Project 0
A Micro Meeting Manager: Getting Started
Due: Friday, Sept 13, 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 this page for the project frequently while you are working on it. Check also the FAQs for the project, even if you don't currently have a question in mind — you can learn a lot by reading FAQs! At a minimum, check the project web pages at the start of every work session.

Introduction

Purpose

An early project for early feedback: This project will:

- Give you early feedback on what the projects are like in this course so you can decide whether to stay in the course.
- Allow you to get thoroughly familiar with the behavior specifications before you need them for Projects 1 - 3.
- Provide some experience in code quality by designing and implementing a function hierarchy and avoiding global variables.
- Help you develop much of the code you need for Projects 1-3. This includes some experience with stream input operations.

In more detail: The Project 0 behavior specifications are almost identical to Project 1, and these specifications are also the basic specifications for Project 2 and 3. *You will write this first project using your pre-existing knowledge of C++ programming*. There will be a few specific code organization requirements, focussed on a couple of the most important principles of good programming, described in this document below. But other than these, you will not be expected to make use of any of the specific techniques covered in this course.

The later Projects 1-3 will require that you organize the code in specific ways and use specific programming techniques (e.g. opaque types in C for Project 1; a DIY String class implementation with move semantics and exception safety in Project 2; thorough use of the STL containers and algorithms in Project 3). The code will be tested for correct input/output behavior using the autograder; these input/output tests are designed to be almost identical to those for Project 1.

In addition, the quality of your code will be assessed with some spot-checks, but unlike the later projects, this spot-checking will be very limited — the evaluation will be on whether your code follows the specific code organization requirements listed below. The code spot-check thus will not depend on using any of the sophisticated features of C or C++ that this course covers. In fact there will be no credit given for "fancy" or advanced code, and in fact, inexperienced programmers often write awful code when they try to use fancy or advanced techniques. This document explains below what code quality principles you will be expected to apply. If you write well-organized, straightforward, and simple code, you will have no trouble "recycling" it for Projects 1-3.

Problem Domain

This project 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. The *Programming Requirements* describes the code quality issues which you must address — the goal is to start learning and using some concepts of "good code," not just hack together some code that behaves right.

Program Specifications

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

1. The user can specify the numbers of the rooms available for meetings. These are kept in a list of rooms ordered by room number - *the room list*.
2. The user can specify the names and phone numbers of individual people who can be participants in the meetings. These are kept in a list of individuals ordered by alphabetical order last name - *the people list*. **Note**: *Throughout this course*, "alphabetical order" for a string means the standard English language order defined by the C and C++ Library functions for ordering character strings (`strcmp`, `std::string::operator<`) . In this order, lower case 'a' follows upper case 'Z'.
3. The user can specify the room number, time, and topic of a meeting; the meetings are kept in a list for that room, ordered by the meeting time — *the meeting list for each room*. In separate transactions, the user can specify which people are participants in a meeting; the participants are kept in a list associated with the meeting, *the participant list for each meeting*. The participant list is ordered by the last name.
4. Rooms are referred to by specifying their number; meetings are referred to by specifying their room number and time, and people are referred to by their last names.
5. The 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 later project in the course.
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. The name is case-sensitive, meaning that McDonald and Mcdonald are two different names. There is no maximum width.

<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. Errors: none. In Project 0, this command outputs the number of people in the person list, the number of rooms in the room list, and the number of meetings currently scheduled in all rooms; it does not show the actual amount of memory allocated for this data.

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 simpler, the last error will result if the user attempts to reschedule a meeting to be in the same room and at the same time as it is currently.

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

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

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

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

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

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

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

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

ld <filename> - load data - restores the program state from the data in the file. Errors: the file cannot be opened for input; invalid data is found in the file (e.g. the file wasn't created by the program). In more detail, the program first attempts to open the file, and if not successful simply reports the error and prompts for a new command. If successful, it deletes all current data, and then attempts to read the people, rooms, and meetings data from the named file, which should restore the program state to be identical to the time the data was saved. If an error is detected during reading the file, the error is reported and any data previously read is discarded, leaving all the lists empty.

qq - terminate the program. 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 these facilities properly, the required code is extremely simple — the streams do almost all the work for you! You should review the basics of C++ streams in the relevant handouts on the course web site; many people don't know certain key ideas and techniques. 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. 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 input functions work when reading an integer from an input stream. 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). If any of these concepts is unfamiliar to you, read the streams handouts on the course website.

