Lab 2 - Standard I/O, Separate Compilation, and Pass By Reference

Due Date: 5:00 p.m., February 17, 2015
All function interfaces are suggested naming and parameter guidelines. If you feel there is a better way, you are free to alter names, functions interfaces, etc, as long as you follow the lab and style guidelines. Output should match exactly unless otherwise stated.

The goal of Lab 2 is to familiarize you with the few basic C++ features and aspects that we have discussed in the first weeks 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 produce formatted output; you will create a full C++ class; 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.

(Parts A must be completed in lab)

Part A: Movie List Program

Please write a C++ program that allows its user to input, change, and show a small amount of generic “Favorite Movie” info. The program should hold the following things, using the indicated C++ types:

- Last Name (C++ string - public)
- First Name (C++ string - public)
- Age (int - public)
- Favorite Movie List (C++ string array of 5 - public)

To accomplish this you should create a C++ class, User, with the above public attributes. You may either statically allocate the movies array (no constructor needed) or use a pointer, which requires you to give the class a constructor that initializes the favorite movies array using new. For simplicity, the program will only have one user object, meaning you should create a single user object on initialization, as use that object throughout the execution of your program.

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

- Type “Create” to add all new information (name, age) and to null out the movie list. This option should allow the user to overwrite the previous information, not really create a new additional account. Your program should hold one set of user information at a time.
- Type “Update” to change some or all of the user information associated with the account. You should only update First, Last, and age. Please simply prompt the user for new information for each item associated with the account.
- Type “View” to see all account information of the user in well-formatted output matching the sample output below.
- Type “Favorites” to update the titles of the favorite movies.
- 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 5 choices above), should generate an error message to the user, 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), 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

Please write a different function for each user specified option above (that is, one function each to implement “Create”, “Update”, “View”, etc.) All of these functions should reside together in a C++ file that is separate from the one that contains main(). Please write **functions**, not **methods**.

You should only need to pass the User object to each of the functions. 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).

Constraints

For full credit, please adhere to the following constraints, in addition to what is described above:

- You may not use any global variables. The variables that you use may be declared within main(), or within some other method, but not as globals.
- You must organize your data within a class (See above)
- Your lab must build from at least two different C++ source files.
    - Please place your main() function in one C++ file, and your command functions 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.
    - Your ‘User’ class itself and the command function declarations should be defined in a header file.
    - The C++ file that contains main() should be called lab2.cpp. The file that contains your other functions should be called User.cpp, and the file that contains your class and function declarations should be called User.h. The two files should build into .o files called Lab2.o and User.o, respectively, and should then link into an executable called lab2. Therefore, after building your program with make, we should be able to type ./lab2 to run it.

Show your TA your code, and that you can compile your code with a makefile.
--END OF IN LAB REQUIRED WORK--

Part B: 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 lab2.cpp” from a shell prompt. You should see your C++ program: the contents of lab2.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 “User” 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 User, as follows:
> cat input.txt | ./lab2
Your 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 have 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 and over until they work, etc. Try it. We will use this mechanism to test your programs. You can check if you code runs against our test cases when you submit ot Mimir.

Part C : Code Organization and Submission

- Required code organization:
    - lab2.cpp
    - User.h
    - User.cpp
    - makefile

        - executable should be called: lab2
            - *do not add a .exe extension*
    - readme
- Create a readme file following the format provided, and add it to your project folder
- While inside your lab 2 folder, create a zip archive with the following command
    - zip -r lab2 *
        - This creates an archive of all file and folders in the current directory called lab2.zip
        - **Do not zip the folder itself, only the files required for the lab**
- Upload the archive to Mimir under 'Lab 2'

Expected Interface and Test Output

Test Commands

- cat input.txt | ./lab2 > output.txt
  diff expected.txt output.txt
    - *Expected Test Output (with our test case):*
        - Test case: input.txt
        - output.txt

Grading Guidelines

- **Part A:**
    - Functions separated into (at least) two separate .cpp files exactly as requested, makefile builds two .o files and links in separate steps, etc. (1 points)
    - All argument passing happens using references. (1 point)
    - Each command implemented in a separate function, etc. (1 point)
    - Accepts commands until user specifies "Quit". Rejects unrecognized commands with a message, then continues to accept input. (1 point)
    - Accepts exactly 5 movies in Favorites. (1 point)
    - "Create", "Update", "Favorites",and "View" work perfectly. (2 points)
- **Part C:**
    - Follows requested project structure and naming conventions and contains written Readme (1 point)
    - Follows formatting guidelines (1 point)
    - Clean submission does not include .o files or binary (1 point)

Formatting Guidelines

- Stores all values in a named variable.

    - No Magic Numbers.
- Uses indentation to identify code blocks.
    - Every Code block should be indented from it's parent block to identify scope.
- No single letter or non-descriptive variable names
    - The only exception to this rule is 'i' in a for loop
- Separates code blocks and logical sections with whitespace
    - Optimize your code for the reader, not the writer
- Output is formatted with an explanation of the output values
    - Format your output so that someone who does not know what the program is supposed to do would know what the output meant
- Each method is preceded by a comment explaining what the method does
- Each significant code block is preceded by a comment explaining what the code block does.
    - A significant code block is more than 3 lines performing a single logical operation
- CONSTANTS are in all caps
- Only data types start with a capital letter
    - Classes, Enums, Structs, etc.
- Do not use the 'using namespace' declaration in a header (.h) file
- In general we will follow the Google C++ style guidelines. If you want more info, you can view them here: https://google.github.io/styleguide/cppguide.html