Project 2
Domain Classes and Support Classes:
The Micro Media Manager in C++
Due: Feb. 10, 2012, 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

This project is to be programmed in pure ANSI C++ only. The purpose of this project is to provide review or first experience with the following:

- Basic C++ console and file I/O.
- Using classes for abstraction and encapsulation of user-defined types that represent concrete types.
- Overloading operators for user-defined types.
- Developing classes that behave like built-in types and properly manage dynamically allocated memory.
- Writing and using templates, including class templates with default template parameters and templated member functions.
- Using some simple static member variables.
- Getting some practice with const-correctness.
- Using constructors to initialize objects from a file stream.
- Using exceptions to simplify error handling and delegate error-detection responsibilities.
- Programming a string class that is a simpler version of std::string that supports input, output, copy, assignment, concatenation, and comparison.
- Programming a linked-list class template similar to the Standard Library list<> template that fully encapsulates the list implementation by using iterators and nested classes. However, unlike std::list<>, it automatically puts the contents in order, similar to std::map<> or std::set<>, but only a linear search of its contents is possible. It fully implements copy and assignment as well.

Problem Domain

The functionality and behavior of this program is almost identical to Project 1; the most important difference is that input strings are no longer restricted in length. The other differences are some changes in the output, especially for the memory allocation information.

Overview of this Document

There are three sections: The *Program Specifications* in this project are very simple, because the program behavior is basically identical to that of Project 1. The second section presents that *Class Design* in the project; study this section for an introduction to how you translate from a problem domain to the design of a set of classes and their interfaces. The third 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. At the end is a section of advice on *How to Build this Project Easily*.

Program Specifications

Unless otherwise specified, the behavior of Project 2 is the same as Project 1 (as amended by the posted Corrections and Clarifications). The difference is that the programming techniques used are much safer, neater, and result in a couple of modules that could be easily re-used in other projects (except they are redundant with the Standard Library). There are four differences in the behavior:

1. Some details of the **pa** command output are different.

2. The record ID counter should be reset when all of the records in the Library are deleted with the **cL** and **cA** commands. Project 1 did not specify this behavior, but it is specified for this project.

3. Because you will be creating and using a String class with automatically-expanding capabilities, there are no restrictions on the length of medium names, titles, file names, or collection names.

4. If the **rA** command detects an invalid input file, instead of leaving the user with an empty Library and Catalog, it implements a "roll back" to the previous program state. Thus the **rA** command has the following behavior, which is modified from Project 1:

- **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 saves the current contents of the Library and Catalog containers in local backup variables and uses the `Record::save_ID_counter` function to save the current ID counter value. It then clears the Library and Catalog containers, resets the record ID counter to zero, 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 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.
- Then if the restore was successful, the original records are deleted, and the backup copies of the Library and Catalog are cleared. The final result will be to restore the program state to be identical to the time the data file was saved except that the record ID counter could have a different value.
- However, if an error is detected during reading the file, the program rolls back its data to what it had at the start of the **rA**. That is, any new Records created while reading the file are deleted, the ID counter is restored to the value it had when the **rA** was started (using the `Record::restore_ID_counter` function), and the contents of the Library and Catalog containers replaced with the values saved in the local backup variables. Then the error is reported.
- Thus the possible results of issuing a **rA** command are: (1) an error message but no other 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, resulting in an error message and the program data in the previous state.
- **Hint:** saving and restoring to/from the backup variables containing copies of Library and Catalog containers should be trivial, given the specification for Ordered_list. If this code has more than a few lines, you have missed an important point.

Class Design - Responsibilities and Collaborations

In this project, you will be implementing classes of two kinds: One kind represents objects in the problem domain: Records and Collections. The second kind define objects that only support the domain classes, these are Strings and Ordered_lists. The contrast is between the computer-science sorts of classes that are handy for coding the program, versus the classes that correspond to things in the world of media management.

A key idea in object-oriented programming is that the structure of the code corresponds to the structure of the domain, and this structure is determined by the responsibilities of each class, and how each class collaborates or relates to the other classes. This is primarily an issue with the domain classes; the support classes are relatively simple in this regard. You might find it helpful to examine the supplied starter files for each class as you read, to see how the responsibilities and collaborations play out in the specified public interface.

Record class. A Record object holds the data about a record using the new String objects. The Record class as a whole is the "keeper" of the ID number scheme - it maintains the `ID_counter` - the source of the next record number - and knows how to assign an ID number to a new record in different situations. In addition, to support reverting to the previous state when an **rA** command fails (see above), the Record class also knows how to save the ID number to a backup variable and restore it from that variable. The Record class t handles this completely on its own, but the main module has to control when to reset, save, or restore the ID_counter, because only the main module knows when these actions need to be done.