- 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 the input functions 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.

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. 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. The number of people, number of rooms, the number of meetings, and number of participants is zero or greater. 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.
- When the file is read by the **ld** command, the program should 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 data, 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.
- A numeric value for the number of items to be read was negative rather than zero or greater.
- 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.

2. *The file was correctly written by a correct program in response to an earlier **sd** command*. If so, you can assume 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.

- 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. *The autograder will not test for unspecified behavior.*

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. You are expected to be able to code Project 0 using your existing knowledge of C++. Consequently, the programming requirements are very few because we haven't yet started on the course material on specific programming techniques.

1. ***Except for the main code file, no specific code files are specified.*** The main code file is specified below. Other than it, you are free to organize your code into as many or as few files as you choose. No Makefile is either supplied or should be submitted for this project, although you are free to use one for your own development work.

2. ***C++ compiler options.*** The program must be written in C++ compatible with gcc 9.1.0 and C++17. This does not mean that you have to use features found only in C++17, it just means that it must compile and build successfully with these options. See the course website for how to access this version of gcc on CAEN. See below for information about submitting your code. Your code must compile and execute correctly using g++ with the following options:

```
-std=c++17 -pedantic-errors
```

3. ***Standard Library only.*** You may make full use of the C++ Standard Library facilities in this project. You may not use any other preexisting libraries or facilities such as databases, parsers, etc. written by others. In other words, except for the C++ Standard Library, all of the code in the project must be written by you. Review the academic integrity rules in the course Syllabus and Policies document.

4. ***No global variables are allowed.*** Definition: in C or C++, *a global variable is a variable declared and defined so that its scope is outside a function, structure type, or class, and it is either internally- or externally-linked*. Inexperienced programmers often use global variables to move information from one part of the program to another. Experienced programmers avoid global variables 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. Using global variables in place of function parameters is a very bad practice; exceptions are rare and will be covered in later

projects. In this course, global variables are absolutely forbidden unless they are explicitly required or allowed in the project specifications. To get started on doing things correctly, *your code for this project must not declare and use any global variables*. [1]

5. ***The top-level code organization must be a well-designed function tree.*** Your program must have a top level implemented in the form of a good function tree, the fundamental programming technique for well-organized code. To get you started on designing and using a good function call hierarchy, your program must be written as follows:

- Your function `main` should be defined in a file named `p0_main.cpp`. Function `main` should contain the top level loop that reads and executes each input command. It does this by reading the two command characters and then calling a separate function, one for each command, that performs the actual work of that command. For example, there should be a separate function that inputs the parameters and carries out the work of the **ai** command, and another function that inputs the parameters and carries out the work of the **am** command, and so forth — a function for each command. Because the quit command **qq** should result in a `return` from `main`, it does not have to be implemented as a separate function.
- Each command function must be implemented by calling a set of "helper" functions that should be well designed to re-use code and give "single points of maintenance" — that is, if the program must do the same thing in handling several commands, it should do it by calling a function that does that thing, and this function is the single piece of code for that piece of the processing. Often these helpers can be well implemented in terms of sub-helpers.
- If you use classes in your program, use them only for the domain objects such as Persons, Meetings, and Rooms that have multiple instances. Do not use classes that have only single instances or that represent non-domain concepts like an input parser, a data base holding all the schedule information, etc. Such design patterns will be discussed later in the course, but for Projects 1-3, a well-designed function tree (see previous bullet points) works very well for organizing the code, and you need practice in this fundamental *procedural programming* technique (which can also be needed for complex functionality within a class).
- See the suggestions below in "How to Build this Project Easily and Well" for more about best practices in code organization and structure.

6. ***Output message text is supplied.*** 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 samples to be sure your program produces output that can match my version of the program. Use Unix `diff` or `sdiff` or an equivalent function on your development platform for the comparison — don't rely on doing it by eye.

7. ***Do not check for end-of-file in reading console input.*** 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

Your project will be autograded (computer-graded) for correct behavior. 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, and how to access gcc 9.1.0 on CAEN, which is the compiler used for autograding. If your code compiles, links, and executes correctly on CAEN gcc 9.1.0 using the following commands, it should work when submitted to the autograder:

```
g++ -std=c++17 -pedantic-errors *.cpp
./a.out
```

Notice that this project does not specify which files or how many files should be used to implement this project. Just be sure to submit *only* the source code files actually needed for the project! If you submit unnecessary .cpp files, such as testing files, they will be included in the build, probably causing a build failure.

