Assignment 2 - 100 points

Part 1

Assignment Overview

In Part 1 of this assignment, you will write a main program and several classes to create and print a small database of book data. The assignment has been split into two parts to encourage you to code your program in an incremental fashion, a technique that will be increasingly important as the semester goes on.

Purpose

This assignment reviews object-oriented programming concepts such as classes, methods, constructors, accessor methods, and access modifiers. It makes use of an array of objects as a class data member, and introduces the concept of object serialization or "binary I/O".

Set Up

1. As in Assignment 1, you should create a subdirectory to hold your files for Assignment 2.
2. In that directory, make a symbolic link to the data file for this part of the assignment:

   ```
   ln -s /home/turing/t90kjm1/CS241/Data/Fall2016/Assign2/bookdata
   ```

3. In this assignment, you will be creating several source code and header files, as described below. You can create each of these files separately using the `nano` editor, just as you did on Assignment 1.
4. To compile and link the program you've created, type:

   ```
   g++ -Wall -std=c++11 -o assign2 assign2.cpp Book.cpp
   ```

   Once you've added the `BookStore` class, you should type:

   ```
   g++ -Wall -std=c++11 -o assign2 assign2.cpp Book.cpp BookStore.cpp
   ```

   (Yes, these commands are rather tedious to type repeatedly. Part 2 of this assignment introduces a new technique for compiling and linking your program files called a *makefile*. Makefiles require a bit more work up front, but save a lot of typing at the command line once the makefile has been created.)

5. To run the executable file created by the previous command, type:

   ```
   ./assign2
   ```

Program

For this assignment, you will need to write three source code files as well as two *header files*. Each of these

files is relatively short, but many inexperienced programmers are overwhelmed by the idea of writing a program as multiple files. "Where do I start?!!" is a common refrain. This assignment sheet attempts to walk you through the steps of writing a multi-file program.

The steps outlined below should not be thought of as a purely linear process, but rather an *iterative* one - For example, work a little on Step 1, then a little on Step 2, then test what you've written (Step 3).

Step 1: Write the Book class declaration

The `Book` class represents information about a book. The code for the `Book` class will be placed in two separate files, which is the norm for non-template C++ classes.

The *header file* for a class contains the class declaration, including declarations of any data members and prototypes for the methods of the class. The name of the header file should be of the form *`ClassName`*`.h` (for example, `Book.h` for the header file of the `Book` class).

A skeleton for the `Book.h` file is given below. As shown, a header file should begin and end with header guards to prevent it from accidentally being `#included` more than once in the same source code file (which would produce duplicate symbol definition errors). The symbol name used in the header guards can be any valid C++ name that is not already in use in your program or the C/C++ libraries. Using a name of the format *`CLASSNAME`*`_H` (like `BOOK_H` in the code below) is recommended to avoid naming conflicts.

```
#ifndef BOOK_H
#define BOOK_H

//*************************************************************
// FILE:      Book.h
// AUTHOR:    your name
// LOGON ID:  your z-ID
// DUE DATE:  due date of assignment
//
// PURPOSE:   Contains the declaration for the Book class.
//*************************************************************

class Book
   {
   // Data members and method prototypes for the Book class go here
            .
            .
            .
   };

#endif
```

Data Members

The `Book` class should have the following four private data members:

- An ISBN (a character array with room for 10 characters PLUS the null character)
- A title (a character array with room for 40 characters PLUS the null character)
- A price (a `double` variable)
- A quantity in stock (an integer)

Note: Make sure you code your data members in THE EXACT ORDER LISTED ABOVE and with THE EXACT SAME DATA TYPES. If you use `float` instead of `double` or only make the title array 40

characters long instead of 41, your final program will not work correctly.

C++11 Initialization Option for Data Members

C++11 adds the ability to initialize the non-static data members of a class at the time you declare them using a "brace-or-equal" syntax. This is very convenient, and can eliminate most or all of the code from your default constructor. Here are a few examples of the kind of initializations you can do in a class declaration:

```
class Foo
   {
   // Data members
   private:

      int x = 0;                                  // Initialize x to 0
      double y = 9.9;                             // Initialize y to 9.9
      char text[21]{};                            // Initialize text to an
                                                  // empty string
      char name[11]{'J', 'o', 'h', 'n', '\0'};    // Initialize name to "John"
      string s{"Hello"};                          // Initialize s to "Hello"

      etc.
   };
```

Feel free to use this option if you want to.

Method Prototypes

The `Book` class declaration should (eventually) contain public prototypes for all of the methods in the `Book.cpp` source code file described in **Step 2** below.

