Project 2
Domain Classes and Support Classes:
Exceptions, Classes, and Basic Templates
Due: Feb. 10, 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 that page frequently while you are working on this project. 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 Standard 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 with copy and move construction and assignment, and both basic and strong exception guarantees.
- 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, substring, insertion, 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, using a function object to specify the order, similar to std::map<> or std::set<>. But only a linear search of its contents is possible.

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 three differences in the behavior:

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

2. 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, collection names, or file names.

3. 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 different behavior from Project 1, and does as follows:

- **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).
- 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 the current Record ID counter value. It then clears the Library and Catalog containers, and then attempts to read the Library and Catalog data from the named file, and creates new records and collections from the file data. At the end of the restore, the Record ID counter should be set to the largest ID number found in the file + 1 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 discarded. The final result will be to restore the Records, Library, and Catalog to be identical to the time the data file was saved.
- 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 Record ID counter is restored to the value it had when the **rA** was started, and the contents of the Library and Catalog containers get 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 and efficient, given the specification for Ordered_list.

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 represents objects that make it easier to write the program; these are `Strings` and `Ordered_lists`. The contrast is between the classes that correspond to things in the world of media management, versus the computer-science sorts of classes that are handy for coding the program.

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 interfaces.

Record class. A `Record` object holds the data about a record using the new `String` objects. 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 (The `Collection` class also has 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; however we can have only one operator< definition; in this project, 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 - for example, changing a record title would disorder the container!

Furthermore, the copy and move constructors and assignment operators are made unavailable, meaning there is no way for client code to "clone" a `Record` - this is a way to represent the idea that like actual physical DVDs, a `Record` object is supposed to be unique object - one Record for each physical DVD. Thus, 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*. An additional effect of this decision is that Record objects cannot be used for call-by-value parameters or returned by value from a function - we handle them strictly through pointers.

Finally, to help practice with using exceptions, 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 `Collection`s - 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 `Record`s - 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. `Collection`s also know how to save and restore themselves. Unlike `Record`s, `Collection`s can be copied, moved, or assigned using the compiler-supplied move/copy operations. `Collection`s are abstractions in the domain, unlike a `Record` which corresponds to an actual physical object.

So the design decision was that it might be useful to be able to copy, move, or assign `Collection`s - they don't have to be unique. This capability is used to copy or move collections into and out of containers. In fact, making Collection be copyable and movable is trivial - we just allow the compiler to supply the copy and move member functions, along with the destructor function. If `Ordered_list` and `String` are correctly defined, then the compiler-supplied functions will do the right thing.

Thus, `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, with the move operations or call-by-reference providing efficiency when appropriate.

`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_list`s, `Record`s, and `Collection`s what to do. It has a Library of `Record`s, 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 `Collection`s (the Catalog) to which it adds and removes `Collection`s. 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 `Record`s and `Collection`s.

A key example of this division of responsibilities is how the main module delegates the many details of saving and restoring data to the `Collection` and `Record` classes. All main does is manage the overall process - for example, in restoring, 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 how `Record`s and `Collection`s are described in the file and how that data is read and used. Finally, the main module manages the Record ID counter - it knows when a new Record has been successfully created, and only then increments the counter.

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 do it in any way that conforms to the general requirements of the project and the guidance provided in this course and its materials.

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 name, parameters, or behavior.
- No specified functions can be removed.
- No public member functions can be added.
- No friend functions can be added beyond those specified.
- 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 some 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" files, and you will complete them to get the actual files. These skeleton 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 `std::string` is that the public interface is considerably simpler. Nonetheless, you can do a lot with `String`s. You can create, destroy, copy, assign, compare, concatenate, access a character by subscript, insert, remove, and access substrings, and 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 project web page also includes some demonstrations of how String works and is used.

The project web page 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 C-string. When 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 adding more characters into the `String`. The logic for this expansion is similar to Project 1's array container. For convenience and speed, the length (the same as given by `strlen()` of the internal C-string) is kept up-to-date in a member variable, and made available through a reader function so that the length can be obtained in constant time. The number of bytes currently allocated is also maintained in a member variable, and is length+1 or greater.

