Assignment 3: Employee Management and Scheduler Application

Dates:

Assigned:	2013-07-11	Thursday
Due:	2013-07-18	Thursday at 11:59 pm

Questions – email rjmfff@mail.umkc.edu

Make sure that your source code is *committed* and *synced* to your student GitHub account. You do not need to turn in the code via Blackboard or anything else.

Note: I will no longer answer questions on assignments on the day they are due, so start it BEFORE Thursday!

Assignment Tips:

- **Do not try to implement all classes at once!**
  Programming is an *iterative process*. Start with one class and work your way up! Don't just copy the UML diagram and then try to fill things in later – this keeps you from being able to **compile** a working program and test functionality on a modular level!

Specification:

This is a phonebook application that will load in a list of people (first name, last name, phone number, and country) from a file, “numbers.txt” and enter them into the system.

Then, the user can view all entries or filter by country or last name.

This assignment will use:

- Operator Overloading
- Templates
- Dynamic Arrays

Step-By-Step

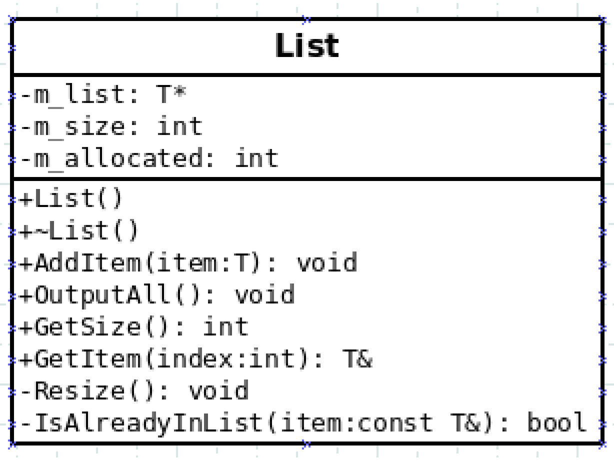
Iteration 1: List Template Class

The first thing we care about is reading in the provided text file, *numbers.txt*, and storing it in a data structure.

First, we need to create that data structure. In the program, we will store a list of strings, as well as our own PhoneNumber class.

Note: With Template classes, <u>everything</u> must go in the header .h file. This includes function definitions (bodies)!

Here is the diagram for the List Template:



`List()`

Our constructor should initialize the size of our list (m_size) to 0, our allocated space (m_allocated) to 10, and then initialize m_list as a new "T" item of size "m_allocated".

```
m_list = new T[ m_allocated ];
```

`~List()`

The destructor is responsible for deleting our dynamic array (m_list).

`void AddItem( T item )`

AddItem will do two checks before adding the item to the list:

1. We must make sure that there is enough space allocated for this item. If the size of the list (m_size) is equal to the amount of allocated space (m_allocated), then we need to call the Resize() function before continuing.

2. Secondly, we want to make sure we're adding a unique element. Have a conditional statement that checks the return value of the "IsAlreadyInList" function. Pass in the item that will be added to the list.

After we build enough space in our list, we will add the item to our array, but only if it's unique.

```
m_list[ m_size ] = item;
m_size++;
```

`void OutputAll()`

This function will output each element of our list (m_list) via a for-loop.

Use a normal cout on the list element. Because we're using the stream operator, when we write our PhoneNumber class we will have to overload the stream operator.

`int GetSize()`

This function will simply return the size of our list (m_size).

`T& GetItem( int index )`

This function will return the item in the list (m_list) at the index given in the parameter.

```
return m_list[ index ];
```

`void Resize()`

Our Resize function works in three steps:

1. Create a tempList dynamic array that has the same allocated space (m_allocated).
   Use a for-loop to iterate through every item in the list (m_list) and copy each element of the list (m_list) to the temp list (tempList).
2. Delete our list (m_list), add 10 to the allocated size (m_allocated), and then create a new dynamic array in our list (m_list) of the size m_allocated.
3. Have another for-loop that iterates over each element over the temporary list (tempList). For each element, copy the item in tempList back over to the main list (m_list).

