Project 1
A Micro Meeting Manager: Mastering C Techniques
Due: Friday, September 27, 2019, 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, a re-do of Project 0, is to write a program that keeps 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. Even though this project should behave almost the same way as Project 0, this document has a complete description of the specifications, meaning there is a lot of repetition with the Project 0 document. 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. This section is much larger and more detailed than Project 0, so read carefully. 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. Except for the memory allocation output from the **pa** command, these specifications are identical to Project 0.

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 meeting times fall within the traditional business day of 9:00 AM to 5:00 PM. Therefore the meeting times are expressed in 12-hour format without an AM/PM designation. For example, “10” is assumed to be 10:00 AM; “2” is 2:00 PM.
6. 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.
7. 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.
8. 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.
9. 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.
10. Rooms can be added or deleted.
11. Meetings can be added, deleted, or rescheduled for a different room and/or time.
12. All of the meetings can be deleted at once.
13. Participants can be added to or removed from a meeting.
14. 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.
15. 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).
16. At any time, the program can save the people, room, and meeting information in a file.
17. At any time, the people, room, and meeting information previously saved in a file can be restored, replacing any current information.
18. 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. In this project, the command provides more detailed information about the amount of memory allocated than in Project 0.

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; new time is out of range; a meeting at the new time already exists in the new room. To keep the logic simple, 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 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, ".234" in this case, 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 a bunch of 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, and this lookup needs to be done anyway to rebuild the schedule. 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. `fscanf` 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. `fscanf` 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 if the program can successfully read the file using the checks described in Case #1 above, the program does not need to do any further validity checks. 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 validity 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 as described in the course materials (see the C Coding Standards document) and any general requirements.

We will use a subset of C99. You must program this project in C, following the 1989 Standard (C89, also known as C90) which is what the K&R book assumes, except that we will actually compile under the 1999 C Standard so that you can use the following two C99 features, which were early C++ features that were so useful that they were back-ported to C:

- You can use // comments in addition to /* */ comments.
- You can declare variables anywhere in a { } block, or in a `for` or `if` statement, not just at the start of the block.

The declare-anywhere feature is valuable because you can declare a variable at the point in the code where you have a useful value to initialize it, which helps prevent errors due to uninitialized variables. The rule: *Do not declare a variable until you have a useful initial value to put in it!* This is a key part of good C++ coding, and it helps in C as well. For example, instead of

```
void foo(void)
{
    int i = 0; // often not a useful initial value, so just wastes CPU time and our typing.
    /* other code */
    i = bar(x, y, z); // now we have a useful value for i !
}
```

You write:

```
void foo(void)
{
    /* other code */
    int i = bar(x, y, z); // now we have a useful value for i !
}
```

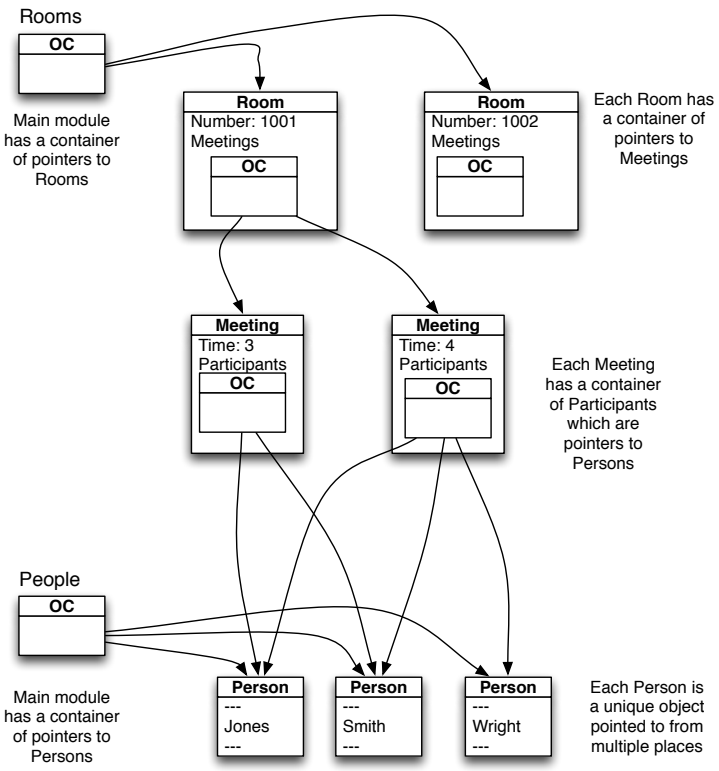
Using these two features will make it easier to convert chunks of your Project 0 code into C code for this project, and in turn, its code into idiomatic C++ for the next project.

