Project 1
Pointers, Functions, & Modules: A Micro Meeting Manager in C

Due: Friday, Sept. 28, 2012, 11:59 PM

Notice:

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

Introduction

Purpose

The purpose of this project is to provide review or first experience with the following:

- Quality procedural program organization with a well-designed function hierarchy for clarity, maximum reuse, and ease of coding and debugging.
- Program modularization using separate compilation and header files that declare module interfaces, and writing modules that interacts with other modules only through their interfaces.
- Using opaque types to hide implementation details from client code, and demonstrating this with two implementations of the same interface.
- Writing code in C to implement and use a generic data structure or container, requiring use of void * pointers and function pointers, to point to dynamically allocated objects, including using multiple containers that point to the same objects.
- Using C techniques for "object-based" programming with good encapsulation and insulation. These techniques are in common use in well-organized modern C software and provide some of the advantages of object-oriented programming with this non-object-oriented language.
- Internal and external linkage for global variables and file-local functions.
- Practice with implementing a node-based and array-based container.
- C techniques for console and file I/O, string processing, and dynamic memory management.

Problem Domain

This project is to write a program that uses linked lists to keep track of a meeting schedule and which people are in what meetings in which rooms. The program does not have to figure out the schedule; rather it is a highly simplified application for maintaining the schedule. It is so simple that all the meetings are on the same day! The rooms are kept in order by room number, the meetings are kept in order of time, and the people are kept in alphabetical order of last name.

A meeting room is specified by its number (any positive integer value) and the meetings being held in it. A meeting is specified by an integer value for the meeting time falling in the traditional 9-5 business hours range, a one-word topic, and a list of participants (people). A person is specified by a one-word first name, a one-word last name, and a phone number (also represented as a character string). Rooms are designated by room number, which must be unique. Meetings are designated by the time, which must be unique in the room. People are designated by last name, which must be unique.

Overview of this Document

There are two major sections: The *Program Specifications* describe what the program is supposed to do, and how it behaves. The *Programming Requirements* describe 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 and Well* - worth taking seriously!

Program Specifications

The basic behavior and functionality of the program is specified in this section. As you read these specifications, study the posted "normal_console" sample of the program's behavior.

1. The user can specify the numbers of the rooms available for meetings. These are kept in a list of rooms ordered by room number - the room list.
2. The user can specify the names and phone numbers of individual people who can be participants in the meetings. These are kept in a list of individuals ordered by alphabetical order last name - the people list. **Note**: Throughout this course, "alphabetical

order" for a string means the standard English language order defined by the C and C++ Library functions for ordering character strings (`strcmp`, `std::string::operator<`) . In this order, lower case 'a' follows upper case 'Z'.

3. The user can specify the room number, time, and topic of a meeting; the meetings are kept in a list for that room, ordered by the meeting time - the meeting list for each room. In separate transactions, the user can specify which people are participants in a meeting; the participants are kept in a list associated with the meeting, the participant list for each meeting. The participant list is ordered by the last name.
4. Rooms are referred to by specifying their number; meetings are referred to by specifying their room number and time, and people are referred to by their last names.
5. The program will not allow two meetings to be scheduled for the same time in the same room, or two people with the same last name to be put into either the people list or a participant list. At this time, there is no check for whether a person is scheduled to be in two different meetings at the same time. This will be corrected in a future version of this program.
6. Upon request, the program will print out the complete information for an individual person, or the whole list of persons. The complete information consists of the first and last name and the telephone number. The output of the list will be in alphabetical order of last name.
7. Upon request, the program will print out the complete meeting information for a specified room, a specified meeting, or the whole schedule of meetings. The complete meeting information for a meeting consists of the meeting time, the meeting topic, and the complete information for each participant. The output of the meeting list for a room will be in order of meeting time. The output for the whole schedule will be each room, in order by room number, with the meeting output for each room. The output of the participant lists will be in alphabetical order of last name. Consult the posted samples for examples of how the output will look.
8. The program will print an error message and request the next command if a specified room, meeting, or person is not found in the corresponding list, or other problems are detected.
9. Rooms can be added or deleted.
10. Meetings can be added, deleted, or rescheduled for a different room and/or time.
11. All of the meetings can be deleted at once.
12. Participants can be added to or removed from a meeting.
13. Individual persons can be added to the list of people, or removed if the person is not a participant in a meeting. That is, before a person can be removed from the people list, they must first be removed as participants in any meetings.
14. All of the people can deleted at once, but only if there are no scheduled meetings (regardless of whether the scheduled meetings have any participants).
15. At any time, the program can save the people, room, and meeting information in a file.
16. At any time, the people, room, and meeting information previously saved in a file can be restored, replacing any current information.
17. The program is controlled by a simple command language, specified below.

The Command Language

The program prompts for a two-letter command. The first letter specifies an action, the second letter the kind of object (room, meeting, participant, individual person) to be acted upon. The command letters are followed by additional input parameters depending on the command. The program executes the command, and then prompts for the next command. See "Input rules" below for specifics about how commands and parameters must be formatted in the input, and how they are checked for validity. See the sample outputs for examples.

The action letters are:

p - print

a - add

r - reschedule (meetings only)

d - delete

s - save

l - (lower-case L) load

The object letters are:

i - individual person

r - room

m - meeting

p - participant

s - schedule (all meetings - delete and print commands only)

g - group (all people - delete and print commands only)

a - all for the delete command, allocations in the print command (memory information, described below);

d - data (all people, rooms, and meetings - save and load commands only)

The possible parameters and their characteristics are:

<room> - the room number, which must be a positive (> 0) integer. Whenever a room number is entered, the program always checks that the number is in this range before the room list is examined for a matching number.

<time> - a meeting time, which conceptually is the starting time for a one-hour meeting. It must be one of the values {9, 10, 11, 12, 1, 2, 3, 4, 5} which is also the order to be used in sorting by time order. Thus the last meeting can start at 5:00 PM, which means it runs after the traditional business hours. Whenever a time is entered, the program always checks that the time is one of these values before examining any of the meeting lists for a matching time.

<lastname> - a person's last name, any characters can be present in the name, but no embedded whitespace characters are permitted, and a maximum of 63 characters can be entered from the keyboard. The name is case-sensitive, meaning that McDonald and Mcdonald are two different names.

