Programming Assignment #7

Learning objective : You are to gain experience using a single class to implement a solution to more than one problem and experience modifying a class that implements a data structure.

Exercise 1

Write a main program called ***palindrome.cpp.*** This program will accept as input a single group of character and your output will be a single answer, either the string is a palindrome or not a palindrome. A palindrome is a word or string of characters that is spelled the same backward or forward.

Write a program that uses a stack object to determine if a string is a palindrome (i.e. the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

Go ahead and start your program by reading in a C-style string from standard input, using the getline function. You may assume a limit of 100 characters on the string (although this can be written, with a little more effort, to accept any size string). Your algorithm must make use of a stack (of type char). Use the Deitel implementation of the Stack from "stack.h" (you **don't** need to change this file).

Ignore spacing and punctuation, as well as special characters and digits (i.e. only count letters as part of a palindrome, and account for upper/lower case. For example, 'B' and 'b' are matching letters).

Sample runs: (user input underlined)

```
Please enter a string:
> ABCDEFGHGFEDCBA

"ABCDEFGHGFEDCBA"  IS a palindrome


Please enter a string:
> The quick brown fox

"The quick brown fox"  is NOT palindrome


Please enter a string:
> Cigar? Toss it in a can. It is so tragic.

"Cigar? Toss it in a can. It is so tragic."  IS a palindrome\
```

Want some palindromes to test you can visit the site www.palindrome.com.

Implementation Details:

- If the user enters a string of characters that contain spaces or special characters you need to reject it (not try to evaluate it) and issue an appropriate error message.

- You need to ignore the case (upper or lower case) which means that you should change the input string to either upper or lower case letters.
- Since you are reading data into a C-style string to begin, you may use any of the libraries <iostream>, <cstring>, and <cctype>, if you like. (The first, of course, for I/O, and the other two, since they deal with C-style strings and characters).

Exercise 2

Modify the List class (file `list.h` so that it has two more functions, which will allow inserts and removes from anywhere in the linked list. Your functions should be called:

- `insertMiddle`
- `removeMiddle`

Your functions should have all the same features as the given `insert` and `remove` functions, except that yours each have one extra parameter. The second parameter on each of your functions should be of type `int`, representing the **position** at which to insert (or delete). Sample calls for a list of integers:

```
L.insertMiddle(345, 5);      // attempts to insert the value 345
                             //  as the 5th item in the list

L.removeMiddle(x, 10);       // attempts to delete the 10th item in the
                             //  list and captures its value into x.
```

For `insertMiddle`, if the position number is larger than the number of items in the list, just insert the item at the **back**. If it's too small (i.e. 0 or less), insert at the **front**. For `removeMiddle`, return false if the position is invalid (without removing anything).

I've modified the menu program of Figure 21.5 so that it adds in two more menu options for testing these features. You can use it to test your class: menu7.cpp

Submitting:

