Project 0
Getting Started: A Simple Media Manager

Due: Tuesday, January 17, 2017, 11:59 PM

Notice:

The project Corrections and Clarifications page posted on the course web site become part of the specifications for this project. You should check the this page for the project frequently while you are working on it. Check also the FAQs for the project, even if you don't currently have a question in mind — you can learn a lot by reading FAQs! At a minimum, check the project web pages at the start of every work session.

Introduction

Purpose

An early project for early feedback: This project will:

- Give you early feedback on what the projects are like in this course so you can decide whether to stay in the course.
- Allow you to get thoroughly familiar with the behavior specifications before you need them for Projects 1 - 3.
- Provide some experience in code quality by designing and implementing a function hierarchy and avoiding global variables.
- Help you develop much of the code you need for Projects 1-3. This includes some experience with stream input operations and some basic string manipulation.

In more detail: The Project 0 behavior specifications are almost identical to Project 1, and these specifications are also the basic specifications for Project 2 and 3. *You will write this first project using your pre-existing knowledge of C++ programming*. There will be a few specific code organization requirements, focussed on a couple of the most important principles of good programming, described in this document below. But other than these, you will not be expected to make use of any of the specific techniques covered in this course.

The later Projects 1-3 will require that you organize the code in specific ways and use specific programming techniques (e.g. opaque types in C for Project 1; a DIY String class implementation with move semantics and exception safety in Project 2; thorough use of the STL containers and algorithms in Project 3). The code will be tested for correct input/output behavior using the autograder; these input/output tests are designed to be almost identical to those for Project 1.

In addition, the quality of your code will be assessed with some spot-checks, but unlike the later projects, this spot-checking will be very limited - the evaluation will be on whether your code follows the specific code organization requirements listed below. The code spot-check thus will not depend on using any of the sophisticated features of C or C++ that this course covers. In fact there will be no credit given for writing "fancy" or advanced code, and in fact, inexperienced programmers often write awful code when they try to use fancy or advanced techniques. This document explains below what code quality principles you will be expected to apply. If you write well-organized, straightforward, and simple code, you will have no trouble "recycling" it for Projects 1-3.

Problem Domain

This project is to write a program that keeps track of a library of movie media (DVDs, VHS tapes, etc) which can be "tagged" - grouped into named collections. Each movie corresponds to a *record*. At present the record holds on the title, medium (DVD vs. VHS, etc), a rating (on a 1-5 scale), and a record ID number assigned automatically by the program. A record can be referred to by either the title or by the ID number. There is no limit on the number of possible records. Records can be created, destroyed, or modified. Each record needs to have a unique title so that it can be reliably retrieved when the title is entered. Two copies of the same movie must be given different titles (e.g. "copy 1" and "copy 2" at the ends).

A *collection* has a string name (like "favorites") and a container of references to records. Collections can created and destroyed, and records added or removed from a collection. There is no limit on the number of collections, nor on the number of records in a collection. A record can be in any number of collections, including zero.

The set of records is called the *Library* and the set of collections is called the *Catalog*.

Later versions of this project will use additional programming techniques and have additional features; you will be able to recycle much of your code for this project in the later projects, so organizing and writing it well on the first project will make the next two projects easier.

Overview of this Document

There are two major sections: The *Program Specifications* describe what the program is supposed to do, and how it behaves. The *Programming Requirements* describes the code quality issues which you must address - the goal is to start learning and using some concepts of "good code," not just hack together some code that behaves right.

Program Specifications

The basic behavior and functionality of the program is specified in this section. As you read these specifications, study the posted samples of the program's behavior.