<firstname> - a person's first name, with same restrictions as <lastname>.

<phone number> - a person's phone number, treated as a character string, and so has the same restrictions as <lastname>.

<topic> - a meeting topic, a character string with the same restrictions as <lastname>.

<filename> - a file name for which the program's data is to be written to or saved from. This is a character string with the same restrictions as <lastname>.

The possible commands, their parameters, meanings, and error possibilities are as follows:

pi <lastname>- print the specified individual person information. Errors: no person with that last name.

pr <room> - print the meetings in a room with the specified number. Errors: room number out of range; no room of that number.

pm <room> <time> - print the time, topic, and participants (full name and phone number) for a specified meeting. Errors: room number out of range; no room of that number; time out of range, no meeting at that time.

ps - print the meeting information (same information as pm) for all meetings in all rooms. Errors: none. (It is not an error if there are no meetings - that is a valid possibility.)

pg - print the individual information (same information as pi) for all people in the person list. Errors: none. (It is not an error if there are no people - that is a valid possibility.)

pa - print memory allocations (described below). Errors: none.

ai <firstname> <lastname> <phone number> - add an individual person to the people list. Errors: A person with that last name is already in the people list.

ar <room> - add a room with the specified number. Errors: room number out of range; room of that number already exits.

am <room> <time> <topic> - add a meeting in a specified room, at a specified time, and on a specified topic. Errors: room number out of range; no room of that number; time out of range; a meeting at that time already exists in that room.

ap <room> <time> <lastname> - add a specified person as a participant in a specified meeting. Errors: room number out of range; no room of that number; time out of range, no meeting at that time, no person in the people list of that name; there is already a participant of that name.

rm <old room> <old time> <new room> <new time> - reschedule a meeting by changing its room and/or time (without changing or reentering topic or participants). Each parameter is read and its value checked before going on to the next parameter. Actually changing the schedule is not done until all parameters have been read and checked. Errors: old room number out of range; old room does not exist; old time is out of range; no meeting at that time in the old room; new room number out of range, new room does not exist; a meeting at the new time already exists in the new room. To keep the logic simpler, the last error will result if the user attempts to reschedule a meeting to be in the same room and at the same time as it is currently.

di <lastname> - delete a person from the people list, but only if he or she is not a participant in a meeting. Errors: No person of that name; person is a participant in a meeting.

dr <room> - delete the room with the specified number, including all of the meetings scheduled in that room - conceptually, unless the meetings have been rescheduled into another room, taking the room out of the list of meeting rooms means that its meetings are all canceled. Errors: room number out of range; no room of that number.

dm <room> <time> - delete a meeting. Errors: room number out of range; no room of that number; time out of range; no meeting at that time.

dp <room> <time> <lastname> - delete a specified person from the participant list for a specified meeting. Errors: room number out of range; no room of that number; time out of range, no meeting at that time, no person of that name in the people list; no person of that name in the participant list.

ds - delete schedule - delete all meetings from all rooms. Errors: none.

dg - delete all of the individual information, but only if there are no meetings scheduled. Logically this is overkill; it would suffice if there are no participants in any meetings, but this specification is made for simplicity. Errors: There are scheduled meetings.

da - delete all - deletes all of the rooms and their meetings (as in dr) and then deletes all individuals in the people list. Errors:none.

sd <filename> - save data - writes the people, rooms, and meetings data to the named file. Errors: the file cannot be opened for output.

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 successful, it deletes all current data, and then attempts to read the people, rooms, and meetings data from the named file, which should restore the program state to be identical to the time the data was saved. If an error is detected during reading the file, the error is reported and any data previously read is discarded, leaving all the lists empty.

qq - delete everything (as in da), and also delete the rooms and people lists, so that all memory is deallocated, and then terminate. Errors: none.

Command Input Rules

The input is to be read using simple input stream operations. You must not attempt to read the entire line and then parse it; such error-prone work is both unnecessary and non-idiomatic when the stream input facilities will do the work for you. Check the samples on the web site to see the general way the commands work, and the type-ahead example to see the consequences of using this simple approach. The input is to be processed in a very simple way that has some counter-intuitive features, but doing it this way, and seeing how it works, will help you understand how input streams work both in C and C++, and several other languages. If you use scanf properly, the required code is extremely simple - scanf does almost all the work for you! Here are the rules; follow them carefully:

- With the following exceptions, your program should read, check, and process each input data item *one at a time*, before the next data item is read. This determines the order in which error messages might appear, and what is left in the input stream after error recovery is done. The exceptions:
  - Read both command letters before checking or branching on either of them. There is only one "Unrecognized command" error message, and it applies if either one or both of the letters are invalid.
  - Read all three data items for a new individual in the **ai** command before testing the last name for validity (duplicate last name).
- The room number and time are supposed to be integer values. Your program should try to read these data items as an integer (i.e. with `%d`). If it fails to read an integer successfully (e.g. the user typed in "xqz" or "a12"), it outputs the error message "Could not read an integer value!". If the user had written a non-integer number such as 10.234, your program must not try to deal with it - just read for an integer, and take what you get - in this case, the value 10 (an integer). The remainder of the input is garbage, ".234" in this case, that is left in the stream to be dealt with on the next read. If this puzzles you, re-read how the `scanf` function works when reading an integer from an input stream. If you are still puzzled, ask for help. The behavior of streams is important to understand; previous courses might have hidden it from you. Once you have successfully read an integer, test it for being in range, then test for validity against the schedule data (e.g. duplicate room number).
- More than one command can appear in a line of input, and the information for a single command can be spread over multiple lines. In other words, type-ahead is permitted, and the program ignores excess whitespace in input commands.
- The program does not care how much whitespace (if any) there is before, between, or after the command letters.
- Whitespace is required in the input only when it is necessary to separate two data items that `scanf` could not tell apart otherwise, such as the room number and time, or the first and last names in a new person entry. When whitespace is not required in the input, it is completely optional. See the posted typeahead sample.
- There should be no punctuation (e.g. commas) between items of input, but your program should not attempt to check for it specifically: the normal error checking will suffice - e.g. ",3" cannot be read as an integer, and "pi,Jones" will be read as a request to print the entry for somebody whose last name starts with a comma.
- Do not attempt to check the person's names and phone number for sensible content - they should be simply treated as whitespace-delimited strings. E.g. a phone number of "Beowulf" is acceptable.

