 0;
      _data = new T[_capacity];
      _data[_top] = t;
   }
   else               // case 2 (more than 0 capacity)
   {
      if (_capacity == (_top + 1))
      {
         // temporary holder
         T* temp = _data;
         // double the vector's capacity
         try
         {
            _capacity *= 2;
            _data = new T[_capacity];
            // copy the content of the temp into the newly allocated vector
            for (int i = 0; i <= _top; i++)
               _data[i] = temp[i];
            // free the temp memory
            delete[] temp;
         }
         catch (std::bad_alloc)
         {
            throw "ERROR: Unable to allocate a new buffer for Stack";
         }
      }
      // the capacity has been handled; add an item to the end
      _data[++_top] = t;
   }
}

/***********************************************
* STACK :: POP
* Remove an item from the end of the stack
***********************************************/
template <typename T>
void Stack <T> :: pop() throw (const char *)
{
   if (_top == -1)
      throw "ERROR: Unable to pop from an empty Stack";
   else
      _top--;
}

/***********************************************
* STACK :: TOP
* Return an item from the top of the stack
***********************************************/
template <typename T>
T & Stack <T> ::top() throw (const char *)
{
   if (_top == -1)
      throw "ERROR: Unable to reference the element from an empty Stack";
   else
      return _data[_top];
}

#endif // CONTAINER_H
```

infix.cpp

```
/***********************************************************************
 * Module:
 *    Week 02, Stack
 *    Brother Helfrich, CS 235
 * Author:
 *    Yurii Vasiuk
 * Summary:
 *    This program will implement the testInfixToPostfix()
 *    and testInfixToAssembly() functions
 ************************************************************************/

#include <iostream>    // for ISTREAM and COUT
#include <string>      // for STRING
#include <cassert>     // for ASSERT
#include "stack.h"     // for STACK
using namespace std;

// i added the libraries
//#include <boost/algorithm/string.hpp>
#include <vector>

using namespace std;
//using namespace boost;

/*****************************************************
 * CONVERT INFIX TO POSTFIX
 * Convert infix equation "5 + 2" into postifx "5 2 +"
 *****************************************************/
string convertInfixToPostfix(const string & infix)
{
    string postfix;

   char token, topToken;
   Stack <char> ops;
   const string BLANK = " ";
   postfix.append(BLANK);

   for (int i = 0; i < infix.length(); i++)
   {
      token = infix[i];

      switch (token)
      {
      // append blanks
      case ' ':
         if (postfix[postfix.length() - 1] != ' ')
            postfix.append(BLANK);
         break;
      // handle parenthesis
      case '(':
         ops.push(token);
         break;
      case ')':
         while (true)
         {
            topToken = ops.top();
            ops.pop();
            if (topToken == '(')
               break;
            postfix.append(BLANK);
            postfix += topToken;
         }
      // handle operators
      case '^':                        // 1
         ops.push(token);
         break;
      case '*': case '/': case '%':  // 2
         while (true)
         {
            if (ops.empty() || ops.top() == '(' ||
               ops.top() == '+' || ops.top() == '-')
            {
               ops.push(token);
               break;
            }
            else
```

```
            {
               topToken = ops.top();
               ops.pop();
               if (postfix[postfix.length() - 1] != ' ')
                  postfix.append(BLANK);
               postfix += topToken;
            }
         }
         break;
      case '+': case '-':            // 3
         while (true)
         {
            if (ops.empty() || ops.top() == '(')
            {
               ops.push(token);
               break;
            }
            else
            {
               topToken = ops.top();
               ops.pop();
               if (postfix[postfix.length() - 1] != ' ')
                  postfix.append(BLANK);
               postfix += topToken;
            }
         }
         break;
      // handle operands
      default:
         postfix += token;
         break;
      }
   }