I plan to do a fast and limited code quality evaluation in the form of a spot-check focussed on the issues described in the Programming Requirements section above, so pay attention to them and the suggestions below. The autograding and spot-checking scores will be combined following the rules described in the syllabus.

---

[1] Technically, the standard streams in the C and C++ Standard Libraries, namely `stdin`, `stdout`, `cin`, `cout`, etc. are global variables; this prohibition does not apply to the standard streams.

How to Build this Project Easily and Well

The Main Code Quality Issue in this Project

Inexperienced programmers fail to organize code into good subfunctions and helper functions, and instead write sloppy duplicated code - *copy-paste coding*. Almost all 381 students make this mistake on the first project. This project has been designed so that the command functions called by the top level loop can be written pretty easily with a few good "building block" helper functions. *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 duplicated bugs) can be avoided by simple "helper" and "sub-helper" functions. Identify them early, and save some time! If you discover them later, refactor your code immediately to use them!

In general, if you find yourself writing code that does the same thing twice, change it into a function - see the ***Designing Functions*** section of the course C and C++ ***Coding Standards*** for more discussion of when duplicated code should be turned into a function. Creating functions for duplicated computations cuts down on errors, makes debugging easier, and clarifies the code. This is a basic principle of well-written code, and this first project is when you should start doing it. Copy-paste coding, and the mess that comes with it, is the most common code-quality problem students have on the first project in this course. Break this habit immediately!

Other Best Practices

Write small amounts of code at a time, and make it good code as you write, cleaning up as you go along. A common beginner's mistake is to write and test the entire project, and then trying to clean up the resulting mess. It is better and more fun to develop and test the code in small pieces, taking advantage of what you learn as you go. For example, often you will discover how your code should be organized while you are writing it, but the secret is to then reorganize the code to improve its design — this is called "refactoring" the code. Learn to do this continuously as you write, so that you are effectively writing good-quality well-organized code as you go. Once you learn how to do this, it will actually save time — things get easier as the code develops, rather than harder! After the code is complete and 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.

Put the function definitions in a readable order. Even if they write subfunctions and helper functions, newbie programmers often list *definitions* for functions 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-organized technical memo: "big picture" first, details later, finest details last. A readable order of functions is a critical aspect of code quality; don't ignore it!

Other Issues

Especially relevant Principles of Good Programming for this project:

- DRY — *Don't Repeat Yourself* — don't duplicate code — see the discussion above.
- KISS — *Keep It Simple, Stupid!* Do the simplest thing that could possibly work. Be especially careful if you have read about various advanced techniques and concepts; often in this course the result is sophisticated but pointlessly complex and elaborate code. Discuss with us before you go down this path — it can easily lead to *over-engineering*, which is a sign of the beginner, not the professional. Good programmers know that the goal is clean and clear code, not a show of technical wizardry.
- YAGNI — *You Aren't Going to Need It* — don't add functionality until you need it — don't overgeneralize your code.

See the "Principles of Good Programming" link on the course home page for more about these and related ideas.

Recommended order of development

Simplify the program writing and debugging process by working on pieces of the program first — writing and trying to test this project all at once is difficult, unpleasant, and just plain stupid! 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. For the main 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 the add person and print person commands. Then do similarly with rooms and meetings.

2. As you add commands, watch for opportunities to *refactor* the code, which means to *improve the design of existing code*. Change some of your code into functions that you can then re-use for the next commands. The patterns and regularity in the input syntax mean that for example, you can define a function that reads a room number and meeting time and returns a pointer to the meeting after doing the required error checks for successful reads of an integer and successful lookup of the room and meeting. Thus the command function only needs to call this one function to get the user's desired meeting. With some thought, you can define a good set of building block functions, for example, a look-up function might be called by two other functions, one of which requires a successful lookup, and the other requires a failed lookup. If you have made good choices, the commands will become increasingly easy to code, mostly just calling already-debugged functions in new combinations. This makes coding more fun!
3. Reserve the save/load command for last. Write the save function first, and examine the resulting file with a text editor; remember the specified format is supposed 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, when 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.
4. **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 — *the autograder does not test for things which are not specified* — and these redundant checks make a mess of your code besides. Instead, make sure that you are correctly implementing the specified checks, and that your program is doing the entire load process as specified, especially in leaving the program in the correct state after a failed load.