Lab8

Lab8

EE 599 Lab8
Spring 2019 Nazarian **100 Points**

Student ID: ____________________ **Name: ____________________**

Assigned: Wednesday, Feb. 27th

Due: Friday March 8th at 11:59pm.

Late submissions will be accepted for two days after the deadline with 15% penalty per day: For each day submissions between 12am and 1am: 2% penalty, between 1-2am: 4%, 2-3am: 8%, and after 3am: 15%. No submissions accepted Sunday after 11:59pm.

Notes:

This assignment is based on individual work. No collaboration is allowed (This means no discussing the technical details of each problem with other students before submitting your work. Also, no copying from students of current or previous semesters, in our lab and Homework assignments). You may post your questions on the discussion forums and/or use the office hours. We may pick some students in random to demonstrate their design and simulations. Please refer to the syllabus and the class regarding the USC and our course AI policies, including the penalties for any violation. If you have any doubts about what is allowed or prohibited in this course, please contact the instructor.

Objective

In this exercise, you will use *test-driven development* to:

1. Build a `Member`class to store member's information.
2. Build a `MemberList`data structure that stores `Member` objects.

Background

You've been hired by a club and assigned the task of writing the software that club will use for its membership management.

The members in a list will be stored in a text file (`input.txt`). Your software needs to open the file and read the member information from the file into memory. Once all the members' information is in memory, your software needs to let the user:

- Find all members by a given username.
- Find all members by a given email.
- Find all members by a given registered year.

To begin solving this problem, we will

1. Create a class named `Member`, capable of storing the information about one member
2. Create a class named `MemberList`that stores the `members`and provides methods to perform the operations above.

We will also write test classes and a testing application to try to ensure that each of these classes works as expected.

Getting Started

Create a new project named `lab08` in which to store the files related to this project.

As described previously, we want to test our software to ensure that it works as expected.

Your boss has supplied you a number of members' information with the following format:

```
Spiderman
Jane Scott
02/17/1998
janescott@gmail.com
2133770909
2017

superman
Tony Lee
01/11/1996
tony11@gmail.com
2131237700
2017

clubusername
Alice Flores
05/22/1990
alice1990@gmail.com
2133306677
2016

newmembers
Mark Brown
03/20/1990
mark320@gmail.com
2133900110
2015

davidclubname
David Smith
01/10/1989
davidsmith@gmail.com
2551020510
2018
```

Note that a member has its username, name, date of birth, email, phone number and registered year, each on separate lines. Within the input file, each member is followed by a blank line to separate it from the next one.

Test-Driven Development

To build this software, we are going to use a methodology called `test-driven development`. The basic idea is that, as we think of an operation that a class should provide, we:

1. Write a test to evaluate whether or not the method works;
2. Write the method; and
3. Run the test to see whether or not the method is working correctly. If the method fails the test, revise the method until it passes the test.

Part 1 The Member Class

Since a `MemberList`is made up of `Members`, we will begin with the `Member`class. First, we will create our testing application, then a `MemberTester`class, and finally the `Member`class itself.

1.1 The Testing Application

To run our tests, we can create a source file named `main.cpp`containing a simple `main()`function:

```
#include "MemberTester.h"
#include <iostream>
using namespace std;

int main()
```

```
{
    MemberTester memberTester;
    memberTester.runTests();
}
```

Note that the program is quite simple: it just creates a `MemberTester`object, and tells that object to run its tests. This simplicity is typical of test-driven development.

However, when we try to build this program, we get compilation errors. The problem is that there is no class `MemberTester`nor header file `MemberTester.h`.

1.2 The MemberTester Class

To fix the compilation errors, we must create a `MemberTester`class to store the tests we will run on the methods of our `Member`class.
A C++ class usually consists of two files:

1. a header file whose name ends in *.h*, and
2. an implementation file whose name ends in *.cpp*

so we also need to create `MemberTester.h`and `MemberTester.cpp`.`MemberTester.h`file contains our `MemberTester`class with a prototype or declaration of a single method named `runTests()`. (We'll be adding other method prototypes shortly.)

```
//
// `MemberTester.h`
//

#ifndef EE599_LAB8_MEMBERTESTER_H
#define EE599_LAB8_MEMBERTESTER_H

class MemberTester {
public:
    void runTests();
};

#endif //EE599_LAB8_MEMBERTESTER_H
```

And in `MemberTester.cpp`implement the `runTests()`method.

```
//
// `MemberTester.cpp`
//

#include "MemberTester.h"
#include <iostream>
using namespace std;

void MemberTester::runTests() {

    cout << "Testing class Member..." << endl;
    cout << "All tests passed!" << endl;
}
```

Note that since we have no actual tests to run (yet), we just provide a minimal definition for our runTests() method for now.
At this point, you should be able to build your project without errors, and run it to produce the messages:

```
Testing class Member...
All tests passed!
```

If you have errors, find and fix them before continuing.

