Project 3
(Almost) Everything is Permitted!
Using Modern C++ in the Micro Media Manager
Due: Friday, Feb. 24, 2017, 11:59 PM

Notice:

Corrections and clarifications posted on the course web site become part of the specifications for this project. You should check the page for the project frequently while you are working on it. Check also the FAQs for the project. At a minimum, check the project web pages at the start of every work session.

Introduction

Purpose

The purpose of this project is to provide review or first experience with the following:

- Using the Standard Library strings and containers instead of your own classes.
- Using the Standard Library algorithms, inserters, iterators, `bind`, `function`, and lambdas to work with the containers.
- Using function objects and function pointers in conjunction with the containers and algorithms.
- Narrow scoping of auxiliary class, function, or constant declarations in header files.
- Using basic object-oriented programming principles to design new functionality.

Problem Domain

The functionality of this program is almost identical to that on Project 2. Unless stated otherwise, your Project 3 solution is supposed to behave exactly like Project 1 and 2 (as amended by the Corrections and Clarifications). Except for a few additional and different output strings in the `strings.txt` file, there are no supplied "starter" or skeleton files for this project - you will be using your Project 2 solution as your "starter".

There are some additional capabilities: there is a "keyword" search of titles, the ability to list the Library in descending order of ratings, displaying some statistics about Collections, and creating new collections by combining two existing collections. Most interesting is that the title of a Record can now be changed to a different title. For example, I built my movie listings with an application that searched various databases on the net for the bar code on my movies; while very useful, some of the titles were full of useless detail or had much more information than a title should have. For example, the classic Fred Astaire & Ginger Rogers film most movie enthusiasts would call *The Gay Divorcee* was in the database as *The Gay Divorcee DVD Authentic Region 1 Starring Ginger Rogers & Fred Astaire 1934*. Even worse, the opera *Sadko* got a title of *Rimsky-Korsakov - Sadko / Vladimir Galouzine, Gegam Grigorian, Sergei Alexashkin, Larissa Diadkova, Nikolai Putilin, Valery Gergiev, Kirov Opera*, which even an opera fanatic would find cumbersome. Even *Alien* got a clunker of a title: *Alien: 20th Anniversary Edition*! Clearly it would be handy to simply give such monstrosities a new simple title. The problem with changing the title is that both the Library and the contents of Collections involve keeping the Records in alphabetical order by title, so we have to ensure that both the Library and the Collections have the new correct result - the same movie should appear with its new title in the correct order, not just in the Library but in all the Collections that it was originally in.

Overview of this Document

There are two main sections: The *Program Specifications* covers only how this project is different from Project 2. The second section presents the *Programming Requirements* - this how you have to program the project in terms of the structure and organization of the code, and which specific techniques you must apply - remember the goal is to learn and apply some concepts and techniques for programming, not just hack together some code that behaves right. This section is in two Steps, described below. At the end is a section of advice on *How to Build this Project Easily*.

This project will be specified in two steps. Step 1 involves switching over from the home-grown Ordered_list and String classes in Project 2 to using the Standard Library string class, and the STL containers, algorithms, iterators, adapters, and binders. This involves a couple of small design changes, and rewriting various bits of your code to use e.g. a Standard Library container instead of Ordered_list. To make sure you get some good practice, the specifications will require that you use a variety of containers and algorithms for parts of your functionality. While the resulting code might be over-elaborate in some cases, the idea is to become familiar with a variety of Standard Library facilities. Then when they are worthwhile in your future work, you will be familiar with the ideas. But actually, my version of the project looked very "clean" as a result of meeting these specifications. You are expected to make thorough use of Standard Library facilities, not just the grudging minimum. Since Step 1 is basically changing implementation and not behavior, I will be examining and scoring your code for both quality and how completely and how well you followed the specifications to use the Standard Library facilities.

Step 2 involves design work in adding the new commands, and especially implementing the title-changing functionality. You should complete Step 1 before embarking on Step 2, but keep the needs for Step 2 in mind when you choose containers in Step 1. The old functionality will be tested separately from the new. So if you first get your Step 1 work completely done, you can check this with an early submission to the grading system, and be assured that you have successfully altered the program without damaging its functionality. Step 2 can then focus on adding the new functionality. You will have to modify some aspects of the design in your Step 1 solution, but it will be easier to modify something that works correctly and is written properly rather than try to do both Steps at the same time.

Program Specifications

What Stays the Same

Unless stated otherwise, your Project 3 solution is supposed to behave exactly like Project 1 and 2 (as amended by the Corrections and Clarifications).

Changed and New Commands

For convenience, here in one place is a description of the command changes for both steps.

