Project 1
Designing Functions and Mastering Pointer Techniques:
A Simple Media Manager in C

Due: Friday, January 27, 2017, 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.

Read "How to do Well on the Projects" on the course web site before going any further. Really. Do it!

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 interact 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.
- Interfacing between type-safe code with our defined pointer types and the generic container that uses `void*` pointers and dealing with some of the issues where C cannot guarantee correct results due to the need to use implicit and explicit casts.
- Using C techniques for "object-based" programming with good encapsulation and insulation. These techniques are in common use in well-organized modern C software and provide some of the advantages of object-oriented programming with this non-object-oriented language.
- Internal and external linkage for global variables and file-local functions.
- Practice with implementing a node-based and array-based container.
- C techniques for console and file I/O, string processing, and dynamic memory management.

Problem Domain

This project is to write a program that uses containers to keep track of a library of movie media (DVDs, VHS tapes, etc) which can be "tagged" - grouped into named collections. Each movie corresponds to a *record*. At present the record holds on the title, medium (DVD vs. VHS, etc), a rating (on a 1-5 scale), and a record ID number assigned automatically by the program. A record can be referred to by either the title or by the ID number. There is no limit on the number of possible records. Records can be created, destroyed, or modified. Each record needs to have a unique title so that it can be reliably retrieved when the title is entered. Two copies of a movie must be given different titles (e.g. "copy 1" and "copy 2" at the ends).

A *collection* has a string name (like "favorites") and a container of references to records. Collections can created and destroyed, and records added or removed from a collection. There is no limit on the number of collections, nor on the number of records in a collection. A record can be in any number of collections, including zero.

The set of records is called the *Library* and the set of collections is called the *Catalog*.

Later versions of this project will use additional programming techniques and have additional features; you will be able to recycle much of your code for this project in the later projects, so organizing and writing it well on the first project will make the next two projects easier.

Overview of this Document

There are two major sections: The *Program Specifications* describe what the program is supposed to do, and how it behaves. The *Programming Requirements* describe how you have to program the project in terms of the structure and organization of the code, and which specific techniques you must apply - remember the goal is to learn and apply some concepts and techniques for programming,

not just hack together some code that by some miracle or dumb luck happens to behave correctly. At the end is a section of advice on *How to Build this Project Easily and Well* - take it seriously!

Program Specifications

The basic behavior and functionality of the program is specified in this section. As you read these specifications, study the posted samples of the program's behavior. These specifications are very similar to the Project 0 specifications; the important differences are that (a) input strings have fixed maximum lengths (typical for C programs), and (b) the rules for input and output are explained in terms of the relevant C stream I/O functions that your code should apply.

1. Throughout this course, "alphabetical order" for a string means the standard English language order defined by the C and C++ Standard Library functions for ordering character strings from "smallest" to "largest" (`strcmp`, `std::string::operator<`) . In this order, lower case `'a'` follows upper case `'Z'`.
2. Throughout this course, "whitespace" means spaces, newlines, tabs, etc. as defined by the C library function `isspace()`. These are characters that when displayed, produce only "whitespace" on the display.
3. To create a record, the user supplies the movie medium (e.g. "DVD" or "BluRay") as a whitespace-delimited character string, and a title as a character string that can include embedded spaces and is terminated by a newline (`\n`) character in the input. The title is stored internally with no leading or trailing whitespace, and all embedded sequences of one or more whitespace characters are replaced with a single space character (`' '`). For example, the entry `"  Much  ado  about  Nothing  "` would get trimmed at both ends and extra spaces compacted so that it stored as `"Much ado about Nothing"`. Note that the newline character is also whitespace and would be trimmed from the end. A title must contain at least one non-whitespace character. The user can specify a record by entering its title; an exact match is required, but the entered title is compacted before being compared to the titles in the records. The comparison is case-sensitive. The rating of a new record is stored in the record as 0, which appears in the output as the character 'u' for "unrated".
4. The program assigns an ID number to each record when it is created. The ID number appears in the program output. The user can enter the ID number to identify records rather than having to type the entire title. The ID number cannot be modified by the user. The program keeps a record ID counter; the first record created is given the ID number of 1, and each record created afterwards is given the next ID number in order. Destroying a record results in its ID number becoming unused, and it remains unused. That is, when another record is created, it is given the ID that is one greater than the last ID number assigned, regardless of which records have been destroyed. In the rest of this document, when the ID record counter is specified as being "reset", it means that it is changed so that the next record created will have an ID number of 1. Thus, if the Library is cleared, the ID counter is reset so that the first record subsequently created is given an ID number of 1.
5. If the user creates a new record whose title is the same as an existing record, it is an error; the new record is discarded, and the record ID counter is not updated, so that the original ID number can be used on the next record that is successfully created.
6. The Library consists of the set of individual records kept in alphabetical order by title. When displaying the Library, the records are displayed in alphabetical order by title, with the ID number and medium shown for each record as well.
7. The user can modify the rating of a record by supplying a rating on a 1-5 scale. The rating subsequently appears whenever the record is displayed.
8. To create a collection, the user supplies a collection name (e.g. "favorites") as a whitespace-delimited character string. The collection is empty when first created. The collection is referred to by its name; an exact match is required. The user can destroy a collection by specifying its name. The user can add or remove a record to/from a collection by specifying the collection name and the record ID number. A record in a collection is a *member* of the collection. When displaying a collection, the collection name is shown followed by each record in the collection in order by title.
9. The Catalog consist of the set of collections kept in alphabetical order by collection name. When displaying the Catalog, the collections are displayed in alphabetical order by name.
10. A record cannot be destroyed if it is a member of a collection. The user can clear (make empty) the Catalog, resulting in each collection being destroyed. Likewise, the user can clear the Library, but only if every individual collection is empty.
11. The program will print an error message and request the next command if a specified record or collection is not found in the corresponding container, or other problems are detected.
12. At any time, the user can save the Library and the Catalog information in a file using a supplied filename. The save file format is described below.
13. At any time, the user can restore the Library and the Catalog information previously saved in a file, replacing any current information. The ID record counter is set after the restore so that the next record created and saved by the user has a value one greater than the largest record ID found in the file, thus ensuring no conflict between old and new record IDs.
14. The user can request that the program display information about the current memory allocations, as explained more below.

