

# Project 1

## A Micro Media Manager in C using Encapsulated Types and Generic Containers

Due: Friday, Jan 29, 2010, 11:59 PM

### Notice:

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

## Introduction

### Purpose

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

- Quality procedural program organization with a well-designed function hierarchy for clarity, maximum reuse, and ease of coding and debugging.
- Program modularization using separate compilation and header files that declare module interfaces, and writing modules that interacts with other modules only through their interfaces.
- Using opaque types to hide implementation details from client code, and demonstrating this with two implementations of the same interface.
- Writing code in C to implement and use a generic data structure or container, requiring use of void \* pointers and function pointers, to point to dynamically allocated objects, including using multiple containers that point to the same objects.
- Using C techniques for "object-based" programming with good encapsulation and insulation. These techniques are in common use in well-organized modern C software and provide some of the advantages of object-oriented programming with this non-object-oriented language.
- Linkage for global variables and file-local functions.
- Using a static local variable.
- 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 names (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 reuse much of your code for this project in the later projects, making it more idiomatic and reorganizing it as necessary.

### Overview of this Document

There are two major sections: The *Program Specifications* describe what the program is supposed to do, and how it behaves. The *Programming Requirements* describe how you have to program the project in terms of the structure and organization of the code, and which specific techniques you must apply - remember the goal is to learn and apply some concepts and techniques for programming, not just hack together some code that behaves right. At the end is a section of advice on *How to Build this Project Easily and Well* - worth taking seriously!

## Program Specifications

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

1. Throughout this course, "alphabetical order" for a string means the standard English language order defined by the C and C++ Library functions for ordering character strings (`strcmp`, `std::string::operator<`). In this order, lower case 'a' follows upper case 'Z'.
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 consecutive whitespace characters shrunk to a single space character (' '). For example, the entry " Much ado about Nothing " would be trimmed at both ends and extra spaces compacted so that it is stored as "Much ado about Nothing". Note that the newline character is also whitespace and would be trimmed from the end. 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 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 to save having to type the entire title. The ID number cannot be modified by the user. The first record 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, but 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.
5. 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.
6. 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.
7. 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.
8. The Catalog consists 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.
9. 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.
10. 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.
11. At any time, the user can save the Library and the Catalog information in a file using a supplied filename. *The record ID numbers are not saved.* The save file format is described below.
12. At any time, the user can restore the Library and the Catalog information previously saved in a file, replacing any current information. *Each record is given a new ID number corresponding to the order that it is created from the input file.*
13. The user can request that the program display information about the current memory allocations, as explained more below.
14. 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). Before being stored in a record, or compared to titles in records, the redundant whitespace is removed to compact the title as described above. The user can type a maximum of 63 characters to enter a title from the keyboard. The title is case-sensitive, so "Mars Attacks!" and "Mars attacks!" are treated as different titles.  
**<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: 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: 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: 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). 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. Note: record ID numbers are not saved. 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, and then attempts to read the Library and Catalog data from the named file, and creates new records and collections accordingly. This should restore the program state to be identical to the time the data was saved, except that the record ID numbers will be different. If an error is detected during reading the file, the error is reported and any data previously read is discarded, 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 `scanf` and `fgets` properly, the required code is *extremely* simple - these functions do almost all the work for you! Here are the rules; 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 an 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.
- The command language has been cleverly designed so that if a title is entered, it is always the last parameter of a command. So, to read in a title string, simply call the C Library function `fgets`, which reads and stores characters from the input stream starting from the current position up to and including the next newline character. Then process the string to remove leading, trailing, and extra whitespace.
- 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`, which includes its own way to protect against buffer overflow (check the handout or a C Library reference).
- 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 confusing error messages, or messed-up titles and names, but this is an approach that is both safe and simple.
- **Important:** When an error is detected, a message is output, and the remainder of the input command line is discarded if necessary. That is, if the last parameter entered was a title, all of the current line will have been read up through the newline, so no action is needed to discard the rest of the command line. Otherwise, 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 any other time.

### Save File Format and Load Input Rules

- The file created by the `SA` command has a simple format that can be read by a human easily enough to simplify debugging the saving and loading code. See the posted samples. It consists of a series of lines containing information as follows:
  - The number of records in the Library on the first line.
  - The 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.
- 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. But suppose the program could read a number as the first item, but suppose that number is large, and then the program hits the end of file before it finished reading the expected data for this many 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.