> **pa** - print allocations; output the information on how many Records and Collections currently exist; this is just a subset of the previous projects' output for this command.

There are five new commands:

> **fs** <string> - find with string. The parameter <string> is a whitespace delimited sequence of characters. The program outputs all records in the Library whose title contains that sequence of characters in the title. The matching sequence in the title does not have to be a separate "word" in the title, and the match is case-insensitive. For example, "fs The" will list all records containing "The", "the", "their", "Theobald" etc. The records are output in alphabetical order by title. Error: No records contain the string.
>
> **lr** - (lower-case L, lower-case R) list ratings. If the Library is empty, the program outputs the same "empty" message as the **pL** command. Otherwise, the program outputs the contents of the Library like **pL** but in descending order of rating; records with the same rating appear in alphabetical order by title. Errors: None.
>
> **cs** - collection statistics. The Library and Catalog is scanned for Records that appear in each Collection, and statistics are output showing how many Records out of the total in the Library appear in at least one Collection, how many appear in more than one Collection, and the total of the number of Records appearing in all Collections (the total size of all Collections). See the sample output. Errors: None.
>
> **cc** <name1> <name2> <new name> - combine collections. The contents of collection <name1> and collection <name2> are combined into a new collection with the name <new name>. Either or both of the collections can be empty. The source collections are unchanged by this command. If the same record appears in both source collections, it appears only once in the new collection. Like all collections, the new collection keeps its records in alphabetical order by title. Errors: Using the same error message strings as in Project 2, check the collection names in order, and report "No collection with that name!" if there is no collection <name1>; "No collection with that name!" if there is no collection <name2>; "Catalog already has a collection with this name!" if there is already a collection with <new name>.
>
> **mt** <ID> <title> - modify title. The record whose ID number is <ID> has its title changed to <title>. All subsequent program output about that record appears with the new title, and in the correct alphabetical order by the new title. Errors: Unable to read an integer; no record with that ID; could not read a title; there is already a record with that title (for simplicity, this includes the case where the new title is the same as the old title).

Notice that the effects of Step 2 are invisible unless the new commands are used — which is why correctly written code can pass the Step 1 tests without implementing the Step 2 requirements.

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 get you to learn certain things and to make the projects uniform enough for meaningful and reliable grading. If the project specifications state that a certain thing is to be used and done, then your code must use it and do it that way. If you don't understand the specification, or believe it is ambiguous or incorrect, please ask for clarification, but if you ask whether you really have to do it the specified way, the answer will always be "yes." If something is not specified, you are free to do it any way you choose, within the constraints of good programming practice as presented in this course and any general requirements.

Design: (somewhat) free at last! In this project, you are allowed to make modifications to the public interfaces of the classes. Consequently, there will not be any testing of your components — their interface and behavior are under your control. However, all such changes must follow the recommendations for good class design presented in this course. In particular:

- You must follow anything this project document specifically requires or inherits from the Project 2 specification — like using the Error exception class. If not specifically exempted in this document, the Project 2 specifications must be followed.
- Any modifications to the classes must follow the guidelines for good design presented in the course.
- In general, although you can change the public interfaces, responsibilities, and collaborations of the classes, the basic program organization and classes should be recognizable as being based on those in Project 2. For example, you should not entertain solutions in which there are no Record objects anymore, or your Collection class has radically different responsibilities, or you compute everything by scanning a single container instead of keeping the information in three containers as in Project 2. So a fundamental redesign should not be necessary. If you are thinking otherwise, ask for clarification well ahead of time.

Step 1. Using the Standard Library Facilities

Program Modules — Changes from Project 2

The project consists of the following modules, described below in terms of the involved header and source files. Your header files can be modified from those you used in Project 2. But the Project 2 files for String and Ordered_list will not be part of your final project. There is no longer any need for p2_globals or any globals. Due to the similarity of the remaining modules with Project 2's, no skeleton header files are needed. Just make the changes specified below to your Project 2 header files.

All modules. Everywhere you used a String, use a `std::string` instead. Remove all references to any of the Project 2 globals, and remove the p2_globals files from your project; *no global variables are allowed*. You will use Standard Library containers instead of Ordered_list, so your final project will not use Ordered_list and its header file.

While you might find it useful to phase out Ordered_list gradually, before you submit the project, remove all references and #includes for String and Ordered_list, and make sure you do not accidentally submit their files with your project.

Utility.h, .cpp. These files have the same specifications as in Project 2.

Record, Collection .h, .cpp. In addition to changing the public interface of these classes, you can add additional functions or declarations as appropriate to work well with the algorithm requirements in Step 1, and then later, to implement Step 2.

