CS 3202 – Spring 2019
Assignment 3 - 20 points
Due date: Tue, Apr 9th at 9a

Learning objectives

- Use explicit memory management in a C++ program.
- Implement a data structure that uses pointers.
- Perform simple file IO.
- Use OO design principles when designing and implementing the solution.
- Apply clean coding practices when developing the code.

Getting started

Getting started

1. Read through the entire assignment description before beginning.

2. Download the starter project, `CardBraider.tar.gz`, from Moodle.

Specifications

The program must be written in C++ must be able to be loaded as a *Code::Blocks* project and executed on the Ubuntu Virtual Machine that we are using in class.

Any project that does not compile will receive a zero.

Project requirements

The project must have the following functionality:

1. The program will be a GUI program using the FLTK C++ GUI toolkit. The starter project, I have provided gives you most of the GUI (view) code that will implement the UI for application.

2. Goals of the assignment include employing good object-oriented design principles, dynamic memory management, and the use of pointers. Therefore, you must build and store the data in a **braided list** (described below). The class designed for the braided list needs to provide constructors, destructors, and access functions, while practicing information hiding.

    a. You must write your own braided list.

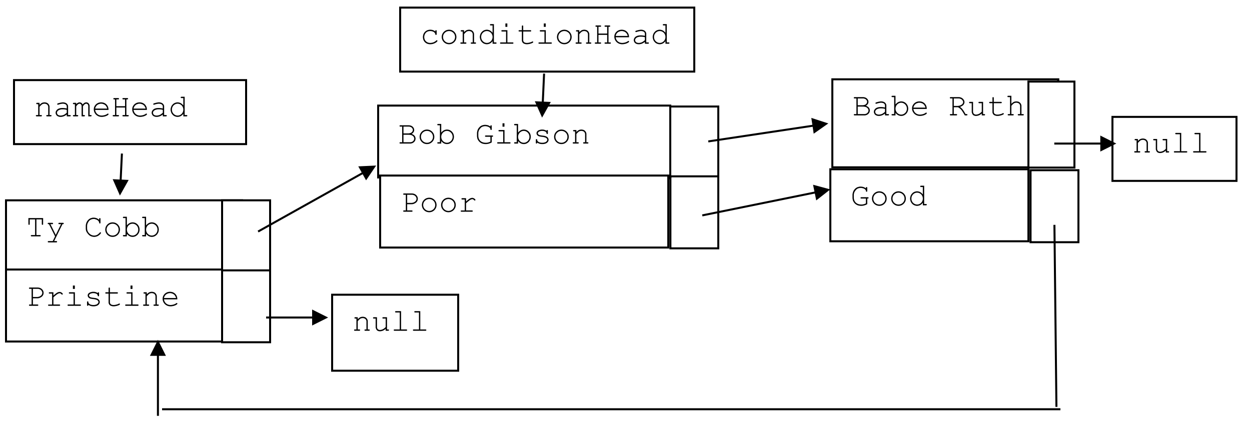
Braided list

The implementation of the list must use a braided list where one braid (link) is the player's name in alphabetic order, the second braid is the year in ascending order, and the third braid will be the condition of the card in ascending order, where the conditions are `Poor`, `Good`, `Excellent`, `Mint`, and `Pristine`. Additionally, a fourth attribute, the card's value, will be stored. However, there will not be a braid for the value of the card.

A braided list is type of a sorted linked list that allows the list to be sorted by more than one field. For example, the entries in the list are maintained in sorted order by last name, year, and condition. This can be implemented by adding multiple links (braids) for each node in the linked list. One link (braid) is used to point to the next node in the name order, another link to the next node in the year order, and a third link that points to the next link in the condition order. A node in the list will contain the following fields:

```
Node
   Player first name
   Player last name
   Year
   Condition
   Value
   Node* nextName;
   Node* nextYear;
   Node* nextCondition;
```

The figure below illustrates the resulting data structure with some of the fields omitted. Note: there will be another braid through this list that follows the year link.



3. Do not use character-based arrays for storing the player's name. You must use the `string` class provided by C++.

4. You must use an enumerated type for the baseball card conditions. The card conditions in order are: Poor, Good, Excellent, Mint, Pristine.

5. The program must provide a GUI to interact with the program. Most of the UI layout is provided for you. It will be your responsibility to tie your code to the provided UI code.

    a. Load a file – allows the user to load a specified file of baseball player cards replacing the currently loaded list of cards. Note: The file loaded may **not** be in alphabetic order according to the player's last name. You will need to provide functionality to read and load the file and store the data in the braided list.

        i. The format of the text file is described below:

        The information for each card will be stored on a single line as a CSV file in the following format:

        `lastname,firstname,year,condition,value`

        where each field is separated by a comma with no spaces.

        The card conditions need to be written to the file as Poor, Good, Excellent, Mint, Pristine. However, it must be stored in the program as an enumerated type.

        The value of the card will be stored as a whole number roundest to the nearest dollar.