Notice: Except for the optional use of `inline` (see below), the above two features are the *only* features of C99 relative to C89 that you can use. The rest of your code must conform to what we read in the K&R book, which is C89 (also known as C90).

Overall Data Structure

Your program should implement the overall data structure shown in the Figure below; study this brief overview, then continue reading for more details. The main module has two top-level containers: Rooms is an Ordered_container that holds pointers to Room objects ordered by number; People consists of an Ordered_container that holds pointers to Person objects, ordered by last name. Each Room contains an Ordered_container that points to Meeting objects, ordered by time. Each Meeting object contains an Ordered_container that points to Person objects, ordered by last name. Each Person is represented by a single unique Person object, pointed to from multiple places; there is no duplication of Person data. Each Meeting and Room is also a unique object, but should be pointed to from only one place. A Meeting can be transferred to another Room simply by moving its pointer.

Project 1 Data Structure



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.

The website project page also contains two skeleton .c files, one for the list and the other for the array implementation. These skeleton .c files contain the struct declarations that you are required to use in your implementation. See below for more about the implementation requirements.

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 by calling its functions) 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 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.

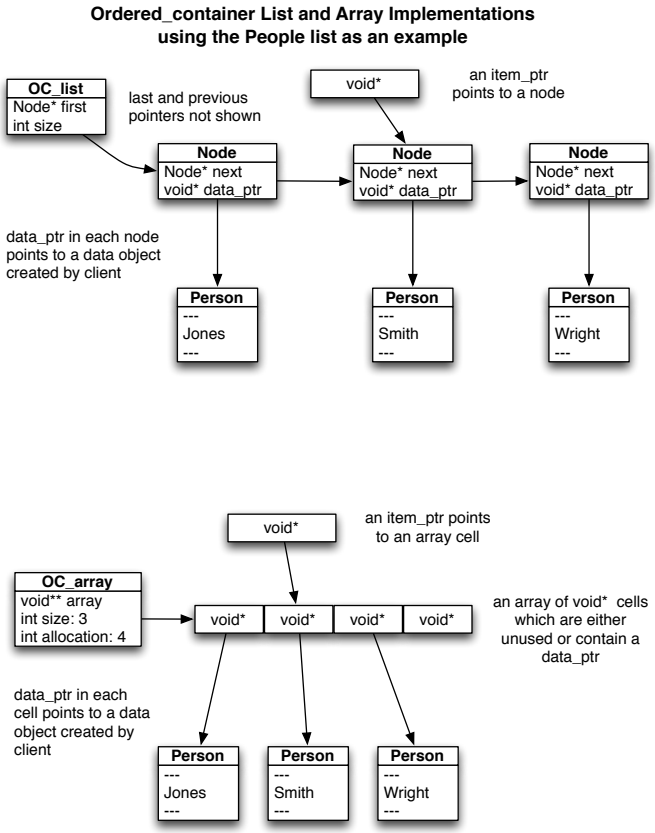
Items can be added to the container only through a function that inserts them in order; the container guarantees that items defined as equal by the ordering function can appear only once in the container. That is, the insertion function checks for a matching item already being present in the container and will not insert a duplicate item; it returns non-zero (for true) if the insertion succeeds, and zero (for false) if the insert fails because a matching item was already present.