Header file issues: You must follow the principles of the Header File Guidelines. Do not clutter header files with function object classes, functions, declarations, or definitions that clients could define in their own modules just as well, or the .cpp file could contain just as well. Put only the declaration in a header file and the definition in the .cpp file, if it makes it possible to `#include` fewer other headers in the header file. In addition, ensure that new declarations in the header files are placed in the narrowest scope possible: prefer making them private in the Record or Collection class to making them public; only if absolutely essential should they be scoped outside a class. Try to design any such facilities so that they work in a narrow scope. All these steps reduce coupling between the modules, making them easier to develop and maintain.

> **Function object classes:** Do not clutter header files for Utility, Record or Collection with publicly accessible function object classes or other declarations or definitions that are not part of or required for the public interface of these classes. For example, if a function object class will only be used in the main module or the Collection class's .cpp file, it is far better to put its declaration in the main module or the Collection.cpp file *just before the function that first uses it, or in the function that uses it (allowed since C++11)*. Check carefully that your #includes and header files follow the course guidelines.

p3_main.cpp. The main function and its subfunctions must be in a file named `p3_main.cpp`. The file should contain function prototypes for all of these subfunctions, then the main function, followed by the subfunctions in a reasonable human-readable order. A function template can be placed before the first call if the compiler requires it. Put function object class declarations immediately before the function that first uses them, or inside the function that uses them, if only one does. Placing them at the beginning of the file helps the human reader not at all and is a code quality failure.

All input text strings should be read from `cin` or a file directly into a `std::string` variable. Single characters should still be read into individual `char` variables.

As in Project 1 and 2, the `main()` function should define three containers, two for the Library, and one for the Catalog.

Top Level and Error Handling

The top level of your program is basically like that of Project 2: it should consist of a loop that prompts for and reads a command into two simple char variables, and then calls a function appropriate for the command. There should be a try-block wrapped around the code that calls the function, followed by the catches which output the message in the exception object, skips the rest of the input line if appropriate, and then allows the loop to prompt for a new command. Error messages are be packed up in exception objects and then thrown, caught, and the messages printed out from a single location — the catches at the base of the command loop. The possible errors, and error-handling behavior, for Step 1 are identical to those of Project 2, amended as described above. See strings.txt and the samples.

You should have the same set of top-level catches and actions as described in Project 2, except that we no longer have `String_exceptions` to catch. As in Project 2, you should have a local `try-catch(…)` around the container insertions of a freshly

new'd object; delete the object in the catch and rethrow the exception. Note we catch *all* exceptions here, not just Error or bad_alloc.

Command functions must be const-correct. In preparation for the requirement in the next paragraph, each command function should take as reference arguments the two Library containers and the Catalog container, and promise not to modify the containers it does not need to modify. Examples: the print-record command functions should promise not to modify any of these containers, and so should take all three of them by const reference; the add-record command function must modify the two Library containers, but should promise not to modify the Catalog container. The delete-all command function must modify all three containers. You should declare and define the command functions to use const reference parameters for these containers wherever the function will not be modifying them. If your Project 2 code had command functions that were not const-correct, or did not take these three containers as parameters[1], modify them before going any further; remember you have to change both the declarations and definitions. If you get problems with const-correctness in helper functions, you can handle it by defining the helpers using templates, as described in the Project 2 document. Be sure your program builds and behaves correctly before going further.

Fun with a map of function pointers! But be const-correct! Do the following to map between commands and functions that do the commands: Concatenate the two characters of the command into a single std::string, and then use it as the key value for a map container that gives you a pointer to the command function. Before processing any commands, the program loads the container with the function pointers for each possible command. Unrecognized command strings should not be allowed to fill up the container, either by not putting them in, or immediately erasing them. Because of its special status, the quit command should be tested for directly; it should not be called using the map container because it is awkward to cause a return from main from a subfunction.

> ***Note:*** In C++, calling exit() should not be done except in an emergency because it bails out to the OS immediately, bypassing any destructors that would be called in a return from the current function and in a return from main(). Thus your project should always terminate with a return from main(). Also, for this course, don't return anything except 0 from main() — it will confuse the current autograder which thinks a non-zero return is a program crash of some sort.