1.3 Testing the Default Constructor

We should be able to declare `Member` objects with the expectation that they will be initialized to the same default values. Such initialization is the job of a class's *default constructor*.
If we choose "empty" strings and zero values as our default values, we might build a test method in *MemberTester.cpp* to test our constructors as follows:

```
void MemberTester::testConstructors() {
    cout << "- constructors... " << flush;
    // default constructor
    Member member;
    assert( member.getUsername() == "" );
    assert( member.getName() == "" );
    assert( member.getDateBirth() == "" );
    assert( member.getEmail() == "" );
    assert( member.getPhoneNumber() == "" );
    assert( member.getYear() == 0 );
    cout << " 0 " << flush;
    cout << " Passed!" << endl;
}
```

That is, we expect to be able to send a Member object messages like `getUsername()`, `getName()`, `getDateBirth()`, `getEmail()`, `getPhoneNumber()` and `getYear()`, to access the values of its instance variables, so we use those to test its values.
In order for this test-method to be a method of class `MemberTester`, we must add a prototype for the method to our `MemberTester`class:

```
class MemberTester {
public:
    void runTests();
    void testConstructors();
};
```

You will also need to invoke this method in our `MemberTester::runTests()`method (in `MemberTester.cpp`):

```
void MemberTester::runTests() {
    cout << "Testing class Member..." << endl;
    testConstructors();
    cout << "All tests passed!" << endl;
}
```

and finally #include any header files this method requires (e.g., `<cassert>`).
Next, try to build your project. The compiler should generate an error message something like the following,

```
MemberTester.cpp: In member function ‘void MemberTester::testConstructors()’:
MemberTester.cpp:20:5: error: ‘Member’ was not declared in this scope
     Member Member;
     ^~~~~~
```

Note that this is expected, as we have not yet created our `Member`class. Let's do so!

1.4 The Member Class

From the structure of our input file, our `Member`class needs to store these attributes about a Member:

- Username
- Name
- Date of Birth
- Email
- Phone number
- Registered year

Instance variables should generally be declared as `private:`, to keep developers from writing software that depend upon them, and in order to access this private field, we need to define a constructor and getter methods as `public:`.
So we can start by creating the following class declaration in `Member.h`:

```
//
// Member.h
//

#ifndef EE599_LAB8_MEMBER_H
#define EE599_LAB8_MEMBER_H

#include <string>
using namespace std;

class Member {
public:
    Member();
    string getUsername();
    string getName();
    string getDateBirth();
    string getEmail();
    string getPhoneNumber();
    unsigned getYear();
private:
    string username;
    string name;
    string datebirth;
    string email;
    string phonenumber;
    unsigned year;
};

#endif //EE599_LAB8_MEMBER_H
```

Since our `MemberTester`class is trying to create a `Member` object, it must see the declaration of class `Member`. To do so, it must `#include "Member.h"` so take a moment to add that directive to `MemberTester.cpp`.
Since a C++ class has both a header file and an implementation file, so we also need to create the file `Member.cpp`as follow:

```
//
// Member.cpp
//

#include "Member.h"

/* Member default constructor
```

```
* Postcondition: username, name, datebirth, email and phonenumber are empty strings, year ==
0.
*
*/
Member::Member() {
    username = "";
    name = "";
    datebirth = "";
    email = "";
    phonenumber = "";
    year = 0;
}

/* getter method for username
 * Return: username
 */
string Member::getUsername() {
    return username;
}

/* getter method for name
 * Return: name
 */
string Member::getName() {
    return name;
}

/* getter method for DateBirth
 * Return: DateBirth
 */
string Member::getDateBirth() {
    return datebirth;
}

/* getter method for email
 * Return: email
 */
string Member::getEmail() {
    return email;
}

/* getter method for phonenumber
 * Return: phonenumber
 */
string Member::getPhoneNumber() {
    return phonenumber;
}

/* getter method for registered year
* Return: year
*/
unsigned Member::getYear() {
    return year;
}
```