The Record class is responsible for how to output a record's data - it does this with the output operator definition, which is a friend function, and therefore part of the public interface for the Record class (both of the domain classes have this responsibility).

Record also knows how to save and restore its data to/from a file stream. The restoration is done by a constructor that uses a file stream as the source of initial values - this is a much more elaborate constructor than you might be used to, so it is excellent practice with a broader concept of what it means to initialize an object.

In addition, the Record class defines an ordering relationship, following the common convention of defining operator< to represent what it means for one record to "come before" another in order; since we can have only one operator< definition, it is chosen to put the records in alphabetical order of title.

The public interface of Record reveals a bit more about Record's relationships with other classes. There is no way to change a record's ID, title, or medium once the Record object has been created. The Record class is responsible for ensuring that these aspects of a Record are immutable - protected from interference. This is important if the data are used to put Records in order - changing a record title would disorder the container!

Furthermore, the copy constructor and assignment operator are declared private, meaning there is no way for client code to "clone" a Record - this is a way to represent the idea that Record objects are supposed to be unique - it doesn't make sense to have more than one copy of a Record, so a Record is always referred to with a pointer to what is supposed to be the unique Record object. This way, the Record object's address is a complete way to identify that unique Record object. In fact, in C++, an object’s identity is given by its address in memory.

Finally, we've delegated to Record the decision of what counts as a valid rating - the modify_rating function will check and throw an exception if the rating is out of range. The main module does not need to handle this responsibility.

Note that while Collection "knows" about Records because it works with them, and so has to use Record's public interface, Record does not "know" about Collections - it has no information on who is going to use it or how it is going to be used. This kind of one-way relationship is very common and makes the design simpler.

Collection class. The Collection class has the responsibility of maintaining its list of member Records - the public interface provides methods for adding and removing members and determining whether a member is present. In other words, the main module handles members by delegating the actual work to the Collection object. Note that there is no way for client code to access the list of members directly, reflecting a design decision that it is strictly up to the Collection object to keep track of its members. Consistent with this responsibility, the Collection functions for adding and removing members do the checking for whether the supplied Record pointer points to a record already in the member list or not. Main does not have to worry about this; it has been delegated to Collection. Using exceptions for error reporting makes such delegation attractively easy to do.

The Collection class also provides a definition of what it means for one collection to come before another, namely in alphabetical order of name. Collections also know how to save and restore themselves. Unlike Records, Collections can be copied and assigned - Collections are abstractions in the domain, unlike a Record which corresponds to an actual physical object - the DVD or VHS tape. So the design decision was that it might be useful to be able to copy and assign Collections - they don't have to be unique. This capability is used to copy collections into and out of containers. This means that Collection, along with String and Ordered_list, should behave like first-class data types: they can be assigned, copied and used as call-by-value arguments and return values just like built-in data types.

Collection must know about the interface for Record, but Record, as noted above, knows nothing about Collection. Likewise, the main module knows about the interface for Collection, but Collection knows nothing about its user, the main module.

Main module. While the main module is not a class, it helps to think about its responsibilities and collaborations. The main module is responsible for interacting with the user - it collects and interprets commands and their parameters, and then tells Ordered_lists, Records, and Collections what to do. It has a Library of Records, and it adds and removes records from the Library. The Library is not a class because it is simply a pair of containers created and maintained by the main module - we give it a fancy name just for convenience in discussion. Likewise, main has a container of Collections (the Catalog) to which it adds and removes Collections. Depending on user commands, it tells a Collection what to do. Thus the main module is the "boss" of the program, but as much as possible, it delegates all of the detail work to the Records and Collections. Another example is how the many details of restoring the data from a file are delegated to the specialized constructors of Collection and Record. All main does is manage the overall process - for example, main knows that the Record data comes first, followed by the Collection data, and what has to happen if there is an error in reading the data. But main does not have to micro-manage the how Records and Collections are described in the file and how that data is read and used.

The key concept: This decentralized delegation of labor to class objects is the hallmark of object-oriented programming; main just supervises, letting a gang of objects cooperate to do as much of the work as possible.

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.

Important! In all the projects in this course, if the public interface for a class is specified and you are told that "*you may not modify the public interface*" for the class, this means:

- All specified public member functions and associated non-member functions must be implemented and behave as specified; you may not change their arguments or behavior.
- No specified functions can be removed.
- No public member functions can be added.
- No friend functions can be added.
- No public member variables can be added.
- No other functions, classes, or declarations can be added to the class header file unless they are *private* members of a class.

Also, if the private members of a class are specified, you must have and use those private members with the same names and types, but otherwise, the choice and naming of private member variables and functions are up to you, as long as your decisions are consistent with good programming practice as described in this course.