What do we map to? And how do we stay const-correct? Recall that the Standard Library containers always hold objects of the same type. So if the map contains function pointers for the command functions, all of the command functions have to match a single function pointer type, so they would have to have identical parameter lists and return types. The only non-horrible way to do this is to have them all take the three containers as modifiable reference arguments, even for functions that don't modify them. But this contradicts the important goal of making the command functions const-correct. So we can't just do the following:

```
// const-correct functions:
pr_command_func(const Lib_ti_t& lib_ti, const Lib_id_t& lib_id, const Cat_t& cat);
ar_command_func(Lib_ti_t& lib_ti, Lib_id_t& lib_id, const Cat_t& cat);
// etc.

// below fails to compile because the function pointer types are incompatible
map<string, void (*)(Lib_ti_t&, Lib_id_t&, Cat_t&)> command_map =
    { {"pr", pr_command_func}, {"ar", ar_command_func}, etc };
```

Using do-it-yourself (DIY) wrappers: If we define some functions that take reference parameters and simply call the actual command function, then we could keep the command functions const-correct, and still have identical signatures to populate the map container. For example:

```
// wrappers that give same parameter list for const-correct command functions
void call_pr(Lib_ti_t& lib_ti, Lib_id_t& lib_id, Cat_t& cat)
    {pr_command_function(lib_ti, lib_id, cat);}
void call_ar(Lib_ti_t& lib_ti, Lib_id_t& lib_id, Cat_t& cat)
    {ar_command_function(lib_ti, lib_id, cat);}
// etc.

//now this compiles because function pointers have the same type:
map<string, void (*)(Rooms_t&, People_t&)> command_map =
    { {"pr", call_pr}, {"ar", call_ar}, etc };
```

This works because you can always pass non-const objects to a function that promises not to modify them - this is never a problem, and the language rules allow it. (What is disallowed is passing const objects to a function that might modify them.) However, writing a wrapper for every command function is remarkably ugly — don't do this.

Getting wrappers for free: We'll double down with the Standard Library: we can use std::function<> to *automatically* generate these wrappers for us in the form of function objects that take the by-reference parameters and call the const-correct functions. Recall that a std::function<> function object can hold *any object that can be called like a function using the specified parameters, and the type of these function objects is always the same, no matter what the called object is like*. This gives us the same effect as the above wrapper functions, but they are generated for us. So instead of a map of strings to function pointers, declare

---

1 Tip: If you do not need to use a parameter in a function, do not give it a name in the parameter list - this means "unused parameter" and the compiler should not pester you with an "unused variable" warning.

```
map<string, std::function<void (Lib_ti_t&, Lib_id_t&, Cat_t&)>>
```

Because the constructor for `std::function` will take a function pointer argument, the initialization for this map is just as simple as the function pointer map in the above examples.

Basic Programming Restrictions

This project is to be programmed in pure and idiomatic Standard C++ only. Everything must be done with typesafe C++ I/O, `new`/`delete`, Standard C++ Library facilities and the classes and templates that you write. Standard C Library facilities are needed only for the assert macro in `<cassert>` and the basic character classification and manipulation functions in `<cctype>`. You do not need, and must not try to use, any declared or dynamically allocated built-in arrays anywhere in your code. This prohibition does not apply to the good practice of defining `const char* const` variables pointing to C-string literals for output messages.

There are still some C++ facilities that you may not use yet:

- You may not use inheritance in this project. In fact, it would add no value.
- You may not use the Standard Library smart pointers yet (`shared_ptr`, `unique_ptr`). We'll use these in a later project.
- You may not use the Standard Library string streams.

Your code is expected to follow the C++ Coding Standards document for the course.

Container Requirements

You will choose which kind of Standard Library container to use for each kind of objects or pointers. However, you must have some variety in your containers, so you must use each of {`map<>`, `set<>`, `vector<>`} *at least once each in the project*. The `map<>` container is already being used for the command map. You must choose the remaining containers under the following restrictions:

- One of the three containers in the Library and Catalog must be a `set<>` , one must be `vector<>`, and the remaining one must be one of {`map<>`, `set<>`, `vector<>`}.
- The Library containers must hold *pointers* to Records; the Catalog container must hold Collection *objects*. Any other containers in the project may hold either pointers or objects as you choose.
- The `map<>` container can thus be used more than once in the whole project.
- Any time a container is *searched* for a supplied Record by title or ID, or a Collection, the search must be done using a method that is logarithmic in time instead of linear (or worse).
- At least one of the uses of `vector<>` must be used with `binary_search` and/or `lower_bound` to search for items and the locations of where to insert them.
- At your option, if you need a sequence container that is not searched, you are free to use a `list<>` for this purpose.
- As before, the Library and Catalog containers must persist from command to command — for example, you can't generate the contents of the Catalog "on demand" for the **pC** command — rather, the Catalog container must be kept up to date when other commands change it, and all the **pC** command does is output the current contents.
- If you use additional containers in the project — such as for Step 2 — they do not have to be persistent across commands. For example, a container of the output data needed for the **lr** command can be generated when processing the command, printed out, and then discarded.
- The other Standard Library containers may **not** be used in this project; they are all relatively specialized variations on the required and optional four listed above; this focus maximizes the learning benefit.