1. Throughout this course, "alphabetical order" for a string means the standard English language order defined by the C and C++ Standard Library functions for ordering character strings from "smallest" to "largest" (`strcmp`, `std::string::operator<`) . In this order, lower case `'a'` follows upper case `'Z'`.
2. Throughout this course, "whitespace" means spaces, newlines, tabs, etc. as defined by the C library function `isspace()`. These are characters that when displayed, produce only "whitespace" on the display.
3. To create a record, the user supplies the movie medium (e.g. "DVD" or "BluRay") as a whitespace-delimited character string, and a title as a character string that can include embedded spaces and is terminated by a newline (`\n`) character in the input. The title is stored internally with no leading or trailing whitespace, and all embedded sequences of one or more whitespace characters are replaced with a single space character (`' '`). For example, the entry `"  Much  ado  about  Nothing  "` would get trimmed at both ends and extra spaces compacted so that it stored as `"Much ado about Nothing"`. Note that the newline character is also whitespace and would be trimmed from the end. A title must contain at least one non-whitespace character. The user can specify a record by entering its title; an exact match is required, but the entered title is compacted before being compared to the titles in the records. The comparison is case-sensitive. The rating of a new record is stored in the record as 0, which appears in the output as the character 'u' for "unrated".
4. The program assigns an ID number to each record when it is created. The ID number appears in the program output. The user can enter the ID number to identify records rather than having to type the entire title. The ID number cannot be modified by the user. The program keeps a record ID counter; the first record created is given the ID number of 1, and each record created afterwards is given the next ID number in order. Destroying a record results in its ID number becoming unused, and it remains unused. That is, when another record is created, it is given the ID that is one greater than the last ID number assigned, regardless of which records have been destroyed. In the rest of this document, when the ID record counter is specified as being "reset", it means that it is changed so that the next record created will have an ID number of 1. Thus, if the Library is cleared, the ID counter is reset so that the first record subsequently created is given an ID number of 1.
5. If the user creates a new record whose title is the same as an existing record, it is an error; the new record is discarded, and the record ID counter is not updated, so that the original ID number can be used on the next record that is successfully created.
6. The Library consists of the set of individual records. When displaying the Library, the records are displayed in alphabetical order by title, with the ID number and medium shown for each record as well.
7. The user can modify the rating of a record by supplying a rating on a 1-5 scale. The rating subsequently appears whenever the record is displayed.
8. To create a collection, the user supplies a collection name (e.g. "favorites") as a whitespace-delimited character string. The collection is empty when first created. The collection is referred to by its name; an exact match is required. The user can destroy a collection by specifying its name. The user can add or remove a record to/from a collection by specifying the collection name and the record ID number. A record in a collection is a *member* of the collection. When displaying a collection, the collection name is shown followed by each record in the collection in order by title.
9. The Catalog consist of the set of collections. When displaying the Catalog, the collections are displayed in alphabetical order by name.
10. A record cannot be destroyed if it is a member of a collection. The user can clear (make empty) the Catalog, resulting in each collection being destroyed. Likewise, the user can clear the Library, but only if every individual collection is empty.
11. The program will print an error message and request the next command if a specified record or collection is not found in the corresponding container, or other problems are detected.
12. At any time, the user can save the Library and the Catalog information in a file using a supplied filename. The save file format is described below.
13. At any time, the user can restore the Library and the Catalog information previously saved in a file, replacing any current information. The ID record counter is set after the restore so that the next record created and saved by the user has a value one greater than the largest record ID found in the file, thus ensuring no conflict between old and new record IDs.
14. The user can request that the program display information about the current memory allocations, as explained more below.
15. The program is controlled by a simple command language, specified below.

The Command Language

The program prompts for a two-letter command. The first letter specifies an action, the second letter the kind of object (record, collection, etc.) to be acted upon. The command letters are followed by additional input parameters depending on the command. The program reads the command, applies error checks as described below, then executes the command, and then prompts for the next command. See "Input rules" below for specifics about how commands and parameters must be formatted in the input, and how they are checked for validity. See the sample outputs for examples.