   // pop remaining on the stack operators
   while (true)
   {
      if (ops.empty())
         break;
      topToken = ops.top();
      ops.pop();
      postfix.append(BLANK);
      if (topToken != ')')
         postfix += topToken;
   }

    return postfix;
}

/*************************************************
 * TEST INFIX TO POSTFIX
 * Prompt the user for infix text and display the
 * equivalent postfix expression
 *************************************************/
void testInfixToPostfix()
{
   string input;
   cout << "Enter an infix equation.  Type \"quit\" when done.\n";

   do
   {
      // handle errors
      if (cin.fail())
      {
         cin.clear();
         cin.ignore(256, '\n');
      }

      // prompt for infix
      cout << "infix > ";
      getline(cin, input);

      // generate postfix
      if (input != "quit")
      {
         string postfix = convertInfixToPostfix(input);
         cout << "\tpostfix: " << postfix << endl << endl;
      }
   }
```

```
   while (input != "quit");
}

/*****************************************
 * CONVERT POSTFIX TO ASSEMBLY
 * Convert postfix "5 2 +" to assembly:
 *     LOAD 5
 *     ADD 2
 *     STORE VALUE1
 *****************************************/
string convertPostfixToAssembly(const string & postfix)
{
   string assembly;

   return assembly;
}

/*************************************************
 * TEST INFIX TO ASSEMBLY
 * Prompt the user for infix text and display the
 * resulting assembly instructions
 *************************************************/
void testInfixToAssembly()
{
   string input;
   cout << "Enter an infix equation.  Type \"quit\" when done.\n";

   do
   {
      // handle errors
      if (cin.fail())
      {
         cin.clear();
         cin.ignore(256, '\n');
      }

      // prompt for infix
      cout << "infix > ";
      getline(cin, input);

      // generate postfix
      if (input != "quit")
      {
         string postfix = convertInfixToPostfix(input);
         cout << convertPostfixToAssembly(postfix);
      }
   }
   while (input != "quit");

}
```

week02.cpp

```
/***********************************************************************
* Program:
*    Week 03, Stack
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise the Stack class.  When you
*    submit your program, this should not be changed in any way.  That being
*    said, you may need to modify this once or twice to get it to work.
************************************************************************/

#include <iostream>    // for CIN and COUT
#include <string>      //
#include "stack.h"     // your Stack class should be in stack.h
#include "infix.h"     // for testInfixToPostfix() and testInfixToAssembly()
using namespace std;


// prototypes for our four test functions
void testSimple();
void testPush();
void testPop();
void testErrors();
```

```
// To get your program to compile, you might need to comment out a few
// of these. The idea is to help you avoid too many compile errors at once.
// I suggest first commenting out all of these tests, then try to use only
// TEST1.  Then, when TEST1 works, try TEST2 and so on.
#define TEST1   // for testSimple()
#define TEST2   // for testPush()
#define TEST3   // for testPop()
#define TEST4   // for testErrors()

/*****************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 *****************************************************************/
int main()
{
   // menu
   cout << "Select the test you want to run:\n";
   cout << "\t1. Just create and destroy a Stack.\n";
   cout << "\t2. The above plus push items onto the Stack.\n";
   cout << "\t3. The above plus pop items off the stack.\n";
   cout << "\t4. The above plus exercise the error handling.\n";
   cout << "\ta. Infix to Postfix.\n";
   cout << "\tb. Extra credit: Infix to Assembly.\n";

   // select
   char choice;
   cout << "> ";
   cin  >> choice;
   switch (choice)
   {
      case 'a':
         cin.ignore();
         testInfixToPostfix();
         break;
      case 'b':
         cin.ignore();
         testInfixToAssembly();
         break;
      case '1':
         testSimple();
         cout << "Test 1 complete\n";
         break;
      case '2':
         testPush();
         cout << "Test 2 complete\n";
         break;
      case '3':
         testPop();
         cout << "Test 3 complete\n";
         break;
      case '4':
         testErrors();
         cout << "Test 4 complete\n";
         break;
      default:
         cout << "Unrecognized command, exiting...\n";
   }

   return 0;
}

/****************************************
 * TEST SIMPLE
 * Very simple test for a Stack: create and destroy
 ****************************************/
void testSimple()
{
#ifdef TEST1
   try
   {
      // Test 1.a: bool Stack with default constructor
      cout << "Create a bool Stack using default constructor\n";
      Stack <bool> s1;
      cout << "\tSize:     " << s1.size()                  << endl;
      cout << "\tCapacity: " << s1.capacity()              << endl;
      cout << "\tEmpty?    " << (s1.empty() ? "Yes" : "No") << endl;