Don't forget to delete the tempList afterwards.

`bool IsAlreadyInList( const T& item )`

This function uses a for-loop to iterate through every element in our list (m_list). For each item, it is going to check whether the parameter (item) is equal to the element in our list (m_list[i]).

If it finds a match on a given element, return true. If it cannot find any matches and it has gone through the entire for-loop, return false.

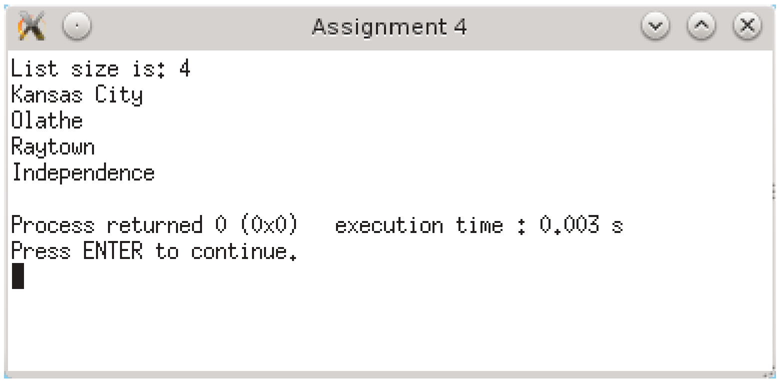
Since this function uses the equal sign == for comparison, we're going to have to make sure to **overload the ==** operator in our PhoneNumber class.

Testing it Out

Now that we should have a working List class, go back to main(). Include our List file, and inside of main add some code like:

```
int main()
{
    List<string> myList;
    myList.AddItem( "Kansas City" );
    myList.AddItem( "Olathe" );
    myList.AddItem( "Raytown" );
    myList.AddItem( "Olathe" );
    myList.AddItem( "Independence" );

    cout << "List size is: " << myList.GetSize() << endl;

    myList.OutputAll();

     return 0;
}
```

When you build and run this, you should get output similar to:



```
List size is: 4
Kansas City
Olathe
Raytown
Independence

Process returned 0 (0x0)   execution time : 0.003 s
Press ENTER to continue.
```

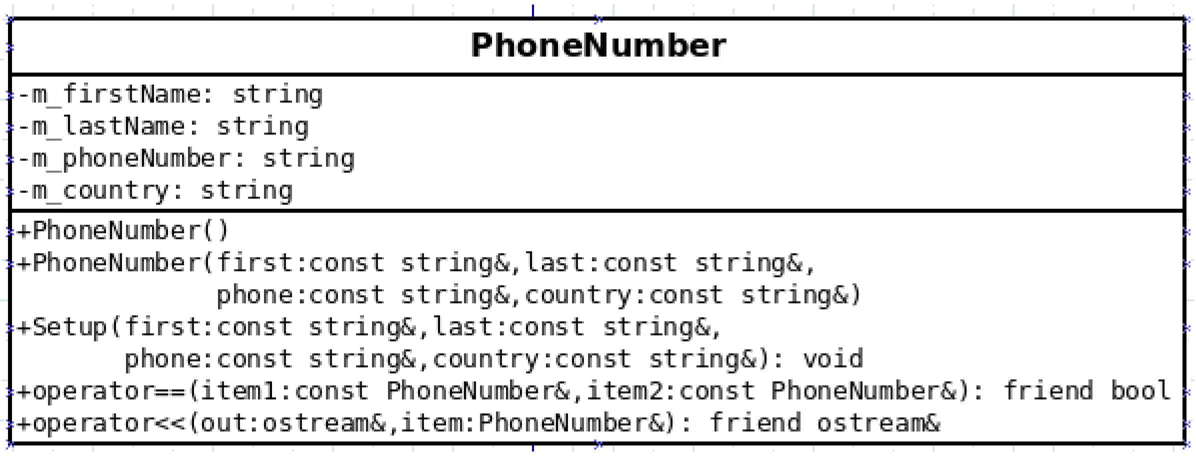
Note how the duplicate (Olathe) is not added twice.