The action letters are:

f - find (records only)

p - print

m - modify (rating only)

a - add

d - delete

c - clear

s - save

r - restore

The object letters are:

r - an individual record, or rating for the modify command

c - an individual collection

m - member for the add and delete commands

L - the Library - the set of all individual records

C - the Catalog - the set of all individual collections

A - all data - both the Library and the Catalog - for the clear, save, and restore commands

a - allocations in the print command (memory information, described below)

The possible parameters and their characteristics are:

<title> - a title string, which can be entered with redundant whitespace before, after, and internally, but is always terminated by a newline character (Return on the keyboard). The whitespace before the title (leading whitespace) cannot contain a newline character. Before being stored in a record, or compared to titles in records, the redundant whitespace (which includes any final newline character) is removed to compact the title as described above. The title is case-sensitive, so "Mars Attacks!" and "Mars attacks!" are treated as different titles. See below for how titles are to be read from the console input and files.

<ID> - a record number, which must be an integer value.

<name> - a collection name, consisting of any non-whitespace characters, but no embedded whitespace characters permitted, entered as a whitespace-delimited string. The name is case-sensitive, meaning that "Favorites" and "favorites" are two different names.

<medium> - a medium name, consisting of any non-whitespace characters, but no embedded whitespace characters permitted, entered as a whitespace-delimited string. Like <name>, the medium name is case-sensitive. Values such as "DVD" and "VHS" are customary, but the program does not require or check for any specific values.

<rating> - a rating value, which must be an integer value in the range 1 through 5 inclusive. Ratings are initially zero, meaning unrated, and are shown in the program output as the character 'u'.

<filename> - a file name for which the program's data is to be written to or saved from. This is a character string with no embedded whitespace characters, entered as a whitespace-delimited string (the platform will impose additional restrictions on possible file names).

The possible commands, their parameters, meanings, and possible errors are:

fr <title> - find and print the specified record with the matching title - the comparison is case-sensitive. Errors: title could not be read (title has less than one character); no record with that title.

pr <ID> - print the specified record with the matching ID number. Errors: unable to read an integer; no record with that ID number.

pc <name> - print collection - print each record in the collection with the specified name. Errors: no collection of that name.

pL - print all the records in the Library. Errors: none. (It is not an error if there are no records - it's a normal possibility.)

pC - print the Catalog - print all the collections in the Catalog. Errors: none. (It is not an error if there are no collections - it's a normal possibility.)

pa - print memory allocations - print the number of records and the number of collections present (see the sample outputs). Errors: none.

ar <medium> <title> - add a record to the Library. Errors: Title could not be read; a record with that title is already in the Library.

ac <name> - add a collection with the specified name. Errors: A collection of that name already exists.

am <name> <ID> - add a record to a specified collection. Errors: No collection of that name; unable to read an integer; no record with that ID number, record already a member of the collection.

mr <ID> <rating> - modify the rating of the specified record with the matching ID number. Errors: unable to read an integer (for the ID); no record with that ID number; unable to read an integer (for the rating); rating out of range.

dr <title> - delete the specified record from the Library. Errors: Title could not be read; a record with that title does not exist in the Library; the record is a member of a collection.

dc <name> - delete the specified collection from the Catalog. Errors: A collection with that name does not exist in the Catalog.

dm <name> <ID> - delete the specified record as member of the a specified collection. Errors: No collection of that name; unable to read an integer; no record with that ID number; record is not a member of the collection.

cL - clear the Library: destroy all of the records in the Library and clear the Library, but only if each collection in the Catalog is empty (has no members). Reset the record ID counter. Errors: There are collections with members.

cC - clear the Catalog: destroy all of the collections in the Catalog, and clear the Catalog. Errors: none.

cA - clear all data: first clear the Catalog as in **cC**, then clear the Library as in **cL**. Errors:none.

sA <filename> - save all data: write the Library and Catalog data to the named file. Errors: the file cannot be opened for output.

