Project 2
The Micro Meeting Manager in C++:
Domain Classes, Templates, Exception Safety, Move Semantics
Due: Friday, Oct. 11, 2019, 11:59 PM

Notice:

Corrections and clarifications posted on the course web site become part of the specifications for this project. You should check this 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 template member functions.
- Using function templates to reduce code duplication, especially for const-correct functions.
- Using some simple static member variables.
- Defining and using simple function object classes.
- 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, simple function objects for ordering, 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.

Because of the variety of C++ programming techniques involved, this is probably the most technically challenging of all the projects in the course. There is *a lot* of conceptual and how-to material in this document in addition to detailed specifications. Plan on taking some time to study this document very carefully before you write any code, and review it frequently while you work on the project to make sure you haven't missed something important — about either the concepts or the specifications.

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, and the fact that a failed load command "rolls back" the containers to their original state.

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 the *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 that 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. The output of the **pa** command is different.

2. Because you will be creating and using a `String` class with automatically-expanding capabilities, there are no restrictions on the length of names, meeting topics, or other user-supplied strings.

3. Some differences in the error reporting for the **am** and **rm** commands result from delegating the error checking to `Room`. See later in this document for specific details.

4. If the **ld** command detects an invalid input file, instead of leaving the user with an empty people list and rooms list, it implements a "roll back" to the previous program state. Thus the **ld** command has different behavior from Project 1, and behaves as follows:

- **ld** <filename>—load data—restores the program state from the data in 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 opening succeeds, the program saves the current people and schedule data in local backup variables and then attempts to read the people, rooms, and meetings data from the named file into empty people and room containers to restore the program state to be identical to the time the data was saved.
- Then if the restore was successful, the backup of the data is discarded.
- 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 **ld**. That is, any new data objects that were created while reading the bad file are deleted, and the contents of the people and room containers get replaced with the values saved in the local backup variables. Then the error is reported.
- Thus the possible results of issuing a **ld** 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 people and room containers, and cleaning up after a successful or failed **ld** should be very simple and efficient, given the specification for Ordered_list. If this code has more than a few lines, or copies the saved/restored data, you have missed an important point.
- *Note:* As specified below, the restore involves reading strings from the file directly into `String` member variables. This means that the total `String` memory allocation shown in **pa** after a restore will likely be different from what it was before the save. This difference in memory does not count as a difference in program state, since the actual data should be the same.

Class Design—Responsibilities and Collaborations

In this project, you will be implementing classes of two kinds: One kind represents objects in the problem domain: Persons, Meetings, and Rooms. The second kind define objects that support the domain classes, these are Strings and Ordered_lists. The contrast is between the things in the world of meeting management versus the computer-science things that programmers like to have 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 domain class, and how each such class collaborates or relates to the other classes. This is primarily an issue with the domain classes; the support classes play no role in this issue, but merely help with the implementation. Examine the supplied starter files for each class as you read the following, to see how the

responsibilities and collaborations play out in the specified public interface. This section walks you through the specified design, and describes some of the pros and cons of the design as specified.

Person class. A Person object has very simple responsibilities. First, it holds the data about a person using String objects. Second, it is responsible for how to output a person's data—it does this with the output operator definition, which is a friend function, and therefore part of the public interface for the Person class (all of the domain classes have this responsibility). Third, it 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 Person class defines an ordering relationship, following the common convention of defining `operator<` to represent what it means for one object to "come before" another in order—in this case, depending on the alphabetical order of last name.

However, the public interface of Person reveals a bit more. There is no way to change a person's data once the Person object has been created. The Person class is responsible for ensuring that a Person is immutable—a person's names and phone number are permanent. Technically, this is incorrect about the real world of persons' names, not to mention their phone numbers, but given the program's functionality, defining the Person class this way makes it very clear what we are assuming about persons. In fact, there is a good effect of this limitation—what would happen to an ordered list of Persons if it were possible to change a Person's last name? By making this immutable, an Ordered_list of Person objects is "safe" from being disordered by accident.