An item can be searched for in the container; if found, it is designated with an 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.

Refer to the Figure below while you read about these two implementations.



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*.

Performance notes: The list is searched linearly to locate items, and to locate where to insert a new item. We insert a new item by creating a new node that comes *before* the appropriate existing node, or at the end of the list, designated with an item pointer of `NULL`. Searching and inserting is linear with the number of items (size) of the container (*O(n)*).

As shown by the declarations in the Ordered_container_list_skeleton.c file, we use a doubly-linked or two-way linked list. Although this requires more code, it has the advantage that once a node is located for removal, or a point of insertion is determined, the change to the list can be made in constant (*O(1)*) time. In addition, as the list is modified, we keep and update a pointer to the last node in the list, as well as the first node. This allows us to add a node to the end of the list without having to scan for its predecessor. Similarly, we keep up-to-date a member variable for the current size (length) of the list, so that client code can get the size in constant time.

Finally, the `insert` and `find` functions must search the list with a linear scan, but they always stop scanning when they have "gone past" where the matching item would be if it was present. This way if the matching item is not present, on the average only half the list will have to be examined. The time complexity is still linear, but this small tweak saves a lot of time even so.

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*.

Performance notes. The array is searched using binary search to locate items, so that searching will be logarithmic with container size (*O(log n)*). Not only the `find` functions use binary search, but also the `insert` function should use binary search to locate where to insert a new item. Note that the next step for insertion 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. Similar to the "gone past" strategy, this will greatly improve average insertion performance even if the time is *O(n)* in the limit.

Implementation notes: While in the linked-list implementation an item pointer points to a list node, in the array implementation, the item pointer simply points to an array cell. Since the array cell is a `void*`, a pointer to an array cell is a `void**`. The interface for the container requires casting this `void**` item pointer to/from a simple `void*` item pointer. Your implementation of the array container should represent item pointers in this way. As shown by the declarations in the Ordered_container_array_skeleton.c file, your code should have a simple `void**` pointer that points to the first cell of the array, and likewise, you should point to cells with a `void**` pointer as well. Be brave and confront the double-void pointer!

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`. Consider writing a binary search helper function that in addition to returning where the item was found (or not) also takes an `int*` parameter and uses it to return the array index where the insertion should be made if the item is not present. Call this function to implement both finding and insertion. Notice that to delete an item, the item pointer designates the to-be-removed item directly — you don't need to search for it.

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. Note that if the to-be-inserted item is already present, the insertion is not done, and so the array would not need to be increased in size. In other words, the process of checking the current allocation and then reallocation, copying the data, etc. is performed only if the new item will be actually inserted.

For this project, the initial size of the array is 3 cells. When a new item is to be inserted and 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.

As shown in the skeleton `.c` files, 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 a container 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.

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. The `OC_insert` function returned value should be used to determine whether the insertion failed because the item was already present in the container; often the data object should be destroyed if so.
- 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. Note that the find step has already located the list node or array cell — there is no need to search for it again. 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, usually before clearing the container.

Are the supplied pointers valid? The Ordered_container interface functions can assume that all supplied pointers 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 only with correct pointer values.

Notice that container pointers and item pointers supplied to the `OC` functions must be dereferenced by the `OC` functions, and so must always be non-`NULL`. You can check for this to detect potentially confusing errors. However, data pointers can be anything the client code wants them to be, including `NULL`. Since data pointers are never dereferenced by the `OC` functions, only by comparison functions supplied by the client, and client code, a `NULL` data pointer is valid as far as the container is concerned. So the `OC` functions should not insist on non-`NULL` data pointers. Of course the comparison functions and client code must check for and handle `NULL` data pointers correctly.

You should 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 an OC function with a `NULL` container pointer argument. Notice that if your function detects a `NULL` pointer and simply does nothing and returns, it is actually *hiding a bug* — this is not helpful! This is why using `assert` is a good idea!