Now that we have a Member class created, we must `#include "Member.h"` in our `MemberTester.cpp` file, so that our `testConstructors()` method can declare its `Member`Object.
With that change, our project should compile without any errors.

```
Testing class Member...
- constructors...  0 Passed!
All tests passed!
```

Congratulations! You've just written your first C++ methods using test-driven development!

Part 1 The Member Class

1.5 Brief Aside

The key to test-driven development is sometimes called *programming by intent*-- you have to know what you *intend* a method to do, write a test-method that thoroughly tests your intent, and then define the methods needed to pass your test.

Test-driven development is the sort of thing that gets easier with practice, so we are starting with "simple" methods before we progress to more complicated ones.

As we will see, test-driven development also supports *regression-testing* -- making sure that changes you make to a class in one method do not cause another method to "break".

1.6 Testing the Explicit-Value Constructor

In addition to a default constructor, most classes should provide an explicit-value constructor that lets the programmer initialize its members to specific values. For example, we might invoke such a constructor for our `Member`class by adding the following code to our `testConstructors()`method:

```
void MemberTester::testConstructors() {
    cout << "- constructors... " << flush;
    // default constructor
    Member member;
    assert( member.getUsername() == "" );
    assert( member.getName() == "" );
    assert( member.getDateBirth() == "" );
    assert( member.getEmail() == "" );
    assert( member.getPhoneNumber() == "" );
    assert( member.getYear() == 0 );
    cout << " 0 " << flush;

    // explicit constructor
    Member Member1("Spiderman", "Jane Scott", "02/17/1998", "janescott@gmail.com",
"2133770909", 2017);
    assert( member1.getUsername() == "Spiderman" );
    assert( member1.getName() == "Jane Scott" );
    assert( member1.getDateBirth() == "02/17/1998" );
    assert( member1.getEmail() == "janescott@gmail.com" );
    assert( member1.getPhoneNumber() == "2133770909" );
    assert( member1.getYear() == 2017 );
    cout << " 1 " << flush;

    cout << " Passed!" << endl;
}
```

Next, try to build your project. As expected, we get a compilation error because the compiler cannot (yet) find a prototype or definition for our explicit-value constructor:

```
MemberTester.cpp: In member function 'void MemberTester::testConstructors()':
MemberTester.cpp:31:102: error: no matching function for call to 'Member::Member(const char
[10], const char [11], const char [11], const char [20], const char [11], int)'
     Member member1("Spiderman", "Jane Scott", "02/17/1998", "janescott@gmail.com",
"2133770909", 2017);
```

To eliminate this error, we must define the explicit-value constructor!

1.7 The Explicit-Value Constructor

Once again, we write a method to satisfy our test. The test indicates the order in which the arguments are passed to the constructor, so add this definition to the definitions in `Member.cpp`:

```
/* Explicit constructor
    * @param: username, a string
    * @param: name, a string
    * @param: datebirth, a string
    * @param: email, a string
    * @param: phonenumber, a string.
    * @param: year, an unsigned int.
    * Postcondition: this -> username == username &&
    *                this -> name == name &&
    *                this -> datebirth == datebirth &&
    *                this -> email == email &&
    *                this -> phonenumber == phonenumber &&
    *                this -> year == year.

    */
Member::Member(const string& username, const string& name, const string& datebirth, const
string& email, const string& phonenumber, unsigned year) {
    this -> username = username;
    this -> name = name;
    this -> datebirth = datebirth;
    this -> email = email;
    this -> phonenumber = phonenumber;
    this -> year = year;

}
```

Note that since `username`,`name`, `datebirth`, `email and phonenumber`are string objects whose values flow into but not out of the constructor, we pass them using pass-by-const-reference. However, `year`is an unsigned int, which is a primitive type, so we pass it using pass-by-value.
Next, add a prototype for this constructor to our `Member`class. You should now have two constructors declared there: a default-value constructor and an explicit-value constructor.

```
class Member {
```

```
public:
    Member();
    Member(const string& username, const string& name, const string& datebirth, const string&
email, const string& phonenumber, unsigned year);
    string getUsername();
    string getName();
    string getDateBirth();
    string getEmail();
    string getPhoneNumber();
    unsigned getYear();
private:
    string username;
    string name;
    string datebirth;
    string email;
    string phonenumber;
    unsigned year;
};
```

