Assignment 4

Inheritance and Template Class Development

Building the User Management Application

First look at the code provided to you:

For this application we are going to be using several classes that are being provided to you. Three classes are used to manage the menu used by the application. These are the **command, menuItem,** and **menuList** classes. One is used to keep track of user ids and passwords. Another class, **userListCommand**, is a subclass of the **command** class and is used to implement part of our application. This is also an abstract class. There are also two subclasses of the **userListCommand** class called **insertCommand** and **printCommand**. These two classes implement the action for inserting a new item into the **userList** object and printing the current items in the **userList**.

These are the provided classes:

- **command** – This is an abstract class that is part of the menu support classes. This is in file **command.h**.
- **menuItem** – This is used to build a menu item for the menu application. A subclass of the **command** class is passed to the **menuItem** when it is constructed. The menu item is displayed by the **menuList** (below) and calls a **command** object. The **menuItem** class is in files **menuList.h** and **menuList.cpp**.
- **menuList** – This displays the menu items. When the menu item is selected by the user the corresponding command is called to actual perform the menu action. The **menuList** class is in files **menuList.h** and **menuList.cpp**.
- **userListCommand** – This abstract class inherits from **command**. It implements some functions and used by the application. This class is in files **userListCommands.h** and **userListCommands.cpp**.
- **insertCommand** – This class inherits from **userListCommand** and implements the **execute** function. The **execute** function implements an insert operation to add a new entry to the **userList**. This class is in files **userListCommands.h** and **userListCommands.cpp**.
- **printCommand** – This class inherits from **userListCommand** and displays all of the entries in the **userList**. This class is in files **userListCommands.h** and **userListCommands.cpp**.
- **userList** – This class implements a simple system to keep track of user ids and their associated passwords. Each user id has one password. The list of entries is kept in sorted order, sorted by user id. The **userList** class is in files **userList.h** and **userList.cpp**.
- Part of the main program has been provided for use in file **assignment4.cpp**.

Here is a copy of the starter code for **assignment4.cpp** that is provided:

```
#include <iostream>
#include <string>

#include "userList.h"
#include "menuList.h"
#include "userListCommands.h"

using namespace std;

int main()
{
	// create the phone book
	userList users;

	// create the commands
	printCommand print(users, std::cin, std::cout);
	insertCommand insert(users, std::cin, std::cout);

	// build the menuList and menuItems
	menuList menu("Userid management:");
	menuItem insertItem('i', "insert", insert);
	menuItem printItem('p',"print", print);

	// add the menuItem values to the menulist
	menu.add(insertItem);
	menu.add(printItem);

	std::cout << "Starting the Userid management application\n\n";

	// run the menu
	menu.start();

	return 0;
}
```

The statement:

```
userList users;
```

Creates the user management object (class **userList**).

The statements

```
printCommand print(users, std::cin, std::cout);
```

```
insertCommand insert(users, std::cin, std::cout);
```

Create an object of type **printCommand** and an object of type **insertCommand**. These classes are defined in the **userListCommands.h** file and implemented in **userListCommands.cpp**.

The statements:

```
menuList menu("Userid management:");
menuItem insertItem('i', "insert", insert);
menuItem printItem('p',"print", print);
```

Tell the **menuList** that we want the menu prompt to be "*Userid management:*".

The two **menuItem** objects will be the insert and print items. Next we add these to the menu:

```
// add the menuItem values to the menulist
menu.add(insertItem);
menu.add(printItem);
```

Finally we start the application with:

```
// run the menu
menu.start();
```

When you run the application the following menu will be displayed:

```
Userid management:
i insert
p print
q exit the menu
```

Note that the "i" and "insert" text were specified in the **menuItem** we created earlier.

When the user types "**i**" and hits the **Enter** key the **menuList** object will call the insertCommand's **execute** member function. This member function contains the following code:

```
void insertCommand::execute()
{
      std::string userName = read("Enter the user id to be added:");
      std::string password = read("Enter the password for user " +
                                  userName + ":");

      bool result = userIds.add(userName, password);

      if (result)
      {
            display("User " + userName+ " was successfully added");
      }
      else
      {
```

```
            display("User " + username +
                    " could not be added and may already exist");
        }
}
```

This above code will insert a new user id and password into the **userList** object **userIds**. This code makes use of the **read** and **display** member functions in class **userListCommand**. The **insertCommand** inherits from **userListCommand**. The **userListCommand** class inherits from **command**. The **menuList** class's **add** member function expects a reference to an object of type **command** to be passed to it. Since **insertCommand** is a subclass of a subclass of **command** we can pass it to the **add** member function of the **menuList** class.

Part 1: Add the missing menu actions.

Currently the application only supports insert and print. In this part we want to add new commands for find, update, and erase functions.