rA <filename> - restore all data - restore the Library and Catalog data from the file. Errors: the file cannot be opened for input; invalid data is found in the file (e.g. the file wasn't created by the program). In more detail, the program first attempts to open the file, and if not successful simply reports the error and prompts for a new command. If successful, it deletes all current data, resets the record ID counter, and then attempts to read the Library and Catalog data from the named file, and creates new records and collections from the file data. By the end of the restore, the record ID counter should be set to one more than the largest ID number found in the file, so that the next new record created by the user will have the next ID number in order. The final result will be to restore the program data to be identical to the time the data was saved. If an error is detected during reading the file, the error is reported and any data previously read is discarded, and the ID counter reset, leaving the Library and Catalog empty.

qq - clear all data (as in **cA**), and also destroy the Library and Catalog containers themselves, so that all memory is deallocated, and then terminate. Errors: none.

Command Input Rules

The input is to be read using simple input stream operations. You must not attempt to read an entire command line at once, or read the line a character at a time, and then parse it to find the commands and parameters; such error-prone work is both unnecessary and non-idiomatic when the stream input facilities will do most of the work for you. Check the samples on the web site to see the general way the commands work, and the type-ahead example to see the consequences of using this simple approach. The input is to be processed in a very simple way that has some odd features (such as how type-ahead works), but doing it this way, and seeing how it works, will help you understand how input streams work both in C and C++, and several other languages. If you use the basic C++ Standard Library stream input functions properly, the required code is *extremely* simple — these functions do almost all the work for you! Here are the rules; follow them carefully:

- With the following exception, your program should *read, check, and process each input data item one at a time, before the next data item is read*. This determines the order in which error messages might appear, and what is left in the input stream after error recovery is done. The exception:
  - Read both command letters before checking or branching on either of them. There is only one "Unrecognized command" error message, and it applies if either one or both of the letters are invalid.
- The *ID number* is supposed to be an integer value. Your program should try to read this data item as a decimal integer. If it fails to read an integer successfully (e.g. the user typed in "xqz" or "a12"), it outputs the error message "Could not read an integer value!". If the user had written a non-integer number such as 10.234, your program must not try to deal with it - just read for an integer, and take what you get - in this case, the value 10 (an integer). The remainder of the input is garbage, ".234" in this case, that is left in the stream to be dealt with on the next read. If this puzzles you, re-read how the input functions work when reading an integer from an input stream. If you are still puzzled, ask for help. The behavior of stream input is important to understand; previous courses might have hidden it from you. Once you have successfully read an integer, then test for validity (e.g. whether it is a valid record number).
- The *rating* is likewise supposed to be an integer value, and the same checks and considerations apply. However, once the rating value has been read, it is then checked to see if it is in range.

- Reading the *title* takes advantage of how the command language has been cleverly designed so that if a title is entered, it is always the last parameter of a command, and is also the last command parameter on a line. This makes it easy to read the title as a whole line, starting with the last character to the newline. Then process the string to remove leading, trailing, and extra whitespace. If the user does not enter any non-whitespace characters before the newline, the compaction process should result in an empty string (length of zero). So after reading and compacting, do the check: if the resulting string is not at least one character long, it is a "can't read a title" error.
- Because of the way the stream input works in C and C++, *type-ahead* is a natural result: more than one command can appear in a line of input, and the information for a single command can be spread over multiple lines; however, since a title is terminated by a newline, the title parameter for a command will always be the last parameter for the last command on a line of input. In other words, type-ahead is permitted where possible, and the program ignores excess whitespace in input commands.
- The program should not care how much whitespace (if any) there is before, between, or after the command letters.
- Whitespace is required in the input only when it is necessary to separate two data items that the stream input functions could not tell apart otherwise, such as the collection name and the following record ID. When whitespace is not required in the input, it is completely optional. See the posted type-ahead sample.
- There should be no punctuation (e.g. commas) between items of input, but your program should not attempt to check for it specifically: the normal error checking will suffice — e.g. ",3" cannot be read as an integer, and "fr,Tobruk" will be read as a request to print the record whose title starts with a comma.
- Do not attempt to check the medium entry for meaningful or standard content like "DVD". The user may enter anything here.
- The input is case-sensitive; "A Wonderful Life" and "a wonderful life" are two different titles. Likewise, "PR" is not a valid command.