Furthermore, the copy and move constructors and assignment operators are all declared as unavailable—they are marked as `=deleted` in modern C++ style (in C++98, you would declare them as private and not define them). This means there is no way for client code to "clone" a Person—this is a way to represent the idea that both real people and Person objects are supposed to be unique—it doesn't make sense to have more than one copy of a Person, so a Person is always referred to with a pointer to what is supposed to be the unique Person object. This way, the Person object's address is a complete way to identify that unique Person object. In fact, in C++, an object's identity is given by its address in memory. Finally, to nail down the immutability of Persons, and provide some practice with const-correctness, we will refer to them everywhere with pointers-to-const, `const Person*`.

However, this last step is problematic: Suppose we wanted to add a command to change a Person's phone number, and added a member function to Person to simply put in a new phone number. We've decreed that Person objects are always referred to with pointers-to-const, which makes them immutable, even if the class interface allows them to be changed. Keep in mind that this decision might be a poor choice if the program functionality is expanded, and if so, it would need be changed throughout the code base rather than being patched-over or "kludged".

Meeting class. The Meeting class has more responsibilities than holding data and knowing how to output, save, and restore itself. It has the responsibility of maintaining its participant list—the public interface provides methods for adding and removing participants and determining whether a participant is present. In other words, the main module handles participants by delegating the actual work to the Meeting object. Note that there is no way for client code to access the list of participants directly, reflecting a design decision that it is strictly up to the Meeting object to keep track of its participants. The Meeting class also provides a definition of what it means for one meeting to come before another, namely in order of time of day. Unlike Persons, Meetings can be copied and moved with construction or assignment—the design decision was that unlike human beings, making a copy of a Meeting is meaningful—for example to make it simpler to reschedule a meeting.

However, there are some problems: A public interface function is provided to alter the time of a Meeting once it has been created. This seems reasonable to simplify rescheduling a meeting, but notice that if a Meeting is in a list ordered by time, and it is possible to access the Meeting in that list, then it is also possible to disorder the list by changing the time of the Meeting while it is in the list. The project design as specified has no easy way to prevent this programming error. Instead, the programmer has to "*program by convention*"—he or she must try to follow a rule which is not represented in the code itself, but is stated outside, or separately from, the code: *Do not change the time for a Meeting while it is in a container ordered by time*. The best way to meet this requirement is the common approach for safely changing the *key* value inside an object being held in an ordered container: (1) make a copy of the object, (2) remove it from the container and discard it, (3) change the key in the copy, and (4) insert the modified copy into the container. Note that In Project 1, because the object was pointed-to by the container, you didn't have to copy it. Rather you merely had to remove the pointer, change the object, and then re-insert the pointer. But in this Project, the Meeting is specified to be an *object* in Room's container, so the copy is necessary.

Room Class. Finally, the Room class is similar to the Meeting class in that it manages its internal contents, in this case, a list of Meetings. Like Meetings, we allow Rooms to be copied and moved. However, the design decision is that Meetings belong to Rooms, and so only Rooms know who the Meetings are. Moreover, except for the brief time while being created or rescheduled, Meetings are always in a Room. Thus, if a Room is destroyed, then any Meetings in the Room should also be destroyed by Room's destructor.

The public interface provides ways to find, add, and remove Meetings. There is no way to directly access the internal list, so if you want to output the Meetings, you have to ask a Room to do it through its output operator. Note that there is no way to change the Room's number once it is created, so we don't have to worry about list orderings being disrupted like we do with Meeting.

There are at least two problems with the design of Room: First, although Rooms claim "ownership" of Meetings, they are willing to give outsiders (the client code) complete access to a Meeting if the outsider can identify it by meeting time—the `get_Meeting` function will return a non-const reference to the Meeting at the supplied time. The idea is that the client code can then use this reference to easily modify the participants in the Meeting through the Meeting public interface. More seriously, the client code could directly change the Meeting time without consulting the Room, meaning that the Room can't guarantee the accuracy of the time-order of the Meeting list. This is definitely not good. To help with this, we have equipped Room with a `const` and non-`const` version of the `get_Meeting` function. This means that if the Room is `const`, the compiler will automatically select the `const`-preserving version of `get_Meeting`. So if the client code is `const`-correct, the compiler will help us avoid modifying a Meeting time while in the Room's container. But notice that we have to be able to modify the participants in a Meeting, so we have to access a Meeting as non-`const` some of the time.