When you have done so, our test-program should compile and run correctly:

```
Testing class Member...
- constructors...  0 1  Passed!
All tests passed!
```

To see what happens when you make a mistake, simulate making a mistake by "commenting out" one of the lines in our constructor:

```
Member::Member(const string& username, const string& name, const string& ISBN, const string&
email, unsigned phonenumber) {
    this -> username = username;
    this -> name = name;
    this -> ISBN = ISBN;
    this -> email = email;
    //this -> phonenumber = phonenumber;
}
```

Now when our test-method runs, the test for the explicit-value constructor will fail:

```
Testing class Member...
- constructors...  0 a.out: MemberTester.cpp:37: void MemberTester::testConstructors():
Assertion `member1.getYear() == 2017' failed.
Abort (core dumped)
```

By checking whether or not your methods pass the tests, test-driven development provides a methodology to help you keep from making mistakes.
"Uncomment" the `this -> year = year;` line in your constructor, rebuild your project, and double-check that your code passes the tests before continuing.

1.8 An Input Method

Since our Members' information is stored in a text file, it appears that we will need a method to fill a `Member`object with data from a file, or more precisely, from an “fstream “to a file. To test such a method, we can use the input file. We can then define a new test-method `testReadFrom()` in `MemberTester.cpp`to test our input method, which we will call `readFrom()`:

```
void MemberTester::testReadFrom() {
    cout << "- readFrom()... " << flush;
    ifstream fin("input.txt");
    assert( fin.is_open() );
    Member member;

    // read first Member in input file
    member.readFrom(fin);
    assert( member.getUsername() == "Spiderman" );
    assert( member.getName() == "Jane Scott" );
    assert( member.getDateBirth() == "02/17/1998" );
    assert( member.getEmail() == "janescott@gmail.com" );
    assert( member.getPhoneNumber() == "2133770909"  );
    assert( member.getYear() == 2017);
    cout << " 0 " << flush;

    // read second Member in input file
    string separator;
    getline(fin, separator);
    member.readFrom(fin);
    assert( member.getUsername() == "superman" );
    assert( member.getName() == "Tony Lee" );
    assert( member.getDateBirth() == "01/11/1996" );
    assert( member.getEmail() == "tony11@gmail.com" );
    assert( member.getPhoneNumber() == "2131237700"  );
    assert( member.getYear() == 2017);
    cout << " 1 " << flush;
    // read third Member in input file
    getline(fin, separator);
    member.readFrom(fin);
    assert( member.getUsername() == "clubusername" );
    assert( member.getName() == "Alice Flores" );
    assert( member.getDateBirth() == "05/22/1990" );
    assert( member.getEmail() == "alice1990@gmail.com" );
    assert( member.getPhoneNumber() == "2133306677"  );
    assert( member.getYear() == 2016);
    cout << " 2 " << flush;

    fin.close();
    cout << "Passed!" << endl;
}
```

Using this as a model, add a final two tests to this method that reads the fourth and fifth Members from our input file, checks that it has the correct values, and if so, displays the value 3 and 4 correspondingly.
Note how the `testReadFrom()` method deals with the blank lines between the 1st and 2nd, and the 2nd and 3rd Members. To work correctly, the test-code you add will need to deal with the blank line between the 3rd and 4th Members and 4th and 5th Members...
Then try to build your project. You should see several error messages. As before, we must:

- place a prototype for this method in our `MemberTester` class
- invoke this method in `MemberTester::runTests()`, and
- add any new `#inlcude` directives this method requires (e.g., `<fstream>`).