Submit Files palindrom.cpp and list.h

```
// * ******************************************************************
  // * * Taken from Deitel & Associates, Inc. and Prentice Hall          *
  // * * Deitel & Deitel How to Probram in C++ 3rd Edition.              *
  // * * Figure 21.13: stack.h                                          *
  // * * Template Stack Class definition derived from class List.       *
  // * *                                                                *
  // * * Must have the files list.h included in order to function       *
  // * * properly.                                                      *
  // * * Additional comments added by Dr. David A. Gaitros              *
  // * ******************************************************************

#ifndef STACK_H
#define STACK_H

#include "list.h"  // List class definition

  // * ******************************************************************
  // * *  Class definition.                                             *
  // * ******************************************************************

template< class STACKTYPE >
class Stack : private List< STACKTYPE > {

public:
  // * ******************************************************************
  // * * This function "push" actually  calls the List function         *
  // * * insertAtFront to place an item at the front of the list.       *
  // * ******************************************************************

   void push( const STACKTYPE &data )
   {
      insertAtFront( data );

   }


  // * ******************************************************************
  // * *  This function "pop" returns an item from the front of the     *
 // * *  stack by simtaneously removing the item from the front of the  *
  // * *  list and returning the value.  This simulates the "pop"       *
  // * *  feature of a stack.                                           *
  // * ******************************************************************


   bool pop( STACKTYPE &data )
   {
      return removeFromFront( data );

   }
  // * ******************************************************************
  // * *  A Boolean function "isStackEmpty" is simply a return value    *
```

```
  // * *  from the List class this->isEmpty.  Essentially, does the      *
  // * *  head of the list point to a null pointer.                       *
  // * ***********************************************************************



   bool isStackEmpty() const
   {
      return this->isEmpty();

   }
  // * ***********************************************************************
  // * *  printStack() simply calls the List class member function       *
  // * *  print() which will print the contents of the list if the       *
  // * *  list is not empty.                                              *
  // * ***********************************************************************

   void printStack() const
   {
      this->print();

   }

};
  // * ***********************************************************************
  // * *     E N D  S T A C K  C L A S S                                   *
  // * ***********************************************************************

#endif

/**************************************************************************
 * (C) Copyright 1992-2003 by Deitel & Associates, Inc. and Prentice      *
 * Hall. All Rights Reserved.                                             *
 *                                                                        *
 * DISCLAIMER: The authors and publisher of this book have used their     *
 * best efforts in preparing the book. These efforts include the          *
 * development, research, and testing of the theories and programs        *
 * to determine their effectiveness. The authors and publisher make       *
 * no warranty of any kind, expressed or implied, with regard to these    *
 * programs or to the documentation contained in these books. The authors *
 * and publisher shall not be liable in any event for incidental or       *
 * consequential damages in connection with, or arising out of, the       *
 * furnishing, performance, or use of these programs.                     *
 *************************************************************************/
```

```
  // * ******************************************************************
  // * * Taken from Deitel & Associates, Inc. and Prentice Hall          *
  // * * Deitel & Deitel How to Probram in C++ 3rd Edition.              *
  // * * Figure 21.4: list.h                                            *
  // * * Template ListNode Class definition.                            *
  // * *                                                                 *
  // * * This class must inherit the class listnode.h in order to       *
  // * * function properly.                                             *
  // * *                                                                 *
  // * * Additional comments and modifications added by                 *
  // * *   Dr. David A. Gaitros                                         *
  // * ******************************************************************
#ifndef LIST_H
#define LIST_H

#include <iostream>
using namespace std;
using std::cout;
  // * ******************************************************************
  // * *  The #include <new> ensures that the operators "new" and       *
  // * *  "delete" and other functions of types composing the           *
  // * *  fundamentals of C++ memory management are provided to the     *
  // * *  class.                                                        *
  // * ******************************************************************

#include <new>
#include "listnode.h"  // ListNode class definition

template< class NODETYPE >
class List {

public:
   List();      // constructor
   ~List();     // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   void insertMiddle( const NODETYPE &, int );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool removeMiddle (NODETYPE &, int);
   bool isEmpty() const;
   void print() const;

private:
   ListNode< NODETYPE > *firstPtr;  // pointer to first node
   ListNode< NODETYPE > *lastPtr;   // pointer to last node


   ListNode< NODETYPE > *getNewNode( const NODETYPE & );

};


template< class NODETYPE >
// * ******************************************************************
// * * Default constructor, set the head and tail pointer to null      *
```

```
// * * indicating a null or empty list.                                  *
// * ***********************************************************************

List< NODETYPE >::List()
   : firstPtr( 0 ),
     lastPtr( 0 )
{


}
// * ***********************************************************************
// * *  Class destructor.  Goes through the entire list and removes       *
// * *  each ListNode one at a time.  When you are done with a            *