This brings us to the second problem, which is more subtle, and reflects some confusion in the design. While apparently Room doesn't try to deal with Meeting participants directly, it has a method for determining whether a Person is a participant in any of the contained Meetings. This function appears to be a good idea, because otherwise a client would need access to the Room's list of meetings in order to search for the presence of a participant. However, why doesn't Room insist on protecting the integrity of its Meeting list otherwise? Should it be in the participant business at all? And if so, shouldn't it be handling all of the participant issues on behalf of its Meetings? But wouldn't that make Room's interface overly complex?

Like all designs, the specified one is a compromise — remember that *no design is perfect; it is always a matter of trade-offs*! Try to become aware of these issues as you build this project, and it will help you understand Object-Oriented Design faster. In fact, here is a good design exercise in preparation for the next project: After you have completed this project, and seen how the issues work out in the actual code, see if you can come up with a better design, or at least one whose trade-offs are different and better in some respects than in the specified design.

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, Persons, Rooms, and Meetings 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 various class objects. A good example is how the main module delegates the many details of restoring the data from a file to the specialized constructors of Person, Room, and Meeting. All main does is manage the overall process — for example, main knows that the Person data comes first in the file, followed by the Room data, and what has to happen if there is an error in reading the data. But main does not have to micro-manage how Persons, Rooms, and Meetings are described in the file and how that data is read and used; those classes are responsible for that work.

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 any way you choose, within the constraints of good programming practice (see the Coding Standards) 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 name, parameters, or behavior.
- No specified functions can be removed.
- No public member functions can be added.
- If the public interface specifications include the output operator for the class, then you are expected to declare it as a friend function if necessary. Otherwise, no friend functions or classes may be added beyond those specified.
- No public member variables can be added.
- Only private functions, classes, or declarations can be added to the class header file.

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" 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" part of the name 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, 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." These operations are adequate for this project; please ask for help if you think you need more.

The project web site 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 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 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. An additional *allocation* member variable keeps track of how much memory is currently allocated to hold the C-string. In general, if the String is non-empty, the allocation is always greater than or equal to the string length + 1.

The default-constructed or 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. Modern high-quality implementations of `std::string` use the *small string optimization* (see the example in Stroustrup): the string object contains a smallish array of char and this is used to store small strings. Only if the string gets large, do we allocate memory for it. Since many strings are small, this really speeds things up. Here, we will use a simpler (and less efficient) idea: the *empty string optimization*. If the String is non-empty, it has an internal pointer to the C-string in allocated memory. If the String is 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`. But this does complicate the memory management logic somewhat: If you add characters to a `String`, you have to check on

whether it is empty or non-empty so you will know whether and how much memory to allocate; likewise, when the `String` is destroyed, you should use `delete[]` only if it is non-empty.

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 by taking advantage of how the compiler will use the `String` constructor to automatically create a temporary `String` object from a C-string (but only if there is already a `String` involved in the comparison). 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 `std::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` 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 a 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. But peeking is fun!
- 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 lines of code in the body. Ask for help if your code is more complex.

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 achieve the final result of deallocating the space originally belonging to the left-hand side object which ends up with a separate copy of the data in the right-hand side object. It must also be safe against aliasing (self-assignment). 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 of "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` 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 (for speed, `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 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 code has a consistent ordering of the output, though the resulting messages are often about the future, rather than current, state of the object.