15. The program is controlled by a simple command language, as 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 (record, collection, etc.) 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:

f - find (records only)

p - print

m - modify (rating only)

a - add

d - delete

c - clear

s - save

r - restore

The object letters are:

r - an individual record, or rating for the modify command

c - an individual collection

m - member for the add and delete commands

L - the Library - the set of all individual records

C - the Catalog - the set of all individual collections

A - all data - both the Library and the Catalog - for the clear, save, and restore commands

a - allocations in the print command (memory information, described below)

The possible parameters and their characteristics are:

<title> - a title string, which can be entered with redundant whitespace before, after, and internally, but is always terminated by a newline character (Return on the keyboard). The whitespace before the title (leading whitespace) cannot contain a newline character. Before being stored in a record, or compared to titles in records, the redundant whitespace (which includes any final newline character) is removed to compact the title as described above. A user can type a maximum of 63 characters to enter a title from the keyboard, including the initial whitespace separator at the beginning and the newline character at the end. After compacting, the resulting final title can be a maximum of 62 characters long, but must be at least one character long. The title is case-sensitive, so "Mars Attacks!" and "Mars attacks!" are treated as different titles. See below for how titles are to be read from the console input and files.

<ID> - a record number, which must be an integer value.

<name> - a collection name, consisting of any non-whitespace characters, but no embedded whitespace characters permitted, entered as a whitespace-delimited string with maximum length of 15 characters accepted from the keyboard. The name is case-sensitive, meaning that "Favorites" and "favorites" are two different names.

<medium> - a medium name, consisting of any non-whitespace characters, but no embedded whitespace characters permitted, entered as a whitespace-delimited string with maximum length of 7 characters accepted from the keyboard. Like <name>, the medium name is case-sensitive. Values such as "DVD" and "VHS" are customary, but the program does not require or check for any specific values.

<rating> - a rating value, which must be an integer value in the range 1 through 5 inclusive. Ratings are initially zero, meaning unrated, and are shown in the program output as the character 'u'.

<filename> - a file name for which the program's data is to be written to or saved from. This is a character string with no embedded whitespace characters, entered as a whitespace-delimited string with maximum length of 31 characters accepted from the keyboard (the platform will impose additional restrictions on possible file names).

The possible commands, their parameters, meanings, and possible errors are:

fr <title> - find and print the specified record with the matching title - the comparison is case-sensitive. Errors: title could not be read (title has less than one character); no record with that title.

pr <ID> - print the specified record with the matching ID number. Errors: unable to read an integer; no record with that ID number.

pc <name> - print collection - print each record in the collection with the specified name. Errors: no collection of that name.

pL - print all the records in the Library. Errors: none. (It is not an error if there are no records - it's a normal possibility.)

pC - print the Catalog - print all the collections in the Catalog. Errors: none. (It is not an error if there are no collections - it's a normal possibility.)

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

ar <medium> <title> - add a record to the Library. Errors: Title could not be read; a record with that title is already in the Library.

ac <name> - add a collection with the specified name. Errors: A collection of that name already exists.

am <name> <ID> - add a record to a specified collection. Errors: No collection of that name; unable to read an integer; no record with that ID number, record already a member of the collection.

mr <ID> <rating> - modify the rating of the specified record with the matching ID number. Errors: unable to read an integer (for the ID); no record with that ID number; unable to read an integer (for the rating); rating out of range.

dr <title> - delete the specified record from the Library. Errors: Title could not be read; a record with that title does not exist in the Library; the record is a member of a collection.

dc <name> - delete the specified collection from the Catalog. Errors: A collection with that name does not exist in the Catalog.

dm <name> <ID> - delete the specified record as member of the a specified collection. Errors: No collection of that name; unable to read an integer; no record with that ID number; record is not a member of the collection.