      // Test 1.b: double Stack with non-default constructor
      cout << "Create a double Stack using the non-default constructor\n";
```

```
      Stack <double> s2(10 /*capacity*/);
      cout << "\tSize:     " << s2.size()                  << endl;
      cout << "\tCapacity: " << s2.capacity()              << endl;
      cout << "\tEmpty?    " << (s2.empty() ? "Yes" : "No") << endl;

      // Test 1.c: copy the Stack using the copy constructor
      {
         cout << "Create a double Stack using the copy constructor\n";
         Stack <double> s3(s2);
         cout << "\tSize:     " << s3.size()                  << endl;
         cout << "\tCapacity: " << s3.capacity()              << endl;
         cout << "\tEmpty?    " << (s3.empty() ? "Yes" : "No") << endl;
      }

      // Test 1.d: copy the Stack using the assignment operator
      cout << "Copy a double Stack using the assignment operator\n";
      Stack <double> s4(2);
      s4 = s2;
      cout << "\tSize:     " << s4.size()                  << endl;
      cout << "\tCapacity: " << s4.capacity()              << endl;
      cout << "\tEmpty?    " << (s4.empty() ? "Yes" : "No") << endl;
   }
   catch (const char * sError)
   {
      cout << sError << endl;
   }
#endif //TEST1
}

/*****************************************
 * TEST PUSH
 * Add a whole bunch of items to the stack.  This will
 * test the stack growing algorithm. It requires
 * Stack::push
 *****************************************/
void testPush()
{
#ifdef TEST2
   try
   {
      // create
      Stack <int> s;

      {
         Stack <int> sTemp;
         cout << "Enter numbers, type 0 when done\n";
         int number;
         do
         {
            cout << "\t> ";
            cin  >> number;
            if (number)
               sTemp.push(number);
         }
         while (number);

         // display how big it is
         cout << "After filling the Stack, the size is:\n";
         cout << "\tSize:     " << sTemp.size() << endl;
         cout << "\tCapacity: " << sTemp.capacity() << endl;
         cout << "\tEmpty?    " << (sTemp.empty() ? "Yes" : "No") << endl;

         // copy the stack to s and delete sTemp
         s = sTemp;
         while (!sTemp.empty())
            sTemp.pop();
      }

      // display how big it is
      cout << "After copying the Stack to a new Stack, the size is:\n";
      cout << "\tSize:     " << s.size() << endl;
      cout << "\tCapacity: " << s.capacity() << endl;
      cout << "\tEmpty?    " << (s.empty() ? "Yes" : "No") << endl;
   }
   catch (const char * sError)
   {
      cout << sError << endl;
   }
```

```
#endif // TEST2
}

#ifdef TEST3
/*****************************************
 * DISPLAY
 * Display the contents of the stack. We will
 * assume that T is a data-type that has the
 * insertion operator defined.
 *****************************************/
template <class T>
ostream & operator << (ostream & out, Stack <T> rhs) throw (const char *)
{
   // we need to make a copy of the stack that is backwards
   Stack <T> backwards;
   while (!rhs.empty())
   {
      backwards.push(rhs.top());
      rhs.pop();
   }

   // now we will display this one
   out << "{ ";
   while (!backwards.empty())
   {
      out << backwards.top() << ' ';
      backwards.pop();
   }
   out << '}';

   return out;
}
#endif // TEST3

/*****************************************
 * TEST POP
 * We will test both Stack::pop() and Stack::top()
 * to make sure the stack looks the way we expect
 * it to look.
 *****************************************/
void testPop()
{
#ifdef TEST3
   // create
   cout << "Create a string Stack with the default constructor\n";
   Stack <string> s;

