Project 2
The Tiny Meeting Manager in C++
Classes for the Domain, DIY Strings and a Container
Also Starring: C++11 Move Semantics!
Due: Oct. 12, 2012, 11:59 PM

Notice:

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

Introduction

Purpose

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

- Basic C++ console and file I/O.
- Using classes for abstraction and encapsulation of user-defined types that represent concrete types.
- Overloading operators for user-defined types.
- Developing classes that behave like built-in types and properly manage dynamically allocated memory.
- Using some simple static member variables.
- Defining and using simple function object classes.
- Writing and using templates, including class templates with default template parameters that use templated function object classes, and STL algorithm-like function templates.
- 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.
- An introduction to the basics of the new C++11 move semantics, which produces faster performance with classes that manage resources, with no change to client code!

Problem Domain

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

Overview of this Document

There are three sections: The *Program Specifications* in this project are very simple, because the program behavior is basically identical to that of Project 1. The second section presents that *Class Design* in the project; study this section for an introduction to how you translate from a problem domain to the design of a class and its interface. The third section presents the *Programming Requirements* - this how you have to program the project in terms of the structure and organization of the code, and which specific techniques you must apply - remember the goal is to learn and apply some concepts and techniques for programming, not just hack together some code that behaves right. At the end is a section of advice on *How to Build this Project Easily*.

Program Specifications

Unless otherwise specified, the behavior of Project 2 is the same as Project 1 (as amended by the posted Corrections and Clarifications). The 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. 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).

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 computer-science things that programmers like to have for coding the program, versus the things in the world of meeting management. A key idea in object-oriented programming is that the structure of the code corresponds to the structure of the domain, and this structure is determined by the responsibilities of each class, and how each class collaborates or relates to the other classes. This is primarily an issue with the domain classes; the support classes are relatively simple in this regard. 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.

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 C++11 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 bad in the longer term, and if so, it should be discarded.

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 their conceptual uniqueness is less important that being able to copy or move them - especially to facilitate rescheduling 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 an list ordered by time, and it is possible to access the meeting in that list, it is then also possible to disorder the list by changing the time of the Meeting. The project as specified has no easy way to prevent this programming error. Instead, the programmer has to follow the conventional method for safely changing the *key* value inside an object being held in an ordered container: make a copy of the object, remove it from the container and discard it, change the key in the copy, and then insert the copy into the container. 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. Either way, this copy-remove-change-reinsert process is common when ordered containers are involved.

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. 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 are also 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 matching 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.

The second problem 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. 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.

Programming Requirements

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

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

- All specified public member functions and associated non-member functions must be implemented and behave as specified; you may not change their arguments or behavior.
- No specified functions can be removed.
- No public member functions can be added.
- No friend functions can be added.
- No public member variables can be added.
- 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" string to get the specified file names.

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

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

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

The default-constructed or empty `String`. It looks like this: the pointer points to a one-byte piece of allocated memory containing only a null byte (the empty C-string ""), the allocation is 1 byte, and the length is zero. This is inefficient, because dynamically allocating a single byte of memory to hold a empty C-string is a ridiculous waste of CPU cycles. The advantage for this project is that it will be simpler for you to implement; there are no special cases: a `String` *always* has a pointer to allocated memory and corresponding allocation and length sizes. Being more efficient in special cases is worthwhile (good library implementations use things like the "Small String Optimization"), but trying to do this would make writing the `String` code more complicated than necessary for the purposes of this project.

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 `Strings` to each other. Comparisons are implemented more easily by taking advantage of how the compiler will use the `String` constructor to automatically create a temporary `String` object from a C-string (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 string class, `String` has the useful feature that the internal memory storing the characters automatically expands as needed; this is especially handy when reading input into the string; you don't have to worry about a crazed user overflowing a fixed-size array! High-quality Standard Library implementations of `std::string` use a variety of tricks to improve performance, but here you will implement a simple form that illustrates the basic ideas. The `String` demo program on the web pages has examples showing this automatic expansion, and read the skeleton header comments carefully for the detailed specifications.

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

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

- It must clear the supplied `String` to an empty 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.
- 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 deallocate space in the left-hand side object and then give it a separate copy of the data in the right-hand side object. It must also be safe against aliasing (self-assignment). You must use the copy-swap logic described in lecture (see the posted notes) to achieve code re-use and exception safety. 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` (see above - e.g. a pointer to one byte of allocated memory containing a null byte).

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

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

Notice: Part of 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 we can test it by itself, your `String` implementation should depend only on your `Utility.h` to get access to the `swapem` function template (see below).

