CA 2: Simple Use of C++ Classes, Arrays, and File I/O
CS-240: Data Structures
Assigned Tuesday September 12, 2017

Goal

The goal of CA 2 is to begin using C++ classes, a dynamically allocated array of C++ objects, and file I/O in a simple way. You will adapt and improve your code for CA 1 to allow your Donor program to (a) hold and manage multiple donor accounts, and (b) save and restore the donor information (all personal information and donation totals) across separate runs of the program. You will organize your code into classes, adhering to the CS240 file naming and organizational conventions for C++ classes (one `.cpp` file and one `.h` file per C++ class, and separate compilation into `.o` files). Finally, you will learn to dig out and use arguments that are passed from the command line through the shell to your C++ program.

Specifications

Some CA 1 information is repeated below, so that this assignment is completely self-contained, but please re-read carefully for several changes and additions.

Please write a C++ program that allows its user to input, change, and show a small amount of "donor information," for use by a fictional funding campaign. The program should support different "accounts" for multiple donor, and should hold the following things for each donor, using the indicated C++ types. Note that you should do more "type checking" of user input than you did in CA 1.

- Last Name (C++ string, contains only letters)
- First Name (C++ string, contains only letters)
- Userid (C++ string, contains at least 5 characters but no more than 10, containing only letters and digits)
- Password (C++ string, contains at least 6 characters, including at least one digit and one character that is neither a letter nor a digit)
- Age (int, must be at least 18)
- Street Number (int, must be positive)
- Street Name (C++ string, *may* include multiple words separated by spaces or tabs)
- Town (C++ string, may also include multiple words separated by spaces or tabs)
- State (C++ enum of 2-character state codes for NY, PA, and the New England states only (RI, NH, VT, MA, CT, ME)
- Zip Code (C++ string, contains exactly 5 digits)
- Amount Donated (float, non-negative, total amount donated must not grow beyond $5,000)

The program should initially prompt the user to make a selection by typing one of the following commands, with the following effects:

- Type "*Login*" to have the user present a *userid* and *password*, and to proceed to a 2nd-level menu (see below). The *userid* and *password* that the user presents should be checked for inclusion in the current set of donor accounts.
- Type "*Add*" to add an additional donor with all new donor information (*userid*, name, address, *password*, etc.) and to zero out the donor's amount donated. Please ensure that the new *userid* is not already in use and that the programmer does not try to create more than the maximum allowable number of donors (see below under "Command Line Arguments").
- Type "*Save*" to store all current donor database information, including donor account information and amounts donated, into a file. After the user enters "*Save*", your program should prompt him or her for the name of the file to save the information to, and should open that file and write into it. If the file exists, your program should warn the user that it exists and offer to overwrite its contents or allow the

user to select a different file.

- Type "*Load*" to restore donor information, including donor information and amounts donated, from a file. Again, the user should be prompted for the name of the file, and any existing donor information "in memory" should simply be overwritten by the contents of the file. The format of the input file for Load should match the format of the output file for Save, so that your program can read and write its own data. You get to decide that format; do not design the file to be used by any program other than your own. In other words, the data does not have to "look pretty" within the file (although it might), it just needs to be accessible to your own code.
- Type "*Report*" to generate a report about how many donors are in the database, and how much money has been donated to the campaign (report one number, namely the sum of all individual contributions).
- Type "*Quit*" to end the program. Before quitting, the program should ask the user whether he or she would like to save donor information into a file, and should then do so (if requested).

All other words that a user may type as input at this menu level (including misspellings, incorrect capitalization, and anything else that does not exactly match the choices above), should generate an error, but should then continue to accept commands. Please make sure your program operates correctly on well-formed "expected" input, but please also have it react appropriately to as many different kinds of malformed input as you can.

Once a user has successfully logged in as a donor, he or she should be able to enter the following commands, with the indicated results.

