Assignment #4 Project (Design and implementation turn-in/compile instructions on canvas.)

A regional library, Safely Housing Hallowed Hardcopies (SHHH), wishes to automate the process of tracking books. This task is simplified by the fact that SHHH has only three types of books — children's books, fiction, periodicals — each sorted differently as described below.

SHHH wishes to be able to allow patrons to check out books, return books, and to display the contents of the library (all three categories) and display a report of a patron's history at any time. All books are displayed by category, sorted within the category. Output must be formatted for easy reading with data in columns. An example will be provided later.

Design and write a program that does the following:
– Initialize the contents of each of the three categories of books
–Process an arbitrary sequence of check-outs, returns, and displays of different kinds of information

<u>Notes</u>
– The library stores:

- Children's books sorted by title, then by author
- Fiction books sorted by author, then by title
- Periodicals sorted by date (year, then month), then by title

Assume each item is uniquely identified by its sorting criteria (other information not used when sorting or retrieving). This data is minimal. In real life, there would be much more data so do not problem solve based on this minimal amount of data per book. Typically, each item would have more data and be more varied.

A data file is used for the initialization. One line in the file contains information on one item. To facilitate processing, the first character of each line indicates this book type: children's books are marked with a 'C' for children, 'F' for fiction, and 'P' for periodicals. After the type is author (comma terminated), then title (comma terminated), and date (year, int type). Note that periodicals do not have an author and that the date will include month and year (both ints). For example,

```
F Pirsig Robert, Zen & the Art of Motorcycle Maint, 1974
P Communications of the ACM, 3 2001
P Communications of the ACM, 12 1998
Z Blah blah blah blah blah blah
C Seuss Dr., Yertle the Turtle, 1950
C Williams Jay, Danny Dunn & the Homework Machine, 1959
```

You can assume the format is correct, but codes may be invalid; e.g., the 'Z' code says the line of data is not valid. The library owns five copies of each item in the data file except periodicals where it only owns one copy. While the data for an item is minimal, do not assume (i.e., design and implement) that is the case. Design and implement as though there may be much more data for an item, e.g., 100 pieces of data on each book. Everything, including the output, could vary greatly if there was more data. Your design and implementation should be well designed and efficiently implemented for any amount of information.

– Library patron information will also be found in a second data file, one line per patron. Sample data includes a unique 4-digit unique ID number, last name, first name. A blank separates fields. For example:

```
1234 Mouse Mickey
```

You can assume correctly formatted data. The assumption about more information applies also to a patron.

–While normally you would expect a program like this to be interactive, to test your program, a third data file is used to simulate the interaction. It contains an arbitrary sequence of commands, one per line. The first char of each line ('C' for check-out, 'R' for return, 'H' to display a patron's history) indicates the action for a patron, or 'D' for library display of the three categories of books, sorted within the category. When it is a patron command, after the character key ('C', 'R', 'H'), there will be a blank, then the patron ID, then the character book type ('F', 'P', 'C'), then the book format (currently only 'H' for hard copy), and then the book data (based on the sorting criteria from above). Commas terminate in the same place as in the book data file.

For example:

```
C 1234 F H Walker Alice, The Color Purple,
C 1234 P H 1996 3 Communications of the ACM,
D
H 1234
R 1234 F H Walker Alice, The Color Purple,
X 5678 C Z Yertle the Turtle, Seuss Dr.,
R 5678 W H Yertle the Turtle, Seuss Dr.,
R 9999 C H Blah Blah Blah, Blah Blah,
```

The data is correctly formatted, but you must handle an invalid action code, an incorrect patron ID (not found), invalid format code, and invalid book (not found). For example, the 'X' is an invalid action code; 9999 is not a valid patron ID (not a data item from the patron data file); 'Z' is an invalid format type; the 'W' is an invalid book code; and there is no children's book with title "Blah Blah" . For bad data, get (read from the data file, getline()) and ignore the rest of the line of data.

–You must handle errors, any invalid data as stated above -- incorrect codes, invalid books or patrons -- and also incorrect commands (a return command when a book was not checked out). Display error messages (as detailed as possible), but do NOT display anything when a successful command is performed.

– When printing, conserve as printing on paper. Use one line per item (some information may be truncated). Displays are to be nicely formatted, in easily read columns, printed in portrait mode without wrapping. For a show of library contents, show each book category (all data for that type) with how many hard copies are checked out and how many remain. For a display of patron history, show a list of whether an item is currently checked out or not, i.e., show a list of every command in the order they occur and specify checked out or returned. (Sample output is provided during the implementation phase.)

–A requirement of this assignment is that you code at least one hash table from scratch. Note that in the design you do not include details of hash functions and tables. You can use as many hash tables as you would like and use the STL for anything except the required one you code yourself. A hash function maps a value to an element in an array, the hash table. If it is a unique mapping, there is no need for searching so an insert or look-up is a O(1) operation. If it is not a unique mapping, the collision must be handled, but it is still efficient. Two simple unique mapping examples help you understand hashing and hash tables:

(1) Suppose you're writing game software where a user needs to guess a letter and you need to keep track of the letters already guessed and those not guessed (e.g., Wheel of Fortune). You can easily map the letters of the alphabet to subscripts of an array ranging from zero to 25. You've checked for a valid letter and made sure it's lowercase. Then the hash function, h(letter), takes a letter and returns the subscript by computing letter-'a' (ascii value of letter minus ascii value of 'a'). So, 'a' maps to 0, 'b' to 1, 'c' to 2, 'd' to 3, and so on. Call the value, letter-'a', returned by the function, subscript. Then, you are examining hashTable[subscript] that gets you to the bool value for the letter you're looking up. Traversing the array is not needed.

(2) Suppose I created an array to hold information about students. I can use your ID number as a subscript into the array to find your information. Students are not contiguously sorted in any manner, but I can quickly do a look-up given your ID number. The array element, table[IDnumber], is your information. Note that there is wasted memory, but that leaves room for growth.

– A requirement of this assignment is to use inheritance. In general, avoid templates for your primary data structures. Best to run your template use by me, as this assignment is designed for you to practice using inheritance. There are no other specific requirements for this assignment, but as always it should be well designed (not violate design principles), easily extensible, efficiently coded, well documented, etc.

– This assignment is to be fully object-oriented so when you store multiple pieces of information, it would be stored in an object. Strings are only to be used in a primitive sense, for example, one name, one title, one of anything. Do **NOT** read long strings (e.g., one line of data). Do **NOT** build long strings that hold information. No string concatenation, no toString(). Functions will **NOT** have a string parameter for any datafile data.