You will need to create three classes that inherit from **userListCommand**. You should use the **display** member function of **userListCommand** to display output to the user. The **read** member function is used to both display a prompt to the user and to read in a value from the user. See the existing **insertCommand** for an example of how these are used. Your new commands must NOT directly read from cin or write to cout. You should use the **read** and **display** member functions. If you need to read or write on your own you need to use std::istream **in** that is inherited from **userListCommand** or std::ostream **out** that is inherited from **userListCommand**. We are doing this to keep the application as independent from using hard coded input and output objects as possible.

The three new classes should be defined in the **userListCommands.h** file and implemented in the **userListCommands.cpp** file. They must all inherit from **userListCommand**. The three new classes are:

- **findCommand**. This will read in the user id (using the **read** member function). If will then use the **userList** class's **find** member function to see if the user id is found or not. An appropriate message must be displayed using the **display** member function. See the sample output for examples of the expected output for the **findCommand**.
- **updateCommand**. This will read in the user id. If the user id exists the user will be asked to enter a new password and the password in the **userList** will be updated with the new password. See the sample output for examples of the expected output.
- **eraseCommand**. The user enters a user id and it is erased. See the sample output for examples of the expected output.

Sample output for part 1 (bold text is user input):

```
Starting the Userid management application

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
i[Enter]
Enter the user id to be added:
albert[Enter]
Enter the password for user albert:
password[Enter]
User albert was successfully added

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
i[Enter]
Enter the user id to be added:
zelda[Enter]
Enter the password for user zelda:
IamZelda[Enter]
User zelda was successfully added

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
i[Enter]
Enter the user id to be added:
bobby[Enter]
Enter the password for user bobby:
12345678[Enter]
User bobby was successfully added
```

```
Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
p[Enter]
Current list of users and passwords

             User id                 Password
              albert                 password
               bobby                 12345678
               zelda                 IamZelda

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
f[Enter]
Enter the user id to search for:
bob[Enter]
User bob was not found

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
f[Enter]
Enter the user id to search for:
zelda
User zelda was found

Userid management:
f find
i insert
u update
e erase
```

```
p print
q exit the menu
u[Enter]
Enter the user id to be updated:
bob[Enter]
User bob was not found

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
u[Enter]
Enter the user id to be updated:
zelda[Enter]
Enter the new password for user zelda:
lkJds38*$[Enter]
The password for user zelda was successfully updated

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
p[Enter]
Current list of users and passwords

             User id                 Password
              albert                 password
                bobby                12345678
                zelda               lkJds38*$

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
e[Enter]
Enter the user id to be erased:
```

```
bob[Enter]
User bob was not found

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
e[Enter]
Enter the user id to be erased:
bobby[Enter]
The entry for user bobby was successfully erased

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
p[Enter]
Current list of users and passwords

             User id                 Password
              albert                 password
               zelda                lkJds38*$

Userid management:
f find
i insert
u update
e erase
p print
q exit the menu
q[Enter]
```

Part 2: Creating a template stack class

For part 2 you need to create a stack class that implements the following functions:

```
stack();
stack(const stack &other);
```

```
const stack& operator=(const stack &rhs);
virtual ~stack();
const Type& top() const;
Type& top();
bool empty() const;
std::size_t size() const;
void push(const Type &value);
void pop();
void debug() const;
void debug(std::ostream &out) const;
```

The stack class needs to be a template class. The stack needs to dynamically create (with push) and destroy (with pop) nodes that contain the data for the stack and contain a pointer to the next entry in the stack. The node class needs to be a template class as well. We will be making the stack class a friend class of the node class.

The **stack.h** file contains the friend class definition and the node class declarations. You will need to implement all of the above constructors, destructor, and member functions. You can create additional private member functions as needed for your implementation.

Here is a brief summary of the various stack functions (listed above)

`stack()`

This creates a new stack and initializes the stack to no entries. The stack needs to have a count of the number of valid values currently in the stack. This will be 0 when the stack is first created.

`stack(const stack &other)`

This is the copy constructor. It will make copies of each of the notes in the “other” stack and add them do the new stack being created. Make sure the order of the items is preserved in the copy.

`const stack& operator=(const stack &rhs)`

The assignment operator is a superset of the behavior in the stack copy constructor. You need to first remove any nodes of the stack being copied into (the this object). You then need to create copies of all of the nodes in the rhs stack and add them to the this stack, again preserving the ordering.

Make sure you return back a copy of the current (this) stack when you are done. Also make sure you check for self assignment. In the case of self assignment your assignment operator should just return without doing anything to the stack being copied into.