Because of the considerable constraints present in the file format, it is reasonable to rely just on this level of checking to detect that an invalid file was specified. Thus, your program should check only for the following specific input errors when reading the file:

  - An expected numeric value could not be read because a non-digit character is present at the beginning of the expected number -i.e. `scanf` with `%d` fails.
  - An expected title for a collection member could not be found in the Library.
  - Premature end-of-file because the program is still trying to read data that is not present - i.e. `scanf` returns EOF unexpectedly.

2. *The file was correctly written by a correct program in response to an earlier 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 buffer overflow; Case #2 does not justify ignoring the possibility of buffer overflow if an invalid file is being read.
- If the program detects invalid data, it prints an error message (which is the same regardless of the cause) and then it deletes all current data that it might have loaded before the problem became apparent, and prompts for a new command. Thus the possible results of issuing 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.

## Programming Requirements

For all projects in this course, you will be supplied with specific requirements for how the code is to be written and structured. The purpose of these specifications is to primarily to get you to learn and practice certain concepts and techniques, and secondarily, to make the projects uniform enough for meaningful and reliable grading.

So, if the project specifications state that a certain thing is to be used and done, then your code must use it and do it that way. If you don't understand the specification, please ask for clarification, but if you ask whether you really have to do it the specified way, the answer will always be "yes." If something is not specified, you are free to do it any way you choose, within the constraints of good programming practice and any general requirements. For example, this project must be done in ANSI C (1989), so you may not use C++ I/O operators in it.

### Program Modules

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

***Ordered\_container.h, Ordered\_list.c, Ordered\_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_list.c`) the other using a dynamic array (`Ordered_array.c`). The course web site and server contains a header file, `Ordered_container.h`, which you must use and `#include` in your two implementation files. `Ordered_container.h` contains the function prototypes for the external interface of this module. You may not modify the contents of this header file - your `.c` files must supply all of the specified functions and behavior, and you may not add any others to the header file. All other functions and declarations in your implementation files must have only internal linkage.

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

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

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

the item pointer to a `get_data_ptr` function that returns the data pointer for the data object. The only way to process all of the items in the container is to write a function that processes a data pointer, and then give a pointer to this function to one of the Ordered\_container "apply" functions; the "apply" function iterates through the items in the container, in order from first to last, and calls your function with the data pointer from the item.

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

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

**Ordered\_list.** In the linked-list implementation, the container is a linked-list and the items are list nodes that contain a void \* data pointer. The nodes are designated to the client by casting node pointers to/from void \*. The list is searched linearly to locate items, and to locate where to insert a new item. This means that searching and inserting is linear with the number of items (size) of the container ( $O(n)$ ). The `OC_get_size` function returns the number of items in the container, which is simply the number of nodes in the list. The `OC_clear` function destroys each node and re-initializes the list to be empty.

**Ordered\_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 or index of a cell to/from a void \*. The array is searched using binary search to locate items, so that searching will be logarithmic with container size ( $O(\log n)$ ). Insertion should use binary search to locate where to insert a new item, but the next step requires moving the items that come after the new item "down" by one to make room for the new item. This means that the insertion time is overall linear, even if binary search is used to speed up the first step.

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 =  $2 * (\text{old size} + 1)$ ). 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.

**Implementation issues.** In both cases, 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.

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

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

To build an executable using one of the implementations of Ordered\_container, build the executable with the .c file for the implementation that you want. To use the other implementation, simply use the other .c file in the build. In an IDE, just add the desired .c file to the project, and remove the other. A makefile can be easily constructed to allow you to choose on the command line which implementation to use - see the starter makefile for the project. Notice that you cannot build a single executable containing both

implementations because of rampant violations of the One Definition Rule - all of the interface functions will be double-defined, not to mention the two different declarations of `struct Ordered_container`.

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

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

- An `Ordered_container` object is created by the `OC_create_container` function which returns a pointer to the opaque `struct` type. The create function is also given a function pointer to the comparison function, which is stored in the container `struct`.
- Data is added to the container by creating a data object (e.g. a `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.
- To find an item in the list, the `OC_find_item` function is called with a `void *` pointer to a data object containing at least the data required by the comparison function. For example, finding a 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. 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. 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.

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 with correct pointer values only. However, follow the guru advice of using the `assert` macro in the functions to help protect yourself against the most common and confusing programming errors that you might make during development - such as accidentally calling with a zero pointer argument.