cL - clear the Library: destroy all of the records in the Library and clear the Library, but only if each collection in the Catalog is empty (has no members). Reset the record ID counter. Errors: There are collections with members.

cC - clear the Catalog: destroy all of the collections in the Catalog, and clear the Catalog. Errors: none.

cA - clear all data: first clear the Catalog as in **cC**, then clear the Library as in **cL**. Errors:none.

sA <filename> - save all data: write the Library and Catalog data to the named file. Errors: the file cannot be opened for output.

rA <filename> - restore all data - restore the Library and Catalog data from 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, resets the record ID counter, and then attempts to read the Library and Catalog data from the named file, and creates new records and collections from the file data. By the end of the restore, the record ID counter should be set to one more than the largest ID number found in the file, so that the next new record created by the user will have the next ID number in order. The final result will be to restore the program data 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, and the ID counter reset, leaving the Library and Catalog empty.

qq - clear all data (as in **cA**), and also destroy the Library and Catalog containers themselves, 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 an entire command line at once, or read the line a character at a time, and then parse it to find the commands and parameters; such error-prone work is both unnecessary and non-idiomatic when the stream input facilities will do most of 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 odd features (such as how type-ahead works), 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 the basic C Standard Library stream input functions properly, the required code is *extremely* simple - these functions do almost all the work for you! Here are the rules, explained in terms of the relevant C stream I/O functions; follow them carefully:

- With the following exception, 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 exception:
  - 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.
- The *ID number* is supposed to be an integer value. Your program should try to read this data item as a decimal integer (i.e. `scanf` with `%d`). If it fails to read an integer successfully (e.g. the user typed in "xqz" or "a12"), it outputs the error message "Could not read an integer value!". If the user had written a non-integer number such as 10.234, your program must not try to deal with it - just read for an integer, and take what you get - in this case, the value 10 (an integer). The remainder of the input is garbage, ".234" in this case, that is left in the stream to be dealt with on the next read. If this puzzles you, re-read how the `scanf` function works when reading an integer from an input stream. If you are still puzzled, ask for help. The behavior of

stream input is important to understand; previous courses might have hidden it from you. Once you have successfully read an integer, then test for validity (e.g. whether it is a valid record number).

- The *rating* is likewise supposed to be an integer value, and the same checks and considerations apply. However, once the rating value has been read, it is then checked to see if it is in range.
- Reading the *title* takes advantage of how the command language has been cleverly designed so that if a title is entered, it is always the last parameter of a command, and is also the last command parameter on a line. This makes it easy to read the title with the C Library function `fgets`, which reads and stores characters into the supplied buffer from the input stream starting from the current position up to and possibly including the next newline character, but no more than the specified number of characters will be read and stored. So, to read in a title string, leave the console input stream as it was after reading the previous data item, and simply call `fgets` with a buffer array that is 64 characters in size. `fgets` will read the rest of the line into the array, and will add the newline and null byte at the end. `fgets` always stops reading soon enough that it can add the null byte at the end, even if it has not yet read and stored the newline. So the maximum length of the stored string is 63 characters. Normally, the first character is a space (because the user normally puts a space between medium and the title), and so the final compacted title is normally at most 62 characters long. Then process the string to remove leading, trailing, and extra whitespace. If the user does not enter any non-whitespace characters before the newline, the compaction process should result in an empty string (length of zero). So after reading and compacting, do the check: if the resulting string is not at least one character long, it is a "can't read a title" error.
- Because of the way the stream input works in C (and C++), *type-ahead* is a natural result: more than one command can appear in a line of input, and the information for a single command can be spread over multiple lines; however, since a title is terminated by a newline, the title parameter for a command will always be the last parameter for the last command on a line of input. In other words, type-ahead is permitted where possible, 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 collection name and the following record ID. When whitespace is not required in the input, it is completely optional. See the posted type-ahead 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 "fr,Tobruk" will be read as a request to print the record whose title starts with a comma.
- Do not attempt to check the medium entry for meaningful or standard content like "DVD". The user may enter anything here.
- The input is case-sensitive; "A Wonderful Life" and "a wonderful life" are two different titles. Likewise, "PR" is not a valid command.
- Your input code can assume that all input text strings will fit into a character array input buffer that is one larger than the specified maximum length for the input parameter type; use `#define` symbols for these sizes.
- **Important:** 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 collection name is longer than 15 characters, it must not be allowed to overflow the input buffer whose size is 16; using `%15s` 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. Titles should be read with `fgets`, whose second parameter is the buffer array size of 64; the function will read and store a maximum number of characters that is one less than the size, and then will always store a null byte after the last character.
- **Also important**: If an input string typed by the user is too long to fit into its buffer (e.g. the user types more than 7 characters for the medium), 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 titles and names, or lots of error messages, but this is approach is safe, simple, and instructive about how stream input works.
- **Very important**: When an error is detected, a message is output, and the remainder of the input command line must be discarded. This will happen automatically if the last parameter entered and processed was a title, because all of the current line will have been read up through the newline. But otherwise, if an error is detected, the characters in the input stream up to and through the next newline are read and discarded, so as to skip the rest of the input line. Thus since type-ahead and excess whitespace (which can include newlines) are 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 if no error message is output*.
- *Clarification of reading a title and handling an error:* The behavior here is subtle even though the code is very simple. In almost all cases, after printing an error message, you skip the rest of the line. The exception is that a title is normally terminated by a newline, but even if it is, it could still produce an error - like no record of that title, or another record already has that title. If you get an error after reading what is supposed to be a title, you do **not** skip the rest of the line. This makes sense because there normally isn't any "rest of the line," and if we did skip through to the next newline, we would be discarding the command(s) on the next line, which makes the type-ahead and error-handling behavior confusing in this case of relatively