If everything builds, runs, and is logically sound – continue on.

Commit & Push!

Iteration 2: PhoneNumber Class

Now we're going to implement our PhoneNumber class.



```
PhoneNumber
-m_firstName: string
-m_lastName: string
-m_phoneNumber: string
-m_country: string
+PhoneNumber()
+PhoneNumber(first:const string&,last:const string&,
             phone:const string&,country:const string&)
+Setup(first:const string&,last:const string&,
       phone:const string&,country:const string&): void
+operator==(item1:const PhoneNumber&,item2:const PhoneNumber&): friend bool
+operator<<(out:ostream&,item:PhoneNumber&): friend ostream&
```

The phone number is going to contain information on the person's name, their country, and their number.

We're going to have a setup function as well as two overloaded operators: the equality comparison (==) and the output stream (<<).

`PhoneNumber()`

Initialize the private variables to something like "Default".

`PhoneNumber( const string& first, const string& last, const string& phone, const string& country )`

Call the Setup function with the same parameters.

`void Setup ( const string& first, const string& last, const string& phone, const string& country )`

Set our private members (m_firstName, etc.) equal to the parameters.

`friend bool operator==( const PhoneNumber& item1, const PhoneNumber& item2 )`

Check whether item1's private members are equal to item2's private members – all of them. If everything matches, return true. Otherwise, return false.

Implement the overloaded operators in PhoneNumber.cpp

`friend ostream& operator<<( ostream& out, PhoneNumber item )`

Output each of the private members using the out stream...

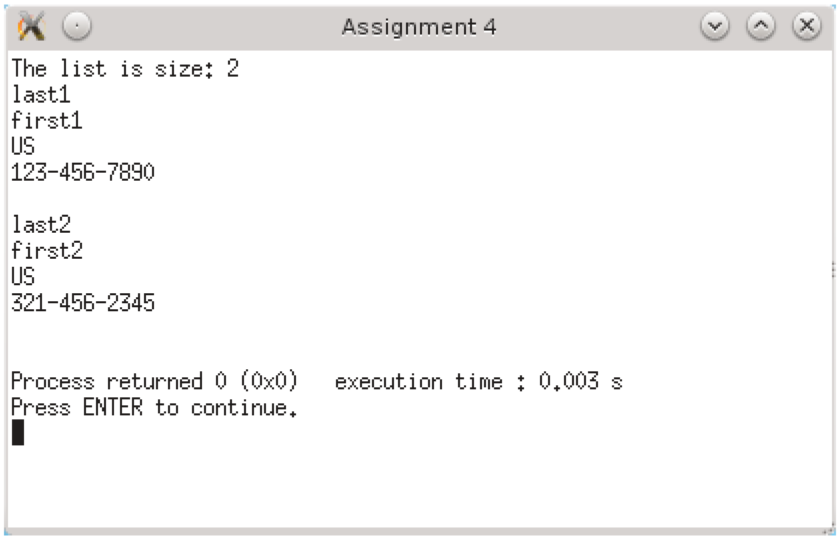
```
ostream& operator<<( ostream& out, const PhoneNumber&
item )
{
    out << item.m_lastName << endl;
    out << item.m_firstName << endl;
    out << item.m_country << endl;
    out << item.m_phoneNumber << endl;
    return out;
}
```

Now we have a PhoneNumber class that will work with our List.

