Last pre-break steps with Program 4: Contact List application

How the test scripts work

This document includes some updates from the prior documents. Changes and additions are highlighted in the same color as this paragraph.

How to approach a project like this

A fundamental aspect of programming is **decomposition -** taking big problems and breaking them down into smaller problems, until we get to "problems" so small that they are solved by commands of the programming language. (For example, we don't need to break the problem "print a string to the console" into a smaller problem because we can solve that one with a line of code.)

(More about decomposition at https://cs.stanford.edu/people/nick/compdocs/Decomposition_and_Style.pdf .)

For this project, we could take either a **top-down** or **bottom-up** approach (see https://www.tutorialspoint.com/difference-between-bottom-up-model-and-top-down-model .)

For our Contact List application, we could start thinking about the options the user could select, then build out each one, breaking it into increasingly smaller parts (top-down); alternatively, because we know the kinds of actions our program needs to be able to carry out (adding, deleting, updating, retrieving contact information), we can build out those low-level capabilities, then combine them into higher-level functionality (bottom-up.)

We'll use a bottom-up approach to start this project, building out the low-level functions that we'll need. One advantage of this approach is that we can more easily do **unit testing** (see https://www.geeksforgeeks.org/unit-testing-software-testing/ .)

After the break, we'll build out the **user interface**. There are many types (see https://stackoverflow.com/a/3191873) but we're building CLI's in our class, not GUI's. We still have a lot of decisions to make with that, but after building out the classes we need, assembling the class methods into a complete program application will be relatively easy.

As you read through the additional member functions and suggestions for implementation, note how we are taking a modular approach, building up more complex functionality by combining smaller parts. (One example would be the use of the Contact.getName method in other functions.)

See the **Getting Started…** document at
https://docs.google.com/document/d/1hFVT4zYfcnMWZZ0uA-QY8VoBb5TF-bIL9EIZDdSeZk4/edit?usp=sharing

In the real world, a development team might start building the "fuly" (really, "mostly" or "largely") designed parts of a new application while other features and functionality are still being worked out. That's what we're mimicking here.

We've reached some decisions about the Contact List class. These decisions are not "right or wrong" - they are just how we decided to build this particular application. Often, we don't find out until later whether our initial decisions are good ones - we find out during testing and use whether our decisions result in some inefficiencies, limitations or errors.

We also *want* you to have the experience of maintaining/updating your code to help convince you of the value of good programming practices such as naming conventions, indentation, whitespace, comments, etc.

As you build each function of the class, test it by calling it in main. If you're unable to complete a particular function, add a **//TODO** comment for yourself with enough detail so you can complete it later.

As we've thought through the idea and tried implementing parts of the *final* application, we've realized some additions and changes we need to make to the Contact and ContactList classes.

Some items have been included in this program description just to give you the opportunity to learn about and practice things like overloading Constructors, functions and operators.

Make sure you review the **Hints, Reminders and new Techniques** document also.
https://docs.google.com/document/d/1nLevItbMf6TpTzHcZGEDiz4o-pX181tH-3hhwwPSL3Y/edit?usp=share_link

Continued on the next page.

ContactList

data member

vector of Contacts

// this should be named **contacts**
// replit and most IDEs have a find/replace feature (CTRL/CMD - F when the editing
// pane is active.

public methods

loadContactsFromFile (accepts a filename string)

// create contacts from the file and add to the vector
// return a message "success: <count> contacts added
// if the named file is not found, "failure: <filename> not found"
// this is a CSV file; you'll need to decide how to split up each line
// there are several ways; one is to use getline with a delimiter
// https://www.geeksforgeeks.org/getline-string-c/

// we decided not to create lots of Constructors for the Contact class; your
// strategy here should be to read in some info, create a contact using the
// (type, first name, last name) constructor, then read more data from the
// csv line and call other methods of the Contact class (e.g., setAddress,
// setEmail, addPhone, etc.)
// when that Contact is as complete as can be, add it to your vector

// in the file, the file header "phone" is home phone, "cell" is cell phone

getCount()

// returns number of contacts

sortContacts ()

// sorts the vector
// search the web for the vector sort method
// will require operator overloading on "less than" (<) for Contacts
// return "success" or "failure"
// **to think about** (but for now, only implement the basic sort)
// what if we wanted to sort the list on state, or type, or something else ?
// should this be public? do we want to control when the vector gets sorted?

findContactsByName(accepts a string)

// returns a vector of integers of the index positions in the Contacts vector
// if the search string is found (**case-insensitive)** in the contact name, include it
// search the web for string find method
// **to think about** (but for now, only implement the basic find)
// what if we wanted to find by type, or birth year, or something else ?