routine errors. This policy applies even if the title was too long to fit in the buffer. In this case, there is in fact something on the rest of the line, and because it wasn't skipped, and we allow type-ahead, the program will attempt to read the remainder of the too-long title as the next command, which will almost certainly be nonsense, and will cause strange errors to be reported. The spec does it this way both for instructional purposes about how streams work, and trying to keep the code simple (the code is simple, but the program behavior is odd). In practical terms, if this happened often enough to be annoying, it means we need to increase the allowed length of titles.

Save File Format and Load Input Rules

The following rules for the file format and input are explained in terms of the relevant C file stream input functions.

- The file created by the **sA** command has a simple format that can be read or edited 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 records in the Library on the first line.

  The ID number, medium, rating, and title for each record in the Library, one line per record, in alphabetical order by title.

  The number of collections in the Catalog on the next line.

  The name and the number of members for the first collection (in alphabetical order by collection name)

  The title of each member, one per line, in alphabetical order.

  Similarly, the next collection 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 (no spaces). Each line (including the last in the file) is terminated with a newline.
- Reading titles from the file: The program could have saved a title that contained all of the characters allowed, and your restore code must be able to get them all back. The title in the file will follow the space after the rating in a Library entry, or the previous newline in a Catalog entry. Your code will have to arrange to skip the separator space or the newline (e.g. with `fgetc`, and then you can read the title with `fgets` and get all of the characters in the saved title. This is easy to do if you rely on how the file is supposed to be formatted. Notice also that the title is supposed to be already compacted, so your restore code does not need to recompact the title and should not waste time doing so. But because `fgets` stores the newline at the end of the array, you will have to get rid of it after reading the title.
- When the file is read by the **rA** command, the program must protect against buffer overflow in the same way as input from the keyboard (see above). In addition, it 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, and the program will eventually detect the problem because the random information is extremely unlikely to be consistent in the way the program expects from the file format. 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 records. If the program cannot successfully read an integer as the first datum, then it knows something is wrong. Also, if the number of records is negative, the program would know something is wrong immediately, because while zero records is a well-defined possibility, a negative number of records is not. But suppose the program could read a number as the first item, and the number is positive, but then the program hits the end of file before it finished reading the expected data for this many records. Again, the program knows something is wrong. Finally, suppose the program manages to get to the collection section of the file, but finds random garbage in what are supposed to be record titles that match those found in the record section of the file. A failure to find a member title in the newly restored Library is a signal that something is wrong; this is easy to check for, and in fact, because a collection consists of a container of pointers to records, it is necessary to look up the record anyway, so this check is naturally part of that process.

  Because of these 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. fs`canf` with `%d` fails.
  - A numeric value for the number of records, collections, or members is negative.
  - An title specified for a collection member could not be found in the Library.
  - Premature end-of-file because the program is trying to read data when no more is present.

2. *The file was correctly written by a correct program in response to an **sA** command*. This means that the file will have no inconsistent or incorrect information (for example, no records with the same title). Thus the program does not have to check the validity of the data any further than in Case #1 above if it can successfully read the file. Notice that although the information is divided up into lines for easier human readability, your program does not have to read the input in whole lines except for titles 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, resets the record ID counter, and prompts for a new command. Thus the possible results of issuing a **rA** 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 Library and Catalog.
- **Hint:** Do not clutter your **rA** code with checks of the data that go beyond those described here. The autograder will not test for unspecified behavior. If your code does not pass the tests involving **rA**, chances are that you have missed some of the specified checks - such as detecting premature EOF in a certain situation. Look for that instead of writing code for checks that are not specified.

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 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: don't 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:

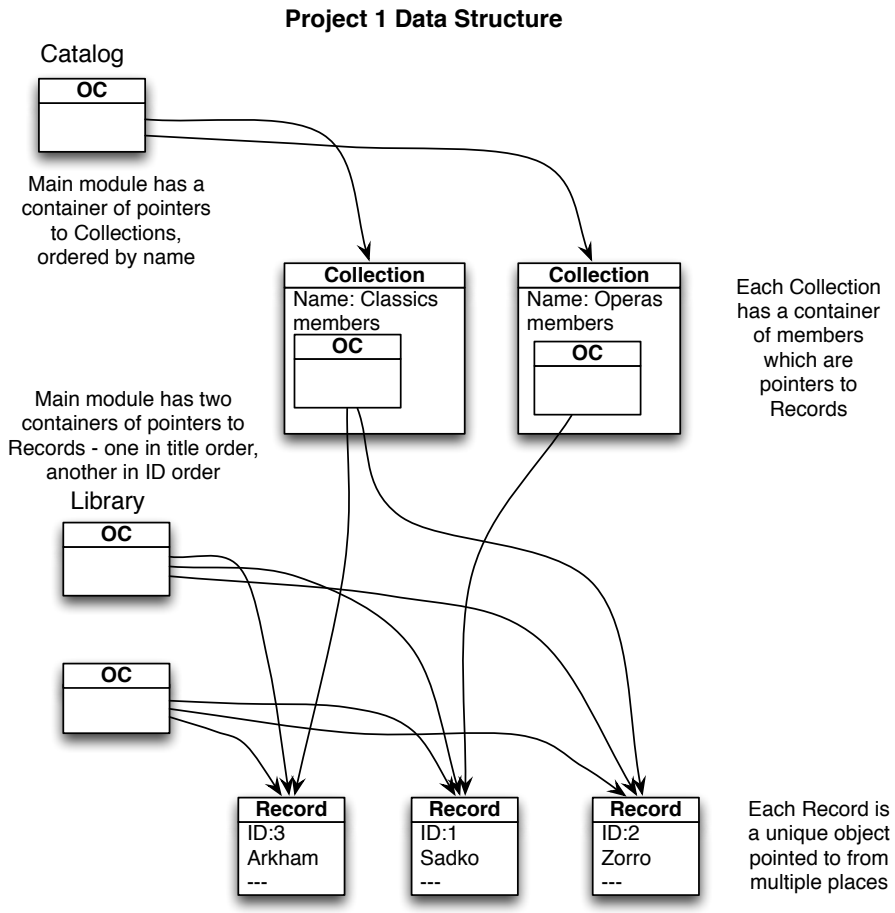
```
void foo(void)
{
    /* other code */
    int i = bar(x, y, z); // now we have a useful value for i !
}
```

Using these two features will make it easier to convert chunks of your Project 1 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 Catalog is an Ordered_container that holds pointers to Collection objects ordered by Collection name; the Library consists of two Ordered_containers that hold pointers to Record objects, one ordered by Record title, the other by Record ID number. Each Collection contains an Ordered_container that points to Record objects, ordered by Record title. Collections and



Records are referred to by pointers held in Ordered_containers, and Records are referred to with pointers in different orders. Each movie is represented by a single unique Record, pointed to from multiple places; there is no duplication of Record data.

Global Variables

Newbie programmers often use global variables to move information from one part of the program to another. The main problem with this approach is that in a program of any complexity, it is too easy to get confused about who changed the variable last, making the 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 specified or allowed in the project documents. This prohibition applies to both program-wide externally-linked global variables and file-scope internally-linked global variables.*

There are a few cases where global variables work acceptably well as reasonable solutions to otherwise horrible problems. These are cases where only one instance of the variable makes sense, the information involved cannot be sensibly moved as parameters or returned values because widely separated levels of the calling hierarchy must be crossed, resulting in severely cluttered code, and where the values conceptually can be modified in only one or two very clear places, so there is no guesswork about who is responsible for the values. The global variables specified in this project that have these properties.

To practice how to use global variables and specify the linkage involved, you will use a few program-wide global integer variables to keep track of the amount of allocated memory being used in the program. Your code should add to these values whenever memory is allocated, and subtract whenever memory is deallocated. Specifics of these variables will be described with the relevant program module in what follows.

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. The project web page will contain certain files in the *exact* form that you are to use them, such as the Ordered_container.h file. Others will be supplied as *skeleton* files, such as for Record_skeleton.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. Finally, you must write some files completely yourself, such as p1_main.c, the main module.

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 (.c) 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, which completely specifies the Ordered_container interface, and the choice of implementation is done at link time. (See the supplied makefile.) This is to give you practice with idea of distinguishing interface from implementation, and seeing how the link step really does determine which modules make up the program.

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

These generic containers are *opaque types*, meaning that the *client code* (code that uses the module) can't "see" into the container - it does not have, and does not need, any information whatsoever about how the container works or what is inside it. Conceptually, regardless of implementation, the Ordered_container contains a series of items. An item contains a data pointer to a data object that has been 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 a item pointer that points to the item in the container. An item can be removed from the container by supplying the item pointer for it. The data object can be retrieved for an item by supplying the item pointer to a `get_data_ptr` function that returns the data pointer for the data object. The only way to process all of the items in the container is to write a function that processes a data pointer, and then give a pointer to this function to one of the Ordered_container "apply" functions; the "apply" function iterates through the items in the container, in order from first to last, and calls your function with the data pointer from the item.