The default-constructed empty `String`. An empty C-string (i.e. "") consists of only a null byte. Allocating a single byte of memory every time we want to create an empty `String` is ridiculously inefficient. In contrast, modern high-quality implementations of `std::string` are very efficient because they use the *small string optimization* (see the example in Stroustrup): the string object actually contains a smallish array of char and this is used to store small strings. We take the time to allocate memory only if the string gets large. Since many strings are small, this really speeds things up.

In this project, we will use a simpler (and less efficient) idea: the *empty string optimization*. If the `String` is first constructed as empty, the internal pointer points to a static `char` member variable that contains only a null byte (`'\0'`), and both the length and allocation are set to zero. This means that default construction (and destruction) of an empty `String` is very fast, requiring no `new` or `delete`. If the `String` is created as non-empty, it has an internal pointer to the C-string in allocated memory, and the length and allocation are set to the usual values. If the `String` later becomes empty because we have removed characters, meaning that `strlen()` of the C-string is zero, then the length is zero but the allocated memory is still kept (like to Project 1's array container).

But this empty-string optimization does complicate the logic somewhat: If you add characters to a String, you have to check on the allocation to whether and how much memory to allocate; likewise, when the String is destroyed, you should use `delete[]` only if the string has a non-zero allocation.

Capabilities. 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 `String`s to each other. Comparisons are implemented more easily (but less efficiently) 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 `std::string`. 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! The `String` demos on the web pages contain examples showing this automatic expansion. Read the skeleton header comments carefully for the detailed specifications for the allocation and when it is changed.

The input operator for `String`, follows the same input scanning rules as `std::string`, which in turn, is the same as C's `%s`. 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. When complete, 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. If properly coded, the `String` variable will contain a valid C-string even if the stream fails during the reading.
- As guidance to help you keep this simple, the instructor's solution implements the input operator with only about 10 very short and simple lines of code in the body. Ask for help if your code is more complex.

Implementing `getline`. The `getline` function for `String` is similar to the C Library function `fgets` and the C++ Library function `std::getline`. *However*, those two Standard function differs in the fate of the final `'\n'` that terminates the line. `fgets` consumes the newline and stores it in the destination string. `std::getline` consumes the newline but does *not* store it in the destination string. `String`'s `getline` function works the same way as `std::getline`: it reads each character, consuming it from the stream, and if it is not a newline, it stores the character, and reads the next. But if the character is a newline, it is not stored, and the read is terminated. Thus to read a title, you don't have to remove the newline like you did for `fgets`. However, as in Project 1, because the newline has been consumed from the cin stream, you'll have to handle user error recovery differently depending on whether the last thing read was a title. We'll do this with a different exception class (see below).

Copy assignment uses 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 copy assignment operator must be safe against aliasing (self-assignment) and arrange to deallocate space in the left-hand side object and give it a separate copy of the data in the right-hand side object. You must use the copy-swap logic described in lecture (see the posted notes) to achieve code re-use and exception safety. This involves implementing and using a `String::swap` member function. Use the obvious generalization to "construct and swap" for assignment from a C-string.

Implementing move semantics. First get the copy constructor and copy assignment operator working correctly. Then add the move constructor and move assignment to automatically improve performance when they apply. The move constructor takes an `rvalue reference` to the original `String`, and simply "steals" the data by a shallow

copy of the member variables, and then sets the original's member variables to be those corresponding to an empty or default-constructed `String`. The move assignment operator is trivial; just call `String::swap`.

Exception safety. If you follow the above specifications, and take care to first allocate memory, and then modify the object, in that order, the `String` class should provide both the basic and strong exception guarantee.

Error handling. The `String` class module 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 illustrate what is happening with your `String`s, 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 `String`s. In addition, there is a `messages_wanted` flag that when set to true, causes the constructors, destructor, and assignment operators to document their activity with output messages. This allows you to see what is happening with these critical member functions.

These messages should be output at the beginning of each function body, before any actual work in the function body is done. In constructors, this would be after the constructor initializers have taken effect. This is the simplest way to ensure that everybody's versions have a consistent ordering of the output, though the messages are now often about the future, rather than current, state of the object.

Notice: Part of the 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. So that I can test it by itself, your `String` implementation should depend only on your Utility.h module at most.