These container requirements apply to the final version of the project that incorporates both Step 1 and Step 2, so you don't have to meet them with only the Step 1 version of your code. In fact, you may want to change your Step 1 choices in light of what you discover when you do Step 2. If your code is well organized (e.g. well-designed "helpers" in the main module) and you use some good typedefs or type aliases, changing containers can be easy. There are choices that make the project code very simple and smooth.

Algorithm Requirements

You may NOT use explicit `for`, `range-for`, `while`, or `do-while` loops in this project; you must use a Standard Library algorithm instead. Use Standard Library algorithms like `find`, `copy`, and `for_each` to operate on the containers instead of writing out explicit loops like

```
for(it = catalog.begin(); it != catalog.end(), ++it) {crunch crunch}
```

This will require using some of the STL iterators, inserters, `bind`, lambda expressions, and writing some additional simple functions and/or function objects, and surveying the list of algorithms to find ones that will work well — go beyond `find` and `for_each` and you will be pleasantly surprised. For example, there are some good places to use `copy_if` and `stable_sort`. The result might be unnecessarily fancy for this simple program, but it is good to get the practice so you will know enough to save tons of coding in more complex cases. In short, now is the time to climb the learning curve on getting acquainted with the Standard Library facilities. If you get stuck on a particular situation, check the code examples on the course home page, and then ask for help.

In fact, after practicing with Step 1, you should find that Step 2 becomes downright fun to write. Each new command typically involves only a few lines of new code to do the work, and most of these are STL algorithm one-liners.

- There are *exactly four* exceptions to this algorithm restriction, and they all *require* explicit loops:
  1. Your top-level command loop that reads in the two command letters and dispatches the command *must* be an explicit `while` or `do-while`.
  2. Your file restore code for implementing the **rA** command in the main module, and Collection class, *must* use explicit loops to control the reading of the data and creation of the objects.
  3. You should have a function that skips the rest of the line for error recovery, and this must use either an explicit `while` loop, or you can use the `ignore()` function instead (see the handout).
  4. **Exactly once** in the project, you must use a `range for` with a container to get a bit of practice using it. To get credit for this, you must put a comment before it that contains the string "the one range for" so that I can find it easily. The body of the `range for` should be plain code (as opposed to using a function object, `bind`, or a lambda expression). The reason for this one-use restriction is to force you to practice using the algorithms even when they are clunky. No such restriction in future projects!
- *No credit will be given for trying to use an STL algorithm for the first three exceptions*, and in fact, trying to do so will produce convoluted code, detracting from the code quality — that's why these exceptions are specified.

Specific requirements for algorithms and containers

- Any time a container is *searched* for a supplied Record <ID> or <title>, or a Collection <name> parameter, the search must be done using a method that is logarithmic in time instead of linear (or worse).
- You must use the `copy` algorithm with an output stream iterator *at least once*, and everywhere it works in a simple way (check Stroustrup, lecture notes, and example code). Remember you can define additional `operator<<` overloads to make this easy (see the Handout).
- You must use a lambda with a captured variable with an algorithm at least once.

  Note: Do not store the lambda object in a variable, even temporarily. Lambdas are supposed to be short and sweet, and written "in place." If the code in a lambda is complicated enough to take several or many lines to be readable, prefer a function or function object instead. Likewise, if you need it in more than one place in the code, define a function or function object class instead — passing lambdas around in variables is an unusual thing to do because what they offer is their value as "in-place unnamed functions."
- At least once with an algorithm, you must use `std::bind` with at least one bound value that is not a placeholder.
- You must use a custom function object class that has private member variables with reader ("getter") functions in combination with an algorithm and a container at least once in the project. *Hint:* Consider the **cs** command.
- Your code for condensing a title may not use any explicit loops, but must use the `std::string` member functions and/or its iterator interface together with algorithms and functions or function objects. If you use a function object for this purpose, it does not count towards the custom function object class requirement above. *Important*: You may not use a `stream string` to help condense the title — instead you should get familiar with the `std::string` interface.
- In all other places in your code where you use a Standard Library algorithm, you must *not* use your own helper function, custom function object class, or a lambda expression if instead you can simply use `bind`, `mem_fn`, a stream iterator, or some other Standard Library facility instead. The goal is to practice using these facilities everywhere they work well. For example, a common error in this project is to solve all problems with lambdas — this limits your skill and often results in more verbose code.
- See the lecture notes or the Coding Standards for how to choose between custom function objects, `bind`, `mem_fn`, or a lambda expression.
- Don't use the C++98 adapters and binders like `bind1st`, `bind2nd`, `mem_fun`, `mem_fun_ref`, `ptr_fun`. These have been deprecated because the C++11 `bind` and `mem_fn` work so much better; they are now historical curiosities.