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

Ordered_list.h. This header file contains a class template for a list container called `Ordered_list`, which is based on the Project 1 list implementation of `Ordered_container`, but it is simpler and cleaner because it does not use the opaque type approach, meaning that you can instantiate a container object as a local stack variable. 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 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. But you may not modify the public interface, and you must follow the specified implementation approaches (such as the use of copy-swap and 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 because in simple cases, the only member is the function that overloads the function call operator, and this function 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, because 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 for common orderings. 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 Person, Meeting, and Room. 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, and puts the new item in the list in front of it. 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 terms of what the less-than operator compares. For example, if x and y are Persons, then `Person::operator<` compares just the lastnames—if the lastnames are the same, then we will treat the two objects as equal even if the other data are different.

Finding objects in the container. 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. The probe object has to be the same type as the objects in the container, which is like the pre-C++14 STL containers; we don't have support for the *heterogenous lookup* approach for Project 1's container,

with its "custom" ordering function for the find function; this support was added to the C++14 STL and will be presented later.

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. For example:

```
Person probe("Smith");
```

or

```
Person probe{"Smith"}; // C++11 style
```

This is an object that will compare equal to the object named "Smith" in the people list—the probe does not need a firstname or phonenumber 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. Note that if you have a container of pointers, you can create a probe object as a local (stack) variable and then use its address; no need to create it on the heap with its risk of a memory leak!

Nodes and Iterators. The `Ordered_list<>` class template uses a separate template class `Node<>` for the list nodes, which has the interesting feature that all of its members are private, but it declares `Ordered_list<>` as a friend. So `Node<>` exists as an implementation detail of `Ordered_list<>` and is not usable by client code.

The `Ordered_list<>` class has a nested class, `Iterator`, which is similar the STL 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 STL iterators. Similarly, `Ordered_list<>` has `begin()` and `end()` member functions that return iterators for the first and next-after-last items in the container, and a `find()` function that returns an `Iterator` pointing to the matching item.

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 objects 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, and destroying the list object is properly implemented as destroying the list node objects, then 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. It is not a container's job to read your mind about when you want pointed-to memory deallocated — maybe you want to keep it, and have some other pointer to it!

Supporting const containers. If an `Ordered_list` has been declared `const`, then it needs to be impossible to alter its contents with iterators. 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 the contents of the container through a `const_Iterator`. The member functions that can modify an object in the container require an ordinary non-`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. For convenience, there is a constructor that converts a plain `Iterator` into a `const_Iterator`.

There is a version of `begin()`, `end()`, and `find()` that are declared as `const` member functions, and return `const_Iterators`. 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 automatically 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_Iterators` 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 technically possible to eliminate the duplication, the template metaprogramming techniques involved are beyond the scope of this project. Fortunately, in all but one case, the functions are trivial — very small and short, and so in the interests of simplicity for this project, you should implement the `const` versions simply by duplicating the code in the non-`const` versions. *The one exception* is the `find` function which is somewhat complicated. A simple solution: define a private finding helper 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 linked-lists in the program — some will contain pointers, others will contain objects (see below).

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 are a type-safe replacement for the `void*` item pointers in Project 1. For example, to output the list, you can step an `const_Iterator` through the list in Standard Library style, from `begin()` to `end()`, and dereference the `const_Iterator` 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`.

Insertion memory management. 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 be sure deallocate the new object. However, thanks to the template facility in C++, we can also have lists of actual objects, instead of just pointers. In this case, to add a new object to the list, we can create it as a stack variable and then call the insert function to copy or move it into the list. 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.

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. (However, 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 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).

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. Your copy constructor simply needs to build a list containing *new* `Nodes` that have the same data member values as in the original `Nodes`. The copy assignment operator simply does a copy/swap using `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. Starting with C++11, the STL containers have a move version of the insertion functions so that if 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, remember that if the new data object is a built-in type like an int or pointer it gets 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, and this project is a good place to learn the basics of exception safety. This takes some care to arrange, so give this information 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 `Node`s. 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. Simply use a loop around a private `push_back` function (or the equivalent code) that allocates the new `Node` before modifying the list.[1] 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 accomplish an exception safe copy constructor 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, which is the same concept as copy-swap: 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 this temporary list. Then use `Ordered_list::swap` to exchange the contents of the temporary list with "this" object. As above, your `push_back` code needs to be exception safe by preserving the list invariant. If copy-construction fails and throws an exception, the compiler will automatically call your list destructor to 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.