When the only error we see is the compiler being unable to find our `readFrom()` method, we are ready to define that method!

```
MemberTester.cpp: In member function 'void MemberTester::testReadFrom()':
MemberTester.cpp:51:12: error: 'class Member' has no member named 'readFrom'
     member.readFrom(fin);
            ^
MemberTester.cpp:63:12: error: 'class Member' has no member named 'readFrom'
     member.readFrom(fin);
            ^
MemberTester.cpp:74:12: error: 'class Member' has no member named 'readFrom'
     member.readFrom(fin);
            ^
MemberTester.cpp:85:12: error: 'class Member' has no member named 'readFrom'
     member.readFrom(fin);
            ^
MemberTester.cpp:96:12: error: 'class Member' has no member named 'readFrom'
     member.readFrom(fin);
```

Note that our test method invokes `readFrom()` multiple times. It is generally a good idea to test a method multiple times, as invoking a method multiple times will sometimes reveal errors that do not show up when the method is only invoked once.

1.9 Defining The Input Method

Now that we know what our input method has to do to pass the test, we write that method. We might try the following definition:

```
/* Member input method...
 * @param: in, an istream
 * Precondition: in contains the username, name, datebirth, email, phonenumber and registered
year data for a Member.
 * Postcondition: the username, name, datebirth, email, phonenumber and registered year data
have been read from in &&
 *                 this -> username == username &&
 *                 this -> name == name &&
 *                 this -> datebirth == datebirth &&
 *                 this -> email == email &&
 *                 this -> phonenumber == phonenumber &&
 *                 this -> year == year.
 */
void Member::readFrom(istream& in) {
    getline(in, this -> username);
    getline(in, this -> name);
    getline(in, this -> datebirth);
    getline(in, this -> email);
    getline(in, this -> phonenumber);
    string yearString;
    getline(in, yearString);
    this -> year = atoi( yearString.c_str());
}
```

Note that because our method modifies the stream it receives through its parameter in, we declare that parameter using pass-by-reference.
Note also that we use the `getline()` method to read in the various string values. We do this because `getline()` will read an entire line of text, whereas the string version of operator>>will only read a word of text.

In order for this method to compile correctly, add a prototype for this method to the Member class, as well as any `#include`directives you need in order for this method to compile correctly (e.g., `<cstdlib>`).
Once our program compiles correctly, run it and verify that it passes all our tests:

```
Testing class Member...
- constructors...  0 1  Passed!
- readFrom()...  0 1  2  3  4 Passed!
All tests passed!
```

Once we have passed all our tests, we can continue to the next operation.

1.10 Testing An Output Method

One of the required `MemberList`operations is to print all the Members with a given word in their username, which means we also need an operation to print/output a Member. Since C++ output is via ostream objects, we will pass our operation an ostream to which a Member can write its information. The ostream class is a superclass of the ofstream class, so this will allow us to use the same method for writing both to the console (via cout) and to a file (via an ofstream). We will name our method `writeTo()`.
Testing our `writeTo()`method using `assert()`takes some effort:

```
void MemberTester::testWriteTo() {
    cout << "- writeTo()... " << flush;

    // declare three Members
    Member member1("Spiderman", "Jane Scott", "02/17/1998", "janescott@gmail.com",
"2133770909", 2017);
    Member member2("superman", "Tony Lee", "01/11/1996", "tony11@gmail.com", "2131237700",
2017);
    Member member3("clubusername", "Alice Flores", "05/22/1990", "alice1990@gmail.com",
"2133306677", 2016);


    // write the three Members to an output file
    ofstream fout("testOutput.txt");
    assert( fout.is_open() );
    member1.writeTo(fout);
    member2.writeTo(fout);
    member3.writeTo(fout);
    fout.close();

    // use readFrom() to see if writeTo() worked
    ifstream fin("testOutput.txt");
    assert( fin.is_open() );
    Member member4, member5, member6;

    // read and check the first Member
    member4.readFrom(fin);
    assert( member4.getUsername() == "Spiderman" );
    assert( member4.getName() == "Jane Scott" );
    assert( member4.getDateBirth() == "02/17/1998" );
    assert( member4.getEmail() == "janescott@gmail.com" );
    assert( member4.getPhoneNumber() == "2133770909"  );
    assert( member4.getYear() == 2017);

    cout << " 0 " << flush;

    // read and check the second Member
    member5.readFrom(fin);
    assert( member5.getUsername() == "superman" );
    assert( member5.getName() == "Tony Lee" );
    assert( member5.getDateBirth() == "01/11/1996" );
    assert( member5.getEmail() == "tony11@gmail.com" );
    assert( member5.getPhoneNumber() == "2131237700"  );
    assert( member5.getYear() == 2017);
    cout << " 1 " << flush;

    // read and check the third Member
    member6.readFrom(fin);
    assert( member6.getUsername() == "clubusername" );
    assert( member6.getName() == "Alice Flores" );
    assert( member6.getDateBirth() == "05/22/1990" );
    assert( member6.getEmail() == "alice1990@gmail.com" );
    assert( member6.getPhoneNumber() == "2133306677"  );
    assert( member6.getYear() == 2016);

    cout << " 2 " << flush;

    fin.close();
    cout << " Passed!" << endl;
}
```