// * *  destructor the list should be empty.                              *
// * ***********************************************************************

template< class NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) {    // List is not empty
 // ***********************************************************************
 //  COMMENTED OUT.  USED IN TESTING TO SEE IF WE ARE DESTROYING THE  *
 //  NODES.                                                           *
 // ***********************************************************************
 //      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr;
      ListNode< NODETYPE > *tempPtr;

      while ( currentPtr != 0 )         // delete remaining nodes
      {
         tempPtr = currentPtr;

 // ***********************************************************************
 //  COMMENTED OUT.  USED IN TESTING TO SEE IF WE ARE DEALLOCATING    *
 //  THE NODES PROPERLY.                                              *
 // ***********************************************************************
 //         cout << tempPtr->data << '\n';

         currentPtr = currentPtr->nextPtr;
         delete tempPtr;

      }

   }

//   cout << "All nodes destroyed\n\n";

}

// * ***********************************************************************
// * * insertAtFront.  Takes a NODETYPE record and inserts it at the  *
// * *   front of the list.  If the list is empty, it places it where *
// * *   both the first and last pointer are pointing to it.          *
// * ***********************************************************************
```

```
// insert node at front of list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;

   else {  // List is not empty
      newPtr->nextPtr = firstPtr;
      firstPtr = newPtr;

   }

}


// * ******************************************************************
// * *  insertAtBack:  Inserts a NODETYPE record at the end of the    *
// * *    list. If the list is empty, it points the first and last    *
// * *    pointer to this record.                                     *
// * ******************************************************************

template< class NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;

   else {  // List is not empty
      lastPtr->nextPtr = newPtr;
      lastPtr = newPtr;

   } // end else

} // end function insertAtBack

} // end function insertMiddle


// delete node from front of list
// * ******************************************************************
// * *  removeFromFront:  Delete a NODETYPE from the front of the     *
// * *      list.  If the list is empty it returns a false.           *
// * *      Also, you must be concerned if you are removing the       *
// * *      last node.                                                *
// * ******************************************************************

template< class NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() )  // List is empty
      return false;  // delete unsuccessful
```

```
   else {
      ListNode< NODETYPE > *tempPtr = firstPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else
         firstPtr = firstPtr->nextPtr;

      value = tempPtr->data;  // data being removed
      delete tempPtr;

      return true;  // delete successful

   } // end else

} // end function removeFromFront

// * ******************************************************************
// * * removeFromBack:  Similar to removerFromFront.  If the list     *
// * *     is empty you return a false.  Else you remove the node     *
// * *     pointed to by the last pointer.  You must also check to    *
// * *     to see if you are creating an empty list by removing       *
// * *     the last node.                                             *
// * ******************************************************************


template< class NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() )
      return false;  // delete unsuccessful

   else {
      ListNode< NODETYPE > *tempPtr = lastPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr;

         lastPtr = currentPtr;
         currentPtr->nextPtr = 0;

      } // end else

      value = tempPtr->data;
      delete tempPtr;

      return true;  // delete successful

   } // end else

} // end function removeFromBack
```

```
// * ******************************************************************
// * *  isEmpty() returns true if the firstPtr is null.               *
// * *   otherwise it returns a false.                                *
// * ******************************************************************

template< class NODETYPE >
bool List< NODETYPE >::isEmpty() const
{
   return firstPtr == 0;

} // end function isEmpty

// * ******************************************************************
// * *   getNewNode().  Not really needed but returns a pointer       *
// * *      to a new node of type NODETYPE.  Usually part of a        *
// * *      template.                                                 *
// * ******************************************************************

// return pointer to newly allocated node
template< class NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
   const NODETYPE &value )
{
   return new ListNode< NODETYPE >( value );

} // end function getNewNode

// * ******************************************************************
// * *   print(); Prints the list if it is not empty.                 *
// * *   NOTE: Here you  must overload the << operator to work with   *
// * *   NODETYPE.                                                    *
// * ******************************************************************

template< class NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) {
      cout << "The list is empty\n\n";
      return;

   } // end if

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;

   } // end while

   cout << "\n\n";

} // end function print
```

```
#endif

/**************************************************************************
 * (C) Copyright 1992-2003 by Deitel & Associates, Inc. and Prentice      *
 * Hall. All Rights Reserved.                                             *
 *                                                                        *
 * DISCLAIMER: The authors and publisher of this book have used their     *
 * best efforts in preparing the book. These efforts include the          *
 * development, research, and testing of the theories and programs        *
 * to determine their effectiveness. The authors and publisher make       *
 * no warranty of any kind, expressed or implied, with regard to these    *
 * programs or to the documentation contained in these books. The authors *
 * and publisher shall not be liable in any event for incidental or       *
 * consequential damages in connection with, or arising out of, the       *
 * furnishing, performance, or use of these programs.                     *
 *************************************************************************/
```