Your own testing should 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*.

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 implementation, but it is simpler and cleaner. You refer to items in the list with `Iterator` and const_Iterator objects, similar to the STL containers. Because `Ordered_list` has a proper copy constructor and assignment operator, you can pass `Ordered_list`s in function calls by value, and get one returned by value. It also implements move semantics for efficiency. Your `Ordered_list` is very capable and very like a Standard Library container. You can learn a lot by building it. But because the find and insert operations take linear time, this type of container is just not efficient enough to be part of the Standard Library, whose ordered containers work in logarithmic time.

The Starters. 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 specification comments about other declarations and functions you must supply to get the final version of the header file, which includes certain private members. Otherwise, the details are up to you - you can add additional private members as you wish. But you may not modify the public interface, and must follow the specifications corresponding to the private members (such as implementing a two-way linked list).

Quick Start for Function Objects. When you declare an `Ordered_list`, you supply two *type parameters*: the first is the type of object in the list; the second is an ordering function in the form of the *name* of a *function object class*. A function object class is simply a class that contains a member function that overloads the function call operator, named `operator()`. An object created from that class can then be used syntactically *just like a function*. For example, the following function object class can be used to compare two integers to see if the first is more than twice as large as the second

```
struct More_than_2X {
    bool operator() (int i1, int i2)
    {
        return (i1 > 2 * i2); // return true if i1 more than twice as big as i2
    }
};
```

Declaration as a `struct` is often used for simple function object classes whose only member is the function that overloads the function call operator and which needs to be public. Once the class is declared we can use an object from that class *as if it were a function*:

```
More_than_2X my_fo;  // create a function object
int a, b;
cin >> a >> b;
bool result = my_fo(a, b); // use the object like a function
if(result)
    cout << "The first is more than twice as big as the second!" << endl;
```

Before creating a function object, the compiler must have seen the class declaration, but then it can find out everything it needs to set up the call to the contained function from the class declaration. Function objects are often used where function pointers would also work, but in general, they are much simpler to use than function pointers, especially in the context of templates. Instead of messy function pointer declarations, you just name the type! Function objects can do much more than function pointers, as we will see later in the course.

Easy setup thanks to templated function object classes. Ordered_list.h contains two templated function object classes that are especially handy for specifying the most common type of ordering relation, namely one that uses the less-than operator (`operator<`) defined for a type. We will define `operator<` for `Record` and `Collection`. The first template, `Less_than_ref`, applies the < relation between two *objects*, which are passed in by const reference to avoid copying. The second, `Less_than_ptr`, assumes that we have *pointers* to the objects to be compared, and so applies the < operator between the dereferenced pointers.

The ordering function object class in the `Ordered_list` template defaults to the `Less_than_ref` function object class, meaning that by default, we order the objects in the container from smallest to largest, as defined by applying the < operator between them. If we want to store pointers in the container, the default ordering would put the objects in order of their addresses, which we would rarely find useful! By specifying the `Less_than_ptr` ordering class, the object pointers would be stored in an order corresponding to how the objects would be in order. If neither one of these suits, we can supply our own function object class and order the objects any way we choose.

Similar to the Project 1 `Ordered_container` logic, the insertion function searches the list for the first existing item that is not less than the new item, checks to make sure that the new item does not match one that is already in the container, and puts the new item in the list in front of it. This check ensures that there are no duplicate items in the list. Once the `Ordered_list` is instantiated and declared, there is no way to change the ordering function, and the only way to add a data item to the list is through the insert member function, so the items are always in order. Following the approach in Project 1, the list must be a two-way linked list, with a pointer to the last node, and a member variable that keeps an updated count of the number of nodes in the list.

Standard equality trick. We use a trick from the Standard Library to test for equality using only the less-than operator or function object. 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 `Record::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.

Finding stuff. 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. Unlike Project 1's container, there is no way to supply a "custom" ordering function for the find operation.