Program Modules

The project consists of the following modules, described below in terms of the involved header and source files. This project has you start with "skeleton" header files, and you will complete them to get the actual header files. These skeleton header files have "skeleton" in their names. You will rename the files to eliminate the "skeleton" string to get the specified file names.

String.h, .cpp. String is a grossly simplified version of std::string. Be careful about the difference in the names - this class starts with capital "S"; the standard one is lower-case "s". By implementing it you will review or learn the key concepts in classes that manage dynamically allocated memory and the basics of user-defined types. The main difference from the std::string is that the public interface is considerably simpler. Nonetheless, you can do a lot with Strings. You can create, destroy, copy, assign, compare, concatenate, access a character by subscript, insert, remove, and access substrings, and finally input and output them. Concatenate means to put together end-to-end, so a String containing "abc" concatenated with one containing "def" would yield one containing "abcdef." In addition, you can modify the contents of a String in a variety of ways: you can change single characters, remove multiple characters, and insert additional characters. These operations are adequate for this project; please ask for help if you think you need more.

The course web site and server contains a skeleton header file, skeleton_String.h, that defines the public interface of String, and which also includes the required members and notes of other declarations and functions you must supply to get the final version of the header file and the corresponding .cpp file. It also specifies the behavior of String, in particular the policy on how much memory is allocated and how it is expanded. Please read this information carefully as you work on this class. Where not specified or restricted, the details are up to you.

How it works. A String object keeps a C-string in a piece of memory that is usually allocated to be just big enough to hold the string. If a String object is destroyed, the allocated memory is "automagically" deallocated by String's destructor function. Moreover, the internal C-string memory is automatically expanded as needed when inputting into a String or concatenating more characters into the String. The similarity to Project 1's array container is deliberate. For convenience and speed, the length of the internal C-string is maintained in a member variable, and made available through a reader function.

Constructors and overloaded assignment operators make it easy to set the internal string to different values. Other overloaded operators allow you to easily output a String or compare two Strings to each other. Comparisons are implemented more easily by taking advantage of how the compiler will use the String constructor to automatically create a temporary String object from a C-string. The overloaded input operator will read in a string using basically the same rules as Standard C++ `operator>>` inputting to a `char *`, which in turn are the same as `scanf`'s `"%s"`. The web page has a demo program with its output. Following is a tiny example of how String can be used:

```
String str1, str2 ("Walrus"), str3;
str1 = "Aardvark";
if (str1 < "Xerox")
    cout << str1 << " comes before " << "Xerox" << " in alphabetical order"
        << endl;
if (str1 < str2)
    cout << str1 << " comes before " << str2 << " in alphabetical order" << endl;

cout << "Enter your name:";
cin >> str1;
String msg("You entered:");
msg = msg + str1;
msg += ", didn't you?";
cout << msg << endl;
// foo is declared as: String foo(String);
// and so uses String in call and return by value.
str2 = foo(str1);
cout << str2 << endl;
```

Like the Standard Library string class, String has the useful feature that the internal memory storing the characters automatically expands as needed; this is especially handy when reading input into the string; you don't have to worry about a crazed user overflowing a fixed-size array! High-quality Standard Library implementations of <string> use a variety of tricks to improve performance, but here you implement a simple form that illustrates the basic ideas. The String demo program on the web pages has examples showing this automatic expansion, and read the skeleton header comments carefully for the detailed specifications.

The input operator for String, like std::string, follows the basic rules for the operation of reading input into a character array. Characters in the input stream are examined one at at time. Any initial whitespace characters are skipped over, then the next characters are concatenated into the result until a whitespace character is encountered. Thus the supplied String variable will contain the next whitespace-delimited sequence of characters in the input stream. The terminating whitespace character is left in the input stream for the next input operation to handle.

Implementing the input operator. The following are some important details and suggestions for your implementation:

- It must use the `clear` function on the supplied String before starting the reading process.
- It should read each character with the `istream::get` function.
- Use the <cctype> function `isspace` to perform the check for whitespace - this will test for all whitespace characters.
- The whitespace character that marks the end of the character sequence must be left in the input stream for this function to behave like the other input operators. But reading a character normally removes it from the stream. How can you read a character and still leave it in the stream? There is an input stream member function, `peek()`, that "peeks" ahead in the stream and returns a copy of the character it finds there without removing the character from the stream. With `peek()`, you can check the next character to see if it is a whitespace; if it isn't, go ahead and read it using the stream get function. If it is whitespace, you stop the input operator processing, and leave the whitespace character in the stream for the next input operation to find. This

can be implemented with a simple loop. Instead of peeking ahead, you can go ahead and `get` the character and then use `putback` or `unget` to put it back into the stream if it is whitespace.

