Project 3
The STL is Your Friend
Due: Friday, Feb. 26, 2010, 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 this 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 STL containers instead of your own classes.
- Using the STL algorithms, iterators, adapters, and binders with the containers, and optionally become introduced to the Technical Report No. 1 (TR1) adapters and binders, which are scheduled to be part of the next Standard Library.
- Using function objects and function pointers in conjunction with the containers and algorithms.
- 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, and creating new collections by combining two existing collections. Most interesting is that the title of a record can now be changed. This is to allow edits of titles as opposed to wholesale changes of titles. For example, I built my movie listings with an application that searched various data bases 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 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 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 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 practice using a variety of containers and algorithms, the specifications will require that you use a variety of containers and algorithms for parts of your functionality. These are not necessarily the best choices in all places, and on occasion will seem like more work than it is worth, but the idea is 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. You are expected to make thorough use of Standard Library facilities, not just the grudging minimum.

Step 2 involves design work in adding the new commands, and especially implementing the title-changing functionality. It is strongly recommended that you complete Step 1 before embarking on Step 2. 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 four new commands:

> **fs** <string> - find with string. The parameter <string> is a whitespace delimited string 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.
>
> **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 in the new collection. Like all collections, the new collection keeps its records in alphabetical order by title. Errors: No collection with name <name1>; no collection with name <name2>; there is already a collection named <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 title. Errors: No record with that ID; there is already a record with that title.

Notice that the effects of Step 2 is 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 to it any way you choose, within the constraints of good programming practice 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. 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.*

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 basic specifications as in Project 2, but remove the `swapem` function template. There is a similar one in the Standard Library, named `swap` which (if necessary) you must use instead.

Record, Collection.h, .cpp. In addition to changing the public interface of these classes, you can add additional functions or declarations as appropriate. **However:** You must follow the principles of the Header File Guidelines. Do not clutter header files for with function object classes, functions, or declarations that clients could define in their own modules just as well. 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. The best place to put function object classes is immediately before the function that first uses them - placing them at the beginning of the file helps the human reader not at all.

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 a single character variable.

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 this followed by a `catch` which outputs the message in the exception object, skips the rest of the input line, 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 catch 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.

Fun with a map of function pointers! 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 this 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; calling `exit()` should *not* be done except in an emergency in C++ because it bypasses any destructors that would be called in a return from `main()`.

Note also that all of the functions called using the map of function pointers must have identical parameter lists and return types, meaning that they will all have to be called with the Library and Catalog containers as arguments. If you wish, you can use an old trick to simplify the parameter lists: Declare a simple `struct` type and make these variables the members of the struct type. Declare a variable of the structure type in `main()`, and then you can simply use this one variable, passed in by reference, as a short-hand to move in all variables by reference at once.

std::getline is different! Note that `std::getline(std::istream&, std::string&)` treats newline differently than both `fgets` and the version of `getline` that you wrote for `String` in Project 2. The function `std::getline` consumes the newline from the stream, but does *not* copy it into the destination string. Thus, as in Project 1, you have to arrange to not skip the rest of the line if the title was the last thing entered before an error was detected. *Hint:* Considering defining another kind of class for exceptions.

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 any 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 this program. This prohibition does not apply to the good practice of defining `constant char * const` variables pointing to C-string literals for output messages. Your code must follow the C++ Coding Standards document for the course.

Container Requirements

You will choose which kind of STL container to use for each collection of objects or pointers. However, you must have some variety in your containers. Because the `map<>` container is already being used for the command map, you can't use it in the rest of your program. Instead, you must use each of { `set<>`, `vector<>`, `list<>` } at least once each. 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.

As in the previous project, the Library containers must hold *pointers* to Records; the container for Catalog must be a container of Collection *objects*. Other containers are your choice. Also as before, the Library and Catalog containers must persist from command to command - 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.

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 it 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. There are choices that make the project code very simple and smooth; take some care with this choice, and use typedefs to simplify matters if your change your mind.

Algorithm Requirements

Use STL algorithms instead of explicit for, while, or do-while loops

A purpose of the project is to get some practice with fully using the STL as it is meant to be used, so you can take advantage of it in the future. Use Standard Library algorithms like `find` and `for_each` and the others to operate on a container instead of writing out explicit loops like `for(it = catalog.begin(); it != catalog.end(), ++it) {crunch crunch}`. This will require using the iterators, adapters, and binders, and surveying the list of algorithms to find ones that will work well (don't stick with only `find` and `for_each`). However, in a few cases, using an STL algorithm will require writing an additional function and/or a function object class. These will be very simple conceptually, but somewhat verbose, and there might be more lines of code involved than in the explicit loop version. But overall, with a good choice of containers and algorithms, the project code will be much simpler and more expressive than the explicit loop version.

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. If you get stuck on a particular situation, check the code examples on the course home page, and then ask for help.

- 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 creation of the objects and reading the data.
  3. You should have a function that skips the rest of the line when needed for error recovery, and you may have another function that reads and discards whitespace or newline characters in an input stream, and these *must* use an explicit `while` loop.
  4. You *must* write a version of an algorithm called `copy_if` and place it in your Utility.h. This algorithm is like `copy`, but only copies an item if a supplied predicate is true for that item. See Stroustrup for an example implementation (other sources are out of bounds - see the Syllabus academic integrity rules). Of course, this algorithm, which can be very handy in the project, uses an explicit `for` loop.
- No credit will be given for trying to use an Standard algorithm for these four exceptions, and in fact, trying to do so will require convoluted code, detracting from the code quality - that's why the exceptions are specified.