---

[1] 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 of the list data object type 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.

Room.h, .cpp, Meeting.h, .cpp, Person.h, .cpp. `Room`, `Meeting`, and `Person` are three more classes, each in its own header-source pair. The design of these classes and their responsibilities are described above. 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. `Room` has an `Ordered_list` of `Meeting` objects), 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.

As in Project 1, the list of participants in a meeting must contain pointers to `Person` objects that are also pointed to by the people list. An individual person's data must be represented only once, in a `Person` object created when that person is added to the people list. All participant lists simply point to the corresponding `Person` objects. Thus removing a participant from a meeting does not result in removing that person from the people list nor destroying that person's data item.

All lists of `Person` pointers must contain pointers-to-const, as in

```
Ordered_list<const Person*, Less_than_ptr<const Person*>> people_list;
```

The `Room`, `Meeting`, and `Person` classes have 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 makes it easy to clean up messes when an error occurs.

Note: If a value for a member variable is read from the file, it should be read directly into the member variable. In particular, 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. Be sure to follow this specification, because the total memory allocation for Strings will be affected.

Utility.h, .cpp. You must have a pair of files, Utility.h and Utility.cpp, to contain any 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 Person is not there)
   throw Error("Person not found!");
```

The error message strings are specified in the 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.

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. Remember that Utility.h must not be used as a dumping ground for all the header #includes needed in the project: the Header File Guidelines take precedent; Utility.h can #include only header files that are needed in order to compile Utility.h by itself. Component tests may fail if you violate this rule.

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.

To practice using containers of both objects and pointers, the main function must declare two `Ordered_list` variables; one must be a list of `Room` *objects*, and one for a list of *pointers* to `const Person` objects. These must be declared as:

```
Ordered_list<Room>
Ordered_list<const Person*, Less_than_ptr<const Person*>>
```

Notice that because we have a container of `Room` objects, and each `Room` has a container of `Meeting` objects, deleting a `Room` should automatically delete the `Meeting`s in it.

Except for the `Ordered_list` instrumentation variables described above for p2_globals.h, .cpp, you may not have any global variables, even if they are local to a file.

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 or locally declared arrays of char are required anywhere in the project, and must not be used. Single character input, as for the commands, should be done into single-character `char` variables.

Using move insertion. Your implementation of the load command in p2_main and the file constructor of `Room` should take advantage of the move version of `Ordered_list::insert()` for greater efficiency. For example, when creating a `Room` from the data file, each `Meeting` constructed from the file should be *moved* in the container of meetings in the `Room`, not just copied in — copying a `Meeting` requires copying the `String` for the topic, and then copying the participant list. If the `Meeting` is moved in, then the topic `String` will get moved during this process, along with the data in the participant list —changing a few pointers and int member variables rather than a lot of memory allocation and data copying. Likewise, in the main module, when a `Room` gets constructed from the file, it can be simply moved into the `Rooms` container, whereupon all of its `Meeting` data gets moved rather than copied.

Since the topic `String`s will get moved in this process, there will be a difference in the memory allocation for `String`s compared to copying — 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 usually larger. So if your code doesn't match the sample output for **pa** after a **ld** command, check to be sure you are moving rather than copying the `Meeting` and `Room` data.

Likewise, your **am** command can move a newly created `Meeting` directly into the `Room` using the move version of the `add_Meeting()` function. In contrast, for the **ar** command, there is little or no advantage of moving a newly created `Room` into the `Rooms` list because it has an empty `Meeting`s list and only a single int member variable — moving this information is no faster than copying it.

Special note about the backup containers. The containers used for the backup in the **ld** command must not be declared in the main function, but rather in your **ld** function—they should not be needed anywhere else.

Your main module command functions must be const-correct. As in Project 1, your main module must be organized around function call tree with a high-level function called by `main` for each command (except the quit command). Most of these functions will have at least one `Ordered_list` parameter. Consider whether this parameter should be a reference to a `const Ordered_list` or not. For example, the command function for printing an individual should treat the Persons container as read-only by taking it as a const reference parameter. A more complex example: To print a specific Room, the command function needs the Rooms container, but does not need to modify it, so it should take it by `const` reference. In contrast, the add-meeting function needs to modify a Room, so the Rooms container parameter must be a non-`const` reference. In both cases the command function should call a Room look-up helper function that reads the room number and returns an iterator pointing to the specified Room in the container. To maintain const-correctness, this lookup function needs to be const-correct.

For the print-meeting case, the look-up helper function should take the Rooms container by `const` reference, and since the container is read-only, it should return a `const_Iterator` pointing to the specified Room; but for the add-participant case, the look-up helper sub-function needs to take the Rooms container by non-`const` reference and return a plain `Iterator` so that the specified Room can be modified. This is essential to make the code const-correct — if you have an unmodifiable container, you need a look-up helper that returns a `const_Iterator`, and if you have a modifiable container, you'll need a look-up helper that returns a non-`const` `Iterator`.

Does this mean you need write two versions of every look-up helper function? Yes, if you just write plain code. You can try writing it that way at first, verify that the code is working, but then you should eliminate the duplicated code with some function templates and some new techniques involving using `auto` as the return type of a function template.

C++14 provided this feature: you can declare the return type of a function as `auto`! The compiler will figure out what the returned type is from the return statement in the function, including the case we need here, where the returned type depends on the parameter type and whether it is `const` or non-`const`. Moreover, the `auto` return type will work with a *function template* if the compiler has seen already seen all the information needed to figure out the returned type before your code calls it. There are two techniques for supplying this information; your solution can use either Technique #1 or #2 described below, or a mixture of both:

Technique 1: Define the template function before all calls. Define the complete function template after you have declared your container types, and before your code that calls it. The function takes a container parameter by reference whose type is the template type parameter. For example, here is a function that looks up something in a container and returns a iterator pointing to it:

```
template<typename T>
auto get_iterator(T& the_container)
{   /* look-up code not shown */
    return the_iterator;
}
```

When this function is called with a container argument whose type is `container_type`, if the container is `const`, then `T` will be `const container_type`, and the iterator code inside the function will be automatically working with a `container_type::const_Iterator`, and the compiler will figure out that `container_type::const_Iterator` should be returned. If the container is non-`const`, then `T` will simply be `container_type`, and a plain `Iterator` should be returned. Other useful helper functions can return a reference to a Room or Meeting; if so, the return type should be `auto&` and the result will be a `const` or non-`const` reference.

Very cool! Now a single definition of this function template gives you both versions of `get_iterator` that you need for const-correctness!

Technique 2: Declare the template function first, define it later. You can forward-declare the function template before the first call of it, and then provide the complete template definition anywhere later in the source file. This is analogous to first declaring an ordinary function with a prototype before the first call of it, and then providing the definition anywhere later in the source file. The syntax for this involves using the `auto` return type with a *trailing return type* and `decltype` (which means "declaration type"). The concept of the trailing return type is that if the return type depends on the function parameter, then we need the compiler to see the parameter type before it tries to figure out the return type. By putting the return type after the parameter, the return type can then be described in terms of the parameter. Using our previous example, the template declaration would be:

```
template<typename T>
auto get_iterator(T& the_container) -> decltype(the_container.begin());
```

The return type is `auto`, the -> arrow marks the trailing return type which is given by `decltype`. The `decltype` expression involves the parameter `the_container`, which will be either a `const` or non-`const` reference. This expression is *not evaluated or computed*; the compiler just studies the expression when instantiating the template to figure out what the return type should be based on the type of `the_container`. If `the_container` is `const`, then the `begin()` iterator will be a `const_Iterator`, and that's what the return type should be; if it is non-`const`, the `begin()` iterator will be a plain `Iterator`, and that is the return type.

When we later define the complete function template, we repeat the function header to match exactly:

```
template<typename T>
auto get_iterator(T& the_container) -> decltype(the_container.begin())
{   /* look-up code not shown */
    return the_iterator;
}
```

Now you can declare the template functions with the regular function prototypes, and then like the regular functions, you define them in a convenient location location in the file. No need to frustrate the code reader with a glob of template definitions at the start of the code!

A couple of nuances: To return a *reference* with this technique, you have to specify it in the trailing return type, not with the auto return type. If the `decltype` expression involves a function call, you may need to supply a dummy argument for the call. For example, to get a const or non-const reference to a Meeting in the Rooms container, we will need to find the Room via an iterator (either const or non-const), and then get a reference to a Meeting in that

Room; this reference needs to be either const or non-const, and the final return type needs to be a reference. We write:

```
// return a reference to the specified Meeting
template<typename T>
auto get_meeting(T& rooms) -> decltype(rooms.begin()->get_Meeting(1))&;
```

Again, note that the `decltype` expression is not actually computed; the compiler uses it to figure out what flavor of room container iterator is involved, which then determines which version of the `get_Meeting` function would get called (which it needs the parameter type to be sure about). Finally, after figuring out the return type, the compiler sees the final `&` and makes the `auto` type a reference of the right flavor.

This technique #2 is one of the "dusty corners" of C++, but trying it out will introduce you to a bit of more advanced template programming. If you want to try it, I suggest using #1 first, and then changing the function templates to use #2 one at time, starting with the simplest return types.

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.

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 additional catches for the following; for each of them, first output an error message of your choice to either `cout` or `cerr` and then terminate the program as if a quit command had been entered (do not call `exit()` — see the Coding Standards).

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 should include an "all exceptions" `catch` as the last `catch` in case one of the Standard Library functions throws an exception that you weren't expecting—seeing a message like "Unknown exception caught!" is much more helpful than your program just suddenly and silently quitting. Again this should arise only from programming errors—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 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!

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 **ld** 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 **ld** 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) 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 Person to the person list, so we allocate memory for it. If a Person with the same last name is already in the person list, we must deallocate the new Person. However, suppose the new Person is not already present, but inside `Ordered_list`, the attempt to allocate memory or copy the data for a new list Node fails, and `Ordered_list` throws an exception. If we allow that exception to propagate out of our add-individual function, then the memory allocated for the new Person will be leaked. The solution is to have a local catch-anything `try-catch(…)` where we deallocate the new Person and rethrow the exception. The better solution is to use smart pointers, but we aren't using them (nor your own equivalent) yet in the course. Use a 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, we will rely on `Meeting` and `Room` to be responsible for preventing duplicate entries. This produces the following deviations from the error-handling pattern in Project 1:

4. `Meeting::add_participant()` throws an exception if the participant is already present in the `Meeting`. Therefore, the main module code for the **ap** command can get the pointer to the specified participant and simply add it to the `Meeting`; it should not check for the participant being in the `Meeting` already.
5. `Room::add_meeting()` (both versions) throws an exception if a `Meeting` is already scheduled at the time. Therefore, the main module code for the **am** command should read the time and check that the time is a legal possible time (i.e., between 9 and 5), then read the topic, but then let `Room` check the time for duplication when we add the `Meeting` to the `Room`. Similarly, when rescheduling a `Meeting`, read the new time and check that it is a legal possible time, and then check on the duplication of the new time by trying to add the `Meeting` to its new `Room`. Note that the `Meeting` should be left in its original `Room` until it has been successfully added to the new `Room`, so that if the rescheduling fails, the `Meeting` is left in its original time and place.

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 Room list, make sure you aren't making a copy of each Room 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>`. Review 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, `std::bind`, `std::function`, lambda expressions, `std::stringstream`, or smart pointers (`shared_ptr` or `unique_ptr`) in this project. We'll use them in later projects. This project is about implementing basic classes and containers and using templates. Project 3 will emphasize (!) using the Standard Library and many other language features. The idea in this project is to learn how some of these 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 and trade-offs.
4. You must use `std::swap` and `std::move` appropriately. These function templates are declared in the `<utility>` Standard Library header. Important: You should not use `move()` except to tell the compiler to make an lvalue, such as a variable, movable by casting it to an rvalue. Also important: `std::swap` cannot substitute for the class member function `swap`, such as `String::swap` or `Ordered_list::swap`, but it is very handy in writing the code for these functions.
5. You may not use inheritance; it is simply not useful in this project and would be nothing more than a complication and a distraction.

6. You may use `auto` throughout the program.
7. You may use only `new` and `delete`—`malloc` and `free` are not allowed.
8. 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.
9. There must be no unnecessary or premature allocations, and no memory leaks, apparent in your code.
10. 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 declared or dynamically allocated arrays are needed or allowed in the `String` member functions.
11. Your String.cpp file can use any functions in the C/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; for example `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.
12. `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.
13. You can and should `#include <cassert>` and use the `assert` macro where appropriate to help detect programming errors.
14. You must follow the recommendations presented in this course for using the `std` namespace and minimizing header file dependencies.
15. Except where specifically required, you may not use any global variables, either externally-linked (program-wide) or internally-linked (scoped in a single file). You may certainly use global constants in a module's .cpp file for things like representing output string text. 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 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

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 your code does follow the specified coding requirements and code quality recommendations. Examples: Does your code use "copy-swap" as described in this course in the specified places? Does it use "this" unnecessarily? Does it include headers 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. Modify the Project 1 versions to be more suitable to how this project's `Ordered_list` works, and how we are using exceptions.

We are doing full-fledged C++, not "C with Classes." 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)`, and `foo()` instead of `foo(void)`. Continue to 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). You can recycle a lot from your Project 1 code. 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 three" of destructor, copy constructor, and copy assignment operator, followed by expanding to the new C++11 "rule of five"—which adds the move constructor and move 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 "rule of five" functions and the swap member function, 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 speedy. In `std::string`, `operator+=` is expected to provide very fast appending. For example,