- Check the status of the input stream after reading each character. If it is in a failed state, the function simply terminates and leaves the supplied String variable with whatever contents it currently has and the stream in whatever failed state it is in.
- As guidance to help you keep this simple, the instructor's solution implements the input operator with only about 10 very short lines of code in the body. Ask for help if your code is more complex.

Implementing `getline`. The `getline` function is similar to the C Library function `fgets` and the C++ Library function `std::getline`. However, it differs from both of them in the fate of the final `\n` that terminates the line. String's `getline` function stops reading and storing characters when it encounters a newline, but *it leaves the newline character in the stream.* This will make the error recovery much more conveniently consistent: after outputting *any* of the error messages, the program will read and discard the input up to and including the next newline character. If a bogus title was the last thing read, then the next newline will be the newline that terminates a title. You can detect and not store the newline by following the above suggestions for finding the terminating whitespace for the String input operator.

Use copy-swap. The String class destructor must deallocate the memory space. The copy constructor must initialize the object by giving it a separate copy of the C-string data in the other String. The assignment operator must deallocate space in the left-hand side object and then give it a separate copy of the data in the right-hand side object. It must also be safe against aliasing (self-assignment). You are to use the copy-swap logic to achieve code re-use and exception safety. Use the obvious generalization of "construct and swap" for assignment from a C-string.

Error handling. The String class includes a class to be used to throw exceptions if the String member functions detect that something is wrong - such as an out-of-range index in the subscript operator (`std::string` does not check `operator[]`). Your top-level function in the main module should include a catch for this class along with the other catches. If your program is written correctly, such exceptions should never be thrown during program execution, but the exception really helps catch bugs!

Instrumentation. To help track of what is happening with your Strings, and to provide some practice using static members, the String class includes some static members that record how many String objects currently exist, and how much total memory has been allocated for all Strings. In addition, there is a messages_wanted flag that when set to true, causes the constructors, destructor, and assignment operators to document their activity. This allows you to see what is happening with these critical member functions.

Notice: Part of our testing and grading will be to attempt to use your String class as a component in a program that exercises it in various ways that should work if you have correctly implemented it according to the specifications. Your own testing should do the same. Test every member function and capability in a stand-alone test harness to be sure that you have built a correct String class. If you discover a bug later and modify the class, be sure to back up and rerun your tests to make sure you haven't broken something - this is called *regression testing*. So that we can test it by itself, your String implementation should depend only on your Utility.h to get access to the `swapem` function template (see below).

Ordered_list.h. This header file contains a class template for a list container called Ordered_list<>, which is based on the Project 1 Ordered list module, but it is more restricted: As before, when you create an Ordered_list, you supply an ordering function, which defaults to the less-than relation defined by operator< between the list items. The only way to add a data item to an Ordered_list is through the insert member function which keeps the list items in order. The insertion function searches the list for the first existing item that is not less than the new item, and puts the new item in the list in front of it. Thus the list is in order from smallest to largest as determined by the supplied less-than relation. If you have pointers in the container, the operator< default is usually inappropriate, because if you compare two pointers with '<', you are comparing the two addresses they contain, not the objects at those addresses! Instead you can supply a function that dereferences the pointers and compare the objects in any way you choose; it is common to simply apply '<' to the dereferenced pointers. This permits you to construct lists ordered in any desired way simply by supplying an appropriate ordering function.

The interface for finding objects in the list is very restricted. The find function returns an Iterator to the object in the list that compares equal to a supplied "probe" object using the ordering function for the container. Like the STL, and unlike Project 1's container, there is no way to supply a "custom" ordering function for the find operation.

Standard ordering trick. We use a trick from the Standard Library to test for equality using only the less-than operator or function. Namely, if both x < y and y < x is false, then we can assume that x and y must be equal. Note that this sense of equality applies only in the sense of what the less-than operator compares. For example, if x and y are Records, then operator< compares just the titles - if the titles are the same, then we will treat the two objects as equivalent even if the other data are different.

To find an object in the list, you must first construct a probe object that will compare as equal to the desired object according to this use of the less-than operator or function. For example, `Record probe("Tobruk")` is an object that will compare equal to the object titled "Tobruk" in the Library - the probe does not need a medium or ID because they are irrelevant to the comparison. Record is specified to include handy constructors for creating probes for both title and ID search.

The Ordered_list<> class template contains two nested classes, `Node` for the list nodes, and `Iterator`, for list iterators. (Note the capital `I` in `Iterator`! The Standard library `iterator` is lower-case!) The `Iterator` class simply encapsulates a `Node *` pointer, and overloads the *, ->, ++, !=, and == operators along the same lines as std iterators. If properly implemented, you can use this template to get ordered lists of any type of object and clients can work with the list without having to mess with any pointers to nodes (and should not be able to, actually!). Because Ordered_list<> has a copy constructor and assignment operator properly defined, you can freely use Ordered_list as a function call-by-value parameter, and as a value returned from a function.