Error Handling. Your program is required to detect the errors listed above and shown in the sample outputs. When an error is detected, a message is output, but the user needs to be able to easily understand what the state of the program is in terms of the command input. In this course, we will follow a simple and clear approach to error reporting: After detecting an error and outputting the message, the program will skip the remainder of the input command line. This will happen automatically if the last parameter entered and processed was a title, because all of the current line will have been read up through the newline. But otherwise, if an error is detected, the characters in the input stream up to and through the next newline are read and discarded, so as to skip the rest of the input line. *Do not skip the rest of the line any other time*.

The course web site has samples illustrating the error messages, and a file containing the strings to be used for the error messages and all other messages - copy-paste these into your code to avoid typing errors.

Clarification of reading a title and handling an error: The behavior here is subtle even though the code is very simple. In almost all cases, after printing an error message, you skip the rest of the line. The exception is that a title is normally terminated by a newline, but even if it is, it could still produce an error - like no record of that title, or another record already has that title. If you get an error after reading what is supposed to be a title, you do **not** skip the rest of the line. This makes sense because there normally isn't any "rest of the line," and if we did skip through to the next newline, we would be discarding the command(s) on the next line, which makes the type-ahead and error-handling behavior confusing in this case of relatively routine errors.

Save File Format and Load Input Rules

- The file created by the **sA** command has a simple format that can be read or edited by a human easily enough to simplify debugging the saving and loading code. See the posted samples. It consists of a series of lines containing information as follows:

  The number of records in the Library on the first line.

  The ID number, medium, rating, and title for each record in the Library, one line per record, in alphabetical order by title.

  The number of collections in the Catalog on the next line.

  The name and the number of members for the first collection (in alphabetical order by collection name)

  The title of each member, one per line, in alphabetical order.

  Similarly, the next collection appears on the next lines.
- General rules for the file format: The file is a *plain ASCII text file*, thus all numeric values are written as the ASCII character representation for decimal integers. Each item on a line is separated by a single space from the previous item on a line. The last item on a line is followed only by a newline (no spaces). Each line (including the last in the file) is terminated with a newline.
- Reading titles from the file: The program should have saved a title that contained all of the characters, and your restore code must be able to get them all back. The title in the file will follow the space after the rating in a Library entry, or the previous newline in a Catalog entry. Your code will have to arrange to skip the separator space or the newline, and then you can read the

title as a whole line and get all of the characters in the saved title. This is easy to do if you rely on how the file is supposed to be formatted. Notice also that the title is supposed to be already compacted, so your restore code should not need to recompact the title and must not waste time doing so.

- When the file is read by the **rA** command, the program should assume that the following two situations are the only ones possible:

  1. *The user provided the name of a file that was not written by the program*. In this case the file contains some other unknown data, and the program will eventually detect the problem because the random information is extremely unlikely to be consistent in the way the program expects from the file format. Thus the program can simply check that it can read a numeric value or string when one is expected, and that the right number of items of information are in the file. For example, the first thing in the file is supposed to be the number of records. If the program cannot successfully read an integer as the first datum, then it knows something is wrong. Also, if the number of records is negative, the program would know something is wrong immediately, because while zero records is a well-defined possibility, a negative number of records is not. But suppose the program could read a number as the first item, and the number is positive, but then the program hits the end of file before it finished reading the expected data for this many records. Again, the program knows something is wrong. Finally, suppose the program manages to get to the collection section of the file, but finds random garbage in what are supposed to be record titles that match those found in the record section of the file. A failure to find a member title in the newly restored Library is a signal that something is wrong; this is easy to check for, and in fact, because a collection consists of a container of pointers to records, it is necessary to look up the record anyway, so this check is naturally part of that process.