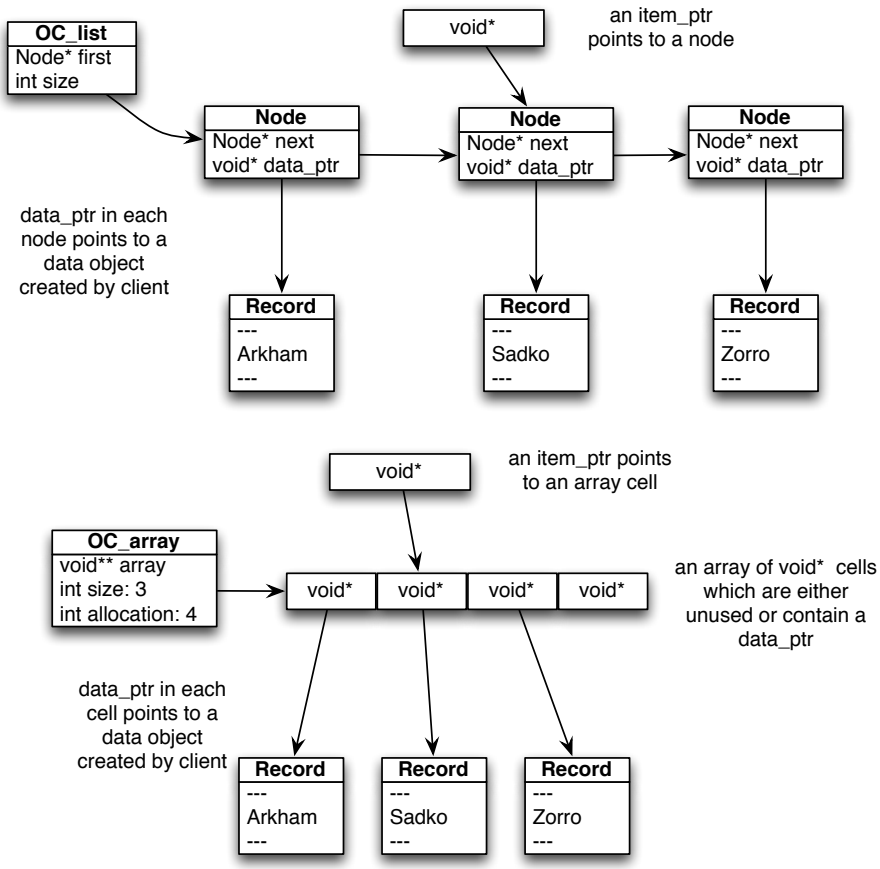
Here is how Ordered_container is a *generic* container: Since the data pointers stored in the items are `void*` pointers, they can point to any kind of data object. Your client code is responsible for creating the data objects - of whatever type they are - and putting void pointers to them into the container. 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*`.

Ordered_container List and Array Implementations
using the Record list as an example



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 has to be 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 (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 data objects when you convert the code to a C++ container class template 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 `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 array should be pointed to by a simple `void**` pointer (which 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 based on the K&R code that in addition to returning whether the item was found (or not) also takes an `int*` parameter and uses it to return the array index where the item is, or 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. For this project, the initial size of the array is 3 cells. When the array is full, a new array is allocated whose size is double the space required to hold the original array plus the new value (new size = 2 * (old size + 1)). This scheme can waste some memory space, but results in fairly fast performance because as the container is filled, fewer new allocation/copy/deallocate operations are required. But the array normally will have cells that are not currently in use; so this implementation has to keep track of how many cells are *in use* to hold items, which is returned by the `OC_get_size` function, and how many cells are *currently allocated* - the size of the current allocated array. To remove an item, the items that come after the removed one are moved "up" by one, but the array is not reallocated - it retains its original size. The only time the array is "shrunk" is with the `OC_clear` function, which discards the entire array and starts over with the initial small allocation.

General implementation issues. In both implementations, the container is always in sort order because items are immediately placed in the correct position; it is never necessary to sort the whole container. The only way to add items to the container is in order. 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 an item is found, and then the container is modified (by adding or removing either the same or a different item) 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 `Record 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 titles in records. The `OC_insert` function returned value

should be used to determine whether the insertion failed because the item was already present in the container; typically the new data object should be destroyed if so.

- To find an item in the container, 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 record in a container can be done with a data object containing only the sought-for title. `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. For example, if you have a title string, you can find a matching record by supplying the title string as the argument, and use a comparison function that compares the supplied title string to the title in the record.
- 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 record is deleted from a container of records, the record object itself must also be deleted. However, removing a record from a collection does not mean that the record is deleted from the Library 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 record in a container of records can be printed using a function that interprets the `void*` data pointer as a pointer to a `Record 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 `Record` pointer as the first argument, and some other pointer type (e.g. a `FILE*`) as a second argument, for every record 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. 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.

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 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, not helping! 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 `Record*`) to and from the `void*` type when interacting with the container. The client code gives the container a data pointer when calling the `OC_insert` function. The container stores the data pointer in one of its internal "items". The item is *list node* in the list implementation, and an *array cell* in the array implementation.

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

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

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

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

Record.h describes the interface functions for creating, destroying, printing, saving, and loading records, along with accessor functions for the data in a record. Note that once created, there is no support for modifying the ID, title, or medium in a record.