Ordered_list.h. This header file contains a class template for a list container called `Ordered_list`, which is based on the Project 1 Ordered list implementation, but it is simpler and cleaner. You refer to items in the list with `Iterator` objects, similar to the STL containers. Because `Ordered_list` has a proper copy constructor and assignment operator, you can pass `Ordered_lists` in function calls by value, and get one returned by value. In all respects, your `Ordered_list` is very capable and very like a Standard Library container. You can learn a lot by building it. But because all of the 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.

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 the only member 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;
```

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

Easy setup thanks to templated function object classes. Ordered_list.h contains two templated function object classes that are especially handy for specifying the most common type of ordering relation, namely one that uses the less-than operator (`operator<`) defined for a type. We will define `operator<` for 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.

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

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

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";
```

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 probe 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!

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

Note that each node contains a member variable whose type is the type of object in the list, unlike Project 1's list which held a `void *` pointer to an external data object. Thus actual objects of any type can be stored in a member variable of the list node instead of just storing pointers. Since the `Node` destructor will automatically destruct all of the `Node` member variables, if you have an `Ordered_list` of objects that have fully functional destructors, 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 memory deallocated - maybe you have some other pointer to the data because you want to keep it around!

Using Ordered_list. Your program must use this `Ordered_list` container for all of the linked-lists in the program. 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`, 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 `Iterator` through the list in Standard Library style, from `begin()` to `end()`, and dereference the `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`. You should do so to get used to doing things in the Standard Library way.

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

Copy and move implementation. So that your `Ordered_list` is as usable and efficient as Standard Library containers, you need to implement both copy construction and assignment, and move construction and assignment. The copy constructor simply builds a list containing *new* nodes that have the same data member value as in the original. The copy assignment operator simply does a copy/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.

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 - you can add additional private members as you wish; the ones shown are for a one-way linked list, but you can add additional ones to make it a two-way list if you choose. But you may not modify the public interface.

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

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 be pointers-to-const, as in

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

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. If a value for a member variable is read from the file, it should be read directly into the member variable (if the variable is a `String`, this will affect its total allocation). 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 using exception processing makes it easy to clean up messes when an error occurs.

Similar to the `String` class, we are keeping a count in a static member of how many `Meeting` objects currently exist, and so the `Meeting` constructors and destructor have to increment and decrement this count. In contrast, the `Room` and `Person` classes do not; the main module can easily obtain the number of these objects from its lists. However, consider carefully what functions you actually need to define in the `Room` class - it is very poor practice to try to define the very same functions that the compiler will supply automatically.

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

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

The error message strings are posted 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.

To provide a bit of practice with function templates, this module must also define a function template named `swapem` that will interchange the value of two variables of any type. This will be useful in your `String` and `Ordered_list` code. For example:

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

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

As in Project 1, the Utility module is reserved for functions or classes that are used by more than module (like the Error class) - do not put anything in here that is used by only one module. 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 `Meetings` in it. For this reason, the program no longer outputs separate messages when a **da** (delete all) command is executed: clearing the rooms automatically clears all the meetings, so a single message is produced (see the sample output and strings.txt).

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. Single character input, as for the commands, should be done into single-character `char` variables.

Top Level and Error Handling

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

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!

You may find it useful for your **ld** command code to catch and rethrow `Error` exceptions (with simply `throw;`), so that you can clean up any bogus entries in the room and person lists resulting from processing a file that turns out to be invalid.

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:

1. `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 does not have to check for the participant being in the `Meeting` already.
2. `Room::add_meeting()` throws an exception if a meeting is already scheduled at the time. Therefore, the main module code for the **am** command should collect all the information required for the `Meeting` - both time and topic - and then simply add it to the `Room` - it should not check the time for duplication before reading the topic. A similar process should be done for the final step on rescheduling a `Meeting` - the last check on the new time should be done by trying to add the `Meeting` to its new `Room`.

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

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

Other Programming Requirements and Restrictions

1. The program must be coded only in Standard C++, and follow C++ idioms such as using the `bool` type instead of an `int` for true/false values, and avoiding `#define` for constants. See the C++ Coding Standards document for specific guidelines.
2. Only C++ style I/O is allowed - you may not use any C I/O functions in this project. This means you can and should be using `<iostream>` and `<fstream>`. See the web handouts on C++ Stream and File I/O for information on dealing with stream read failures.
3. You may use only `new` and `delete` - `malloc` and `free` are not allowed.
4. Before your program terminates, all dynamically allocated memory must be deallocated - in this project, correctly working destructor functions will do this automatically in many, but not all, cases. Notice that like the Standard Library containers, the destructor for a `Ordered_list` of pointers does not do a `delete` on its contained pointer values - it does not presume to manage memory on your behalf.