     Because of these considerable constraints present in the file format, it is reasonable to rely just on this level of checking to detect that an invalid file was specified. Thus, your program should check only for the following specific input errors when reading the file:

     - An expected numeric value could not be read because a non-digit character is present at the beginning of the expected number.
     - A numeric value for the number of records, collections, or members is negative.
     - An title specified for a collection member could not be found in the Library.
     - Premature end-of-file because the program is trying to read data when no more is present.

  2. *The file was correctly written by a correct program in response to an **sA** command*. This means that the file will have no inconsistent or incorrect information (for example, no records with the same title). Thus the program does not have to check the validity of the data any further than in Case #1 above if it can successfully read the file. Notice that although the information is divided up into lines for easier human readability, your program does not have to read the input in whole lines except for titles and should not try to - it would be just an unnecessary complication.

- If the program detects invalid data, it prints an error message (which is the same regardless of the cause) and then it deletes all current data that it might have loaded before the problem became apparent, resets the record ID counter, and prompts for a new command. Thus the possible results of issuing a **rA** command are: (1) no effect because the file could not be opened; (2) a successful restore from a successfully opened file, or (3) a successfully opened file that contained invalid data, producing an error message and completely empty Library and Catalog.
- **Hint:** If your program appears to have trouble with **rA** in the autograder, make sure you have implemented **sA** correctly, and then correctly implemented the specified checks in #1 above. Do not add additional validity checks to your **rA** command code — they are unnecessary and will just waste your time. The autograder will not test for unspecified behavior. If your code does not pass the tests involving **rA**, chances are that you have missed some of the specified checks - such as detecting premature EOF in a certain situation. Look for that instead of writing code for checks that are not specified.

Programming Requirements

For all projects in this course, you will be supplied with specific requirements for how the code is to be written and structured. The purpose of these specifications is to primarily to get you to learn and practice certain concepts and techniques, and secondarily, to make the projects uniform enough for meaningful and reliable grading. You are expected to be able to code this project using your existing knowledge of C++. For this Project 0, the programming requirements are very few because we haven't yet started on the course material on specific programming techniques.

1. ***Except for the main code file, no specific code files are specified.*** The main code file is specified below. Other than it, you are free to organize your code into as many or as few files as you choose. No Makefile is either supplied or should be submitted for this project, although you are free to use one for your own development work.

2. ***C++ compiler options.*** The program must be written in C++ compatible with gcc 5.1.0 and C++14. This does not mean that you have to use features found only in C++14, it just means that it must compile and build successfully with these options. See the course web page for how to access gcc 5.1.0 on CAEN. See below for information about submitting your code. Your code must compile and execute correctly under gcc 5.1.0 with the following options:

```
-std=c++14 -pedantic-errors
```

3. ***Standard Library only.*** You may make full use of the C++ Standard Library facilities in this project. You may not use any other preexisting libraries or facilities such as databases, parsers, etc. written by others. In other words, except for the C++ Standard Library, all of the code in the project must be written by you. Review the academic integrity rules in the course Syllabus and Policies document.

4. ***No global variables are allowed.*** To be clear, in C or C++, a global variable is a variable declared and defined so that its scope is outside a function, structure type, or class, and it is either internally- or externally-linked. Inexperienced programmers often use global variables to move information from one part of the program to another. Using global variables in place of function parameters is a very bad practice. Experienced programmers avoid global variables because decades of experience shows that if the program gets complex, you easily become confused about who changed the variable last, making a complex program extremely hard to debug and maintain. In this course, global variables are absolutely forbidden unless they are explicitly required or allowed in the project specifications. Notice that read-only constants can be defined as const variables with global scope, but these are not actually global variables. To get started on doing things correctly, *your code for this project must not declare and use any global variables*. [1]

5. ***Top-level code organization must be a well-designed function tree.*** Your program must have a top level implemented in the form of a good function tree, the fundamental programming technique for well-organized code. To get you started on designing and using a good function call hierarchy, your program must be written as follows:

- Your function `main` should be defined in a file named `p0_main.cpp`. Function `main` should contain the top level loop that reads and executes each input command. It does this by reading the two command characters and then calling a separate function, one for each command, that performs the the actual work of that command. For example, there should be a separate function that inputs the parameters and carries out the work of the **ar** command, and another function that inputs the parameters and carries out the work of the **ac** command, and so forth — a function for each command. Because the quit command **qq** should result in a `return` from `main`, it does not have to be implemented as a separate function.
- Each command function must be implemented by calling a set of "helper" functions that should be well designed to re-use code and give "single points of maintenance" — that is, if the program must do the same thing in handling several commands, it should do it by calling a function that does that thing, and this function is the single piece of code for that piece of the processing. Often these helpers can be well implemented in terms of sub-helpers. See the suggestions below for more discussion about best practices.

6. ***Output message text is supplied.*** The output messages produced by your program must match exactly with those supplied in the sample output and the supplied text strings on the project web page. Copy-paste the text strings into your program to avoid typing mistakes, and carefully compare the output messages and their sequence with the supplied output samples to be sure your program produces output that can match my version of the program. Use Unix `diff` or `sdiff` or an equivalent function on your development platform for the comparison — don't rely on doing it by eye.

7. ***Do not check for end-of-file in reading console input.*** Some of you may have been told to do this in previous courses, but don't do it in this one! The normal mode of operation of this program is interactive on the console. It will keep prompting for input until you either tell the OS to force it to quit (e.g. `cntrl-C` in Unix) or you enter a quit command. I will be testing your programs with redirected input from files just for convenience, not as part of the concept of its operation. My test files will always end with a quit command (**qq**) and yours should too! When you do input from a file, checking for end-of-file is a central part of the logic (be sure you do it correctly!). But for interactive programs, checking for `eof` on keyboard input just creates incredible clutter in the code.

Project Evaluation

Your project will be autograded (computer-graded) for correct behavior. I will announce when the autograder is ready to accept project submissions. See the instructions on the course web site for how to submit your project, and how to access gcc 5.1.0 on CAEN,

---

[1] Technically, the standard streams in the C and C++ Standard Libraries, namely `stdin`, `stdout`, `cin`, `cout`, etc. are global variables; this prohibition does not apply to the standard streams.

which is the compiler used for autograding. If your code compiles, links, and executes correctly on CAEN gcc 5.1.0 using the following commands, it should work when submitted to the autograder:

```
g++ -std=c++14 -pedantic-errors *.cpp
./a.out
```

Notice that except for the `p0_main.cpp` file, this project does not specify which files or how many files should be used to implement this project. Just be sure to submit *only* the source code files actually needed for the project! If you submit unnecessary .cpp files, such as testing files, they will be included in the build, probably causing a build failure.

I plan to do a fast and limited code quality evaluation in the form of a spot-check focussed on the issues described in the Programming Requirements section above, so pay attention to them and the suggestions below. The autograding and spot-checking scores will be combined following the rules described in the syllabus.

How to Build this Project Easily and Well

The Main Code Quality Issue in this Project

Newbie programmers fail to organize code into good subfunctions and helper functions, and instead write sloppy duplicated code - *copy-paste coding*. Almost all 381 students make this mistake on the first project. This project has been designed so that the command functions called by the top level loop can be written pretty easily with a few good "building block" helper functions. *Absolutely do not write or paste the same code over and over again for each command*. In general, watch out for places where code duplication (and attendant duplicated bugs) can be avoided by simple "helper" and "sub-helper" functions. Identify them early, and save some time! If you discover them later, refactor your code immediately to use them!

In general, if you find yourself writing code that does the same thing twice, change it into a function - see the ***Designing Functions*** section of the course C and C++ ***Coding Standards*** for more discussion of when duplicated code should be turned into a function. Creating functions for duplicated computations cuts down on errors, makes debugging easier, and clarifies the code. This is a basic principle of well-written code, and this first project is when you should start doing it. Copy-paste coding, and the mess that comes with it, is the most common code-quality problem students have on the first project in this course. Break this habit immediately!