- The input is case-sensitive; "jones" and "Jones" are two different names. Likewise, "AI" is not a valid command.
- Use the `%ns` format specifier to ensure that input of strings with `scanf` and `fscanf` will not overflow the buffer. That is, if the input lastname is longer than 63 characters, it must not be allowed to overflow an input buffer whose size is 64; using `%63s` for the scanf format will ensure it. Avoid maintenance problems presented by "magic strings" for these formats by using one of the techniques discussed in lecture.
- If an input string typed by the user is too long to fit into its buffer (e.g. the user types more than 63 characters for the lastname), the program simply accepts as many characters as will fit into the buffer, and leaves the rest in the stream to be read as the next parameter or as the next command. In no case must the program overflow the input buffers. The result might be messed-up names, or various error messages, but this is approach is safe, simple, and instructive about how stream input works.

Error Handling. Your program is required to detect the errors listed above and shown in the sample outputs. When an error is detected, a message is output, and then the characters in the input stream up to and through the next newline are read and discarded. Thus the input on the rest of the input line is skipped, but since type-ahead and excess whitespace (which includes newlines) are both permitted, the exact results of skipping the rest of the line depend on exactly what the input is. *Do not skip the rest of the line any other time*.

The course web site has samples illustrating the error messages, and a file containing the strings to be used for the error messages and all other messages - copy-paste these into your code to avoid typing errors.

Save File Format and Load Input Rules

- The file created by the **sd** command has a simple format that can be read by a human easily enough to simplify debugging the saving and loading code. See the posted samples. It consists of a series of lines containing information as follows:

  The number of people in the people list on the first line.

  The firstname, lastname, and phonenumber for each person in the people list, one line per person, in alphabetical order by last name.

  The number of rooms in the room list on the next line.

  The room number and the number of meetings in the room for the first room (in order by room number)

  The time, topic, and number of participants in the first meeting (in order by time)

  The lastnames of each participant, one per line, in alphabetical order.

  Similarly, the next meeting appears on the next lines.

  Similarly, the next room appears on the next lines.
- General rules for the file format: The file is a *plain ASCII text file*, thus all numeric values are written as the ASCII character representation for decimal integers (e.g. with `fprintf` using `%d`). Each item on a line is separated by a single space from the previous item on a line. The last item on a line is followed only by a newline (`\n` with no previous spaces). Each line (including the last in the file) is terminated with a newline.
- When the file is read by the **ld** command, the program can assume that the following two situations are the only ones possible:

  1. *The user provided the name of a file that was not written by the program*. In this case, the file contains some other unknown data. In this case, without any special checking, the program will eventually detect the problem because the random information is extremely unlikely to be consistent in the way the program expects when it reads the file. Thus the program can simply check that it can read a numeric value or string when one is expected, and that the right number of items of information are in the file. For example, the first thing in the file is supposed to be the number of people in the people list. If the program cannot successfully read an integer as the first datum, then it knows something is wrong. But suppose the program could read a number as the first item, but suppose that number is large, and then the program hits the end of file before it finished reading the expected data for this many people. Again, the program knows something is wrong. Finally, suppose the program manages to get to the Rooms and Meeting section of the file, but finds random garbage in what are supposed to be Person lastnames that match those found in the Person section of the file. A failure to find a participant lastname in the newly restored set of Persons is a signal that something is wrong; this is easy to check for. Because of the considerable constraints present in the file format, it is reasonable to rely just on this level of checking to detect that an invalid file was specified. Thus, your program should check only for the following specific input errors when reading the file:
     - An expected numeric value could not be read because a non-digit character is present at the beginning of the expected number -i.e. `scanf` with `%d` fails.
     - An expected meeting participant name could not be found in the person list.
     - Premature end-of-file because the program is trying to read data that is not present - i.e. `scanf` returns `EOF` unexpectedly.

2. *The file was correctly written by a correct program in response to an earlier **sd** command*. This means that the file will have no inconsistent or incorrect information (for example, no people with the same last name). Thus the program does not check the validity of the data any further than in Case #1 above if it can successfully read the file. Notice that although the information is divided up into lines for easier human readability, your program does not have to read the input in whole lines at all, and should not try to - it would be just an unnecessary complication.

- **Important:** the restoration code must protect against input buffer overflow; Case #2 does not justify ignoring the possibility of buffer overflow if an invalid file is being read.
- If the program detects invalid data, it prints an error message (which is the same regardless of the cause) and then it deletes all current data that it might have loaded before the problem became apparent, and prompts for a new command. Thus the possible results of issuing an **ld** command are: (1) no 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, producing an error message and completely empty people and room lists, with no meetings.
- **Hint**: If your program appears to have trouble with **ld** in the autograder, make sure you have implemented **sd** correctly, and then correctly implemented the specified checks in #1 above. Do not add additional checks to your **ld** command code - they are unnecessary and will just waste your time.

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 primarily to get you to learn and practice certain concepts and techniques, and secondarily, to make the projects uniform enough for meaningful and reliable grading.

So, 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, 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 and any general requirements. For example, this project must be done in ANSI C (1989), and so you may not use C++ I/O operators in it.

Program Modules

The project consists of the following modules, described below in terms of the involved header and source files. You must submit *exactly* this set of files to the autograder - no more, no fewer, and with these names and specified contents. All header files must have include guards to protect against multiple inclusion. Some files will be for you to write in their entirety, but the project web page will contain some of the files in either the exact form that you are to use them, such as the Ordered_container.h file, or as *skeleton* files, such as for Person.c. A skeleton file has some code in it which your project is required to use and conform to, such as a particular struct type declaration, but the rest of the code is for you to write.