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. Refer to the Figure above that illustrates how the two container implementations work.

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 a *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 an 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.

Ordered_container global variables. The Ordered_container.h header file contains declarations for three global variables. These variables keep track of the total number of Ordered_container structs and items in use and total number of items that are allocated. The pa command outputs the current values, as well as accessing the containers to determine how many Persons and Rooms currently exist. 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. 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 implementations, and your Record, Collection, and main modules mixed with mine, these program-wide global variables must have the same types (int) and names throughout all the code. The customary way to ensure consistent declarations of global variables is the scheme described in lecture and the handout that guarantees that each global variable has a unique definition and single point of maintenance: The globals .h file declares the variables as `extern`, the .cpp file includes the header and defines and initializes the variables; all other modules simply include the header file.

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, or a built-in data item, 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 use these three 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.

Check the above figure on the overall data structure. 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_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 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, but it is reasonable to do so.

Utility.h, .c. This module is for functions, definitions, or declarations that are used in more than one module. You must modify the supplied skeleton header file to supply a correct global variable declaration, and submit the .h and corresponding.c files with your project.

You can place additional definitions and functions in this module of your own choice. Learning how to group functions in modules is an important basic skill and a key feature of good code quality. The functions and definitions in Utility are restricted to those needed by *more than one* of the other modules, and so 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." (A possibly better name for "Utility" would have been "Shared".)

In addition, Standard Library #include statements needed by more than one module should not be placed in Utility.h. The Header File Guidelines take precedence: A header file X.h should only #include the headers that are needed for the X.h contents to be successfully compilable by themselves (see the Guidelines). For example, many of the modules will need to use the FILE symbol which is defined in <stdio.h>. But this #include <stdio.h> should not be in Utility.h because Utility.h does not (or should not) need it! As another example, several module .c files will need <stdlib.h> for memory allocation — but this #include should not be in Utility.h (or in any other project header files!).

When I test your modules separately or in combination with mine, I will build executables that mix your modules with mine, but your Utility files will always be in the build, so your code can use your Utility contents. However, if your Utility violates the above specification, the builds might fail because I will include in the builds only your other modules that are logically needed. For example, your Ordered_container modules logically should not depend on the Person, Meeting or Room modules, but can and should use functions in the Utility module. There is a good reason for Utility to depend on Person because it can have Person-related functions needed in more than one module, but there is no logical reason for Utility to depend on Meeting or Room. Thus a build to test your Ordered_container modules will include your Utility module and your Person module, but not your Meeting or Room module.

If the autograder reports build errors involving Utility and listing undefined functions that you know you defined, be sure your code meets this specification.

Global variables

Newbie programmers often use global variables to move information from one part of the program to another. Experienced programmers avoid this because decades of experience shows that if the program gets complex, you easily become 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. In this course, *global variables are **absolutely forbidden** unless they are explicitly required or allowed in the project specifications. This prohibition applies to file-scoped global variables even if they are internally-linked* — both program-wide and module-wide global variables make a mess; the only difference is in the size of mess.

However, there are a few cases where global variables work acceptably well as reasonable solutions to otherwise horrible problems, and this project requires you to use a few of them to illustrate this point. These are cases where

- only one instance of the variable makes sense conceptually;
- the information involved cannot be sensibly moved as parameters or returned values because widely separated levels of the calling hierarchy must be crossed, which would result in severely cluttered code;
- conceptually, the values can be modified in only one or two very clear places, so there is no guesswork about who is responsible for the values.

The Standard I/O streams `stdin` and `stdout` are actually global variables, and work well for these reasons.