Probe objects. So 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 object. For example:

```
Record probe("Tobruk");
```

or

```
Record probe{"Tobruk"}; // C++11 style
```

This is an object that will compare equal to the `Record` with title of "Tobruk" in the Library title list - the probe does not need a ID or rating because they are irrelevant to the comparison. Our classes are specified to include handy constructors for creating probes without having to create more than the minimum amount of data. Notice that if you have a container of pointers, you can create a probe object as a local (stack) variable and then use its address in the `find` function; there is no need to waste time and risk a memory leak by creating the probe on the heap!

Nodes and Iterators. The `Ordered_list<>` class template contains three nested classes, `Node` for the list nodes, and `Iterator` and const_Iterator for list iterators. (Note the capital `I` in `Iterator`! The Standard library `iterator` is lower-case!) The iterator classes simply encapsulate a `Node*` pointer, and overload the *, ->, ++, !=, and == operators along the same lines as STL iterators. If properly implemented, you can create 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!).

Note that each `Node` contains a member variable whose template type `T` 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 object*s of any type can be stored in a member variable of the list node instead of just storing pointers.

Since the `Node` destructor will automatically destruct all of the `Node` member variables, if you have an `Ordered_list` of objects that have fully functional destructors, and destroying the list object is properly implemented as destroying the list node objects, then the data objects in the list nodes will be "automagically" destroyed as well. However, `Ordered_list`s of pointers require that the client code itself manage the memory for the pointed-to objects, analogous to Project 1. It is not a container's job to read your mind about when you want pointed-to memory deallocated - maybe you have some other pointer to the data because you want to keep it around!

Supporting const containers. If an `Ordered_list` has been declared `const`, then it needs to be impossible to alter its contents through an iterator. Similar to the STL, `Ordered_list` provides `const_Iterator` for this purpose. A `const_Iterator` is just like an `Iterator` except that it encapsulates a `const Node*`, and when it is dereferenced, yields a reference to the data item that is `const`. This means you can't modify an item in the container through a `const_Iterator`.

However, somewhat counterintuitively, the `erase()` member function requires a const_`Iterator`. The idea is that removing an item from the container is not the same thing as changing the value of the item; all the iterator is doing is designating the location of the item, not providing a way changing the value of the item. The C++ Standard container erase functions work this way, so our container's `erase()` function is defined the same way.

There is a version of `begin()`, `end()`, and `find()` that are declared as `const` member functions, and return `const_Iterator`s. The compiler will automatically call these functions as needed. To see this, check the declaration of these two member functions:

```
Iterator begin();
const_Iterator begin() const;
```

If you declare a container as `const`, then call its `begin()` member function, the compiler will automatically choose the `const` member function, which returns a `const_Iterator`. If the container is not `const`, then the compiler will map the same call to the non-`const` member function that returns an ordinary `Iterator`. Likewise, calling the `find()` member function will return a `const_Iterator` if the container is `const`.

In addition, we follow the STL design in that `Ordered_list` also has two member functions, `cbegin()` and `cend()`, that return `const_Iterator`s when called on either `const` or non-`const` containers; this allows you to write a loop that promises not to modify a modifiable container.

Implementing the `const_Iterator` *functions*. Basically, the functions that return `const_Iterator` are just "clones" of the functions that return an `Iterator`. Except for the difference in the type of the returned value and the encapsulated `Node*`, they do the same thing. The simplest way to implement them is to (shudder) duplicate the code — while it is possible to eliminate the duplication, the techniques involved are beyond the scope of this project[1]. In all but one case, the functions are trivial - very small and short, and so for simplicity for this project, implement the `const_Iterator` versions simply by duplicating the code in the Iterator versions. *The one exception* is the `find` function which is somewhat complicated. Try implementing a private helper finding function in `Ordered_list` that returns a `Node*`; call this function from the `const` and non-const versions of `find`, and use the returned value to initialize a `const_Iterator` or `Iterator` to return from the `find` function.

Using `Ordered_list`. As in Project 1, your program must use this `Ordered_list` container for all of the containers in the program - some will contain pointers, others will contain objects (see below).

---

[1] If you are completely satisfied with your project 2 implementation, and have the time and desire to learn more, check with the instructor for some pointers on using some relatively simple template metaprogramming to remove the duplication.