Other algorithm requirements

- 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 instead of linear. Note that this rules out using a `list<>` container for certain purposes.
- Your program must use an output stream iterator wherever appropriate.
- Your program must use the adapters like `mem_fun`, the inserters, and the binders everywhere that they can be used in an algorithm to accomplish a task. Write a function object class for the other cases.
- Your code for condensing a title may not use any explicit loops, but must the `std::string` member functions and/or its iterator interface together with algorithms and function objects. *Hint*: an elegant and remarkably simple solution, needing only a few lines of code, is possible with an algorithm and a somewhat "smart" function object.

- At your option, you can use the TR1 facilities such as `mem_fn` and `bind` instead of the Standard Library facilities. These are already available if you are using gcc 4.x (on Unix/Linux or Mac Xcode). See the posted examples. If you are using MSVS, you may have to download and install a TR1 library. This is not recommended unless you embark on it immediately, to give time to test the installation and back out if necessary.

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, decide which class/module is best suited to handle it because it "knows" the relevant information. Another consideration is the possibilities for future modifications - for example, the **fs** command might be enhanced so that multiple strings could be entered, and the search would output only the records that contained all the strings. Which module is the most natural site for such changes? It is probably the one that should be responsible for the current version of the command.

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 misprogrammed 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 get to work out a good design for implementing this capability. As an aside 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 duplicating data with no compensating advantage), and retain a good class design for the classes. Remember that the key to a good design is a clear concept of which responsibilities each class or module will take on. Only a very few additional member functions or variables should be involved; 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.

Project Grading

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.

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.

I will examine the final version of your code and evaluate it for quality, along the lines of Project 1. Pay attention to commenting, organization, clarity, and efficiency consistent with good organization and clarity. Study the Syllabus information on how the autograder and code quality scores will be determined and weighted. Read the web page “How to do Well on the Projects”. Study the C++ Coding Standards handout. 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 `0` instead of `NULL`, `bool` instead of `int` where appropriate). 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 Standard Library containers, algorithms, adapters, and binders, and how well you used them. Check the project specifications carefully.
- How good your design solutions in Step 2 are, both in how well they follow the course concepts, and whether it results in a simple and clear code structure with good division of responsibilities and collaborations.

How to Build this Project Easily

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 your lecture notes and the posted code examples to see how to use the containers, algorithms, adapters, and binders. Remember: don't going rummaging through possible trash on the web; use the provided resources first, and then ask for help.

First 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 to use the map of function pointers instead of the switch. Next, choose your containers; choose wisely! You can change them one at a time and verify your program still behaves correctly.

Gentle introduction to algorithms, adapters, and binders. To get gently introduced to using the algorithms, I suggest keeping or modifying your explicit loops from Project 2, and then change them to use the Standard Library algorithms, adapters, and binders one at a time. 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. 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, almost all of the code using STL algorithms will stay the same! If this isn't true, your code is poorly written - fix it!

Make a copy if it helps. 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. Of course, you must use an algorithm or the container's constructor to make that copy - no loops allowed for this purpose! Note that the temporary container type counts towards the different types you must use.

Adapters & binders or custom function objects? There are only two cases concerning whether and how you use the adapters and binders:

1. Using the Standard Library (or TR1) adapters and binders will be easy and simple in a particular place in the code. Just check the course materials and code examples to see how it can be done.
2. You have to write a custom function object class that does exactly what you want in an algorithm with no help from the adapters and binders. (Because it is a custom class, you are not required to make it adaptable, which is generally a poor use of time anyway.)

You have to learn enough to tell whether you are in case 1 or case 2, and write your code accordingly. Don't be too hasty in deciding in favor of case 2. Most of the cases in this project are case 1.

A couple of examples: There is an issue with reference-to-reference in the Standard. If you try to give a stream (which must be passed by reference) to one of the Standard Library binders, the template instantiation can result in a reference-to-reference error, meaning that using the binder won't work in this case. Some of these cases can be handled with a stream iterator; others must be done with a custom function object class containing a reference-type member variable that you initialize with the stream. As another example, trying to use a member function with `for_each` might look like it won't work unless you remember that a member function has to be called differently because of the hidden "this" parameter, which is the *first* parameter of the function; you can often arrange that with a Standard Library binder.

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 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 or pointers to Collection or Record objects, resulting in simpler and more understandable code. Remember that if something has gone wrong, an error 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 under the Standard, its objects are supposed to be unmodifiable to make sure that contents stay in order. If you want to use `set<>` for changeable objects, it can be done simply and safely if you change the object while it is not in the container - the copy-remove-change-insert strategy. 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 with 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 `const_cast` is a sign of design failure - don't go there - and don't abuse `mutable` (see the Coding Standards for the only acceptable use).

Using a vector? Use binary search! The `binary_search` and `lower_bound` algorithms provide a fast log-time search when used with a sorted `vector<>`, but these functions can be confusing. 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! If you need to know both whether the item is present, and where to find it or insert it, you can either do a `binary_search` followed by a call to `lower_bound` if needed, or you can do it all with a single call to `lower_bound` as follows:

- 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).
- So 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.
- The problem is that a non-`.end()` value doesn't tell you whether or not it is there - it 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 Record* that has the sought-for title). If it matches, then "here it is." If not, it means "here is 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.