Ordered_container.h, Ordered_container_list.c, Ordered_container_array.c. Your program must use a generic ordered container module for all of the containers in the program, and you must supply two different implementations of this container, one using a linked-list (Ordered_container_list.c) the other using a dynamic array (Ordered_container_array.c). The course web site and server contains a header file, Ordered_container.h, which you must use and `#include` in your two implementation files. Ordered_container.h contains the function prototypes for the external interface of this module. You may not modify the contents of this header file - your .c files must supply all of the specified functions and behavior, and you may not add any others to the header file. All other functions and declarations in your implementation files must have only internal linkage.

Note that it is more common for header and implementation files to have the same name, differing only in the "extension" of ".h" or ".c". So commonly, there would be a "Ordered_container_list.h" file and an "Ordered_container_array.h" header file, and the rest of the program would #include the header file corresponding to the desired implementation. But in this project, there is only the single header file, and the choice of implementation is done at link time. (See the supplied makefile.) This is to give you practice with idea of distinguishing interface from implementation, and seeing how the link step really does determine which modules make up the program.

As will become clear in the discussion below, the Ordered_container_list and Ordered_container_array implementations should produce *identical* behavior of the program, with two exceptions: the run time characteristics will be different (which would be hard to perceive unless large amounts of data were involved), and there will be a difference in memory usage, which will appear in the output of the **pa** command, described later.

These generic containers are *opaque types*, meaning that the client code (code that uses the module) can't "see" into the container - it does not have, and does not need, any information whatsoever about how the container works or what is inside it. Conceptually, regardless of implementation, the Ordered_container contains a series of items. An item contains a data pointer to a data object created by the client code (the calling code). The items are always in the order specified by an ordering function supplied when the container is created. The ordering function examines the contents of the data objects to decide which item comes before or after another item, or if the two items should be considered as equal. Items can be added to the container only through a function that inserts them in order. An item can be searched for in the container; if found, it is designated with a item pointer that points to the item in the container. An item can be removed from the container by supplying the item pointer for it. The data object can be retrieved for an item by supplying the item pointer to a `get_data_ptr` function that returns the data pointer for the data object. The only way to process all of the items in the container is to write a function that processes a data pointer, and then give a pointer to this function to one of the Ordered_container "apply" functions; the "apply" function iterates through the items in the container, in order from first to last, and calls your function with the data pointer from the item.

Here is how Ordered_container is a *generic* container: Since the data pointers stored in the items are void * pointers, they can point to any kind of data object. Your code (the client code) is responsible for creating the data objects - of whatever type they are - and putting void pointers to them into the container. When you get a data pointer out of the container, you have to cast the void * to the correct type in order to access the data in the data object.

Here is how Ordered_container is an *opaque type* whose implementation is completely independent of its interface: Notice that Ordered_container.h includes only an *incomplete* type declaration for the structure type `struct Ordered_container`. When a container is created, the structure is dynamically allocated in the "create" function and the pointer to it is returned. The functions that operate on the container require a pointer of this type that designates which container is being operated on. Also, item pointers are simply void * pointers. All of the interface functions work in terms of pointers of these types. Because the container type is incomplete, and items are designated with void *, and the client of the Ordered_container only includes the header file, the client does not have access to the internal members of the container, and has no way of "knowing" anything about the layout or organization of the container or the items in it, and thus can and must work with the container completely in terms of the supplied interface functions. Thus the container is opaque - the client code can not "see inside" the container or its items. To demonstrate how this technique separates interface from implementation, you have to supply the two different implementations of the Ordered_container interface.

Ordered_container_list. In the linked-list implementation, the container is a linked-list and the items are list nodes that contain a void * data pointer. The nodes are designated to the client by casting node pointers to/from void *. The list is searched linearly to locate items, and to locate where to insert a new item. This means that searching and inserting is linear with the number of items (size) of the container (*O(n)*). However, if the desired item is not present, the search should stop as soon as it has "gone past" where the item should have been, so that on the average, only half of the list must be examined if the item is not present, which means the linear search is faster in this case than it would be otherwise; this applies to both find functions. The `OC_get_size` function returns the number of items in the container, which is simply the number of nodes in the list. The `OC_clear` function destroys each node and re-initializes the list to be empty.

Implementation note: One approach to implementing a linked-list involves creating dummy nodes to simplify the handling of empty lists. This approach is not recommended for this project for two reasons: (1) it involves ugly special-case counting of the number of nodes in empty lists; (2) more seriously, it will lead to the creation of bogus objects when you convert the code to a C++ templated container class in the next project.

Ordered_container_array. In the dynamic array implementation, the container is a dynamically allocated array in which each cell contains a void *. The cells of the array are designated to the client code by casting the address of a cell to/from a void *. The array is searched using binary search to locate items (this applies to both find functions), so that searching will be logarithmic with container size (*O(log n)*). Insertion should use binary search to locate where to insert a new item, but the next step requires moving the items that come after the new item "down" by one to make room for the new item. This means that the insertion time is overall linear, even if binary search is used to speed up the first step.

Implementation note: The C Standard Library includes a binary search function, `bsearch`, which is great, but it will not tell you where to insert an item if it is not present. Fortunately, there is a simple modification to the binary search algorithm that will give you both whether or not the item is present, and if it is not, where it should be put. This is a good justification for writing your own, more general-purpose, binary search function. For example, refer to the example binary search function in K&R, p. 58 - if you drop out of the loop, the item was not found, but the place where it should be inserted would be `high+1`.

Growing the array. The array is dynamically allocated when the container is created, and initially contains a small number of cells. As items are added, the array is reallocated as needed: if there is no room for the new item, a bigger array is allocated, and the data pointers copied over, and the old array is deallocated. For this project, the initial size of the array is 3 cells. When the array is full, a new array is allocated whose size is double the space required to hold the original array plus the new value (new size = 2 * (old size + 1)). This scheme can waste some memory space, but results in fairly fast performance because as the container is filled, fewer new allocation/copy/deallocate operations are required. But the array normally will have cells that are not currently in use; so this implementation has to keep track of how many cells are *in use* to hold items, which is returned by the `OC_get_size` function, and how many cells are *currently allocated* - the size of the current allocated array. To remove an item, the items that come after the removed one are moved "up" by one, but the array is not reallocated - it retains its original size. The only time the array is "shrunk" is with the `OC_clear` function, which discards the entire array and starts over with the initial small allocation.