printContacts()

```
        // output the contacts
        // <separator>  (20 dash characters "-")

        // move this into the Contact.getAsString() method
        // last name, first name
        // contact type display
        // street number name city state postalcode
        // dob
        // email
        // phoneNumbers, one per line <type display>:<sp><number>

        // if you wanted to go further with modularization, this printContacts
        //   function could create a vector of index positions for the entire vector
        //   which it would pass in a call to the other printContacts function

        // such choices may seem odd when you're new to programming, but
        //   they greatly increase the maintainability of a project;  in this case, there
        //   would only be one function to update if there was some desired change
        //   to the "printing" of contacts

printContacts(vector<int> of index positions in the vector)

// sometimes we don't want to print the entire list – maybe just the results of a search
// might be useful to have a Contact.getAsString () method that returns a string

// this function and the above could call that for each, and add the separator

addContact (accepts a Contact object)
        // adds to the vector
        // returns "success: contact <lastname>, <first name> added
        //  or "failure: unable to add contact
```

~~deleteContact (hmmmm…what should this accept?)~~

```
deleteContact (int - the index position to delete)
        // returns "success: deleted contact <contact name>
        // or "failure: unable to delete contact <index position>" ~~<contact name>~~
```

string tester();   // you'll **declare** this but not define it; we'll do that in the test script

Continued on the next page.

class **Contact**

- **data**
  - char
    - contactType // P(ersonal) or W(ork)
  - string
    - firstName lastName,streetName,city,state,email, dateOfBirth
  - int
    - streetNumber, postalCode
  - vector of PhoneNumber objects named phoneNumbers
  - PhoneNumber is (a struct with string number and char phoneType
    - // W(ork), H(ome), C(ell)

- **constructors**
  - create with just a contact type, first name and last name
  - create with a contact type and phone number   // not tested in current test bench

- **basic getters and setters**
  - getFirstName
  - getLastName
  - getContactType
  - getState
  - getEmail
  - getDateOfBirth
  - setFirstName
  - setLastName
  - setContactType
  - setEmail
  - setDateOfBirth

- **other "setters"**
  - setAddress (number, street, city, state, postalCode)

- **other public member functions**
  - getName
    - // returns ln,<sp>fn
  - getAddress
    - // returns number<sp>street,<sp>city,<sp>state<sp>postalcode
  - addPhone(type, number)
    - // if type is invalid, return "failure: invalid phone type - <type>

```
          // else "success: added number <number> <type display>

          // at some point, we'll need to add some validation of the type char
          // and the phone number, but we'll leave those for later.

          // we could define our app to allow only one number of each type,
          //  but for now, we'll allow multiple numbers of the same type

          // at some point, you'll also want a helper function to get the type display
          // for a given type code


deletePhone (int index position in vector of phone numbers)
          // returns "success: deleted phone <number>  (for now, the index position)
          // or "failure: unable to delete phone <index position>"




getAsString()

          // returns contact as one string, with newline characters
          // last name, first name
          // contact type display
          // street number name, city, state postalcode
          // dob
          // email
          // phoneNumbers, one per line <type display>:<sp><number>


          // note that you can use your existing getName and getAddress
          // functions here

string tester();   // you'll declare this but not define it;  we'll do that in the test script
```

Continued on next page

Other

you'll need an overload on the "less than" operator so that we can sort contacts by name.   You can make use of the Contact.getName() method here.

Continued on next page

How the test scripts work

The test bench includes many scripts that evaluate various aspects of your Class, both data members and member functions.

They aren't meant to be obscure or difficult to understand.

The test name - **createContact, then setAddress, getAddress** for example - indicates what the script attempts to do. Here's the relevant code for that test:

```
        Contact c('W',"David", "Hayes");

        c.setAddress(123, "Main St", "Chicago", "Illinois", 60606);

        string studentResult = c.getAddress();

        string expectedResult = "123 Main St, Chicago, Illinois 60606";

        if (studentResult == expectedResult) {

           return true;

        } else {

           testFeedback << "Student code returned *" << studentResult
<< "*; expected *" << expectedResult << "*" << endl;

           return false;

        }
```

Note that this isn't evaluating the output of your **main()** function. The script works with the class data members, constructors and functions, and compares what is returned from *your* code with what is expected.

We didn't include all the test script code because it's not necessary for you to have it - the test name tells you what functions are being called and tested.

You can and should be writing such tests in your main() as you develop each function.