Note that each node contains a member variable whose type is the type of object in the list, unlike Project 1's list which held a void * pointer to an external data object. Thus actual objects of any type can be stored in the list node instead of just pointers. Since the Node destructor will destruct all of the Node member variables, if you have an Ordered_list of objects that have fully functional destructors, destroying the list object itself will “automagically” destroy all of the data objects in the list as well. However, Ordered_lists of pointers require that the client code itself manage the memory for the pointed-to objects, analogous to Project 1.

Your program must use this Ordered_list container for all of the linked-lists in the program. You use Ordered_list<> similarly to the list module from Project 1; you can use the apply functions (which are more reliable because they are type-safe templates). But in addition, your client code can operate on individual items in the list with Iterators, which are a type-safe replacement for the void * item pointers in Project 1. For example, to output the list, you step an Iterator through the list in Standard Library style, from begin() to end(), and dereference the Iterators to produce the output. Occasionally you will need to traverse a list for other purposes, but in almost all cases, when you need to search the list, you should use the find function, which works like the Standard Library approach for ordered containers like `std::set<>`. You should do so to get used to doing things in the Standard Library way.

Just like the Standard Library iterators, our Iterators are not foolproof. They simply encapsulate a pointer. If you have an Iterator that points to a Node that does not exist, we say that the Iterator is *invalid* - this means that trying to advance it with ++ or dereference it with * or -> will produce *undefined* results. Trying to make iterators "smarter" about this would cause huge performance problems, so it hasn't been done either here or in the Standard Library. So it is the user's responsibility avoid using iterators in an undefined fashion. But your code can and should use `assert` to detect simple programming errors, such as trying to advance an iterator that points to 0 (the end() value).

The course web site and server contains a skeleton header file, skeleton_Ordered_list.h that defines the public interface of Ordered_list<>, and which also includes the required members and notes about other declarations and functions you must supply to get the final version of the header file. Otherwise, the details are up to you - you can add additional private members as you wish; the ones shown are for a one-way linked list, but you can add additional ones to make it a two-way list if you choose. But you may not modify the public interface.

Notice: Part of the testing and grading will be to attempt to use your Ordered_list template in a program that exercises it in various ways that should work if you have correctly implemented it according to the specifications. Your own testing should do the same. Test every member function and capability in a stand-alone test harness to be sure that you have built a correct Ordered_list class template. If you discover a bug later and modify the template, be sure to do regression testing.

Record.h, .cpp, Collection.h, .cpp. Collection and Record are two more classes, each in its own header-source pair. Skeleton header files are also provided for them. You must supply the .cpp files and fill out the class declarations in the headers as needed. Again you may add any private members that you wish, but you may not change the public interfaces. If a private member is specified in the skeleton header file (e.g. Collection has an

Ordered_list of Records), you must use this private member and work out the code in terms of that member - there are important lessons to be learned by doing so. The design of these classes and their responsibilities are described above.

These two classes include a constructor function that takes an input stream argument, allowing the object to be initialized from a file in a fully encapsulated way compatible with how we usually create and initialize objects. These constructor functions should throw an Error exception if they encounter invalid data in the file (using the same rules as in Project 1). This provides valuable practice in how to handle a failing construction, and illustrates the amazing way in which exception processing can automatically clean up messes when an error occurs.

As in Project 1, the list of members in a Collection must contain pointers to Record objects that are also pointed to by the Library lists. An individual record's data must be represented only once, in a Record object created when that record is added to the Library. All member lists simply point to the corresponding Record objects. Thus removing a member from a collection does not result in removing that record from the Library nor destroying that record's data item.

In Project 1, the Record had a file-local (internally-linked) global variable that served as the ID counter to produce a unique ID number every time we created a Record. In this Project, we use a better technique: Record has a *private static member variable*, called `ID_counter`, which is initially zero. Record has four different constructors. When a record is created to add to the Library (title and medium are both supplied), the `ID_counter` is incremented and the new value used for the new record ID. When a record is created for use as a probe in a title search, only the title is supplied and `ID_counter` is not incremented or used. A probe can be created for an ID search, in which case only the supplied ID number is used to initialize the record's ID, and again `ID_counter` is not incremented or used. There are also two additional functions (see the skeleton header) that saves the `ID_counter` in a second static member variable, and restores from that second variable. This is used to recover from a failed **rA** command (see above).

Finally, there is a constructor used to create a Record from a file stream. This function reads the record ID from the file. However, if the ID from the file is greater than the current value of `ID_counter`, it sets `ID_counter` to that value - this way the next record created by the user with an **ar** command will have the next ID number.