        Example file contents:
        ```
        Cobb,Ty,1909,Good,751234
        Gibson,Bob,1968,Poor,12
        ```

    b. Save a file – writes the current list of loaded cards to a user-specified file in the format mentioned above.

        i. When writing to a file, if an existing file is selected, go ahead and overwrite the file.

        ii. The file must be written out in ascending order by the last name.

c. Delete card – prompts the user for the player's last name and deletes all players that match the last name from the list. If a player with the last name is not found, the user should be informed.

    i. The search for the player's last name should be case insensitive.

    ii. When a card is deleted then make sure to free the memory for the card node.

d. Add card – prompts the user for all the fields needed and adds the card to the current list of cards. If there is not a currently loaded list, this should create a new list.

e. Display cards ascending by last name – use a **recursive** function to traverses the last name braid in order to obtain and display the card information in sorted ascending order via player's last name, as follows:

```
Ty Cobb        1909      Good            $751,234.00
Bob Gibson     1968      Poor                 $12.00
```

    i. **Each column for all required output must be aligned in the output box.**

f. Display cards descending by last name – uses a **recursive** function that uses the name braid to obtain and display the card information in reverse sorted order via player's last name, as follows:

```
Bob Gibson     1968      Poor                 $12.00
Ty Cobb        1909      Good            $751,234.00
```

g. Display cards ascending by condition – use a **recursive** function to traverse the condition braid in order to obtain and display the condition information in sorted ascending order via condition, (if condition same, then by name), as follows:

```
Bob Gibson     1968      Poor                 $12.00
Ty Cobb        1909      Good            $751,234.00
```

h. Display cards descending by condition – uses a **recursive** function that uses the condition braid to obtain and display the card information in reverse sorted order via condition, (if condition same, then by name) as follows:

```
Ty Cobb        1909      Good            $751,234.00
Bob Gibson     1968      Poor                 $12.00
```

i. Print cards ascending by year – use a **recursive** function that traverses the year braid in order to obtain and display out the card information in sorted ascending order via year, (if the year is the same, then by name), as follows:

```
Ty Cobb        1909      Good             $751,234.00
Bob Gibson     1968      Poor                  $12.00
```

j. Print cards descending by years – uses a **recursive** function that uses the year braid to obtain and display the card information in reverse sorted order via year, (if the year is the same, then by name) as follows:

```
Bob Gibson     1968      Poor                  $12.00
Ty Cobb        1909      Good             $751,234.00
```

k. Exit – exits the program. Upon exiting the program it must clean-up all allocated resources.

6. File I/O must use the stream classes.

7. When starting the application the title bar of the main window must display your name:

```
John Doe’s Baseball-card collection
```

8. Update the `notes.txt` – list any known bugs or missing functionality in this file before you export and submit the project.

Milestone 1 demo

1. There will be a demo of your work on the assignment on Tuesday, April 2nd. To receive full credit on this demo your program should be able to load from a file a list of cards, be able to add a new card to the collection, and display them sorted by name (ascending and descending).

Guidelines

- You must do this assignment by yourself.
- Apply clean coding practices and good design practices that incorporate model-view separation, place code in appropriate folders, etc.
- Make sure all memory that is allocated is freed.
- Be consistent in your formatting. The formatting for indentation can be done with Code::Blocks, but make sure you are consistent in the use of white space.
- Every public constructor, function, and field must have a comment header with a summary statement, precondition, post condition, as necessary, and the params and returns fields, as necessary.
- The naming standards must adhere to the following:
    - Names representing types (classes) must be in mixed case starting with upper case.
    - Variable names must be in mixed case starting with lower case.
    - Named constants (including enumeration values) must be all uppercase using underscore to separate words.
    - Names representing methods or functions must be verbs and written in mixed case starting with lower case.
    - Names representing namespaces should be all lowercase.
    - Folder names should all be lowercase.
    - Do NOT use Hungarian notation.

Submission

To submit the project do the following:

1. Update the notes.txt file to list any known bugs.
2. Clean the project to remove all files that can be regenerated upon building the project.
3. Close *Code::Blocks*.
4. Navigate to the root folder for the project and compress the project and name the compressed file *FirstnameLastname*A3.tar.gz and submit the zip file to Moodle by the due date.

Grading rubric

Any program that does not compile will receive a 0. Partial credit is not possible for any program that does not compile.
If a program is only partially complete and a category cannot be accurately assessed you will not receive full credit for that category.

	Exceptional	Acceptable	Amateur	Unsatisfactory
Functionality	10 pts. – grading on continuous scale.			
Milestone demo	2 pts.	1 pt.	NA	0 pts.
Readability/formatting/ organization	2 pts.	NA	1 pt.	0 pts.
Implementation/reusability	4 pts.	3 pts.	2 pt.	1 pt.
Documentation	2 pts.	NA	1 pt.	0 pts.

Grading description

	Exceptional	Acceptable	Amateur	Unsatisfactory
Readability/ formatting/ organization	The program is exceptionally well organized, very easy to follow, and there are not any compiler warnings.	The code is fairly easy to read and there are a few compiler warnings that were not legitimately explained as to why they still exist.	The code is readable only by someone who knows what the code is supposed to be doing and/or there are many unresolved compiler warnings.	The code is poorly organized and very difficult to read.
Implementation/r eusability	The code follows best practices in implementation and could be reused as a whole or each routine could be reused.	Most of the code follows best practices and could be reused in other programs.	Some parts of the code follows best practices and could be reused in other programs.	The code does not follow best practices and is not organized for reusability.
Documentation	The required documentation is well written and clearly explains what the code is accomplishing. There is not any redundant inline commenting.		The required documentation is there, but not complete or is only somewhat useful in understanding the code.	The documentation is poor and very incomplete.