Step 2. Adding New Capabilities

Adding the new commands to the project in some cases is very simple, but in other cases, you have to do some significant design work because you must modify the responsibilities and collaborations of the classes and the main module. For each piece of work to be done, figure out which class/module is best suited to handle it — this is the class or module that "knows" the relevant information. Avoid making the main module the "*god module*" *that knows all, sees all, and does all* to solve a problem — the basic approach in object-oriented programming is that the main module delegates the work to the components best suited to do it.

The new **mt** command requires the most careful thought. In Project 2's version of Record, there was no way to modify the title of a Record once it had been created. This helped ensure that a container of Records ordered by title could not be accidentally disordered by incorrect client code. Thus we expressed a design concept in a way that enables the compiler to help us follow and stick to the design concept. However, for this Project, we need to be able to change the title of a Record without corrupting or confusing the program's data. You must come up with a good design for implementing this capability. To stimulate thinking more widely, note that we need to get at least the *effect* of changing the title of a Record. Whether this is done by changing the existing Record object or creating a new Record object is a design decision for you to make.

You are allowed to modify the Record and Collection classes, including their public interfaces, to arrive at good solutions for the new commands. A good solution will have a direct expression of the design concept in the code, be compact and economical (such as not grossly duplicating data), and retain a good class design for the classes. Remember that the key to a good class design is a clear concept of which responsibilities each class or module will take on.

You should be considering only a very few additional member functions or variables; the interface and members from Step 1 should not be affected very much. If you discover a need to make drastic changes, you might be pursuing a poor solution — get some sleep or discuss it with me.

Algorithm and Container requirements

The restrictions and requirements stated above for Step 1 also apply to Step 2. But they apply to the complete project, not the code for the individual steps.

Project Grading

We 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.

There will be two sets of autograder tests. The first set will be similar to those of Project 2, and will not involve the new commands. Thus your program should be able to pass these tests after Step 1. The second set will involve the new commands.

Since you are allowed to modify the public interfaces of the classes, I will not be testing individual components of your code by mixing them with my own.

This project will be both autograder tested and hand-graded, so study the Syllabus information on how the autograder and code quality scores will be determined and weighted. Pay attention to commenting, organization, clarity, and efficiency consistent with good organization and clarity. Study and apply the C++ Coding Standards handout. Review the feedback you got on your Project 1 solution. Do not expect to do well on code quality by slapping the project together at the last minute.

The evaluation will include:

- Quality issues similar to the Project 1 evaluation (in their C++ version). Be sure you modified your code to follow C++ idioms instead of C idioms (e.g. using `nullptr` instead of `NULL`, `bool` instead of `int` for true/false values). The C++ Coding Standards handout covers many of these issues. Be sure to assess your code against the C++ Coding Standards before your final submission.
- Whether your code met the above specifications for using the above requirements such as the Standard Library facilities, and how well you used them. Check your code against the project specifications carefully.
- How good your design solution in Step 2 is, both in how well it follows the course concepts, and especially whether it results in a simple and clear code structure with good division of responsibilities and collaborations.

How to Build this Project Easily and Well

Step 1.

If you were thinking of getting a copy of Josuttis, now is the time. Otherwise, keep your Stroustrup handy and open to the containers and algorithms discussion. Check also the lecture notes, handouts, and Stroustrup's *Tour* for C++11 facilities, and the posted code examples to see how to use the containers, algorithms, adapters, and binders. Remember: don't go rummaging through possible trash on the web; use the provided resources first, and then ask for help.

First, refactor and clean up your code in response to the Project 1 code quality evaluation. The same issues (or C++ versions of them) will be evaluated in Project 3, so take advantage of the feedback!

Second, convert your Project 2 over to use Standard Library facilities. Changing over to `std::string` instead of String is trivial, so get that out of the way first. Most IDE's have a multiple-file search and replace that makes this a minute's work. Change the top-level command dispatching.

Third, choose your containers; choose wisely! Some container choices will be clumsy, others will be very clean and elegant. You can change them one at a time and verify your program still behaves correctly. To specify the ordering relations, follow the syntactically simpler and idiomatic approach: *declare the container with a function object class* instead of initializing the container with a function pointer (you can do this with the STL containers, but you should reserve it for the case where your code must choose the ordering function at run time when the container is created). Consider using heterogenous lookup instead of creating probe objects.