Back in main(), add this code:

```
int main()
{
    List<PhoneNumber> myList;

    PhoneNumber tempNumber( "first1", "last1", "123-456-7890",
          "US" );

    myList.AddItem( tempNumber );
    myList.AddItem( tempNumber );

    tempNumber.Setup( "first2", "last2", "321-456-2345", "US" );
    myList.AddItem( tempNumber );

    cout << "The list is size: " << myList.GetSize() << endl;

    myList.OutputAll();

     return 0;
}
```

You should get output like:



```
The list is size: 2
last1
first1
US
123-456-7890

last2
first2
US
321-456-2345


Process returned 0 (0x0)   execution time : 0.003 s
Press ENTER to continue.
```

If this builds, runs, and is logically sound, continue on.

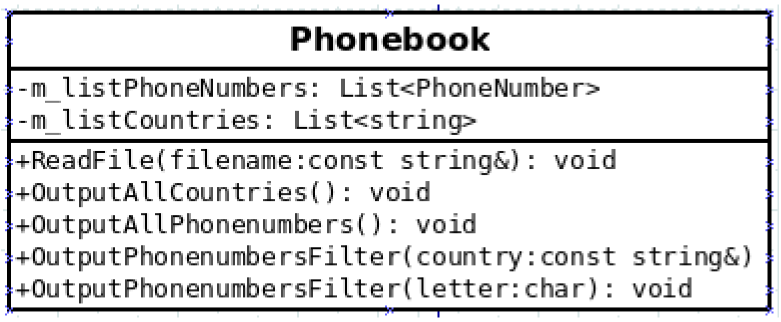
Iteration 4: The Phonebook

Our Phonebook class will handle all of the interfacing with the lists and phone numbers. Main will only be able to access simple functions via the phonebook itself.

First, go back to PhoneNumber.h and add the declaration under public:

```
friend class Phonebook;
```

Here is the diagram:



```
Phonebook
-m_listPhoneNumbers: List<PhoneNumber>
-m_listCountries: List<string>
+ReadFile(filename:const string&): void
+OutputAllCountries(): void
+OutputAllPhonenumbers(): void
+OutputPhonenumbersFilter(country:const string&)
+OutputPhonenumbersFilter(letter:char): void
```

The phonebook class has two Lists: one holding just strings (names of countries) and one holding PhoneNumber classes. As we read the file of numbers, we're going to add to both of these lists.

`void ReadFile( const string& filename )`

This function will open a file, using the filename passed in as a parameter.

Create four temp variables: first, last, phone, and country (all strings).

Create a while-loop to read in four variables at a time, and then store them in the list.

```
string first, last, phone, country;
while ( infile >> first >> last >> phone >> country )
{
    PhoneNumber tempPhone( first, last, phone, country );
    m_listPhoneNumbers.AddItem( tempPhone );
    m_listCountries.AddItem( country );
}
```

Afterwards, close the file.

`void OutputAllCountries()`

In this function, simply call the “OutputAll()” function that belongs to the List class. Make sure to call it with respect to the m_listCountries list.

`void OutputAllPhonenumbers()`

In this function, simply call the “OutputAll()” function that belongs to the List class. Make sure to call it with respect to the m_listPhoneNumbers list.

`void OutputPhonenumbersFilter( const string& country )`

In this function, rather than calling the list's “Output” function, we're going to have to iterate through each element in our List. We have access to list elements via the GetItem(index) function, and we can get the List's size via the GetSize() function (both belong to the List class).

This function deals specifically with m_listPhoneNumbers, we don't care about the country list right now.

So, create a for-loop that goes from 0 to the size of the List.

Within the for-loop, create a PhoneNumber variable named *item*. This is where we're going to store the *current element of the List* so we can output the information.

Set the *item* variable equal to the current element in our list:

```
PhoneNumber item = m_listPhoneNumbers.GetItem( i );
```

Then, check whether the country of this item is equal to the country parameter passed in. If it is, display this item via cout.

```
cout << item << endl;
```

`void OutputPhonenumbersFilter( char letter )`

This function also only deals with the PhoneNumber list, not the country list.