   // instructions
   cout << "\tTo add the word \"dog\", type +dog\n";
   cout << "\tTo pop the word off the stack, type -\n";
   cout << "\tTo see the top word, type *\n";
   cout << "\tTo quit, type !\n";

   // interact
   char instruction;
   string word;
   try
   {
      do
      {
         cout << "\t" << s << " > ";
         cin  >> instruction;
         switch (instruction)
         {
            case '+':
               cin >> word;
               s.push(word);
               break;
            case '-':
               s.pop();
               break;
            case '*':
               cout << s.top() << endl;
               break;
            case '!':
               cout << "\tSize:     " << s.size() << endl;
               cout << "\tCapacity: " << s.capacity() << endl;
               cout << "\tEmpty?    " << (s.empty() ? "Yes" : "No") << endl;
               break;
```

```
            default:
               cout << "\tInvalid command\n";
         }
      }
      while (instruction != '!');
   }
   catch (const char * error)
   {
      cout << error << endl;
   }
#endif // TEST3
}

/*****************************************
 * TEST ERRORS
 * Numerous error conditions will be tested
 * here, including bogus popping and the such
 *****************************************/
void testErrors()
{
#ifdef TEST4
   // create
   Stack <char> s;

   // test using Top with an empty stack
   try
   {
      s.top();
      cout << "BUG! We should not be able to top() with an empty stack!\n";
   }
   catch (const char * error)
   {
      cout << "\tStack::top() error message correctly caught.\n"
           << "\t\"" << error << "\"\n";
   }

   // test using Pop with an empty stack
   try
   {
      s.pop();
      cout << "BUG! We should not be able to pop() with an empty stack!\n";
   }
   catch (const char * error)
   {
      cout << "\tStack::pop() error message correctly caught.\n"
           << "\t\"" << error << "\"\n";
   }
#endif // TEST4
}
```

Test Bed Results

```
a.out:

------------------------------------------------------------
Starting Test 1

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > 1
Create, destory, and copy a Stack
   > Create a bool Stack using default constructor
   >    Size:     0
   >    Capacity: 0
   >    Empty?    Yes
   > Create a double Stack using the non-default constructor
   >    Size:     0
   >    Capacity: 10
   >    Empty?    Yes
   > Create a double Stack using the copy constructor
   >    Size:     0
   >    Capacity: 10
```

```
   >    Empty?    Yes
   > Copy a double Stack using the assignment operator
   >    Size:     0
   >    Capacity: 10
   >    Empty?    Yes
   > Test 1 complete

Test 1 passed.
------------------------------------------------------------

------------------------------------------------------------
Starting Test 2

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > 2
Create an integer Stack with the default constructor\n
This test will exerise the grow() function
   > Enter numbers, type 0 when done
   >    > 9
   >    > 8
   >    > 7
   >    > 6
   >    > 5
   >    > 4
   >    > 3
   >    > 2
   >    > 1
   >    > 0
The capacity should be a power of two
   > After filling the Stack, the size is:
   >    Size:     9
   >    Capacity: 16
   >    Empty?    No
We will copy the stack and destroy the old.
   > After copying the Stack to a new Stack, the size is:
   >    Size:     9
   >    Capacity: 16
   >    Empty?    No
   > Test 2 complete

Test 2 passed.
------------------------------------------------------------

------------------------------------------------------------
Starting Test 3

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > 3
This will test pushing, poping, toping, and copying of a stack
   > Create a string Stack with the default constructor
   >    To add the word "dog", type +dog
   >    To pop the word off the stack, type -
   >    To see the top word, type *
   >    To quit, type !