Collections and Records can be compared using the '`<`' operator, which simply compares the names or the titles respectively.

Utility.h, .cpp. You must have a pair of files, Utility.h and Utility.cpp, to contain any utility functions and classes shared between the modules. The supplied skeleton header file Utility.h contains a class declaration for an "Error" class that must be used to create exception objects to throw when an error occurs. This class simply wraps a const char * pointer which the constructor initializes to a supplied text string. Thus, to signal an error, a function simply does something like

```
if(whatever condition shows the Record is not there)
    throw Error("Record not found!");
```

The error message strings are specified in the posted starter materials. Except for the Unrecognized command message, these messages must appear in throw Error expressions, not in individual output statements as in Project 1. See the description of error handling below.

Any module that wants to throw Error exceptions to report input errors must #include this header.

To provide a bit of practice with function templates, and simplify some of your other code, this module will also define a function template named "swapem" that your other code should use. It will interchange the value of two variables of any type. For example:

```
int i = 5, j = 3;
swapem(i, j);
// i is now 3, j is 5
```

In all component tests, your Utility.h, .cpp files will be included with your other code.

As in Project 1, the Utility module is reserved for functions or classes that are used by more than module (like the Error class) - do not put anything in here that is used by only one module.

p2_globals.h, .cpp. These files define two global variables that are used to monitor the memory allocations for the Ordered_list template. Your Ordered_list code should increment and decrement the variables, and your p2_main should output them in the **pa** command. You may not include any other declarations or definitions in this module.

p2_main.cpp. The main function and its subfunctions must be in a file named p2_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.

Analogous to Project 1, your main function must declare three Ordered_list variables; two are the Library and are lists of Record *pointers*, with one ordered by title and the other by ID. The third, the Catalog, must be a list of Collection *objects*, using the default ordering supplied by operator< , declared as:

```
Ordered_list<Collection>
```

Important: The Catalog must be a container of ***objects***, not pointers.

Note: The containers used for the backup in the **rA** command must not be declared in the main function, but rather in your **rA** function - they should not be needed anywhere else.

All of the list manipulations must be done using the public interface for Ordered_list<> and its Iterators. Ask questions or seek advice if you think this interface is inadequate for the task.

All input text strings should be read from cin or a file stream directly into a String variable using the overloaded input operator - if correctly implemented, this cannot overflow, and so no limited-length character buffers are required. Single character input, as for the commands, should be done into single-character char variables.

Except for the Ordered_list instrumentation variables described above for p2_globals.h, .cpp, you may not have any global variables.

Top Level and Error Handling

The top level of your program in p2_main should consist of a loop that prompts for and reads a command into two simple char variables, and then switches on the two characters to call a function appropriate for the command. There should be a try-block wrapped around the switches followed by a `catch(Error& x)` which outputs the message in the Error object, skips the rest of the input line, and then allows the loop to prompt for a new command.

As described above, your **rA** command code has to restore (or "roll back") the data to its original values if the file reading fails. A good way to do this is to catch any Error exceptions thrown by functions that it calls, do the roll-back, and then rethrow the exception so that the top level try-catch can handle the error in the usual way..

The "Unrecognized command" message can be simply output from the default cases in the switches, as in your code for Project 1, but all other error messages must be packed up in Error objects and then thrown, caught, and the messages printed out from a single location - the catch at the end of the command loop. Once adopted, this pattern greatly simplifies your code. Try it; you'll like it!

To reinforce how exceptions can simplify code, and how we can delegate responsibilities to classes more easily with them, we will rely on Collections and Records to be responsible for identifying bogus modifications. This produces the following deviations from the error-handling pattern in Project 1:

1. `Collection::add_member()` throws an exception if the member is already present in the Collection. Therefore, the main module code for the **am** command can get the pointer to the specified member and simply add it to the Collection; it does not have to check for the member being in the Collection already.
2. Likewise, `Collection::remove_member()` throws an exception if the member is not present in the Collection, meaning that the main module **dm** command does not need to check this, but simply supplies the record pointer and lets Collection check it out. The function `Collection::is_member_present` should be used only to implement the **dr** command.
3. `Record::set_rating()` throws an exception if the supplied rating is out of range. Therefore, the main module code for the **mr** command should collect the new rating and then simply attempt to modify the Record - it should not check the new rating for out-of-range.

Your program is required to detect the same set of errors as in Project 1 - see the supplied list of text strings for error messages. Notice that it is specified which messages must be output from which components - follow this specification carefully to avoid problems in the component tests. See the sample output for some examples. In addition, to make your program well-behaved, your top-level try/catch setup should have catches for the following; For each of them, print an error message of your choice to either `cout` or `cerr` and terminate the program as if a quit command had been entered.