Specifying ordering in the associative containers. `map<>` and `set<>` assume a default ordering using `less<T>`, an STL function object class that simply applies `T::operator<`. This is used to order the key values in the `map<>`, or the objects in the `set<>`, respectively. You can supply your own function object class to use as an ordering as the optional third template parameter for `map<>` and the optional second template parameter for `set<>`. Compare this to Project 2's `Ordered_list` template and the default `Less_than_ref` templated function object class - this was modeled after the STL approach. See the code examples on the course website for examples.

Gentle introduction to algorithms and gizmos. To get gently introduced to using the algorithms and gizmos like lambda, bind, function objects, stream iterators, etc, keep your explicit loops from Project 2 (after modifying them as needed to suit the container), and then change them one at a time to use the Standard Library algorithms and gizmos. Recompile and check the program after you change each one. This will mean you have to deal with crazy template error messages on only one thing at a time. It will get easier as you learn more.

Change the container type if it helps. As you work through the project, you may discover that a choice you made for the container was not a good one — while it worked well in one place, it was too awkward in another. The `map<>` container is especially clumsy because it contains `pair` items which often interferes with simple ways of doing things. Don't hesitate to change the container type for a better overall result. If your code is well organized, and you used typedefs, changing the container will be very easy — for example, most of the code using STL algorithms will stay the same!

Key concept. The STL algorithms like `for_each` all run an iterator over a container and do something, like call a function, with *the value from the dereferenced iterator.* The key to using them easily is to keep in mind what kind of a object the dereferenced iterator is. Your function/function object/bind object/lambda object must accept a parameter of this type. This is especially important for the `std::map<>` container, which holds `std::pairs`; if you dereference a `std::map<>::iterator`, you get a `std::pair`. To use an algorithm that traverses the map you usually need a way to pick out the `.second` member. Sometimes you can do this easily, such as writing an `operator<<` that takes a pair and outputs only the `.second` member, or a lambda, helper function, or custom function object that does takes a pair parameter. The `bind` or `mem_fn` facility is the preferable solution if you already have a function or member function and just need to call it.

> *Sanity preserving hint:* Instead of declaring the full `pair` type as a function parameter, take advantage of how the STL containers contain a `typedef` or type alias for the type of object in the container, namely `value_type`. For example, to declare a function that takes a `pair` from a `map<int, string>`, simply declare:
>
> ```
> void my_function(map<int, string>::value_type& the_pair);
> ```
>
> Of course, if the `map` container has been `typedef`'d or type aliased this declaration could be even simpler:
>
> ```
> void my_function(My_map_t::value_type& the_pair);
> ```

Beware of egregiously inefficient code. A bit of inefficiency can be a good trade for code that is especially simple and easy to work with in the future. But egregiously inefficient code is either grossly inefficient, or inefficient for no good reason — either there is no design or simplicity advantage, or the code could have been made more efficient with a bit more thought (lazy or rushed programming is no justification, although in the real world it can be unavoidable sometimes).

Good choices of containers and algorithms will eliminate many cases of pointless inefficiency. In addition, this project has several opportunities to easily apply move semantics and related efficiency techniques. So look for places to use move insertion or emplace when you put an object into a container, and for opportunities to use member-function swap and move construction or assignment.

Remember that moving a built-in type such as a pointer into a container is the same thing as copying it into the container, so the move operations are only relevant to dealing with complex objects.

A couple of additional goals: In the processing of a command, avoid unnecessary data movement or data copying, or creating the same data multiple times, or searching for the same object more than once. You can usually refactor your design to make these inefficiencies unnecessary.

But make a copy of a container if it helps with using STL algorithms in this project. Some containers work very well for some purposes but not for others. For the purpose of the learning goals of this project, it is OK to copy a container's data into a temporary new container of another type if it allows a particular problem to be solved especially well with the STL. This would not count as egregious inefficiency in the project evaluation if the solution is a good STL exercise (I like my **mt** implementation!) Of course, you must use an algorithm or the container's constructor to make that copy — no loops allowed for this purpose!

Look beyond `for_each`. There are lots of algorithms that do useful things. C++11 has one of my favorites, `copy_if`, which was accidentally left out of C++98! Now is the time to learn something about them — study Stroustrup's list. Keep in mind that some algorithms make sense only if applied to some kinds of containers, and alternately, that some container types will do work directly that would require using algorithms to accomplish with other container types.

Choose function return types wisely. Don't get stuck on the approach of returning iterators from helper functions — while often handy, the problem with returning an iterator is that the client usually has to declare the iterator type (which `auto` helps with) or access the container to interpret the iterator (e.g. to tell whether it is `== .end()`). Elegant and more useful helper functions can be written that return references to Collections and pointers to Records, resulting in simpler and more understandable code. Remember that if something has gone wrong, an exception will be thrown, so you generally don't have to worry about how to make the returned value mean "not good" nor check the returned value for validity. Your code won't be thinking about a returned value if the function threw an exception!