- Type "*Manage*" to change some or all of the donor information (other than the *userid* and *password*) associated with the record for the donor who is logged in. Please allow the user to select which fields to update, keeping all other information the same.
- Type "*Passwd*" to change the password of the donor who is logged in. Prompt the user for the old password, and require her to type the new one correctly twice. (Don't worry about not echoing input characters for the password.) If the user does not successfully enter the exact same password twice, your program should disallow the password change.
- Type "*View*" to see all donor information associated with the currently logged in donor, in well-formatted, professional, "easy to read" output.
- Type "*Donate*" to donate money to the campaign (associating it with the donor who is currently logged in). Your program should prompt the user for an amount to donate, should not allow users to enter a negative amount for deposit, and should not allow donors to donate more than a total of $5,000.
- Type "*Total*" to display the last name of the currently logged-in donor, followed by the amount he or she has donated (formatted with a dollar sign and exactly two decimal places of accuracy) on the same line of output, as follows (for example):
    - Lewis $45.33
- Type "*Logout*" to log back out to the previous top-level menu.

Your program should continue to accept commands for the current donor, until the user enters "*Logout*". As with the top-level menu, your program should support well-formed "expected" input, but should also recognize and react appropriately to as many malformed inputs as possible.

Command Line Arguments

Your program should read arguments from the command line to support the following interface:

```
./Donate <max donors> <filename>
```

The `<max donors>` parameter specifies the largest number of donors that your program will accept. The `<filename>` parameter indicates the name of a file from which to read initial donor database information. The user may omit the `<filename>` parameter, but must specify a `<max donors>` value that falls between

1 and 1000. So, for example, the user may run your program as follows, to support an initially empty donor database that can hold information for up to 25 donors:

```
$ ./Donate 25
```

Or the user may run your program to support a donor database that can hold up to 100 donors, with initial donor information found in the file named `mydonors.txt`, as follows:

```
$ ./Donate 100 mydonors.txt
```

Your program is not expected to support input files that we provide; it should read only input files that it had previously created in response to its own "*Save*" or "*Quit*" commands.

To support command line arguments in C++, parse the `argc` and `argv` parameters passed to `main()`. This will give you practice working with C/C++ arrays. The type of `argv` is "pointer to pointer to char" or equivalently "pointer to an array of character pointers." Base your solution for parsing command line arguments on the simple example in Blackboard.

Design

Please create two C++ classes, one for *Donor* objects, and one for a single *DonorDatabase* object. Each C++ class should have an associated `.h` file and a `.cpp` file to hold the class definition and member function implementations, respectively. File names should match the class names, so you should have `DonorDatabase.cpp`, `DonorDatabase.h`, `Donor.cpp`, and `Donor.h`. Set up your `makefile` to build `DonorDatabase.o` from the code for the *DonorDatabase* class, and `Donor.o` from the code for the *Donor* class, then link those object code files against a third file that includes your compiled `main()` program. As with CA 1, link everything into a single executable called "Donate".

Please reorganize and update any code that you decide to use from CA 1 into functions that reside where they "should." For example, whereas in CA 1 you may have supported the *Total* option with a free floating function (i.e. not a member function), that function should now become a public member function of the *Donor* class. Your program should hold donor account information in data members of *Donor* objects. Please also update your code to change data appropriately. For example, you may have used "pass by reference" quite a bit in CA 1. A single *Donor*'s account data should now be associated with a C++ object, and updating values should now be done more "safely" by member functions of the *Donor* class.

Your *DonorDatabase* class should be instantiated exactly once into a single *DonorDatabase* object, which should contain a dynamically allocated array of `<max donors>` *Donor* objects. Not all of these array slots may contain data at any given moment.

When you write information out to a file, or read it in from a file, the *Donor* class should "know how to" write its information to the file and read it from a file. The *DonorDatabase* class should therefore use the *Donor* class to store the data associated with however many donors are currently part of the donor database; same with reading the data from the file into memory.

Submission

Please follow the Submission Conventions described in CA 0, to produce an appropriately named "tar ball" for CA 2. Everything should unpack for us into a single directory named appropriately. Submit your code to Blackboard.