1. A catch for `bad_alloc` in case of a memory allocation failure. You should `#include <new>` at the appropriate point to access the declaration of the Standard Library `bad_alloc` exception class.
2. You should have a catch for `String_exception` to handle a programming error; in a correct program, there should be no way the user could trigger this exception from the String class.
3. You may want to include a "catch all exceptions" catch in case one of the Standard Library functions throws an exception that you weren't expecting - seeing something like "Wow! Unknown exception caught" is much more helpful than your program suddenly and silently quitting. Again this should be programming errors only - in a correct program, there should be no way the user could cause such an exception.

Other Programming Requirements and Restrictions

1. The program must be coded only in Standard C++, and follow C++ idioms such as using the `bool` type instead of an int for true/false values, and avoiding `#define` for constants. See the C++ Coding Standards document for specific guidelines.
2. Only C++ style I/O is allowed - you may not use any C I/O functions in this project. This means you can and should be using <iostream> and <fstream>. See the web handouts on C++ Stream and File I/O for information on dealing with stream read failures.
3. You may use only `new` and `delete` - `malloc` and `free` are not allowed.
4. Before your program terminates, all dynamically allocated memory must be deallocated - in this project, correctly working destructor functions will do this automatically in many, but not all, cases.
5. There must be no memory leaks apparent in your code.
6. You must not use any declared or allocated built-in arrays for anything anywhere in the program. The single exception is that your String class must use `new` to dynamically allocate a `char` array to store the internal C-string.
7. You may not use the Standard Library `<string>` class or any of the Standard Library (or "STL") containers, algorithms, or iterators in this project. The idea is to learn how these things are done by writing some simplified versions yourself. Subsequent projects will allow (actually require) full and appropriate use of the Standard Library.
8. Your String.cpp file can use any functions in the C++ Standard Library for dealing with characters and C-strings, including any of those in <cctype>, or <cstring>. See a Standard Library reference, or the online brief reference pages in the Handout section of the course web site. Any non-standard functions must be written by yourself. *Hint*: The String class can be coded easily using only basic functions from the C part of the C++ Standard Library like isspace, strcmp, strcpy, strlen, and strncpy; seek help if you think you need some of the more exotic library functions. Note that all other modules in the project will work only with individual char variables (e.g. for the commands) or String objects. You can use <cctype> functions such as isspace wherever they are appropriate.
9. String::operator+= must not create any temporary String objects or unnecessarily allocate memory. See above for more discussion. The skeleton String.h file comments specify which operators and functions create temporary String objects.
10. You can and should #include <cassert> and use the assert macro where appropriate to help detect programming errors.
11. You must follow the recommendations presented in this course for using the std namespace and minimizing header file dependencies.
12. Except where specifically required, you may not use any global variables. You may certainly use file-global constants in a module's .cpp file for things like representing output string text, and when appropriate in Utility.h. Note that in C++, `const` variables automatically get *internal linkage*, so you don't have to declare them static or in the unnamed namespace in a header file.
13. 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 sample to be sure your program produces output that can match our version of the program.

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.

Your project will be computer-graded for correct behavior, and component tests will test your classes separately and mixed with mine, (so be sure all functions work correctly, especially any you did not need to use in this project). Your Utility module will be used with your code in all component tests.

I do not plan to do a full code quality evaluation on this project, but you should still try to write high-quality code anyway - once you get used to it, it will help you work faster and more accurately than simply slapping stuff together. It make writing code a lot more fun, also. However, I plan to do a spot-check as described in the Syllabus and deduct points from your autograder score if you code does not appear to represent an effort to follow the specified coding requirements and code quality recommendations. This will be things that I can screen for quickly. Examples: Did you use copy-swap where specified? Did you put "using" statements in header files? Did you use "this" unnecessarily?

How to Build this Project Easily

As in Project 1, there are many places in the main module where code duplication (and attendant possible bugs) can be avoided by simple "helper" functions. Entertain changing them some from their Project 1 version to be more suitable to how this project's Ordered_list works.

Be sure to translate your C code into C++ idiomatic code along the way - e.g. use the `bool` type instead of zero/non-zero ints, write `while(true)` instead of `while(1)`, and declare variables at the point of useful initialization instead of all at the start of a block.

You can base the Ordered_list and String class code on any previous code authored by you (review the academic integrity rules). However, the code must conform to the specifications, so expect to do some surgery.

Both Ordered_list and String can be built and tested separately. Do so; absolutely, positively, do not attempt to do anything with the rest of the program until you have completely implemented and thoroughly(!) tested these two components. Few things are as frustrating as trying to make complicated client code work correctly when its basic components are buggy!