General implementation issues. In both implementations, the container is always in sort order because items are immediately placed in the correct position; it is never necessary to sort the whole container. The ordering function supplied to `OC_find_item_arg` must imply an ordering that is consistent with the ordering function supplied when the container was created, and which is the ordering used by `OC_find_item`. The effects of using a function in `OC_find_item_arg` that involves an ordering that is inconsistent with the original ordering is not defined.

Your implementation `.c` files must each contain a complete declaration for `struct Ordered_container`. The list nodes and array cells must contain a pointer of type void * - the data pointer. Thus each node in the generic linked-list, or cell in the generic dynamic array, points to a data item; by using a void pointer, the node can point to a data item of any type. The user must use the function `OC_get_data_ptr` to access the data pointed to by a node or cell, and must cast the returned void* to the proper type in order to use the pointer to access the actual data.

If a `void *` pointer designating an item is found, and then either the same or a different item is added or removed from the container, then the pointer to the found item is not guaranteed to be valid and can't be used to locate the data object. This is easy to see in the case of the dynamic array implementation, but because the container interface is supposed to work the same way in both implementations, this restriction must be obeyed in using both implementations: *Don't use an item pointer after the contents of the container have been changed; you must find the item again*.

To build an executable using one of the implementations of Ordered_container, build the executable with the .c file for the implementation that you want. To use the other implementation, simply use the other .c file in the build. In an IDE, just add the desired .c file to the project, and remove the other. A makefile can be easily constructed to allow you to choose on the command line which implementation to use - see the starter makefile for the project. Notice that you cannot build a single executable containing both implementations because of rampant violations of the One Definition Rule - all of the interface functions will be double-defined, not to mention the two different declarations of `struct Ordered_container`.

You are free to implement the linked list and dynamic array in any way you choose as long as it is *your own code* (review the Academic Integrity rules in the syllabus) and as long as your implementation supports the interface functions defined in the supplied Ordered_container.h and the specifications in the skeleton files for the two implementations. For example, you can make the Ordered_container_list either a one-way or two-way list.

Using Ordered_container. Study the supplied demos. In overview, the generic Ordered_container module is used as follows.

- An Ordered_container object is created by the `OC_create_container` function which returns a pointer to the opaque `struct` type. The create function is also given a function pointer to the comparison function, which is stored in the container struct.
- Data is added to the container by creating a data object (e.g. a `Person struct`) and then inserting into the container an item containing a pointer to the data object using the `OC_insert` function. The data objects themselves will typically reside in dynamically allocated memory which must be managed separately. The comparison function returns negative, zero, or positive (analogous to `strcmp`) for whether the new item should come before, is the same as, or comes after, an item already in the container. For example, this function can use `strcmp` to compare lastnames in Persons.
- To find an item in the list, the `OC_find_item` function is called with a void * pointer to a data object containing at least the data required by the comparison function. For example, finding a Person in a container can be done with a data object containing only the sought-for lastname. `OC_find_item` returns a `void *` item pointer to the item containing a data pointer to the data object that matches the supplied data (the comparison function returns zero), or NULL if such a node is not found.
- To avoid having to create a data object simply to search for a matching item, the `OC_find_item_arg` function can be given its own comparison function and an argument pointing to the data used in the function. If you have a lastname string, you can find a matching Person by supplying the lastname string as the argument, and use a comparison function that compares the supplied lastname string to the lastname in the Person.
- To remove an item from the container, find the item and then call the `OC_delete_item` function to remove the item. The program must deallocate the memory for the data object itself separately. E.g. if a Person is deleted from a container of Persons, the Person object itself must also be deleted. However, removing a Person from a Meeting does not mean that the Person is deleted from the People container! In other words, the Ordered_container module manages the memory for its items, but does not manage the memory for the pointed-to data objects. The client code is fully responsible for creating and destroying the data objects.
- A function may be applied to every item in the container using the `OC_apply` function, which simply takes a pointer to a function and calls it with the data pointer from each item in order. For example, each Person in a container of Persons can be printed using a function that interprets the `void *` data pointer as a pointer to a `Person struct` and outputs the contents appropriately. The `OC_apply_arg` function does the same thing, but includes a second `void *` argument to be passed to the called function, making it possible to call a function that for example, has a `Person` pointer as the first argument, and some other pointer type (e.g. a `FILE *`) as a second argument, for every Person in the container. The "if" versions of the apply functions can be used to scan the container for desired information; if the supplied function returns a non-zero, the iteration is halted and the value returned.

- To find out how many items are in the container, call the `OC_get_size` function; the `OC_empty` function is a convenient way to determine whether the container is empty or not and should be used instead of testing for the size being equal to zero. To empty the container (delete all items), call the `OC_clear` function which returns the container to its initial state. As with `OC_delete_item`, the client code is responsible for destroying any of the pointed-to data objects separately.

The Ordered_container interface functions can assume that all supplied pointers to containers and items are valid. This is because there is no reasonable way to guarantee that the functions can detect erroneous supplied pointer values in all cases or with acceptable efficiency (think about it!). Therefore, the client code is responsible for calling these functions with correct pointer values only. However, follow the guru advice of using the `assert` macro in the functions to help protect yourself against the most common and confusing programming errors that you might make during development - such as accidentally calling with a zero pointer argument. However, a data pointer can be null - the programmer of the client code might find it convenient to store a null pointer in an array cell or list node, so using `assert` to check for non-null data pointers should not be done.

A summary: Item pointers vs. data pointers. One purpose of this project is to get completely comfortable with pointers. Understanding how pointers are used to point to container items and data objects is a valuable step to this goal.

Both item pointers and data pointers are `void *` pointers so that they can be used to point to any type of container item or data object.

The client code "knows" what the data objects are, and casts pointers to data objects (like `Person *`) to and from the `void *` type when interacting with the container. The client code gives the container a data pointer when calling the `OC_insert` function. The container stores the data pointer in one of its internal "items". The item is *list node* in the list implementation, and an *array cell* in the array implementation.