Check the supplied demo for examples of usage. You can use the `apply` functions (which are more reliable than Project 1 because they are more type-safe). These are modeled after the STL *algorithm function templates*: they take two `Iterators`, (or `const_Iterators`), and apply a function to each element in the specified range. Your client code can also access individual items in the list with iterators, which thus are a type-safe replacement for the `void*` item pointers in Project 1. For example, to output the list, you can step a `const_Iterator` through the list in Standard Library style, from `begin()` to `end()`, and dereference it to produce the output. 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`.

Iterator validity. 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 their validity would cause huge performance problems, so it hasn't been done either here or in the Standard Library. (Some implementations support a "debug mode" version of the Standard Library.) So it is the user's responsibility to avoid using iterators in an undefined fashion. But your container code can and should use `assert` to detect simple programming errors, such as trying to advance an `Iterator` that points to `nullptr` (the `end()` value).

Finally, as in Project 1, if we allocate a new object in dynamic memory, and then we try to store it in an Ordered_list that already contains a matching item, we must deallocate the new object and signal an error to the user. However, we can also insert actual objects in the list (not just pointers); the insert function attempts to copy or move them into the list Node. If the insert function reports a failure due to the presence of a matching item already in the container, there is nothing to dynamically deallocate - the object on the stack will just be discarded when we leave the relevant function.

Copy and move implementation. So that your `Ordered_list` is as usable and efficient as Standard Library containers, you need to implement both copy construction and assignment, and move construction and assignment. The copy constructor simply builds a list containing *new* `Nodes` that have the same data member value as in the original `Nodes`. The copy assignment operator simply does a copy/swap with `Ordered_list::swap`. Again, implement and test the copy construction and assignment before the move functions. Move construction amounts to just "stealing" the original data - simply shallow copy over the original's member variables (including the pointers), and set the original's pointer members to `nullptr` values to show that it is now empty. Move assignment is also easy to do - simply swap the lhs and rhs member variable values.

Move insertion. In C++98, when you inserted new data into a STL container, the container always made a copy of the supplied object. This was simple, but inefficient. In C++11, the STL containers have a move version of the insertion functions so that if you know that the data object isn't going to be needed any more (e.g., it is an rvalue), the insertion code moves the data into the container instead of making a copy of it. `Ordered_list` provides this capability with a second version of the `insert` function that takes an rvalue reference. This requires a `Node` constructor that takes an rvalue reference for the data object. Implementing this version is surprisingly simple. However, notice that it is pointless to use move insertion if the new data object is a built-in type like an int or pointer — these get copied anyway because there is no pointed-to data to steal.

Exception safety. It is valuable to ensure that `Ordered_list` provides both the basic and strong exception safety guarantees. This takes some care to arrange, so give this discussion some careful attention.

There are two key places when an operation on an `Ordered_list` will fail and throw an exception, and both of these happen when we try to add items - which are new `Nodes`. First, our attempt to allocate memory for the new list `Node` might fail; second, our attempt to copy the new information into a new `Node` might fail (for example, each `Node` might contain a `String` whose allocation fails when we try to copy it — of course, for some kinds of objects, copying might fail for some other reason). Exception safety means that if either of these cases happen, nothing bad or messy results.

Let's start with the list `insert` function: to get the new datum into the list, you'll have to create a new `Node` whose constructor should initialize its datum member using the supplied datum value. You should arrange to create the new `Node` before modifying the list in any way. Thus if the construction fails and throws an exception, you won't have modified the list, and you get both the basic and strong guarantee: failed `new` will automatically clean up its own business, including any `Node` member variables constructed before the `new` failed, and you haven't touched the list yet, so its invariant is maintained and its original contents are still there.

Copy construction of a list involves adding a series of `Node`s to "this" list, each of which has a copy of the data in the corresponding `Node` in the original list. A loop around a private `push_back` function (or the equivalent code) that allocates the new `Node` before modifying the list is the way to do this.[2] If creating one of these `Node`s fails, the basic guarantee requires cleaning up (deleting) any `Node`s (and their data) that were already copied. You can do this in two ways:

(1) The copying loop is wrapped in `try-catch-everything` block; in the `catch(…)`, you walk "this" partially constructed list and delete each `Node` (the same logic as in the destructor). This will work if your `push_back` code for adding each `Node` always leaves the list invariant intact - so you can still walk it successfully in the `catch`. This approach is simple conceptually, but doesn't reuse existing code very well.