Two things are worth noting about this test:

1. To automate the test, we open an `ifstream`to a file and use our `writeTo()`method to write three Members to that file. After closing the stream, we then open an `ofstream` to the file and use our `readFrom()` method to read the Members from the file into new `Member`variables. We can then assert what the values in these Members should be. By using files instead of interactive I/O, this approach *automates* the test, so that the user need not enter any values manually
2. To thoroughly test the `readFrom()` and `writeTo()` methods, we use multiple Members.

After placing a prototype of `testWriteTo()` in our `MemberTester` class and invoking this method in `runTests()`, the only error we should generate is due to `writeTo()` not being defined:

```
MemberTester.cpp: In member function 'void MemberTester::testWriteTo()':
MemberTester.cpp:122:13: error: 'class Member' has no member named 'writeTo'
     member1.writeTo(fout);
             ^
MemberTester.cpp:123:13: error: 'class Member' has no member named 'writeTo'
     member2.writeTo(fout);
             ^
MemberTester.cpp:124:13: error: 'class Member' has no member named 'writeTo'
     member3.writeTo(fout);
             ^
```

This means it is time for us to define that method!

1.11 The Output Method

Since the test tells us what we expect our method to do, we just have to define our method in a way that passes the test:

```
/* Member output...
 * @param: out, an ostream
 * Postcondition: out contains username, a newline,
 *                             name, a newline,
 *                             datebirth, a newline,
 *                             email, a newline,
 *                             phonenumber, a newline,
 *                             year, a newline.
 *
 */
void Member::writeTo(ostream& out) const {
    out << this -> username << '\n'
    << this -> name << '\n'
    << this -> datebirth << '\n'
    << this -> email << '\n'
    << this -> phonenumber  << '\n'
    << this -> year  << '\n';
}
```

Note that our method changes its parameter `out`so we define that parameter using call-by-reference.
Note also that an output method should not change any of its class's instance variables, so we define `writeTo()`as a const method. This tells the compiler that if we should inadvertently change an instance variable in this method, the compiler should generate an error to alert us to our mistake.
Define this method and place a prototype for it in your `Member`class. Then compile and run your test-application. If all is well, pass all the tests should pass.

```
Testing class Member...
- constructors...  0 1  Passed!
- readFrom()...  0 1  2  3 4  Passed!
- writeTo()...  0 1  2  Passed!
All tests passed!
```

Congratulations! You have just built a reasonably complex Member class!

Part 2 The MemberList Class

Our next class is the `MemberList`class, in which we will store the Members of a given input file. Since it must store multiple Members, we might envision a `MemberList`object named `aMemberList`containing the Members from `input.txt`.

2.1 Testing the MemberList Class

To test this new class, we will create a `MemberListTester`class, similar to the `MemberTester`class we built previously.
We start by returning to `main.cpp`and adding code that (a) `#includes`the header file `MemberListTester.h`, (b) creates a `MemberListTester`object, and (c) invokes its `runTests()`method:

```
#include "MemberTester.h"
#include "MemberListTester.h"
#include <iostream>
using namespace std;
```

```
int main()
{
    MemberTester MemberTester;
    MemberTester.runTests();
    MemberListTester aMemberListTester;
    aMemberListTester.runTests();
}
```

If we try to build our project, we get errors because neither class `MemberListTester`nor the file `MemberListTester.h`exist. To proceed, we must create them, which we might do as follows:

```
//
// MemberListTester.h
//

#ifndef EE599_LAB8_MEMBERLISTTESTER_H
#define EE599_LAB8_MEMBERLISTTESTER_H

class MemberListTester {
public:
    void runTests();
};


#endif //EE599_LAB8_MEMBERLISTTESTER_H
```

and

```
//
// MemberListTester.cpp
//

#include "MemberListTester.h"
#include <iostream>
using namespace std;

void MemberListTester::runTests() {
    cout << "\nTesting class MemberList..." << endl;

    cout << "All tests passed!" << endl;
}
```