Step 2: Write the Book class implementation

The *source code file* for a class contains the method definitions for the class. The name of the source code file should be of the form *`ClassName`*`.cpp` or *`ClassName`*`.cc` (for example, `Book.cpp` for the source code file of the `Book` class).

The `Book` class implementation should (eventually) contain definitions for all of the methods described below. Make sure to `#include "Book.h"` at the top of this file.

- `Book` default constructor - This "default" constructor for the `Book` class takes no parameters. Like all C++ constructors, it does not have a return data type.

  This method should set the ISBN and title data members to "null strings". This can be done by copying a null string literal (`""`) into the character array using `strcpy()` or by setting the first element of the array to a null character (`'\0'`). The price and quantity data members should be set to 0.

  (If you're working in C++11 and you initialized the data members at declaration as described above under *C++11 Initialization Option for Data Members*, this method's body can be empty. You still need to code the method though, even though it won't actually do anything.)

- Alternate `Book` constructor - Write another constructor for the `Book` class that takes four parameters: 1) a character array that contains a new ISBN, 2) a character array that contains a new title, 3) a `double` variable that contains a new price, and 4) an integer that contains a new quantity. **DO NOT GIVE THESE PARAMETERS THE SAME NAMES AS YOUR DATA MEMBERS.**. Like all C++

constructors, this constructor does not have a return data type.

Use `strcpy()` to copy the new ISBN parameter into the ISBN data member and the new title parameter into the title data member. Call the `setPrice()` and `setQuantity()` methods to set the price and quantity data members to the new price and new quantity passed into the constructor.

- `getISBN()` - This accessor method takes no parameters. It should return the ISBN data member. In C++, the usual return data type specified when you are returning the name of a character array is `char*` or "pointer to a character" (since returning an array's name will convert the name into a pointer to the first element of the array, which in this case is data type `char`.
- `getTitle()` - This accessor method takes no parameters. It should return the title data member.
- `getPrice()` - This accessor method takes no parameters. It will have a return data type of `double`. It should return the price data member.
- `getQuantity()` - This accessor method takes no parameters. It will have a return data type of `int`. It should return the quantity data member.
- `setPrice()` - This method takes a `double` argument, a new price. It returns nothing. The method should check if the new price is greater than or equal to 0. If it is, it should set the price data member to the new price. Otherwise, it should set the price data member to 0.
- `setQuantity()` - This method takes an integer argument, a new quantity. It returns nothing. The method should check if the new quantity is greater than or equal to 0. If it is, it should set the quantity data member to the new quantity. Otherwise, it should set the quantity data member to 0.
- `fulfillOrder()` - This accessor method takes one parameter, an integer that represents the quantity of this book that has been ordered. The method returns an integer, which is the quantity of this book that the book store is actually able to ship at this time.

  The logic for this method should be as follows:

  - If the order quantity is less than zero, the order is in error. The number shipped should be zero. Do not alter the quantity in stock for the book.
  - If the order quantity is less than or equal to the quantity in stock, the order can be completely filled. The number shipped should be the same as the order quantity, and the order quantity should be subtracted from the quantity in stock.
  - Otherwise, this order can not be completely filled. The number shipped should be the quantity in stock, and the quantity in stock should be set to zero.

- `print()` - This method takes no parameters and returns nothing. The method should print the ISBN, title, price, and quantity members on the console using `cout`. Use `setw()` to line the printed values up in columns (a width of 14 for the ISBN, 44 for the title, 5 for the price, and 6 for the quantity will match the sample output). The ISBN and title should be left justified; the price and quantity should be right justified. The price should be printed using fixed-point notation with two places after the decimal point.

Step 3: Test and debug the Book class

As you write your declaration and implementation of the `Book` class, you should begin testing the code you've

written. Create a basic main program called `assign2.cpp` that tests your class. This is not the final version of `assign2.cpp` that you will eventually submit. In fact, you'll end up deleting most (or all) of it by the time you're done with the assignment. An example test program is given below.