(2) Use the more elegant RAII approach: Build a temporary copy of the source list, and let the list destructor automatically clean it up if an exception if thrown during the copying. Then swap its contents with those of "this" object. In more detail: First, initialize "this" object to an empty state, and then create an empty temporary list in a local variable, and use your `push_back` to copy the source data into the temporary list (instead of "this" list). Then use `Ordered_list::swap` to exchange the contents of the temporary list with "this" object. If the `push_back` code is exception safe, you can rely on the intact invariant so that your list destructor code can properly delete any `Node`s already created in the temporary list. (See Stroustrup's rather more complex example for an alternative picture of the concepts involved.)

Either way, to match the instructor's solution for failed copy construction, be sure that you delete any left-over nodes in the order that they were created - which is what you get if you walk the list from the front. Now you have an exception-safe copy constructor that you can use in a copy-swap method for the copy assignment, and this will then be exception-safe as well.

We'll also provide the no-throw guarantee in the form of `noexcept` specifications on all relevant member functions.

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 `Record*`), 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 simplify cleaning up messes when an error occurs. In these constructors, the input text strings should be read from a file stream directly into a `String` member variable using the overloaded input operator - if correctly implemented, this cannot overflow, and so no limited-length character buffers are required.

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.

---

[2] Don't try to use the `insert` function for two reasons: (1) It is ridiculously inefficient – you already know that each node should be added at the end, so why search for where to put it? (2) The `insert` function needs to use the comparison function, and depending on details of how the code is arranged, this can result in having to `#include` complete declarations where conceptually only an incomplete declaration should be needed. But putting even an incomplete declaration of a specific data object type in what is supposed to be a generic `Ordered_list` class implementation is a fundamental design error.

Thus removing a member from a collection does not result in removing that record from the Library nor destroying that record's data item.

`Record` has four different constructors. When a `Record` is created to add to the Library, the ID, title, and medium are supplied. When a `Record` is created for use as a probe in a title search, only the title is supplied. 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. Finally, there is a constructor used to create a `Record` from a file stream.

`Collection`s and `Record`s 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.

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. However, compared to Project 1, there is much less need for certain helper functions that would be shared between modules. In fact, the autograder's standalone test of Ordered_list assumes that it does not have to include Record or Collection in the build. If you have functions in Utility that call functions in Record or Collection, this build will fail. So take them out!

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. However, 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. This is to illustrate what we can do with templates that we couldn't do with C's approach to generic containers.

Important: To add a Collection to the Catalog, you have to create a Collection object then add it to the Catalog container. Arrange your code so that the new Collection is *moved* into the Catalog container, not copied. Do this both for the **ac** command, and when restoring a Collection from the save file in the **rA** command. Notice that the name String and the Record* container should get automatically moved in this process, saving time. The efficiency difference is small in the case of the **ac** command, but potentially large in the **rA** command. Notice that the same issue does not appear for the Library containers since they contain only pointers, not objects, and moving pointers into the container is no faster than copying them.

Since the Collection name `String`s will get moved in this process, there will be a difference in the memory allocation for `String`s compared to copying Collections— notice that if you copy a `String`, the copy gets minimum allocation, but if you move a `String`, the original allocation is retained — which is larger. So if your code doesn't match the sample output for **pa** after a **ld** command, check that you are moving rather than copying the Collection data during restoration.

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.

Be const-correct: when designing your main module functions that take an `Ordered_list` parameter, consider whether this should be a reference to a `const Ordered_list` or not. For example, in a look-up function that promises not to modify a container supplied by reference, you can't return a plain `Iterator`, only a `const_Iterator`. Start by writing the functions that you think you need, and then see if you can remove any duplicated code with one or more helper template functions.

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. The input of the individual characters for the commands should be done into single-character `char` variables, analogous to how they were read in Project 1.

Except for the `Ordered_list` instrumentation variables specified 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 `switch`es 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.