This should be sufficient to allow our project to build and run successfully:

```
Testing class Member...
- constructors...  0 1  Passed!
- readFrom()...  0 1  2  3  4 Passed!
- writeTo()...  0 1  2  Passed!
All tests passed!

Testing class MemberList...
All tests passed!
```

Note that we do not replace our `MemberTester`tests. We want to continue to continue to re-run the `Member`tests each time we test our `MemberList`methods, to make sure that nothing we write breaks anything that worked previously. This kind of testing -- in which we rerun all of our tests every time we modify the system to ensure that our modifications haven't broken something -- is called regression testing. Regression testing is a powerful technique for writing reliable software.

2.2 Testing the MemberList Constructor

Recall that a Member list's Member-data is stored in a file. Because of this, it really doesn't make much sense to build a default-constructor for our `MemberList`class, at least not yet. Instead, we will build an explicit constructor to which we pass the name of an input file as an argument. Our constructor can get the name of the file via its parameter, and then read the Member list's data from that file.

To test our constructor, we can once again use the `input.txt`file. We can then write a method to test our `MemberList`constructor:

```
void MemberListTester::testConstructors() {
    cout << "- constructors..." << flush;
    MemberList bList("input.txt");
    assert( bList.getNumMembers() == 5 );
    cout << " 0 " << flush;

    cout << " Passed!" << endl;
}
```

Note that our test method (a) invokes an explicit-value `MemberList`constructor, passing it the name of our input file; and (b) checks that this worked by checking that the `MemberList`contains five Members, the number in the input file.

We must of course invoke this method in runTests():

```
void MemberListTester::runTests() {
    cout << "\nTesting class MemberList..." << endl;
    testConstructors();
    cout << "All tests passed!" << endl;
}
```

and place its prototype in our `MemberListTester`class:

```
class MemberListTester {
public:
    void runTests();
    void testConstructors();
};
```

When we try to build, the compiler will generate errors because we have not yet defined a `MemberList`class. That means it's time to do so!

```
MemberListTester.cpp: In member function 'void MemberListTester::testConstructors()':
MemberListTester.cpp:18:5: error: 'MemberList' was not declared in this scope
     MemberList bList("input.txt");
     ^
MemberListTester.cpp:18:16: error: expected ';' before 'bList'
     MemberList bList("input.txt");
                ^
MemberListTester.cpp:19:13: error: 'bList' was not declared in this scope
     assert( bList.getNumMembers() == 5 );
             ^
MemberListTester.cpp:19:40: error: 'assert' was not declared in this scope
     assert( bList.getNumMembers() == 5 );
                                        ^
```

2.3 The MemberList Class

Given the functionality required by our `testConstructors()`method, we might declare class `MemberList`as follows:

```
//
// MemberList.h
//

#ifndef EE599_LAB8_MEMBERLIST_H
#define EE599_LAB8_MEMBERLIST_H

#include <string>
using namespace std;

class MemberList {
public:
    MemberList(const string& fileName);
    unsigned getNumMembers() const;
};
#endif //EE599_LAB8_MEMBERLIST_H
```

Note that we declare an explicit-value constructor, which has a parameter to store the name of the input file; and a `getNumMembers()`method, since the test requires that. Note also that since the parameter for our `MemberList`constructor is a string (which is a class type), we declare it using pass-by-const-reference. Note finally that since the `getNumMembers()`method should not change any of our instance variables, we declare it as a const method.

Take a moment to `#include "MemberList.h"`in `MemberListTester.cpp`, so that our test-methods can see class `MemberList`; and `<cassert>`so that they can use `assert()`.

If you try to build the project at this point, there will still be errors because we have declared the `MemberList()`explicit-value constructor and the `getNumMembers()`method, but we have not defined them. We can fix that by adding the following stub definitions to `MemberList.cpp`:

```
//
// MemberList.cpp
//
#include "MemberList.h"

/* MemberList constructor
    * @param: fileName, a string
    * Precondition: fileName contains the name of a input file.
    */
MemberList::MemberList(const string& fileName) {

}

/* Retrieve length of the Member list
 * Return: the number of Members in the list.
 */
unsigned MemberList::getNumMembers() const {


}
```

With that addition, our program should build without errors. Make sure that this is the case before proceeding.