Similar to in the OutputPhonenumbersFilter function above, we're going to loop through each element and output the element's information ***only*** if it meets the filter condition.

Here, we are filtering so we only display entries whose last name begins with the letter provided.

We can get the 0th character of a string with the subscript operator:

```
if ( item.m_lastName[0] == letter )
```

Now, back in main(), we need to test!

```
int main()
{
    int pauser;

     Phonebook phonebook;
     phonebook.ReadFile( "numbers.txt" );

     phonebook.OutputAllCountries();
     cin >> pauser;

     phonebook.OutputAllPhonenumbers();
     cin >> pauser;

     phonebook.OutputPhonenumbersFilter( "France" );
     cin >> pauser;

     phonebook.OutputPhonenumbersFilter( 'T' );
     cin >> pauser;

     return 0;
}
```

If this code builds, runs, and is logically sound, then continue!

Iteration 5: The Interface

Now we're going to update main() with a user interface. The user will use this to be able to output and filter phone numbers.

All of the code currently in main() is just throwaway test code. Go ahead and remove it.

Create an instance of Phonebook with the name “phonebook”. Call the “ReadFile” function on the phonebook instance and pass in our phone number file, “numbers.txt”.

Also create a “done” boolean and set it to false.

Now we start the main loop: Create a while loop for “while the program is not done”.

Now we need to create our menu of choices. We are going to create another enumeration similar to Assignment 3. This goes **outside** of main().

```
enum MenuOptions {
    QUIT,
    VIEW_ALL_ENTRIES, FILTER_BY_COUNTRY, FILTER_BY_LETTER
};
```

Then, within our while loop in main, we can output each option like this:

```
cout << VIEW_ALL_ENTRIES    << ". View all Entries" << endl;
```

Then, we create if statements for each command:

```
if ( choice == VIEW_ALL_ENTRIES )
```

Use the appropriate functions for each case...

`VIEW_ALL_ENTRIES`

Call the OutputAllPhonenumbers function on our phonebook instance

`FILTER_BY_COUNTRY`

First, call the OutputAllCountries function. Prompt the user to enter the name of a country. We will store this choice in a string.

After the user has typed in a country (this is case-sensitive), call the OutputPhonenumbersFilter function of the phonebook class, and pass in the country variable.

FILTER_BY_LETTER

Have the user enter a character (tell them to make it upper-case). After the user enters their char, call OutputPhonenumbersFilter in the phonebook class and pass in the char variable.

QUIT

Set the “done” variable to true.

Build, run, test!

Output Samples

View all entries...

```
Read file numbers.txt   41 phone numbers        8 countries
1. View all Entries
2. Filter by Country
3. Filter by Letter
0. Quit
1
ALL PHONE NUMBERS:
LAST_NAME
FIRST_NAME
COUNTRY
PHONE_NUMBER

Akiyama
Kazue
Japan
81-3-549-5823

Bandyopadhyay
Suresha
India
91-79-3424-5943

(etc)
```

Filter by Country...

```
1. View all Entries
2. Filter by Country
3. Filter by Letter
0. Quit
2
ALL COUNTRIES:
COUNTRY
Japan
India
France
Portugal
Ireland
Chinese
Poland


Please type a country name (case-sensitive): Portugal

PEOPLE WHO LIVE IN COUNTRY Portugal
Ferreira
Viriato
Portugal
351-38-281-9218

Medeiros
Eufemia
Portugal
351-23-383-1912

1. View all Entries
2. Filter by Country
3. Filter by Letter
0. Quit
```

Filter by Letter...

```
1. View all Entries
2. Filter by Country
3. Filter by Letter
0. Quit
3


Please type a letter (capital) to filter by: M

PEOPLE WHOSE LAST NAME STARTS WITH M
Matsumoto
Airi
Japan
81-24-6-484-2892

Medeiros
Eufemia
Portugal
351-23-383-1912

1. View all Entries
2. Filter by Country
3. Filter by Letter
0. Quit
```