You also need an additional exception class, called `Title_error`, whose declaration should just be a clone of the supplied `Error` class. Your code should throw a `Title_error` exception if the last thing read from `cin` was a title and using that title produced some kind of error. A `catch` for `Title_error` should follow the `catch` for `Error`. and should output the message in the exception object, but then must not skip the rest of the input line (because the newline has already been read and consumed), and continues the loop to prompt for a new command. The declaration for `Title_error` should not be in Utility because it is only needed in the main module.

In addition, to make your program well-behaved, your top-level `try-catch` setup should have catches for the following conditions. For each of these, print an error message of your choice to `cout` and *terminate the program in the same way as a quit command* —"cleaning up" as in a normal program termination:

1. 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, so it is there to catch programming errors.
2. 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.
3. You should have a "catch all exceptions" `catch` in case one of the Standard Library functions throws an exception that you weren't expecting - seeing something like "Unknown exception caught!" is much more helpful than your program suddenly and silently quitting. Again this should be for programming errors only — in a correct program, there should be no way the user could cause such an exception.

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 user error messages must be packed up in `Error` or `Title_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!

Any additional `try-catch` blocks in this program are a sign of bad design - ask for help if you think you need them. But there are two exceptions:

1. 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 for your **rA** command function to catch internally any exceptions thrown by the functions that it calls, do the roll-back, and then rethrow the exception (simple: a `throw;` statement in a `catch` is how you should do a rethrow - whatever the exception object we caught, the same object is rethrown) so that the top level `try-catch` can then finish handling the error in the usual way.

2. If we really want to be exception-safe, we need to ensure that any allocated memory gets deallocated if an exception is thrown. Here is a scenario: We are adding a new Record to the Library, so we allocate memory for it. If one with the same title is already in the Library, we must deallocate the new Record, as described above. However, suppose it is not already in the Library, but inside `Ordered_list`, the attempt to allocate memory or copy the data for a new Node fails for some reason, and `Ordered_list` throws an exception. If we allow that exception to propagate out of our add-record function, then the memory allocated for the new Record will be leaked. The solution is to have a local `try-catch(…)` where we deallocate the new Record and rethrow the exception. The better solution is to use smart pointers, but we aren't using them (or a your own equivalent) yet. Use the local try-catch for this problem instead; remember that it is only necessary when we are trying to put the pointer to a newly allocated object into an `Ordered_list`.

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

4. `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.
5. 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.
6. `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.

Container Iteration Requirements

You *must do the following in your main module* to get some experience with how container iterations are done "STL style".

- Use a `range for` on the Ordered_list container at least once - it should work! If you use `range for` on the Catalog, make sure you aren't making a copy of each Collection as you go - not only is this is inefficient, but also it may result in the code not working correctly, depending on what you are trying to do.
- Use one of the Ordered_list "`apply`"-type functions at least once. Notice what the comments in the skeleton header say about these.
- Use an ordinary `for` loop with an explicitly declared `Ordered_list::const_Iterator` as the loop variable at least once (that is, don't declare the iterator type with `auto`).
- You are free to use whatever forms of iteration you want in the other modules.

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 not using `#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 not use the Standard Library `<string>` class or any of the Standard Library (or "STL") containers, algorithms, binders, iterators, adapters, or `std::bind`, `std::function`, lambda expressions, stringstreams, or smart pointers (`shared_ptr` or `unique_ptr`), in this project. This project is about implementing basic classes and containers, and using templates. Project 3 will emphasize(!) use of the Standard Library. The idea in this project is to learn how Standard Library things are done by writing some simplified versions yourself. Subsequent projects will allow (actually require) full and appropriate use of C++ features and the Standard

Library. Do not attempt to recreate the powerful and general Standard Library facilities. For example, don't try to write your own version of smart pointers - the approach in this course is to start with "raw" pointers, even if they are clumsy, so that when we start using smart pointers, you will understand their value.