Collection.h shows the corresponding functions for a collection. As shown in the skeleton Collection.c file, a collection has a name and an Ordered_container of for the members of the collection. These are supposed to be pointers to records. The interface for the Collection module does not have any functions for accessing the container directly; instead there are functions which operate on the container as needed.

You must use these two 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 Record.c and Collection.c.

Any functions not declared in the header files for a module 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 specified header file; all other functions are not part of the interface, and must be inaccessible from outside the module.

The record and collection structs must contain `char*` pointers to C-strings for the record medium and title and the collection name. Creating an record or collection means allocating memory for the struct object itself plus the relevant strings and copying in their supplied values. Deleting one of these structs requires first deallocating the memory for the strings as well as the object struct itself.

Note that the container of records in a collection must contain simply pointers to record objects that are also pointed to by the Library containers. Thus, an individual record's data must be represented only once, in a record object created when that movie is added to the Library. Thus the members in a collection simply point to the corresponding record objects, and removing a record from a collection does not result in removing that record from the Library nor destroying that record object.

p1_globals.h, .c. To practice how to use global variables and specify the external linkage involved, in addition to the global variables in Ordered_container.h, you will use an additional global integer variables in this module to keep track of the number of bytes allocated to store string data in the program. Your code should add to these values whenever memory for strings is allocated, and subtract whenever the memory is deallocated. The **pa** command outputs the current values, as well as accessing the containers to determine how many Records and Collections currently exist. See the sample output for the format. The variables that track OC_container memory use are part of the OC_Container module, and are discussed above.

Because I will be testing your Ordered_container implementations, and your Record, Collection, and main modules mixed with mine, this program-wide global variable must have the same type (int) and name throughout all the code. The customary way to ensure

consistent declarations of global variables is the scheme described in lecture and the handout. There is a p1_globals_skeleton.h file that contains the required variable name for the string memory tracking variable. Fill in the rest of the file following the described scheme and rename it to "p1_globals.h" for use in your project. It must contain the declaration, in the form presented in the course materials, to allow the global variable to be accessed from any other module in the program (external linkage) simply by #including this file. The file p1_globals.c then defines the global variable.

If you decide to define the additional global variables allowed in this project specification (discussed below), you must place their declaration and definition in this module. The files p1_globals.h and p1_globals.c *must not contain any other variables, functions, definitions, or declarations*. Any other content is a violation of this specification. Your p1_globals module will always be used together with your code, so if you use the additional global variables, they will always be available to your modules.

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 the subfunctions in a reasonable human-readable order (see the Coding Standards).

Refer to the first Figure. Your `main` function must create *three* Ordered_container objects, one for the Catalog, containing pointers to Collections, and *two* for the Library because the Library is accessed both by record title and record ID number. Thus one container of Record pointers must be ordered by the title, and a second container of Record pointers ordered by record ID number. The different commands search the appropriate container depending on whether the command requires the record title or the record ID number.

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

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

Utility.h, .c. This module is for functions, definitions, or declarations that are shared between the modules or used in more than one module. You must supply these files with your project even if they are empty.

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

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

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

Combining Type-Safe and Generic Code

Use actual Record and Collection pointers wherever possible, use void* only in Ordered_container and its interface. As specified in Record.h and Collection.h, Records and Collections are referred to using pointers of the appropriate type: `struct Record*`, `struct Collection*`. 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_Record` function is declared in Record.h as requiring a `struct Record*` pointer as a parameter. If you try to call `print_Record` with a `struct Collection*` 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 Records and Collections. For example, the `print_Record` function would have a `void*` parameter, and so would the `print_Collection` function. In this case we could accidentally call `print_Record` with a pointer that actually points to a Collection, not a Record. 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 Record*` and `struct Collection*` 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 Records and Collections. We could handle this by pointing to Records and Collection 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 Record and Collection 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 Records, Collections, 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 Record*` 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 Records and Collections; 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 are not 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 Record or Collection interface, and are specified there as taking some combination of `struct Record*`, `struct Collection*`, or `FILE*` parameters. Your code must do the following:

- In at least two cases of applying Record or Collection 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_Collection` to a `void(*)(void*)` function pointer type and by casting `save_Record` to a `void(*)(void*,void*)` function pointer type.
- In at least two cases of applying Record or Collection 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_Record` and a wrapper for applying `is_Collection_member_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 Record or Collection 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 C under the restricted C99 Standard described on p. 7 above. You are free to use any facilities in the Standard C library, and are expected to do so whenever appropriate - unnecessarily duplicating the Standard Library 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. These extra functions actually turn out to be of very little value for the projects in this course. Check the handouts section of the course web site for lists of 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 each and 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. This applies to both program-wide and internally-linked global variables.

4. You may define global "variables" as constants to hold output message strings in the files p1_main.c, Record.c, and Collection.c (see the C Coding Standards). Each module should define only the strings that it outputs (see strings.txt) and these const 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 records and collections must be held in dynamically allocated memory, using the minimum allocation that will correctly hold the strings. Similarly, Record, Collection, and Ordered_container objects and their items must be dynamically allocated with the correct size.