```
s1 += s2;
```

will run fast because if the left-hand-side (lhs) object has enough space, then `s2`'s characters can be simply copied over at 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 right-hand-side (rhs) involves creating a temporary object copied from `s1` and then concatenating `s2` into it, and then copying or moving the contents of the temporary object into the lhs variable, and finally discarding the original lhs contents. There is an idiom in C++ for avoiding this inefficiency: prefer `std::string`'s += 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. 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 routine but confusing 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, if a function returns a `String` by value, "technically" the returned value is supposed to get copy or move-constructed as the function returns. But this construction is often optimized away—the "Returned Value Optimization (RVO)" — instead of constructing the variable to be returned on the local function stack frame, the compiler constructs the returned object directly into the place on the stack where the returned value will be after the function returns! Some of the traditional elisions are now mandatory in C++17, meaning that you can depend on them to be present. You can see this by turning on the String messages and noticing when copies and moves appear, and you can see the activity of the optional optimizations if you turn them off in g++ using the option `-fno-elide-constructors`. Different compilers differ in how aggressively they elide optional constructors; the course is using gcc/g++ 9.1.0 with C++17 as the "standard", so upload your code and build on CAEN with this option to see this stuff happening. See the demos for examples, and a detailed discussion.

Building the Ordered_list template. Templates can be a pain to debug because most compilers and debuggers generate really verbose and confusing messages about template code. First code `Ordered_list` as a plain class that e.g. keeps a list of `int`'s ordered using a hard-coded function for the < operator, 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.

Important: Remember that with current compilers, if you want to use a template class or function in a source code file, all of the code for the 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 function, 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. Do not submit an Ordered_list.cpp file!

Dummy nodes are a bad idea. As mentioned in Project 1: 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 do not exist in the domain. For example, if you use a dummy node, even if your Room list is empty, there is still a Room object in existence! And by the way, what is its number? 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. Remember that member functions have access to private members of any object in the same class, not just in "this" object. So 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 code will be more convoluted than going directly to the member variables of the list and its nodes.

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 very efficient.