You do not have to have written all of the methods for the `Book` class before you begin testing it. Simply comment out the parts of your test program that call methods you haven't written yet. Write one method definition, add its prototype to the class declaration, uncomment the corresponding test code in your test program, and then compile and link your program. If you get syntax errors, fix them before you attempt to write additional code. A larger amount of code that does not compile is not useful - it just makes debugging harder! The goal here is to constantly maintain a working program.

```
#include <iostream>
#include "Book.h"

using std::cout;
using std::endl;

int main()
   {
   char isbn1[11] = "1111111111";
   char title1[41] = "Learn C++ Now";
   char isbn2[11] = "2222222222";
   char title2[41] = "Learn Java Later";
   int numShipped;

   // Test default constructor
   Book book1;

   // Test alternate constructor
Book book2(isbn1, title1, 39.99, 5);

// Test data validation
Book book3(isbn2, title2, -22.99, -6);

// Test print() method and whether constructors
// properly initialized objects
cout << "Printing book1\n\n";
book1.print();
cout << endl << endl;

cout << "Printing book2\n\n";
book2.print();
cout << endl << endl;

cout << "Printing book3\n\n";
book3.print();
cout << endl << endl;

// Test accessor methods
cout << book2.getISBN() << endl;
cout << book2.getTitle() << endl;
cout << book2.getPrice() << endl;
cout << book2.getQuantity() << endl;

// Test the fulfillOrder() method
numShipped = book2.fulfillOrder(-5);
cout << "\nShipped " << numShipped << endl;
cout << "Quantity now " << book2.getQuantity() << endl;
```

```
numShipped = book2.fulfillOrder(3);
cout << "Shipped " << numShipped << endl;
cout << "Quantity now " << book2.getQuantity() << endl;

numShipped = book2.fulfillOrder(4);
cout << "Shipped " << numShipped << endl;
cout << "Quantity now " << book2.getQuantity() << endl;

return 0;
}
```

Once your `Book` class has been thoroughly tested and debugged, it's time to write the second class for this assignment.

Step 4: Write the BookStore class declaration

The `BookStore` class represents a database of `Book` objects. Like the `Book` class, the code for this class will be placed in two separate files.

Place the class declaration in a header file called `BookStore.h`. Like the file `Book.h` you wrote in Step 1, this file should begin and end with header guards to prevent it from accidentally being `#included` more than once in the same source code file.

After the header guard at the top of the file but before the class definition, make sure to `#include "Book.h"`.

Data Members

The `BookStore` class should have the following two private data members:

- An array of 30 `Book` objects
- An integer that specifies the number of `Book` objects actually stored in the array

Note: Once again, make sure you code your data members in THE EXACT ORDER LISTED ABOVE and with THE EXACT SAME DATA TYPES.

Method Prototypes

The `BookStore` class declaration should (eventually) contain public prototypes for all of the methods in the `BookStore.cpp` source code file described in **Step 5** below.

Step 5: Write the BookStore class implementation

The `BookStore` class implementation should (eventually) contain definitions for all of the methods described below. Make sure to `#include "BookStore.h"` at the top of this file.

- `BookStore` default constructor - This "default" constructor for the `BookStore` class takes no parameters. Like all C++ constructors, it does not have a return data type.

  This constructor is called to create an empty database, so this method should set the number of books data member to 0.

  (As with the `Book` class, if you initialize the number of books data member to 0 when you declare it, this method's body can be empty. You still need to code the method with an empty body.)

- Alternate `BookStore` constructor - Write a second constructor for the `BookStore` class that takes one parameter: a pointer to a constant character (data type `const char*`), which will point to an array of characters that contains the name of an existing database file. Like all C++ constructors, this constructor does not have a return data type.

  This constructor should do the following:

  1. Declare an input file stream variable (the code below assumes it is named `inFile`).

  2. Open the file stream for binary input. Check to make sure the file was opened successfully as usual.

  3. Read the database file into your `BookStore` object. You can do this with a single statement:

     ```
     inFile.read((char*) this, sizeof(BookStore));
     ```

  4. Close the file stream.

- `print()` - This method takes no parameters and returns nothing.

  This method should first print a descriptive header line (e.g., "Book Inventory Listing"). It should then loop through the array of `Book` objects and print each of the elements that contain book data, one per line. Here we see some of the power of object-oriented programming: since each element of the array is an object, we can call a method for that object. We've already written a `print()` method for the `Book` class, so printing an element of the array is as easy as calling `print()` for the array element. The syntax for calling a method using an array element that is an object is pretty straightforward:

  ```
  arrayName[subscript].methodName(arguments);
  ```

Step 6: Write the main program

Since most of the logic of the program is embedded in the two classes you wrote, the `main()` routine logic is extremely simple.

- Create a `BookStore` object using the alternate constructor you wrote. Pass the filename string `"bookdata"` as an argument to the constructor.
- Call the `print()` method for the `BookStore` object.

Other Points

- You do not need to submit this part of the assignment, only Part 2.

- Make sure to document your program according to the standards listed in the Course Notes book. In particular, each class method or function should have a documentation box describing its purpose, the input parameters (if any), and the return value (if any). There should also be a documentation box for the program as a whole.