While our program builds, when we try to run it, it fails our test:

```
Testing class Member...
- constructors...  0 1  Passed!
- readFrom()...  0 1  2  3  4 Passed!
- writeTo()...  0 1  2  Passed!
All tests passed!

Testing class MemberList...
- constructors...a.out: MemberListTester.cpp:21: void MemberListTester::testConstructors():
Assertion `bList.getNumMembers() == 5' failed.
Abort (core dumped)
```

To pass the test, we must fill in the stubs of our `MemberList`constructor and `getNumMembers()`methods.

2.4 The MemberList Constructor

Before we can complete the constructor definition, we have to decide how we are going to store the MemberList's Member objects. One way we might do so is using the vector class-template from the C++ standard template library (STL). To do this, we can add an instance variable to our `MemberList`class whose type is `vector<Member>`and whose name is `totalMembers`, as shown below:

```
//
// MemberList.h
//

#ifndef EE599_LAB8_MEMBERLIST_H
#define EE599_LAB8_MEMBERLIST_H

#include "Member.h"
#include <vector> // STL vector
#include <string>

using namespace std;

class MemberList {
public:
    MemberList(const string& fileName);
    unsigned getNumMembers() const;

private:
    vector<Member> totalMembers;
};
#endif //EE599_LAB8_MEMBERLIST_H
```

A vector is an array that can grow or shrink as the program runs. For this reason, a vector is sometimes called a dynamic array.
With a vector data structure in which to store our Members, we can define our `MemberList()`constructor as follows:

```
/*  MemberList constructor
    * @param: fileName, a string
    * Precondition: fileName contains the name of a input file.
    */
MemberList::MemberList(const string& fileName) {
    // open a stream to the Memberlist file
    ifstream fin( fileName.c_str() );
    assert( fin.is_open() );

    // read each Member and append it to totalMembers
    Member Member;
    string separator;
    while (true) {
        Member.readFrom(fin);
        if ( !fin ) { break; }
        getline(fin, separator);
        totalMembers.push_back(Member);
    }

    // close the stream
    fin.close();
}
```

To pass our test, our `getNumMembers()`method just needs to return the number of Members in `totalMembers`.

```
/* Retrieve length of the Member list
 * Return: the number of Members in the list.
 */
unsigned MemberList::getNumMembers() const {

    return totalMembers.size();
}
```

With that change, your program should build correctly, and when you run it, it should pass all our tests. Make certain this is the case before proceeding.

```
Testing class Member...
- constructors...  0 1  Passed!
- readFrom()...  0 1  2  3  4 Passed!
- writeTo()...  0 1  2  Passed!
All tests passed!
```

```
Testing class MemberList...
- constructors... 0  Passed!
All tests passed!
```

Using this TDD concept and this working model that we have designed to implement three Memberlist's operations, **implement search by username, search by email and search by registered year.**
You are asked to use the provided code, make the necessary updates and submit the updated package. Make sure your submission includes all the files including the ones you updated. Please also add in your readme.pdf a summary of your updates, any notes the grader should be aware of, any non-working parts or issues, etc.

Submission

(1) Zip all the files you need to submit into a zip file named: "EE599_lab8_firstname_lastname".zip.

(2) Your zip file should include all the coding parts the assignment asks for, and also a Readme.pdf. For this assignment it is okay that your zip file would only include one file, i.e., your Readme.pdf.

(3) In your Readme.pdf, include any information that you think the course staff, especially the grader should know while grading your assignment: references, any non-working part, any concerns, etc.

a) Any non-working part should be clearly stated.

b) The citations should be done carefully and clearly, e.g.: *"to write my code, lines 27 to 65, I used the Diijkstra's shortest path algorithm c++ code from the following website: www.SampleWebsite.com/..."*

c) The Readme file content of labs and PAs can be hand-written or typed. In case you decide to hand-write, then please scan and include in your Readme.pdf.

(4) Use the provided BB submission link to submit your zip file for this assignment.

Lab8 Submission Link