To practice how to use global variables appropriately, and specify the external linkage involved, you are required to use a few global integer variables to keep track of the amount of allocated memory being used in the program — these required global variables are named in the supplied headers and skeleton header. Your code should add to these values whenever memory is allocated, and subtract whenever memory is deallocated. One variable tracks the total number of bytes currently allocated for the storage of C-strings. Three variables keep track of the total number of Ordered_container structs and items in use and total number of items that are allocated. Another global variable tracks the number of Meeting structs allocated. The number of Persons and Rooms can be easily determined from the container sizes (see the specifications for p1_main.c above). 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 all of the lists. However, for the dynamic array implementation, the number of items in use in the containers 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 a container, and the number of items allocated is increased whenever the array is reallocated, and decreased only if a 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 header file handout: A global variable is declared `extern` in a single header file, corresponding to the module it is associated with, and this header provides the interface to that global variable. The defining declaration for the global variable then appears in the .c file for that module; you must provide this defining declaration for each global variable. The supplied "starter" files show which modules the global variables are associated with.

Combining Type-Safe and Generic Code

Use actual Person, Meeting, and Room pointers wherever possible, use void* only in Ordered_container and its interface. As specified in Person.h, Meeting.h, and Room.h, these data items are referred to using pointers of the appropriate type: `struct Person*`, `struct Meeting*, struct Room*`. These are strong types and thus enable type safety — the compiler will not allow you to use the wrong type of pointer. For example, the `print_Person` function is declared in Person.h as requiring a const `struct Person*` pointer as a parameter. If you try to call `print_Person` with a `struct Meeting*` pointer, the compiler will disallow it. This helps prevent programming errors.

Compare this situation to what could happen if instead of the strongly typed pointers, we simply used `void*` pointers everywhere to point to Persons, Meetings, and Rooms. For example, the `print_Person` function would have a `void*` parameter, and so would the `print_Meeting` function. In this case we could accidentally call `print_Person` with a pointer that actually points to a Meeting, not a Person. The compiler cannot detect this error — a `void*` can point to any kind of data. Thus the program would build without errors, but it would produce incorrect results.

The conclusion is that we should give the compiler as much information as possible about the type of pointed-to objects so that it can help us write correct code. Thus, as much as possible, your code should be type-safe by using the "strongly-typed" `struct Person*`, `struct Meeting*, and struct Room*` pointers everywhere possible.

However, the `Ordered_container` module is a generic container because it works in terms of `void*` pointers to the actual data objects. To use it, you have to supply `void*` pointers that point to the actual Persons, Meetings, and Rooms. We could handle this by pointing to Persons, Meeting, and Rooms with `void*` pointers instead of strongly-typed pointers, but as just discussed above, this sacrifices type safety and opens many possibilities for programming errors. Thus you should use the strongly-typed pointers wherever possible, but use casts to convert the Person, Meeting, and Room pointers to/from `void*` pointers at the interface between the generic `Ordered_container` code and the type-safe code in the rest of the program.

Dealing with incompatible function pointers. A difficult situation arises with function pointers that you supply to the `Ordered_container` functions. These are pointers to the comparison functions supplied to `OC_create_container`, `OC_find_item`, and `OC_find_item_arg`, and the "apply-functions" supplied to the `OC_apply` family of functions. The `Ordered_container` code is going to call your comparison and apply-functions with arguments that are `void*` pointers to data objects. But your comparison and apply-functions will have to treat those `void*` addresses as addresses of Persons, Meetings, Rooms, C-strings, etc, so at some point, the `void*` pointers have to converted into the corresponding strongly-typed pointers.

Review the lecture notes on casting function pointers. If we write a comparison function that takes `struct Person*` pointers, the compiler will not allow us to simply use that function name as a pointer to a function that takes `void*` pointers — the function pointer types are *incompatible*. As described in the lecture notes, there are two solutions to this problem. One is to use a wrapper function that accepts `void*` parameters and calls the actual function, and the other is to use a function pointer cast, which should be valid because on the architectures we are using, the function call stack contents should look the same for `void*` and the strongly-typed pointers. To get some practice with handling this issue, your code must meet the following requirements (the bullet point items below):

Notice that the comparison functions are not a specified part of the interface for Persons, Meetings, and Rooms; therefore we are free to give them compatible parameters so that they will be compatible with the function pointer types required by `Ordered_container`:

- Write the comparison functions so that their parameters are `void*` pointers. They must not be simply wrappers to the "real" comparison functions, but actually carry out the comparisons — they do the comparison work internally after casting the `void*` parameters to the correct types (either explicitly or implicitly — your choice).

