

Cuyahoga.com Database

For this project you will implement a very simple database of the sort that might be used by a company that sells books. The company maintains, in different files, records about authors and records about books. We will call these the author database (dB) and book database files, respectively. The syntax of these files is described in a later section.

On startup, the program will read in two dB files, one for author information and one for book information. The program will use the data read to initialize two in-memory data structures, which we will refer to as the author dB and the book dB. Each of the in-memory databases will be stored using a array template, which will maintain its contents in sorted order. For the author dB, the records will be sorted by author ID; for the book dB, the records will be sorted by ISBN. When a new record is added to either dB, it must be added in the correct location for the given sort order for that dB.

Then the program will read a script file and carry out the commands it contains, writing any output to a log file. The names of the two dB files, the script file and the log file will be specified as command-line arguments to the program.

Program Invocation:

Your program **must** take the names of the input and output files from the command line — failure to do this will irritate the person for whom you will demo your project. The program will be invoked as:

```
dB <book dB> <author dB> <script> <log>
```

If one of the specified input files does not exist, the program should print an appropriate error message and either exit or prompt the **user** for a correction.

Book Database File Description:

A book record consists of six fields: ISBN, category label, title, author ID, units sold, and year of publication. There will be one book record per line, and the components of a book record will be tab-separated, with a single newline following the publication year. The number of lines is arbitrary. There will be no empty lines. Here is a sample book database file:

```
0-345-43481-1  FICTION  The Last Full Measure  S0001  13  1998
0-812-55138-0  FICTION  The Skystone           W0001  19  1996
0-8041-1188-X  MYSTERY  Defend and Betray     P0001  105  1992
0-380-79856-5  FICTION  The Ape Who Guards the Balance  P0002  217  1998
0-06-102070-2  FANTASY  The Light Fantastic    P0003  98   1986
. . .
```

Author Database File Description:

An author record consists of two fields: author name and ID. There will be one author record per line, the components will be tab-separated, and there will be a single newline following the author ID. The number of lines is arbitrary; there will be no empty lines. Here is a sample author database file:

```
Peters, Ellis  P0005
Shaara, Jeff   S0001
Whyte, Jack   W0001
Perry, Anne   P0001
Peters, Elizabeth P0002
Pratchett, Terry P0003
. . .
```

Full versions of both input files are available on the Projects page of the course website.

Script File Description:

As usual, lines beginning with a semicolon (` ; `) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the input file.

Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of “tokens” which will be separated by single tab characters. **Bold** text indicates command keywords that will be used verbatim. Tokens will never contain a tab character. A newline character will immediately follow the final “token” on each line.

addBook <ISBN> <category> <title> <author ID> <units sold> <year>

If there is an author in the author dB with the specified ID, this causes the addition of a new book record to the book dB. If the author dB does not contain a matching ID, then the book is not added to the book dB. Of course, there must be room in the book dB for another record. If there is not, then the book is not added. **If the book dB already contains a record with the given ISBN, then the book is not added.**

addAuthor <name> <author ID>

This causes the addition of a new author record to the author dB. Of course, if there is no room in the author dB for another record, then the author is not added. **If the author dB already contains a record with the given author ID, then the author is not added.**

delBook <ISBN>

If there is a book in the book dB with the specified ISBN, this causes the deletion of that book record from the book dB. If the book dB does not contain a matching ISBN, then the deletion fails.

listBooks

listAuthors

These cause a nicely labeled and formatted display of all the contents of the relevant dB to be written to the log file. These should be arranged in a more readable and attractive format than the database files. **You must include, with each printed record, the index at which that record is stored.**

updateSales <units sold> <ISBN>

If there is a book in the book dB with the specified ISBN, this causes the sales figures for the relevant book to be increased by the given number of units. **If** the book dB does not contain a matching ISBN, then the update fails.

findAuthor <author ID>

This causes the author dB to look for the record for the specified author. If no record is found, an error message is logged. If the record is found, log the author's name, a list of all the books from the book dB that s/he has written, and the total sales for all books by that author.

saveBooks <file name>

saveAuthors <file name>

These cause your program to write the specified in-memory database to disk, taking the given name and appending the extension ".dB" to it. You must use the same format for the saved files as is specified above for the initial book and author database files. In neither case will the corresponding in-memory database be cleared. If, for some reason, it is impossible to create the output file, then the save operation will fail, and an error message will be logged.

exit

This causes your program to deallocate all dynamic memory and then terminate immediately. The current in-memory databases are not automatically saved to disk. The script file is guaranteed to contain an `exit` command.

Legend: in the commands above:

<ISBN>	a string which is a unique identifier for a book
<category>	a string (FANTASY, FICTION, MYSTERY, NONFICTION, REFERENCE, SCIFI)
<title>	a string
<author ID>	a string which is a unique identifier for an author
<units sold>	a non-negative integer
<year>	a non-negative integer
<name>	a string
<file name>	an alphanumeric string containing no whitespace characters

Note that the ISBN is a primary key for book records, and the author ID is a primary key for author records. A primary key is a field for which different records are not allowed to contain duplicate values.

Each command must be echoed to the log file, along with an informative message indicating the results when the command was processed. The output from each command should be delimited in some manner, similar to the parsing homework assignment.