An item pointer is a pointer to a *list node* in the list implementation, and a pointer to an individual *array cell* in the array implementation. The OC function code, when given an item pointer, will cast it to a pointer to a node in the list implementation, or a pointer to a array cell, in the array implementation. When returning an item pointer, an OC function will cast a pointer to node to `void*` in the list implementation, and a pointer to array cell to `void *` in the array implementation, and return the `void *` result as the item pointer.

Now suppose the client code gets an item pointer from `OC_find_item`. It can then provide this item pointer to `OC_delete_item`, to tell the container which item to remove from the container. In the list implementation, the item pointer points to the node to cut out of the list. In the array implementation, it points to the array cell that we want overwritten by moving the cells that follow it up by one.

Or the client code can give the item pointer to `OC_get_data_ptr`, which will look into the item and return the data pointer stored in the item. In the list implementation, `OC_get_data_ptr` simply returns the data pointer stored in the node pointed to by the item pointer. In the array implementation, `OC_get_data_ptr` simply returns the data pointer stored in the array cell pointed to by the item pointer. The client code can then cast the data pointer to the right type (say `Person *`) to then access the data object.

If you remember using the STL, the item pointer corresponds roughly to an iterator - it points to an item in the container, except you can't change it yourself (e.g. no ++), you can only get it from the `find` function. The data pointer in the item corresponds to what you get when you dereference the iterator, except you have do this with the `OC_get_data_ptr` function.

Room.h, .c, Meeting.h, .c, Person.h, .c. Rooms, meetings and people make up three more modules, and so three more header files are also provided. Like Ordered_container, these are also opaque types in that the other code does not have access to the actual structure of these data objects. (Even in procedural programming, a struct-type collection of data is often called an "object.") Thus the header files contain only an incomplete declaration of the structure types. The skeleton .c files provide the struct declaration that you are required to use, and which must only appear in your .c files.

Room.h describes the functions for creating, destroying, and printing Rooms, along with functions for accessing and modifying the data in a Room. As shown in the skeleton Room.c file, a Room has a list of meetings and a room number. Meeting.h and Person.h similarly describe the Meeting and Person modules, and the skeleton .c files provide the structure declarations. Each Meeting has a list of Persons who are participants. You must also use these header files, without modification, in your project; they specify the interface to the modules. You must use the specifications in the skeleton files and complete the implementation files named Room.c, Meeting.c, and Person.c.

Any functions in a module.c file that are not declared in the module header file must have internal linkage - they are not allowed to be accessible from other modules. The complete interface for a module, and only the interface, is in the header file; all other functions are not part of the interface, and must be inaccessible from outside the module.

The Meeting and Person objects contain `char *` pointers to C-strings for the person's name or meeting topic. Creating an item means allocating memory for the object itself plus the relevant strings and copying in their supplied values. Deleting one of these items requires deallocating the memory for the strings as well as the object struct itself. Be sure your code does this! A similar consideration applies for the list members of Room and Meeting.

Meetings can only be held in Rooms, so if a Room is destroyed, the Meetings contained in it are also destroyed. But note that the list of participants in a Meeting must contain simply pointers to Person objects that are also pointed to by the people list. Thus, 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 object.

If a Meeting is rescheduled for a different time or place, the other data about the meeting must not be destroyed and recreated. Rather the pointer to the Meeting is simply moved to a different place or to another list.

p1_globals.h, .c. Newbie programmers often use global variables to move information from one part of the program to another. The main problem with this approach is that in a program of any complexity, it is too easy to get confused about who changed the variable last, making a complex program extremely hard to debug and maintain. For this reason, global variables should be avoided if at all possible, and are absolutely forbidden in this course unless they are explicitly specified. For example, there are a few cases where they work acceptably well as reasonable solutions to otherwise horrible problems. These are cases where only one instance of the variable makes sense, the information involved cannot be sensibly moved as parameters or returned values because widely separated levels of the calling hierarchy must be crossed, resulting in severely cluttered code, and where the values conceptually can be modified in only one or two very clear places, so there is no guesswork about who is responsible for the values.

To practice how to use global variables and specify the external linkage involved, you will use a few global integer variables to keep track of the amount of allocated memory being used in the program. Your code should add to these values whenever memory is allocated, and subtract whenever memory is deallocated. Two variables keep track of the total number of Ordered_container structs and items in use and total number of items that are allocated. Another variable tracks the total number of bytes currently allocated for the storage of C-strings. A global variable tracks the number of Meeting structs allocated. The number of Persons and Rooms can be easily determined directly (see the specifications for p1_main.c below). The **pa** command outputs the current value of these variables upon request. See the sample output for the format.

Note that for the linked-list implementation, the number of container items in use and the number of items allocated will be identical; the two global variables will be incremented or decremented together because they are both counting the number of nodes in the list. However, for the dynamic array implementation, the number of items in use in the container will always be less than or equal to the number of items allocated. The number of items in use is incremented/decremented whenever a data pointer is added/removed from the container, and the number of items allocated is increased whenever the array is reallocated, and decreased only if the container is cleared.

Because I will be testing your Ordered_container, Person, Meeting, Room, and main modules mixed with mine, the global variables must have the same types (int) and names throughout all the code. The best way to ensure consistent declarations of global variables is the scheme described in lecture and the handout. There is a p1_globals_skeleton.h file that contains the required variable names for all of the global variables. Fill in the rest of the file following the described scheme and rename it to "p1_globals.h" for use in your project. It must contain the declarations, in the form presented in the course materials, to allow the global variables to be accessed from any other module in the program (external linkage) simply by #including this file. The file p1_globals.c then defines the global variables. The files p1_globals.h and p1_globals.c *must not contain any other variables, functions, definitions, or declarations*. Any other content is a violation of this specification.

p1_main.c. The main function and its subfunctions must be in a file named p1_main.c. The file should contain function prototypes for all of these subfunctions, then the main function, followed by definitions for well-chosen subfunctions and "helper functions" in a reasonable human-readable order.

Your main function must create two Ordered_container objects, one for the room container, and one for the people container.

All of the container manipulations must be done using the interface specified in the supplied Ordered_container.h file. Ask for help if you think this interface is inadequate for the task.