7. Your code must detect an failed memory allocation, and print a message of your choice and terminate immediately by calling `exit`. This message can be printed to either `stdout` or `stderr`; your choice.

8. 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. Follow the advice in lecture about how to make memory management problems less likely.

9. Unnecessary memory allocations are egregiously inefficient, and error-prone, and must not be done. In particular, character array buffers to temporarily hold input strings should be declared as local variables, not dynamically allocated chunks of memory. Likewise, when searching for an existing object, do not create a Record or Collection object simply to search for it; that's what the `find_arg` function is for. Create new Records or Collections 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`, `fscanf`, and `fgets` for all I/O of output messages and record and collection information. For `fscanf` and `scanf`, use format specifications of `%c`, `%d`, and `%s`, and for `printf` and `fprintf`, `%d` and `%s` will suffice. You can use the `getchar` function to skip unwanted characters in the input as part of error recovery. 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 should implement the command language using `switch` to select a function to execute given the individual command letters - this is what `switch` is for - and the compiler can often optimize the code to be much faster than the corresponding structure of `if-elses`. A mess of unnecessary `if-elses` is ugly and a sign of poor code quality when a `switch` can be used. Note that you can and should, for this project, embed a `switch` inside a `case` of another `switch` — thus a *two-level* `switch` structure is required in this project. Each `case` should then *call a function* that does the actual work of the command. Include `default` cases in the switch to handle unrecognized commands. Because the quit command **qq** should result in a return from main, it does not have to be implemented as a separate function. The result will be a simple and clear (though somewhat 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 some beginning programmers suffer from – do not use function pointers except where specified in this project. Function pointers are a way to implement a facility that *must* be generic; but if over-used the result is cryptic obfuscated code with no advantages over simple function call trees — especially in the main module.

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 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 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. 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.

17. 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.

18. 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 p1_globals 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_Record()` in Record.c and `load_Collection()` in Collection.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 the setup would be much simpler.

19. 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 and the program recognizes it. I will test 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 (e.g. **qq**) and yours should too! When you do input from a file, checking for end-of-file is a central part of the logic (be sure you do it correctly!). But for interactive programs, it just creates incredible clutter in the code; it isn't idiomatic.

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 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 following the specifications. Your Utility module will be used with your code in all component tests.

I will do a human-graded code quality evaluation on this project, so pay attention to the guidelines presented in the C Coding Standards document and the general concepts for code 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. Review this document, and the course and web page material on quality code organization and code design.

How to Build this Project Easily and Well

Recommended order of development

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

1. Survey what kinds of work the project code will need to do, so you can anticipate what might be useful Utility functions - either write them first, or be ready to create them as soon as you find that you need the same code twice. For example, you'll be dealing with a lot of C-strings stored in allocated memory. 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 Record 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_Record should give you a pointer that you can give to print_Record, and it should output what you gave create_Record. Again, if this component is completely debugged, it will make testing and debugging the rest of the project *much* easier.
5. Next do the same approach with the Collections module - only in this case, your test setup will need to include the Record module and the Ordered_container module. Get this module working, and all that's left is the main module, and you have working components to build it with!
6. For the main module, build the command functionality a bit at time, testing and debugging as you go. It is always easier to work with a small program, and with only a little bit of new code at a time. For example, a good place to start is with the commands for working with records in the Library, such as first adding a record and then finding or printing a record.
7. As you add commands, watch for opportunities to *refactor* the code = *improve the design of existing code*. Change some of your code into functions that make you can then re-use for the next commands. In fact, the main module should consist of a structure in which the switch statements call a function for each command. This command function handles all of the work for the command. It reads the input from the user by calling other functions, and then calls other functions to do the actual work of the command. The patterns and regularity in the input syntax means that for example, you can define a function that reads a record number and returns a pointer to the record, doing the required error checks for successful read of an integer and successful lookup of a record. Thus the command function only needs to call this one function to get the user's desired record. 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 do - mostly just calling already-debugged functions in new combinations. This makes coding more fun!
8. On the other hand, if you find yourself repeatedly copy-pasting a chunk of code for a command, you are doing it wrong! Stop it! Take a break, and then refactor the code and do it right! This is the most important code quality issue in this project and in your future as a programmer!
9. Reserve the save/restore command for last. Write the save function first, and examine the resulting file with a text editor - remember it is formatted to be human-readable. Try the save function out in a variety of situations and verify that the contents of the file are correct. Especially check boundary conditions - e.g. no records are in the Library, or some collections have no members. Only when you are satisfied that the save file is correct, should you try to get the restore command working. You should not have a "god" function in p1_main that reads all of data and then stuffs it into records and collections. Rather, the details of creating a record or collection from the file are supposed to be delegated to the Record and Collection `load` functions.

Overall suggestions

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 Records and Collections 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 — see the C Coding Standards for recommendations on error return values. 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 - Records, Collections, 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 Collection?

Create a good 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) 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 essay 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. Copying *all* of the very lengthy 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 a 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 possibly 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!