The apply-functions are usually those specified in the Person, Meeting, or Room interface, and are specified there as taking some combination of `struct Person*`, `struct Meeting*`, `struct Room*`, or `FILE*` parameters. Your code must do the following:

- In at least two cases of applying Person, Meeting, or Room functions to a container, you must use a function pointer cast when calling the `OC_apply` family function. For example, this requirement would be met by casting `print_Meeting` to a `void(*)(void*)` function pointer type and by casting `save_Person` to a `void(*)(void*,void*)` function pointer type.
- In at least two cases of applying Person, Meeting, or Room functions to a container, you must use a pointer to a wrapper function when calling the `OC_apply` family function. For example, this requirement would be met with a wrapper for applying `print_Person` and a wrapper for applying `is_Meeting_participant_present`.
  - At your option, you can declare the wrapper functions internally linked in the .c file and declare them as `inline` to eliminate the function call overhead. You are allowed to use this C99 feature only for this purpose in this project.
- The Person, Meeting, or Room functions used to satisfy the above two requirement must be different functions. For all other cases of apply-functions, it is your choice which approach to use in resolving the incompatible function pointer types.

Other Programming Requirements and Restrictions

1. The program must be written in ANSI C, compiled under C99, but using only the two features of C99 as described above, and C89 features otherwise. C89 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 extensions have actually 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. Any variable that is declared outside of a function is a global variable. This prohibition applies to both program-wide and internally-linked global variables.

4. You may define global "variables" as constants to point to output message strings (declare as `const char* const`) 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. If these are properly declared as constants, they do not have the problems that actual global variables do, and so technically are not global variables, but global constants.

5. The functions defined in p1_main.c should not be called by any other modules, but customarily, such functions are not declared with internal linkage, but you may do so at your option. In fact, because their prototypes are not in a header file, they could be called only with obviously flakey hard-coded local declarations in the other modules.

6. 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.

7. No memory leaks are allowed, and upon normal termination (with the quit command) the program must free all allocated memory before terminating. This policy of "clean up before quitting" is good practice, even if it is often not essential in modern OSs.

8. Your code must detect a failed memory allocation, and print a message of your choice and immediately terminate with a call to `exit()`. The message can be printed to either `stdout` or `stderr`; your choice. For simplicity in this project, your program does not need to try to clean up its memory allocations in this exceptional situation; we will rely on the OS.

9. Unnecessary memory allocations are egregiously inefficient, and error-prone, and must not be done. For example, character array buffers to temporarily hold input strings should be declared as local variables, not dynamically allocated chunks of memory. A good check: if the memory can or should be freed when the function that does the allocation returns, then the memory allocation is almost certainly unnecessary or premature. Similarly, when searching for an existing object, do not create a Person object simply to search for it; that's what the `find_arg` function is for. Create new Persons, Meetings, or Rooms only when it is supposed to get stored in a container; destroy them if they turn out to be duplicates according to the `insert` function.

10. Use only `printf`, `fprintf`, `scanf`, and `fscanf` for all I/O of command, meeting, and person information, with format specifications of `%c`, `% 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.

11. 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; the compiler can often optimize the code to be much faster than the corresponding structure of `if-else`s. A mess of `if-else`s is ugly and a sign of poor code quality if a `switch` can be used instead. You should embed a `switch` inside a `case` of another `switch` to select on the two command letters. 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 of using this two-level `switch` will be a simple and clear (though long) top-level structure for the program; once in place, it will be easy to add and modify the code for individual commands, which makes developing the program a lot easier.