**`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 (and convenient typedefs). 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 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 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 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.*** 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. But there are a few cases where they work acceptably well as reasonable solutions to otherwise horrible problems. These are cases where only one instance of the variable makes sense, the information involved cannot be sensibly moved as parameters or returned values because widely separated levels of the calling hierarchy must be crossed, resulting in severely cluttered code, and where the values conceptually can be modified in only one or two very clear places, so there is no guesswork about who is responsible for the values.

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

Note that for the linked-list implementation, the number of container items in use and the number of items allocated will be identical; the two global variables will be incremented or decremented together. 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, the 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. There is a `p1_globals_skeleton.h` file that contains the required variable names for all of the global variables. Fill in the rest of the file following the described scheme and rename it to "`p1_globals.h`" for use in your project. It must contain the declarations, in the form presented in the course materials, to allow the global variables to be accessed from any other module in the program (external linkage) simply by `#including` this file. The file `p1_globals.c` then defines the global variables. The files `p1_globals.h` and `p1_globals.c` *must not contain any other variables, functions, definitions, or declarations*. Any other content is a violation of this specification.

***p1\_main.c.*** The main function and its subfunctions must be in a file named "`p1_main.c`". The file should contain function prototypes for all of these subfunctions, then the main function, followed by the subfunctions in a reasonable human-readable order.

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. When I test your modules separately or in combination with mine, I will combine yours with your Utility files as well, so your code can always use your Utility.

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

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

## Other Programming Requirements and Restrictions

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

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

3. There must be no global variables that are not specifically required or allowed by the project specifications.

4. You may define global variables as constants to hold output message strings in the files `p1_main.c`, `Record.c`, and `Collection.c`. Each module should define only the strings that it outputs (see `strings.txt`) and these variables should have internal linkage.

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. In fact, because their prototypes are not in a header file, they could be called only with obviously flakey hard-wired 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. Your code must detect a failed memory allocation, and print a message of your choice and terminate. This message can be printed to either `stdout` or `stderr`; your choice.

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

8. 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.

9. Use a static local integer variable in the `create_Record` function for generating the record ID number.

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.

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 embed a `switch` inside a case of another `switch`. Each case should then *call a function* that does the actual work of the command. Include default cases in the switch to handle unrecognized commands. 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.

12. The output messages produced by your program must match exactly with those supplied in the sample output and the supplied text strings on the project web page. Copy-paste the text strings into your program to avoid typing mistakes, and carefully compare the output messages and their sequence with the supplied output sample to be sure your program produces output that can match our

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

13. 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 am testing your programs with redirected input from files just for convenience, not as part of the concept of its operation. Our 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. Note that if your program reading and error recovery logic is incorrect, it might not manage to read the quit command, and so will hang in a loop complaining about the input. The autograder will eventually time it out or kick it off because it writes too much output.

## Project Grading

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

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_array` and `Ordered_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 a 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 - changing some of your code into functions that make you can then re-use for the next commands. If you do a good job with this, the commands will become increasingly easy to do - mostly just calling already-finished subfunctions in new combinations. This makes coding more fun!
8. 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

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

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: it involves special-casing the number of nodes in empty lists; more seriously, it will lead to the creation of bogus objects when you convert the code to a C++ templated container class in the next project.

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

An important code design concept is that of division of responsibility. In this project, a variety of objects are being created, destroyed, and manipulated - 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?

Beware especially of copy-paste coding - too easy for bugs to creep in. This project has been designed so that the main module can be written pretty easily with a few good "building block" functions and the functions specified in each module. Absolutely do not write or paste the same code over and over again for each command. In general, watch out for places where code duplication (and attendant possible bugs) can be avoided by simple "helper" functions. Identify them early, and save some time! In general, if you find yourself writing code that does the same thing twice, change it into a function - see the C Coding Standards for more discussion of when duplicated code should be turned into a function. Creating functions for duplicated work cuts down on errors, makes debugging easier, and clarifies the code. This is a basic principle of well-written code, and this first project is when you should start doing it. Copy-paste coding, and the mess that comes with it, is the most common code-quality problem students have on the first project in this course. The grade consequences are pretty serious, so breaking the habit on the first project is an extremely good idea.

Finally, plan your work on the project with a goal of having a working version at least a day or two before the first deadline. Try to write good-quality well-organized code as you go - it will actually save time in this relatively complicated project. If things go poorly, you've still got a chance of meeting the final deadline. But if they go well, you can then take a bit more time to further improve your code quality and program organization. Not only will a higher code quality score result, but you'll be able to more easily recycle your code for later projects.