You may assume that the input file will conform to the given syntax, so syntactic error checking is not required. However, it is certainly likely that search commands may specify non-existent books or authors and your program must deal with that gracefully. If an error occurs during the parsing of the command file, there's an error in your code. However, your program should still attempt to recover, by "flushing" the current command and proceeding to the next input line. Here is a very simple script file:

```
; Sample input script for dB project.
;
listbooks
listauthors
;
addBook      0-7868-8930-6      FICTION      Cimmaron Rose      B0001  389700      1997
findAuthor   B0001
;
delBook      0-06-102070-2
findAuthor   P0003
;
addAuthor    Gardner, John      G0003
addAuthor    Gardner, John      G0004
;
; That's right, two different authors with identical names.
;
updateSales  3      0-380-79856-5
findAuthor   P0002
;
saveBooks    Books01
saveAuthors  Authors01
;
exit
```

Check the website after Spring Break for additional samples.

Design:

For this project, we are providing a list of some required and some recommended classes in the project specification. You are expected to conform to this design. We will not, however specify all details of the relationships among these objects.

The determination of class relationships, and of reasonable attributes and responsibilities for each class is largely left to you. That does not mean that all decisions are equally good, nor that all decisions will receive full credit. Your goals should be to place responsibilities where they most logically belong, and to provide for maximum cohesion and reusability. We may choose to address the specific implications of these goals for this project in class.

The following classes are required (the given names are not mandatory):

- There must be an Array template for the in-memory databases. The template must allocate its array dynamically, but the array will be fixed in size once it is created. The size of the array will depend on the number of records in the relevant initial dB file; the array dimension must be 1.25 times the number of records in the dB file. The template must be a pure container; that is, it must know nothing about the data members and virtually nothing about the public interface of the data type actually stored.

The template will make the following assumptions regarding the data type used:

- There will be a default constructor.
- If necessary, there will be adequate support for deep copy operations and dynamic deallocation.
- The data type will overload at least the equality and less-than operators: == and <.

The template will perform absolutely no file I/O operations. The template will have responsibility for managing insertions and deletions to preserve the correct ordering of the data elements it stores. The template will provide deep copy operations for itself, and a suitable destructor.

- There must be a Book class and an Author class. Each must encapsulate the appropriate data, and provide for the expectations of the array template stated above.
- There must be a Database class that encapsulates the book and author databases and serves as an interface for all accesses to and modifications of those databases. For example, the Database object would be told to add a book and provided the appropriate parameters; the Database object then has the responsibility of using the book and author databases to carry out the addition. The Database class will do absolutely no file input, but it is permitted to do file output (preferably through an associated object, but directly is acceptable).
- There must be a Manager class with responsibilities similar to that of the Manager in the first project. The Manager has the responsibility of interpreting commands from the script file, but not of actually carrying them out, with the possible exception of the exit command. The Manager class will do absolutely no file input directly, but it is permitted to do file output (preferably through an associated object, but directly is acceptable).
- There must be a ScriptInput class with the responsibility of serving up commands from the script file. The ScriptInput class may perform limited parsing (tokenization) but it should not interpret commands. This is essentially the same as the FileManager from the first project.

The following classes are recommended but you will not be penalized for not using them:

- A LogOutput class to manage all writing to the log file.

Some additional explicit requirements are:

- You may not use inheritance.
- You may not use any STL container templates instead of the specified array template.

Log File Description:

Since this assignment will be graded by TAs, rather than the Curator, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct. The first few lines should contain your name, course (CS 2704), and project title.

The remainder of the log file output should come directly from your processing of the script file. You are required to echo each command that you process to the log file so that it's easy to determine which command each section of your output corresponds to. You are also required to delimit the output from each command. A sample log file that corresponds to the initial database files and script file given earlier **is available on the course website**.

Submitting Your Program:

You will submit a zipped file containing your project to the Curator System (read the *Student Guide*), and it will be archived until you demo it for one of the GTAs. Instructions for submitting are contained in the *Student Guide*. You will find a **generic** list of the required contents for the zipped file on the course website. Follow the instructions there carefully; it is very common for students to suffer a loss of points (often major) because they failed to include the specified items.

Be very careful to include all the necessary source code files. It is amazingly common for students to omit required header or cpp files. In such a case, it is obviously impossible to perform a test of the submitted program unless the student is allowed to supply the missing files. When that happens, to be fair to other students, we must assess the late penalty that would apply at the time of the demo.

You must also include an MS Word doc containing a revised class diagram and revised class forms reflecting your final design.

You will be allowed up to five submissions for this assignment, in case you need to correct mistakes. Test your program thoroughly before submitting it. If you discover an error you may fix it and make another submission. Your last submission will be graded, so fixing an error after the due date will result in a late penalty.

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

Programming Standards:

The GTAs will be carefully evaluating your source code on this assignment for programming style, so you should observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Evaluation:

You will schedule a demo with your assigned GTA. The procedure for scheduling your demo will be **the same as for the first project, although the partitioning of students among GTAs will be changed**. At the demo, you will download your submitted project, perform a build, and run your program on the supplied test data. The GTA will evaluate the correctness of your results. In addition, the GTA will evaluate your project for good internal documentation and software engineering practice.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your main source code file.

```
// On my honor:  
//  
// - I have not discussed the C++ language code in my program with  
//   anyone other than my instructor or the teaching assistants  
//   assigned to this course.  
//  
// - I have not used C++ language code obtained from another student,  
//   or any other unauthorized source, either modified or unmodified.  
//  
// - If any C++ language code or documentation used in my program  
//   was obtained from another source, such as a text book or course  
//   notes, that has been clearly noted with a proper citation in  
//   the comments of my program.  
//  
// - I have not designed this program in such a way as to defeat or  
//   interfere with the normal operation of the Curator System.  
//  
// <Student Name>
```

Failure to include this pledge in a submission is a violation of the Honor Code.