Customarily, functions defined in the main module are *never* called by other modules - the main module is at the top of the calling tree. One way to enforce this is to declare the subfunctions in main as static functions so they can't be linked to by other modules. However, this clutters the code unnecessarily. If you follow the concept that each module's interface is declared only in its header file, then the fact that the main module has no header file means that it has no interface - other modules thus aren't supposed to try to use the main module as a sub-module. Therefore, under these rules for program organization, declaring the functions static in the main module is not necessary.

Utility.h, .c. This module is for functions, definitions, or declarations that are shared between the modules or used in more than one module. You must supply these files with your project even if they are empty. When I test your modules separately or in combination with mine, I will combine it with your Utility files as well, so your code can always use your Utility.

Learning how to group functions in modules is an important basic skill and a key feature of good code quality. Utility functions or definitions are supposed to be functions and definitions that are needed by more than one of the other modules, and placed in Utility to give a single point of maintenance (see the C Coding Standards document). *Any functions that are called from only one module, or* *definitions used in only one module, should be in that module, not in Utility*. Thus, "utility" means "useful in multiple places," not "dumping ground for miscellaneous odds and ends."

Implicit in this concept is that your Utility functions will refer to functions in other modules only as needed, and so your Utility.h and Utility.c should #include the headers for other modules only when really necessary (see the Header File Guidelines document).

Other Programming Requirements and Restrictions

1. The program must be written in ANSI C according to the C89 Standard - this is basically the Standard C described in Kernighan and Ritchie, as updated in the course materials. You are free to use any facilities in the Standard C library, and are expected to do so whenever appropriate - unnecessarily re-coding the wheel is bad programming. In particular, plan to use the C-string facilities in string.h, such as strcmp and strcpy to compare and copy the string data. Beware of "extensions" to the Standard library. For example, some C implementations contain a strdup function, but it is not specified in the Standard, and so you should not use it. These actually turn out to be of very little value for the projects in this course. Check the handouts section of the course web site for quick references about functions that might be especially useful.

2.Your code is expected to conform to the C Coding Standards for this course. Give that document an initial reading when you begin the project. Then as you finalize your code, *go through every item* in the Coding Standards document and check that your code is consistent with it. If you code violates these Standards, it will be scored as having low code quality.

3. There must be no global variables that are not specifically required or allowed by the project specifications. In addition to file-scope variables with external linkage, this restriction also applies to file-scope variables with internal linkage.

4. You may define global variables as constants (const char * const) to hold output message strings in the files p1_main.c, Person.c, Meeting.c, and Room.c. Each module should define only the strings that it outputs (see strings.txt) and these variables should have internal linkage. As it says in the Coding Standard

5. The string data in Persons and Meetings must be held in dynamically allocated memory, using the minimum allocation that will correctly hold the strings. Similarly, Person, Meeting, Room, and Ordered_container objects (and their components) must be dynamically allocated with the correct size. Your code must detect an failed memory allocation, and print a message of your choice and terminate. This message can be printed to either stdout or stderr; your choice.

6. No memory leaks are allowed, and upon normal termination (with the quit command) the program must free all allocated memory before terminating.

7. Unnecessary memory allocations are egregiously inefficient, and error-prone, and must not be done. In particular, character array buffers to temporarily hold input strings should be declared as local variables, not dynamically allocated chunks of memory.

8. Use only printf and scanf for all I/O of command, meeting, and person information, with format specifications of %c, %d, and %s. You can use the getchar function to skip unwanted characters in the input following an error. Ask for help if you think you need anything additional. Unless an error message is printed, the unread characters must remain in the input stream to be read by the next scanf call. That is, do not skip the rest of the characters in the input line unless an error message has been printed. See the samples.

9.Your input code can assume that all input text strings will fit into a 64-character array input buffer; use a `#define` symbol for this size. **Important:** Use the `%ns` format specifier to ensure that input of strings with `scanf` will not overflow the buffer. That is, if the input string is longer than 63 characters, it must not be allowed to overflow the input buffer; using `%63s` for the format will ensure it. You may define a global variable to hold a format string if you want to generate this string only once at run time for the entire program to use; it is egregiously inefficient to generate it for every read, so a global variable is permitted for this purpose to enable you to experiment with this technique.

10. You must implement the command language using `switch` statements to select a function to execute given the individual command letters - this is what `switch` is for; compilers can often optimize the code generated. Using a mess of conditionals when you can use `switch` is ugly, will always run slowly, and is a sign of poor code quality. Note that you can embed a `switch` inside a `case` of another `switch`. Each case should then *call a function* that does the actual work of the command. Include default cases in the switch to handle unrecognized commands. Because the quit command **qq** should result in a `return` from `main`, it does not have to be implemented as a separate function. The result will be a clear top-level structure for the program; once in place, it will be easy to add and modify the code for individual commands.

11. 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 my version of the program. Use Unix diff or sdiff or an equivalent function in your IDE for the comparison - don't rely on doing it by eye.

12. Do not check for end-of-file in reading console input with `scanf` or `getchar`. Some of you may have been told to do this in previous courses, but don't do it in this one! The normal mode of operation of this program is interactive on the console. It will keep prompting for input until you either tell the OS to force it to quit (e.g. cntrl-C in Unix) or you enter a quit command. I will be testing your programs with redirected input from files just for convenience, not as part of the concept of its operation. My test files will

always end with a quit command (**qq**) and yours should too! When you do input from a file, checking for end-of-file is a central part of the logic (be sure you do it correctly!). But for interactive programs, it just creates incredible clutter in the code.

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 mix and match my modules with yours, and I will test your Ordered_container module separately (so be sure all functions work correctly, especially those you did not need to use in this project). I also may interchange your .h files with those you are required to use as-is, as a check on following the specifications. Your Utility module will be used with your code in all component tests.

I will do a code quality evaluation on this project, so pay attention to commenting, organization, clarity, and efficiency consistent with good organization and clarity. Study the Syllabus information on how the autograder and code quality scores will be determined and weighted. Read the web page "How to do Well on the Projects". See the sample code quality grading checklist in the course handouts web page. Check you code against each item in the Coding Standards document. Review the course and web page material on quality code organization and design.

