Coding Assignment 1:
Standard I/O, Separate Compilation, and Pass By Reference
CS-240: Data Structures
Tuesday August 29, 2017

Goal

The goal of Coding Assignment 1 is to familiarize you with the few basic C++ features and aspects that we have discussed in the first week or so of classes. In particular: you will write several functions that use different parameter passing styles (call by value, reference, and/or constant reference); you will use `iostream` to read in variables of different types from a user, and to produce formatted output; and you will do just a bit more separate compilation, by building two separate object (i.e. `.o`) files, and then separately linking them into a single executable. Finally, you will learn a little about how to use the linux pipe mechanism from the command line, to send input to your program from another source, and to redirect output from your program to a file.

Specifications

Please write a C++ program that allows its user to input, change, and show a small amount of generic "Donor Information", for use by a fictional Go Fund Me style fundraiser. Your program will allow its user to type in donor information, and will store that information along with an amount donated. The program should hold the following things, using the indicated C++ types:

- Donor Last Name (C++ string)
- Donor First Name (C++ string)
- Age (int)
- Street Number (int)
- Street Name (C++ string)
- Town (C++ string)
- Zip Code (C++ string)
- Amount Donated (float)

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

- Type "*Add*" to add all new donor information (name, address, etc.). This option should allow the user to *overwrite* the previous information, not really create a new additional "account". So your program should hold one set of donor information at a time. (We will extend this functionality for the next assignment, after we cover arrays, structs, and objects.)
- Type "*Manage*" to change some or all of the donor information. Please simply prompt the user for all new information, for each item associated with the account, other than the amount donated, which should remain unchanged after the update.
- Type "*View*" to see all account information in well-formatted output matching the sample executable solution that we give you.
- Type "*Donate*" to donate money to the Fundraiser. The user should be prompted for an amount to add to the Amount Donated so far.
- Type "*Total*" to display the current balance (formatted with a dollar sign and exactly two decimal places of accuracy), as follows (for example):
    - $1020.33
- Type "*Quit*" to end the program.

All other words that a user may type as input (including misspellings, incorrect capitalization, and anything else that does not exactly match the 6 choices above), should generate an error, but the program should then continue to accept commands. Please do not worry too much about other kinds of errors, such as entering a letter when a digit is expected (this will likely throw your program into an infinite loop), zip codes that do not look like zip codes, etc. For the most part, we would like your program to operate correctly on well-formed "expected" input.

Your program should continue to accept commands until the user enters "Quit".

Design

If you are a Java or C++ programmer, then the natural way to store a collection of information about a single entity (in this case a donor), is within an *object*. If you are a C programmer, you might be more inclined to use a C *struct*. If you know how to use classes/objects or structs, fell free to use them. If not, you may instead simply use separate variables for each value that you need to store.

Please write a *different function for each user specified option* above (that is, one function each to implement "Add", "Manage", "View", etc.) (You do not have to include a separate function for the "Quit" command.) All of these functions should reside together in a C++ file that is separate from the one that contains `main()`.

You should pass to a function only the information that the function needs to do its job. No more, no less. Always use the *most efficient* and *safest* calling semantics. Recall that "call by value" copies the data, "call by reference" allows the called function to change the data (which may be necessary in some cases), and "call by constant reference" sort of represents a hybrid approach (but may not always be appropriate, depending on what the function has to do). Integers take up 4 bytes of storage space on the lab machines. When considering your program's efficiency, design your code with the expectation that most strings are more than 4 characters long, and that copying even an extra byte or two when making a function call is wasteful and "matters" (even if you think it does not, in this case).

Constraints

Your lab must build from at least two different C++ source code files. Please place your `main()` function in one C++ file, and the implementation of your other supporting files in a second C++ file. These two files should build into separate `.o` files (as specified in your modified `makefile`), and the two `.o` files should link into a single executable. The C++ file that contains `main()` should be called **CA1.cpp**. The file that contains your other functions should be called **LastFirstDonorFunctions.cpp** (with "Last" and "First" replaced by your names). The two files should build into `.o` files called **CA1.o** and **LastFirstDonorFunctions.o**, respectively, and should then link into an executable called **Donate**. Therefore, after building your program with make, we should be able to type **./Donate** to run it.

Testing and Running Your Code

Even though your program will be written to retrieve input from `cin`, which can correspond to the *stream* of input that the user types into the program, you need not interact directly with your program to be able to test it. The unix "`cat`" command reads a file and prints it out to the screen. Try typing "`cat CA1.cpp`" from a shell prompt. You should see your C++ program: the contents of `CA1.cpp`. The shell *pipe* operator, '`|`' (a single vertical bar), connects the output of one program to the input of another. This means that you can type commands meant for your "`Donate`" executable into a text file, which could be called "`input.txt`", for example. You could then "`cat`" the `input.txt` file and pipe its output into `Donate`, as follows:

```
cat input.txt | ./Donate
```

Your `Donate` program should then behave exactly as if you typed the contents of `input.txt` into the program, as prompted. Of course, this requires that `input.txt` contains the right kind of input, formatted appropriately, in the right order, etc. But once you get it set, you can include many commands, you can test the same sophisticated test cases over an over until they work, etc. Try it. We will use this mechanism to test your programs.

The next thing you can do is to redirect the output of your program (or any linux program), into a file, using the linux redirection operator '>' (the greater than sign). The following linux command reads input from `input.txt`, pipes it into `Donate`, and redirects the output into a file called `output.txt` (which will be created if it doesn't already exist):

```
cat input.txt | ./Donate > output.txt
```

It may look like nothing happens when you run this command, but look what's in `output.txt`! (perhaps with "`cat`").

Submission

Please follow the submission conventions described in Coding Assignment 0, to produce an appropriately named "tar ball" for CA 1. Everything should unpack for us into a single directory named **Lastname_Firstname_CA1**. Submit your code to Blackboard.