Try String first. Writing the String class first of all will help you get comfortable with the operator overloading and the "big three" of destructor, copy constructor, and assignment operator. Put off the input operator at first, but do write the output operator right away to make it easier to test the String as you add functions.

Important hint: Many of the String functions and operators can be easily implemented in terms of a few of the other ones. So: write and test the constructor, destructor, copy constructor, and assignment operator, and then the += concatenation operator. Save operator+ and operator>> for last. As you work, take care to consider how you can use the functions you have already implemented when you write each function. Look for opportunities to use private helper functions to simplify coding and debugging - these make a huge difference!

The += operator is your friend. In std::string, operator+= is expected to provide very fast appending. For example,

s1 += s2;

will run fast because if the lhs object has enough space, then s2's characters can be simply copied over to the end of the existing s1 characters. If not, only a single action of memory allocation/copy/deallocation will be required to expand the lhs object. In contrast:

s1 = s1 + s2;

will run slowly because the rhs involves creating a temporary object copied from s1 and then concatenating s2 into it, and then copying the contents of the temporary object into the lhs, involving another bout of memory allocation/copy/deallocation. Thus it is a C++ idiom to favor += over (= and +) if it does what you want.

You have to give String::operator+= the same speed advantage, so you must implement += without creating any temporary String objects and by working directly with the pointers and memory allocations. In turn, this function and its helpers can serve as a building block for several other functions - which is why you should build it first. With careful coding, your operator+= will produce the expected result even if the rhs and lhs are the same object, as in:

```
s1 += s1;         // "doubles" the contents of s1
```

The key is to not deallocate the old lhs space, and not update the lhs pointer member variables, until you have copied the data from the rhs; done properly, there is no problem if lhs and rhs happen to be the same object.

Testing String. Write a simple main module test harness and beat the daylights out of this class. Test the copy constructor and assignment operator by calling a function with a call-by-value String parameter, and returning a String that gets assigned to a String variable. Use the static members to track whether your constructors and destructors are being called properly - you should see the amount of memory increasing and decreasing like it should. Put the messages_wanted output in the code from the beginning, and turn it on. This will tell you when your constructors, destructors, and assignment operators are being called - it will help you debug, and will be very instructive if you haven't seen these processes in action before.

Note: If you are using gcc/g++, use the option `-fno-elide-constructors` which will turn off g++'s optimization that eliminates some calls of the copy constructor - this will result in the copy constructor getting called "by the book."

Building the Ordered_list template. Templates can be a pain to debug because most compilers and debuggers generate really verbose and confusing messages about templated code. First code Ordered_list as a plain class that e.g. keeps a list of int's and only after it is completely tested and debugged, turn it into a template. Do something like `typedef int T;` so that you can use `T` for the template type parameter throughout the code; delete the typedef when you change the code into a template.

Test the daylights out of your pre-template version of Ordered_list, Then after you've turned it into a template, test the daylights out of it again with both simple and pointer types. You will waste huge amounts of time if you struggle with debugging Ordered_list in the context of the whole project.

Template code goes only in header files. Remember that with current compilers, if you want to use a template class in a source code file, all of the code for a class template must be seen by the compiler. The customary way to do this is to put all of the template code in the header file - there is no .cpp file for a template class or template functions, only the header file. This might be different from what you were taught before, but it is the normal industry practice, and it is what you must do. Suggestion: After finishing the non-template version, simply paste the .cpp code into the header file after the class declaration, and then modify it to turn it into the template. Be sure to remove the old .cpp file from your project.

Sentinel nodes are a bad idea. Dummy nodes for the first and/or last nodes in a list can allow simpler code, but this trick is rarely used in C++. It violates a basic assumption in OOP that the objects in the program correspond to objects in the domain. The dummy nodes will hold objects that aren't really there in the domain. For example, if you use a sentinel node, even if your Catalog list is empty, there is still a Collection object and its member variables in existence! And by the way, what is its name? This also means that any side effects of creating objects will no longer make sense in the domain. In this project, the side effects are minor (the counts of some of the objects will be wrong), but in complex applications, the side effects can be serious. The amount of code simplification from using this approach is not worth the other problems of trying to kludge the side effects and dealing with violations of the design concept.

A common beginner's error with classes like Ordered_list and String is to not take advantage of the fact that member functions have complete access to the "guts" of the object. Instead, newbies try to implement member functions only through the public interface; sometimes the results are extraordinarily clumsy and round-about. When writing member functions, by all means call a public member function if it does exactly what you need. But do not hesitate to directly go to the private member variables - member functions have this privilege, and there is no virtue in not taking advantage of it.

In particular, notice that the Iterators in Ordered_list are intended for the *clients* of the public interface - there is absolutely no need to use them in member function code, and doing so is silly since the results will be more convoluted than going directly to the member variables of the list and its nodes.