Other Best Practices

Write small amounts of code at a time, and make it good code as you write. A common beginner's mistake is to slap together the entire project, and then try to clean up the resulting mess. It is better and more fun to develop and test the code in small pieces, taking advantage of what you learn as you go. For example, often you will discover how your code should be organized while you are writing it, but the secret is to then reorganize the code to improve its design — this is called "refactoring" the code. Learn to do this continuously as you write, so that you are effectively writing good-quality well-organized code as you go. Once you learn how to do this, it will actually save time - things get easier as the code develops, rather than harder! After the code is complete and working, take some more time to further improve your code quality and program organization. Obviously you have to start early to give yourself a chance to do this. Not only will a higher code quality score result, but you'll be able to more easily recycle your code for later projects - well-written code is reusable; a messy pile of garbage just has to be thrown away.

Put the function definitions in a readable order. Even if they write subfunctions and helper functions, newbie programmers often list *definitions* for functions in a arbitrary or haphazard order, making reading the code a *hellish experience of confusion and rummaging around*. Fix this problem as follows: *Declare* the functions at the top of the source code file(s) to make it possible to order the *definitions* to make the code easily human-readable. A good human-readable order for the functions is something like a top-down breadth-first order, with the top-level function coming first and the lowest-level helper functions coming last. Definitions of subfunctions or helper function should *always* appear *after* the first function that calls them, or possibly all functions that call them, *never before*. The idea is that you or I should be able to read the code like a well-organize technical memo: "big picture" first, details later, finest details last. Function and variable names should be chosen to make this work well. A readable order of functions is a critical aspect of code quality; don't ignore it!

Especially relevant Principles of Good Programming for this project:

- DRY — *Don't Repeat Yourself* — see the discussion above about not duplicating code.
- KISS — *Keep It Simple, Stupid!* Do the simplest thing that could possibly work. Be especially careful if you have read about various advanced techniques and concepts; often such students in this course write sophisticated but pointlessly complex and

elaborate code. This is *over-engineering*, which is a sign of the amateur beginner, not the professional. Good programmers know that the goal is clean and clear code, not a show of technical wizardry.

- YAGNI — *You Aren't Going to Need It* — don't add functionality until you need it — which means don't overgeneralize your code.

See the "Principles of Good Programming" link on the course home page for more about these and related ideas.

Recommended order of development

Simplify the program writing and debugging process by working on pieces of the program first - writing and trying to test this project all at once is difficult, unpleasant, and just plain stupid - don't do it! Your program should *grow* - you add, test, debug one piece at a time, rearranging and refactoring the code as you go, until you are finished. The result should be a well-organized body of code that was easy to build and easy to further modify or extend. Here is the recommended order of development:

1. For the main module, build the command functionality a bit at time, testing and debugging as you go. It is always easier to work with a small program, and with only a little bit of new code at a time. For example, a good place to start is with the commands for working with records in the Library, such as first adding a record and then finding or printing a record.
2. As you add commands, watch for opportunities to *refactor* the code, which means to *improve the design of existing code*. Change some of your code into functions that make you can then re-use for the next commands. The patterns and regularity in the input syntax mean that for example, you can define a function that reads a record number and returns a pointer to the record, doing the required error checks for successful read of an integer and successful lookup of a record. Thus the command function only needs to call this one function to get the user's desired record. With some thought, you can define a good set of building block functions – for example, a look-up function might be called by two other functions, one of which requires a successful lookup, and the other requires a failed lookup. If you have made good choices, the commands will become increasingly easy to do - mostly just calling already-debugged functions in new combinations. This makes coding more fun!
3. Reserve the save/restore command for last. Write the save function first, and examine the resulting file with a text editor - remember it is formatted to be human-readable. Try the save function out in a variety of situations and verify that the contents of the file are correct. Especially check boundary conditions - e.g. no records are in the Library, or some collections have no members. Only when you are satisfied that the save file is correct, should you try to get the restore command working.