12. To avoid getting sucked into "overengineering" — a problem that afflicts some beginning programmers — *do not use function pointers except where specified in this project*. Function pointers are a way to implement a facility that *must* be generic (as in the Ordered_container module); but over-use of function pointers produces cryptic obfuscated code that has no advantage over simple direct function calls.

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 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.

14. Follow good programming practice by ensuring that program parameters, such as maximum string lengths, and the corresponding `scanf` input formats, are specified as named constants using `#define`. See the lecture notes and the C Coding Standards for guidance on avoiding "magic numbers" and "magic strings" in the code.

15. Review and follow the requirements in the above section on *Combining Type-Safe and Generic Code*

16. Your input code can assume that all input text strings will fit into a 64-character array input buffer; in the right place in the code, use a `#define` symbol for this size to avoid a "magic number" maintenance problem in the code.

17. 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.

18. Avoid a "magic string" like "%63s" for the scanf/fscanf format for reading into the character array; if this literal string appears in every scanf/fscanf call, it presents a maintenance nightmare if we changed the character input buffer size. Use one of the techniques described in the C Coverage lecture notes to solve this problem.

19. If you want to experiment with generating `%ns` format strings at run time using `sprintf`, you must generate them only once, at program start up, and save the results in one or more additional global variables of your own that you must correctly declare and define in the Utilities module. **Important:** These are the only additional global variables you are allowed to have. **Very Important:** Using these generated format strings is optional, and your code must be arranged so that the autograder, which assumes it is optional, does not reject your program depending on whether it does or does not use this option. If you do not take this approach, you do not need to do anything special. But if you take this approach, you *must* do the following:

Define a function similar to this in your Utility module (it can have your own name):

```
void setup_global_format_strings(void)
{
   static int format_strings_set_up = 0;
   if(format_strings_set_up)
      return;
   /* put the code to generate the format strings in global variables here */
   format_strings_set_up = 1;
}
```

Call this function at the beginning of your `main()`, and then at the beginning of `load_Person()` in Person.c and `load_Meeting()` in Meeting.c. This is a bit clumsy and inefficient, but much less so than creating the format strings every time a read is needed, and makes sure that in the component tests, your components have the strings generated before they use them. In the meantime, my components will be doing their own thing, and it won't interfere with yours. In a real application, generating and using these strings would be part of the overall specification and so this kludgy setup would be unnecessary.

20. 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, checking for `eof` on keyboard input just creates incredible clutter in the code.

Project Evaluation

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.

The supplied header files must be submitted with your code and must not be modified in any way at all. Certainly you can clean up the formatting in your own working copies, but your submitted code must contain the these exact files with no modifications at all.

The autograder will check your program for correct behavior, and component tests will mix and match my modules with yours, and will test your Ordered_container_array and Ordered_container_list modules separately (so be sure all functions work correctly, especially any that you did not need to use in the complete program). The autograder will also use the fully specified (use as-is) header files in place of yours as another check on whether your code follows the specifications. Your Utility module will be used with your code in all component tests.

I will do a full 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 your 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

Beginners have a habit of writing the whole program first, and then try to test and debug it. This approach is difficult, frustrating, and grossly inefficient! You end up trying to debug the whole thing at once, and if you made a poor design choice at the beginning, you have throw away a lot of code! Don't do it! Instead, 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. At this point, you should be able to implement all of the Person-oriented commands in a first version of the main module that makes use of your Ordered_container module. Build the command functionality *one command 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 and printing a Person, and then deleting a Person.
6. Next do the same approach with the Meetings module. Get this module working, then go on to the Room module. After that, all that's left is the remaining command in main!
7. As you add commands, watch for opportunities to *refactor* the code — change some of your code into functions that you can then re-use for the next commands. If you do a good job with this, the commands will become increasingly easy to implement — mostly just calling already-debugged subfunctions in new combinations. This makes coding more fun!
8. Reserve the save/load command and the load and save functions in the modules for last. Write the save function first, and examine the resulting file with a text editor; remember its format is designed 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, for example, if 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, this work should be *delegated* to the `load` functions in each component.
9. **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 because (1) the autograder tests do not test for things which are not specified; and (2) the redundant tests make a mess of your code. 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