5. There must be no memory leaks apparent in your code.
6. You must not use any declared or allocated built-in arrays for anything anywhere in the program. The single exception is that your `String` class must use `new` to dynamically allocate a `char` array to store the internal C-string.
7. You may not use the Standard Library `<string>` class or any of the Standard Library (or "STL") containers, algorithms, or iterators in this project. The idea is to learn how these things are done by writing some simplified versions yourself. Subsequent projects will allow (actually require) full and appropriate use of the Standard Library. You can use `<cctype>` functions such as `isspace` wherever they are appropriate.
8. Your String.cpp file can use any functions in the C++ Standard Library for dealing with characters and C-strings, including any of those in `<cctype>`, or `<cstring>`. See a Standard Library reference, or the online brief reference pages in the Handout section of the course web site. Any non-standard functions must be written by yourself. *Hint*: The `String` class can be coded easily using only basic functions from the C part of the C++ Standard Library like `isspace`, `strcmp`, `strcpy`, `strlen`, and `strncpy`; seek help if you think you need some of the more exotic library functions. Note that all other modules in the project will work only with individual `char` variables (e.g. for the commands) or `String` objects.
9. `String::operator+=` must not create any temporary `String` objects or unnecessarily allocate memory. See above for more discussion. The skeleton String.h file comments specify which operators and functions create temporary `String` objects.
10. You can and should `#include <cassert>` and use the `assert` macro where appropriate to help detect programming errors.
11. You must follow the recommendations presented in this course for using the std namespace and minimizing header file dependencies.
12. You may not create any global variables, even if they are scoped in a single file. You may certainly use file-global constants in a module's .cpp file for things like representing output string text, and when appropriate in Utility.h. Note that in C++, file-scope `const` variables automatically get *internal linkage* by default so you don't have to declare them static or in the unnamed namespace in a header file.
13. The output messages produced by your program must match exactly with those supplied in the sample output and the supplied text strings on the project web page. Copy-paste the text strings into your program to avoid typing mistakes, and carefully compare the output messages and their sequence with the supplied output sample to be sure your program produces output that can match our version of the program.

Project Grading

I will announce when the autograder is ready to accept project submissions. See the instructions on the course web site for how to submit your project.

Your project will be computer-graded for correct behavior, and component tests will test your classes separately and mixed with ours, (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. For example, one spot-check would be whether your code uses "copy-swap" as described in this course in the specified places. Another would be whether it uses "this" unnecessarily.

How to Build this Project Easily

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

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

You can base the `Ordered_list` and `String` class code on any previous code authored by you (review the academic integrity rules). 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 constructor, destructor, copy constructor, and copy assignment operator, and then the += concatenation operator. Save `operator+` and `operator>>` for last. As you work, take care to consider how you can use the functions you have already implemented when you write each function. Look for opportunities to use private helper functions to simplify coding and debugging - these make a huge difference! For example, a private helper function to "set up" a `String` given a `char *` and a length is really handy!

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

s1 += s2;

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

s1 = s1 + s2;

will run slowly because the rhs involves creating a temporary object copied from `s1` and then concatenating `s2` into it, and then copying the contents of the temporary object into the lhs, involving another bout of memory allocation/copy/deallocation. This is why we have a C++ idiom: prefer += over (= and +) if it does what you want. If a move assignment operator is available, the last step will be faster, but using += is still overall faster than +.

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

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

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

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

Note: Compilers *love* to "elide" (eliminate) many calls to copy or move constructors - a traditional and important optimization. Unfortunately the result can be confusing - it doesn't look like what is supposed to happen. For example, the copy or copy construction of a String being returned from a function will often get optimized away - the "Returned Value Optimization (RVO)." You can see the "by the book" activity if you can turn off these optimizations. If you are using g++, use the option `-fno-elide-constructors` to cause all copy or move constructors to be left in the generated code. Xcode 4.4.1 LLVM Clang has this option, but it is buggy, so don't use it. MSVS 2012 doesn't appear to have this sort of option. Because different compilers differ in how aggressively they elide constructors, you will see different results when you turn on the messages. The course is using gcc/g++ 4.7.0 as the "standard" so upload your code and build on CAEN with this option to see this stuff happening.

Building the Ordered_list template. Templates can be a pain to debug because most compilers and debuggers generate really verbose and confusing messages about templated code. First code `Ordered_list` as a plain class that e.g. keeps a list of int's 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 in a source code file, all of the code for a class template must be seen by the compiler. The customary and common way to do this is to *put all of the template code in the header file - there is no .cpp file for a template class or template functions, only the header file. If you were told to do this 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.

Sentinel 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 aren't really there in the domain. For example, if you use a sentinel 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.