4. You may not use inheritance; it is simply not useful in this project and would be nothing more than a complication and a distraction.
5. You may use `auto` throughout the program (but see the exception under *Container Iteration Requirements* above).
6. You may use only `new` and `delete` — `malloc` and `free` are not allowed.
7. Before your program terminates with the quit command, all dynamically allocated memory must be deallocated — in this project, correctly working destructor functions will do this automatically in many, but not all, cases. See above for details.
8. There must be no unnecessary or premature memory allocations or memory leaks apparent in your code.
9. 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 the single `char` array that stores the internal C-string; no other dynamically allocated arrays are allowed in the `String` member functions.
10. 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>` and `<cstring>`. See a Standard Library reference, or the 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 the simple functions from `<cctype>` and `<cstring>` part of the C++ Standard Library such as `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 `String` objects or individual `char` variables (e.g. for the commands). This means that `<cstring>` should not need to be #included in any other module.
11. `String::operator+=` must not create any temporary `String` objects or unnecessarily allocate memory. See above for more discussion. The comments in the skeleton String.h file specify which operators and functions create temporary `String` objects.
12. You can and should `#include <cassert>` and use the `assert` macro where appropriate to help detect programming errors.
13. You must follow the recommendations presented in this course for using the `std` namespace and minimizing header file dependencies.
14. Your submited `Ordered_list` code must be all contained in the header file; no .cpp file for this component should be submitted. See below for more discussion.
15. Except where specifically required, you may not use any global variables, both internally- or externally-linked 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++, file-scope `const` variables automatically get *internal linkage* by default, so you don't have to declare them static or in the unnamed namespace in a header or source file.
16. 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

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.

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.

We will not 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, we plan to do a spot-check as described in the Syllabus

and deduct points from your autograder score if your code does not appear to represent an effort to follow the specified coding requirements and code quality recommendations. The spot check will cover things that we can screen for quickly. Examples: Did you use copy-swap where specified? Did you include headers unnecessarily? 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.

We are doing full-fledged C++, not "C with cout." So 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 `int`s, write `while(true)` instead of `while(1)`, write `foo()` instead of `foo(void)`, and declare variables at the point of useful initialization instead of all at the start of a block, and `#include` any C Standard Library headers using their C++ header names, (like `<cstring>`) instead of the C header names (like `<string.h>`)

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 "rule of five". Put off the input operator at first, but 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, swap member function, and copy assignment operator, move constructor and assignment, and then the += concatenation operators. Then do the remaining functions. Save `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 speedy. 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. Thus it is a C++ idiom to favor += over (= and +) if it does what you want.

You have to give your `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 before those others. 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. Then add the move operations. Keep in mind that move just "steals" the data along with the member variable values that go with it. Because it does not make a fresh copy of the

data, this means that if the allocation was originally "too big", it will remain so after move assignment - unlike the case with copy assignment.

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 the 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.

Constructor Elision - a confusing routine optimization: Compilers *love* to "elide" (eliminate) many calls to copy or move constructors - a traditional and important optimization. Unfortunately the result can be confusing - it doesn't look like what is supposed to happen. For example, the copy construction of a `String` being returned from a function will often get optimized away — the "Returned Value Optimization (RVO)." You can see the "by the book" activity if you can turn off these optimizations. If you are using `g++`, use the option `-fno-elide-constructors` to cause all copy or move constructors to be left in the generated code.[3] Because different compilers differ in how aggressively they elide constructors, you will see different results from different compilers when you turn on the `String` messages. The course is using `gcc/g++ 5.1.0` as the "standard" so upload your code and build on CAEN with this option to see this stuff happening.

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 class objects and pointer types. You will waste huge amounts of time if you struggle with debugging `Ordered_list` in the context of the whole project.

Important: 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 and common 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. If you were told to use and #include a .cpp file in previous courses, it was wrong, or at least seriously deviant from normal practice*. 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, and then delete the old .cpp file.

Dummy 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, even if your Catalog list is empty, there is still a couple of `Collection` objects and their member variables in existence! And by the way, what are their names? 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 and function call overhead isn't an issue (which it would be for a trivial function). But do not hesitate to directly go to the private member variables such as the actual `Node` pointers. 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 code will be more convoluted than going directly to the member variables of the list and its nodes.

---

[3] Xcode LLVM Clang has this option, but it has been buggy, so don't use it until it has been fixed. MSVS doesn't appear to have this sort of option.

Finally, because we have a pretty complete implementation of the important member functions of `Ordered_list`, providing the "roll-back" for a failed load should be really easy and efficient. See if you can implement this in a way that is both simple and super-efficient.