Especially relevant Principles of Good Programming for this project: DRY — don't repeat yourself by duplicating code; create functions instead. KISS— keep it simple, stupid! Don't over-engineer the code. Do the simplest thing that could possibly work. YAGNI — you aren't going to need it! Don't add functionality until you need it; don't overgeneralize the code. See the "Principles of Good Programming" link on the course home page for more about these and related ideas.

Converting Project 0 C++ code into C for this project. Well-written C and well-written C++ procedural code are actually very similar. Here are a few basic rules: Change any classes you defined to structs with associated functions instead of member functions

— see the specifications for Persons, Rooms, and Meetings as an example. Change your functions that take no arguments to have a parameter list of `(void)` — an empty parameter list in C means something different than in C++. If you used exceptions for error handling, replace them with the traditional tedious checking of return values; properly done, the result is reasonable and is a common pattern in much code, though usually inferior to exceptions. The C stream I/O functions and error conditions are conceptually isomorphic to C++ stream operations, but the details of error detection and handling are different; study the C streams handout closely — everything you need is there.

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

Watch your casts! 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.

Design the responsibilities. 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?

Create a tree of helper functions in the main module. 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 if you did't do it in Project 0, you must start doing it now.

Write good code as you go. Finally, try to write good-quality well-organized code as you go — once you learn how to do this, 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; a messy pile of garbage just has to be thrown away.

Comments. Appropriate comments are part of good coding. Elaborate and formal comments are not necessary in this course, but very sparse or pointless comments are a failure of code quality. The basic idea is to help the reader of the code (certainly me in the course, but which could be you in the same or next project). The reader must be able to understand what is happening in the code without having to reconstruct all of the reasoning you went through to write it. A few key points:

- See the essay on the course web page and the Coding Standards for guidance on comments — this is the level and style of commenting that I expect — more formal or elaborate comments are OK, but not required.
- Saving comments until after you finish the code is a bad idea. Properly done, comments will help you write the code by clarifying what has to be done and helping you keep track of what you are doing. See the essays for some examples.
- Function declarations in a header file that are part of the interface for a module should be commented there so that whoever is writing the client code does not need to find and study the function definition in the implementation .c file in order to see how to call the function. The supplied header files have many examples of such comments, but the very long and detailed comments are part of the project specifications, and would not normally be in real software code.
- In an implementation .c file, functions that are not part of the module interface should be declared near the top of the file so that function definitions can be listed in a human-readable order instead of a compiler-determined order (see discussion below). These declarations should not be commented because humans usually will not be looking at the top of the file when examining

the function definitions. Rather, each function definition should be preceded by a comment, even if they were commented in the header file.

- If the function was declared in a header file, it is very helpful to copy the basics of the header file comment into the comment before the definition. However, some of the comments in the supplied header files are very lengthy descriptions that are part of the project specifications. Copying *all* of theses specification comments in some of the supplied headers is not helpful — the basics are enough.

Put the main module code in a readable order. Even if they write subfunctions and helper functions, newbie programmers often list *definitions* for functions in the main module in an arbitrary or haphazard order, making reading the code a *hellish experience of confusion and rummaging around*. Fix this problem as follows: *Declare* the functions at the top of the source code file(s) to make it possible to order the *definitions* to make the code easily human-readable. A good human-readable order for the functions is something like a top-down breadth-first order, with the top-level function coming first and the lowest-level helper functions coming last. Definitions of subfunctions or helper function should *always* appear *after* the first function that calls them, or all functions that call them, *never before*. 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!