~stack()

This is the stack destructor. Make sure you delete all nodes in the stack. To not allow your program to have memory leaks.

const Type& top() const

This returns the top element in the stack. You can assume the stack has one or more entries. If you want you can also add an assert to your program:

```
#include <cassert>

// lots of code goes here
const Type& top() const
{
   assert(firstNode != nullptr); // firstNode should be replaced by
                      // the pointer to the top of the stack.
}
```

Type& top()

This returns the top element in the stack. This will be the same as the code in your other **top()** member function. The only difference is that this returns a reference that allows the caller to change the data in the top element.

bool empty() const

Returns `true` if the stack is empty and `false` if the stack has at least one entry.

std::size_t size() const

Returns the number of entries in the stack.

void push(const Type &value)

This creates a new node that contains value and adds it to the top of the stack.

void pop()

This removes the top entry from the stack. Make sure you delete the node. If the stack is empty pop should do nothing.

void debug() const
void debug(std::ostream &out) const

The debug member functions are provided. They assume the count of the number of valid entries is in a member variable named **count** and that the pointer to the top of the stack is named **firstNode**. If you

use different names for these data members of the stack class you will have to update the **debug(ostream &)** member function.

Make sure you write code to thoroughly test your stack before moving on to part 3.

Part 3: Update your user id management application

In part 3 you are going to update the **userList** class to include a **printReverse** function, actually two functions:

```
void printReverse(std::ostream &out) const;
void printReverse() const { printReverse(std::cout);}
```

Note that the **printReverse** that takes no parameters, is inlined, and has been written for you. It simply calls the other version of **printReverse**.

In your **printReverse(ostream &out)** member function you need to create a stack and add the entries from the user id list to the stack. See the **print(ostream &out)** member function for an example of iterating through all of the user ids and their passwords.

Here is a copy of the loop (it is a range based for loop):

```
for (const auto &elem : ids)
{
      out << std::setw(NAME_LEN) << elem.first;
      out << std::setw(PASSWORD_LEN) << elem.second << std::endl;
}
```

Note that the elem type is specified as auto. That means C++ will decide the type based on the type of elements in the map named ids.

If you look at the map in the .h file you will see the following:

```
std::map<std::string, std::string> ids;
```

The map contains two strings. The first one is the key (the user id) and the second one is the data (the password). When the key/value part is retrieved from the map the type of object returned is actually of type:

```
std::pair<const std::string, std::string>
```

So for an element **elem** of type **std::pair** we get the first value with **elem.first** and the second one with **elem.second**.

When you create the stack the type for the stack template class will be **std::pair<const std::string, std::string>**. In the range based for loop you will retrieve the element from the map and push in onto the stack.

After you have processed all of the elements in the map the stack will contain all of the elements, but in reverse order. You can then create a loop to go through the elements of the stack and print the top one and then pop it off of the stack.

Here is some sample output for part 3 (input in bold)::

```
Starting the Userid management application

Userid management:
f find
i insert
u update
e erase
p print
r print in reverse order
q exit the menu
i[Enter]
Enter the user id to be added:
sally[Enter]
Enter the password for user sally:
jxqE98&2[Enter]
User sally was successfully added

Userid management:
f find
i insert
u update
e erase
p print
r print in reverse order
q exit the menu
i[Enter]
Enter the user id to be added:
albert[Enter]
Enter the password for user albert:
password[Enter]
User albert was successfully added

Userid management:
f find
i insert
u update
e erase
p print
r print in reverse order
```

```
q exit the menu
p[Enter]
Current list of users and passwords

User id             Password
albert              password
sally               jxqE98&2

Userid management:
f find
i insert
u update
e erase
p print
r print in reverse order
q exit the menu
r[Enter]
Current list of users and passwords in reverse order

User id             Password
sally               jxqE98&2
albert              password

Userid management:
f find
i insert
u update
e erase
p print
r print in reverse order
q exit the menu
q[Enter]
```

Comments, variable names and so on.

Make sure your variable names have meaningful names and that you have included appropriate comments in your application.

You must include the following as your first two comments:

```
// Assignment 4 for CS 1337.013
// Programmer: <Your name goes here>
// Description:
//  <Comments here to describe what the application does>
```

Make sure you replace **<Your name goes here>** with your name. Make sure you include comments at the top of your cpp file that describes what you are doing in the application code.

You need to ensure you have comments throughout your program and not just the comments at the top of the program.

Your program must have the name **assignment4.cpp**. You only need to upload the final version of your program to eLearning. Make sure you upload all of the .h and .cpp files.

Grading:

20% of the grade will for comments you have in your program, for having meaningful variable names, and for formatting your code as is shown in the C++ text book.

80% will be for the program itself. Make sure you do all of the parts and that your output matches what is shown in the sample runs.

If you have any questions at all about this assignment let me know.