Test pushing items onto the stack
   >    { } > +Genesis
   >    { Genesis } > +Exodus
   >    { Genesis Exodus } > +Levidicus
   >    { Genesis Exodus Levidicus } > +Numbers
   >    { Genesis Exodus Levidicus Numbers } > +Deuteronomy

Test accessing the last item on the stack with top()
   >    { Genesis Exodus Levidicus Numbers Deuteronomy } > *
   > Deuteronomy

Test popping items off the stack
```

```
   >    { Genesis Exodus Levidicus Numbers Deuteronomy } > -
   >    { Genesis Exodus Levidicus Numbers } > -
   >    { Genesis Exodus Levidicus } > -
   >    { Genesis Exodus } > -
   >    { Genesis } > *
   > Genesis

Test pushing items after we have poped a few
   >    { Genesis } > +Matthew
   >    { Genesis Matthew } > +Mark
   >    { Genesis Matthew Mark } > *
   > Mark
   >    { Genesis Matthew Mark } > +Luke
   >    { Genesis Matthew Mark Luke } > +John
   >    { Genesis Matthew Mark Luke John } > +Acts
   >    { Genesis Matthew Mark Luke John Acts } > -

Now we will look at the size and capacity.
Since the maximum number of items was 5, there should be a capacity of 8
   >    { Genesis Matthew Mark Luke John } > !
   >    Size:     5
   >    Capacity: 8
   >    Empty?    No
   > Test 3 complete

Test 3 passed.
---------------------------------------------------------------

---------------------------------------------------------------
Starting Test 4

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > 4

Test to make sure we cannot top off of an empty stack
   >    Stack::top() error message correctly caught.
   >    "ERROR: Unable to reference the element from an empty Stack"

Test to make sure we cannot pop off of an empty stack
   >    Stack::pop() error message correctly caught.
   >    "ERROR: Unable to pop from an empty Stack"
   > Test 4 complete

Test 4 passed.
---------------------------------------------------------------

---------------------------------------------------------------
Starting Test 5

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > a
   > Enter an infix equation.  Type "quit" when done.

Simple test where order of operations is not verified
   > infix > 4 + 6
   >    postfix:  4 6 +
   >

If - is wrong, you are not taking order of operations into account
   > infix > a + b * c ^ d - e
   >    postfix:  a b c d ^ * + e -
   >

Another test exercising the order of operations
   > infix > a ^ b + c * d
   >    postfix:  a b ^ c d * +
   >
```

```
This test will verify that tokens can consist of more than one letter
   > infix > 3.14159 * diameter
   >    postfix:  3.14159 diameter *
   >

This test exercises the code's ability to see where one token begins
and another token ends. The best way to do this is to create a Token
class that defines the extraction operator. The rules for the end of
of a variable are quite different than the rules for the end of a number
   > infix > 4.5+a5+.1215  +   1
   > \tpostfix:  4.5a5 +.1215 +  1 +\n
Exp: \tpostfix:  4.5 a5 + .1215 + 1 +\n
   >

This is really no different than the previous test
   > infix > pi*r^2
   > \tpostfix:  pir2 ^ *\n
Exp: \tpostfix:  pi r 2 ^ *\n
   >

This too is no different than the previous test
   > infix > (5.0  /  .9)*(fahrenheit - 32)
   > \tpostfix:  5.0 .9 / )fahrenheit 32 -  *\n
Exp: \tpostfix:  5.0 .9 / fahrenheit 32 - *\n
   >
   > infix > quit

Test 5 failed.
---------------------------------------------------------------

---------------------------------------------------------------
Starting Test 6

   > Select the test you want to run:
   >    1. Just create and destroy a Stack.
   >    2. The above plus push items onto the Stack.
   >    3. The above plus pop items off the stack.
   >    4. The above plus exercise the error handling.
   >    a. Infix to Postfix.
   >    b. Extra credit: Infix to Assembly.
   > > b
   > Enter an infix equation.  Type "quit" when done.

Simple test to see if a single triplet of assembly statements can be generated
   > infix > 4 + 6
   > infix >
Exp: \tLOAD 4\n
   >
Exp: \tADD 6\n
   >
Exp: \tSTORE VALUE1\n

Another simple triplet
   >
Exp: infix >
3.14159 * diameter

vas14001@byui.edu
```