Keep const promises. A `std::set<>` container has a simpler interface than `std::map<>`, but its items are supposed to be unmodifiable so that the contents cannot be changed in ways that will invalidate the order in the container. If you want to use it for changeable objects, remember that the most preferable approach is to change the object while it is not in the container. If you keep pointers in the container, then the pointers cannot be changed, but you can point to the objects with non-const pointers. But in this case, it is up to you to ensure that the changes you make will not result in disordering the container. Review the lecture notes summary on this issue.

Using a `const_cast` is a sign of design failure — if you need to change something, why did you make it unmodifiable to start with? — and don't abuse `mutable` (see the Coding Standards for the only acceptable use in this course). Correct the design rather than making a mess!

Dealing with set<>. A `std::set<>` container has a simpler interface than `std::map<>`, but under the Standard, its objects are made unmodifiable to make sure that contents stay in order. If you want to use `set<>` for changeable objects, it can be done safely if you change the object while it is not in the container — the copy-remove-change-insert strategy. But if the objects are complex, this is a very inefficient and clumsy container choice — use something better. In contrast, if you keep pointers in the container, then the pointers cannot be changed, but you can point to the objects with non-`const` pointers, which enables you to change the object using the pointer. But in this case, it is up to you to ensure that the changes you make will not result in disordering the container. Review the lecture notes summary on this issue.

Using a vector? Use binary search! The `binary_search` and `lower_bound` algorithms provide a fast log-time search when used with vector<>, but their behavior is somewhat puzzling. When applied to a sorted sequence container, the `binary_search` algorithm will tell you whether the matching item is present, but not where it is! `lower_bound` also does a binary search and returns an iterator, but it doesn't necessarily tell you whether the item is present! Specifically:

- If the matching item is present, `lower_bound` returns an iterator that points to the matching item in the sequence.
- If the matching item is not present, the iterator points to where the sought-for item should be inserted (e.g. with a call to the `insert` member function that takes an iterator argument).

If you need to know both whether the item is present, and where to find it or insert it, you can do it all with a single call to `lower_bound` as follows:

- A returned value of `.end()` means either that the item is not present, or it should be inserted at the end (which the insert function would automatically do if given this iterator value). So a `.end()` result is unambiguous: if you are looking for the item, it is not there; if you want to know where to put it as a new item, it's at the end.
- A non-`.end()` value means either "here it is" or "here is where to put it". How do you tell? Test to see if the iterator is pointing to an item that matches what you are looking for (such as a Collection that has the sought-for name). If it matches, then "here it is." If not, it means "here is where to put it."

Consider creating your own function template for binary searching that, like the suggestion in Project 1, gives you both "is it there" and "where is it/where to put it."

Step 2.

When you start on Step 2, keep an open mind about whether and how you might modify your Step 1 solution. If you did a good job in Step 1, you will find it relatively easy to make whatever design changes will help you do Step 2. Don't let your Step 1 solution prevent you from arriving at a really nice Step 2 solution.

The single most important concept in Step 2 is *making good decisions on which component should be responsible for what part of the work*. The component that is responsible for the data and has the relevant information should be doing the work. Learning how to think about this is essential to being able to take advantage of Object-Oriented Programming! Set aside some time for this; it is not as easy as you might think!

Don't just jump in and start coding. Think about the design and sleep on it.

- Design Tip #1: Design the public interface(s) first; don't worry, or even think, about the private implementation. Getting distracted about how you will code stuff before you are even sure what you need is the main enemy of creative design.
- Design Tip #2: Delay writing code for as long as you can stand it. For example, sketch out how the public interfaces will be used by clients in pseudocode. Actual code has a habit of setting your thinking in concrete; you need an open mind.
- Design Tip #3: Once you have worked out a design idea, try to think up at least one more reasonable (non-stupid) alternative design. Then consider the pros and cons of each design idea, and pick the best one. There are always tradeoffs because no design is perfect, but if you go with the first design that you happen to think of, you can end up with a horrible one instead a good one.
- Design Tip #4: If your design idea turns out to be a bad one, throw it away and start over, maybe with one of the alternatives. This is almost always faster than trying to get a bad design working properly. This is because you now understand the problem better, and good designs code easily because it is clear what needs to be done, and where it will be done.

A final suggestion: Spend some time thinking about the design before you start on Step 1, and whenever you take a break while working on Step 1. Doing this thinking before you actually start on Step 2 will help you follow Tips #1 and #2 above.