How to Build this Project Easily and Well

Recommended order of development

Simplify the program writing and debugging process by working on pieces of the program first - writing and trying to test this project all at once is difficult, unpleasant, and just plain stupid - don't do it! Your program should *grow* - you add, test, debug one piece at a time, rearranging and refactoring the code as you go, until you are finished. The result should be a well-organized body of code that was easy to build and easy to further modify or extend. Here is the recommended order of development:

1. Survey what kinds of work the project code will need to do, so you can anticipate what might be useful Utility functions - either write them first, or be ready to create them as soon as you find that you need the same code twice. For example, you'll be dealing with a lot of C-strings stored in allocated memory - how can you handle this? Review the C Coding Standards document for advice on how to recognize and create reusable functions.
2. Then, write and debug the generic linked-list version of the Ordered_container module completely separately, using a simple main module test harness (something as tiny as the posted demo), and storing something simple in the list, such as addresses of a set of int variables or string literals (like the posted demo). Check the essays on debugging and testing on the course web site. Test the daylights out of this module first - a completely debugged list module will make the rest of the project a lot easier. Notice that the posted demos are just demos, not complete tests!
3. You can develop the array container version either next, or after you have completed the rest of the project. However, if you wait until later, do not try to test and debug the array container with the whole project - that's doing it the hard way! Instead, take advantage of how the two implementations are supposed to behave the same, and reuse your test harness from the linked-list development. Much easier!
4. Next, if you write a simple testing driver, you can test the Person module by itself (or with just Utility) - call the interface functions with appropriate parameters, and see if you get the correct output printed out and correct values returned. E.g. create_Person should give you a pointer that you can give to print_Person, and it should output what you gave create_Person. Again, if this component is completely debugged, it will make testing and debugging the rest of the project *much* easier.
5. Next do the same approach with the Meetings module - only in this case, your test setup will need to include the Person module and the Ordered_container module. Get this module working, then go on to the Room module. After that, all that's left is the main module, and you have working components to build it with!
6. For the main module, build the command functionality a bit at time, testing and debugging as you go. It is always easier to work with a small program, and with only a little bit of new code at a time. For example, a good place to start is with the commands for working with Persons in the people container, such as first adding a Person and then finding or printing a Person.
7. As you add commands, watch for opportunities to *refactor* the code - changing some of your code into functions that make you can then re-use for the next commands. If you do a good job with this, the commands will become increasingly easy to do - mostly just calling already-debugged subfunctions in new combinations. This makes coding more fun!
8. Reserve the save/load command for last. Write the save function first, and examine the resulting file with a text editor - remember it is formatted to be human-readable. Try it out in a variety of situations and verify that the contents of the file are correct. Especially check boundary conditions - e.g. no Person data is present, or some Meetings have no participants. Only when you are satisfied that the save file is correct, should you try to get the load command working. You should not have a "god" function in `p1_main` that reads all of data and then stuffs it into Persons, Meetings, and Rooms. Rather, the details of creating a Person, Meeting, or Room from the file are supposed to be delegated to the `load` functions. **Important:** If your code

does not pass the tests for handling bad data files, do not try to fix the problem by stuffing in all kinds of checks for additional possible problems in the data file beyond those specified above. You will be wasting your time - I do not test for things which are not specified - and making a mess of your code besides. Instead, make sure that you are correctly implementing the specified checks, and that your program is doing the entire load process as specified.

Overall suggestions

You can use any specific algorithms you want for the generic list and array modules, and can base them on any code authored by you (review the academic integrity rules). However, the code must conform to the specifications, so expect to do some surgery.

In your code for accessing the data in an item (node or array cell), be very careful with casting the item data pointer to the correct structure type. It is very easy to make an error here. Stay aware of just what is being pointed to by the items in each container. The problem with such generic code in C is a severe lack of type safety - the compiler can't help you keep your code consistent when converting to and from `void` pointers. The symptom of a bad cast will be a peculiar run-time error such as a crash or garbage output. A good debugger will help make the situation obvious, because it will interpret the pointed-to data based on the cast that is present in your code, so if it looks wrong, check your cast.

An important code design concept is that of division of responsibility. In this project, a variety of objects are being created, destroyed, and manipulated - Persons, Meetings, Rooms, containers, and items in containers. As much as possible, you want the individual modules of the code to be as fully responsible for all of the work associated with an object; it should not be scattered around the program, but should be inside the module as much as that module's definition will allow. One place where this is especially important is in the modification of the global variables - whose responsibility is it to increment/decrement the items in use? Another example is who is responsible for reading and using the data to restore a Meeting?

Really important: Newbie programmers fail to organize code into good subfunctions and helper functions, and instead write sloppy duplicated convoluted code. This project has been designed so that the main module can be written pretty easily with a few good "building block" functions and the functions specified in each module. Absolutely do not write or paste the same code over and over again for each command. In general, watch out for places where code duplication (and attendant possible bugs) can be avoided by simple "helper" functions. Identify them early, and save some time!

In general, if you find yourself writing code that does the same thing twice, change it into a function - see the C Coding Standards for more discussion of when duplicated code should be turned into a function. Creating functions for duplicated work cuts down on errors, makes debugging easier, and clarifies the code. This is a basic principle of well-written code, and this first project is when you should start doing it. Copy-paste coding, and the mess that comes with it, is the most common code-quality problem students have on the first project in this course. The grade consequences are pretty serious, so breaking the habit on the first project is an extremely good idea.

Even if they write subfunctions and helper functions, newbie programmers often list *definitions* for functions in the main module in a arbitrary or haphazard order, making reading the code a *hellish experience of confusion and rummaging around*. A good human-readable order for the subfunctions is something like a top-down breadth-first order, with the helper functions coming last. Definitions of subfunctions or helper function should *never* appear before the main function, or before the first function that calls them. The idea is that I (or you) should be able to read the code like a well-organize technical memo: "big picture first", details later, finest details last. Function and variable names should be chosen to make this work well. A readable order of functions is a critical aspect of code quality; don't ignore it!

Finally, try to write good-quality well-organized code as you go - it will actually save time in this relatively complicated project. Once the code is working, take some more time to further improve your code quality and program organization. Obviously you have to start early to give yourself a chance to do this. Not only will a higher code quality score result, but you'll be able to more easily recycle your code for later projects - well-written code is reusable, garbage just has to be thrown away.