Assignment Tutorial Letter 2024

Advanced Programming

COS3711

Year module

Computer Science Department

Assignment 3 Questions

BARCODE

Assignment 3

1. Introduction

Please use CMake (and not qmake) when setting up your assignment projects.

Qt Designer should not be used to design user interfaces, and you are expected to manually set up GUIs to ensure that you properly handle memory using Qt’s parent-child functionality.

Good programming practices should be followed.

Follow standard naming conventions: class names start with a capital letter, variable and function names start with a lowercase letter, using camelCase for names made up of multiple words.

Ensure consistent code layout and use of blank lines. Use forward class declarations in header files.

Use initialiser lists in constructors.

Have proper GUI management: setting cursor focus, sequential tabbing, clearing input widgets (like text input fields being cleared and spin boxes being returned to some default value), and enabling and disabling buttons as appropriate.

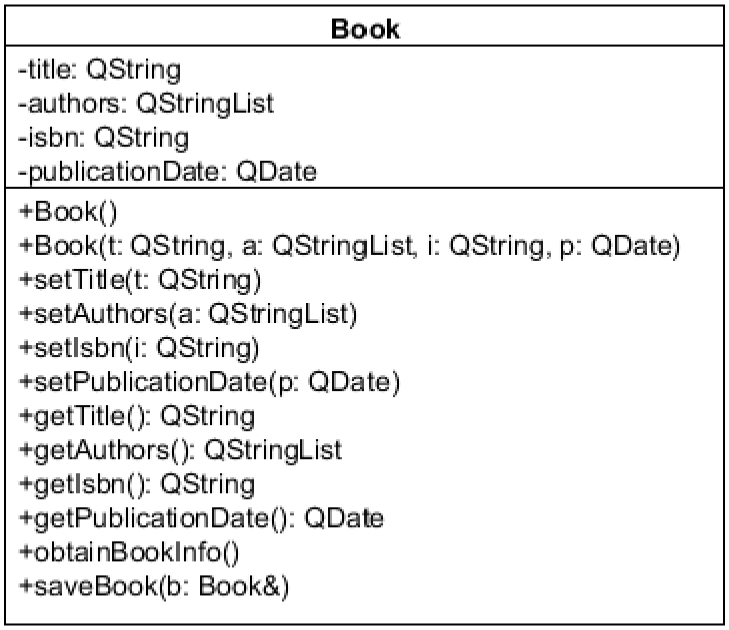
Provide appropriate user feedback.

Your code should build and run without any warnings.

2. Questions

Question 1

Refer to the UML diagram to answer this question.



```
Book
-title: QString
-authors: QStringList
-isbn: QString
-publicationDate: QDate
+Book()
+Book(t: QString, a: QStringList, i: QString, p: QDate)
+setTitle(t: QString)
+setAuthors(a: QStringList)
+setIsbn(i: QString)
+setPublicationDate(p: QDate)
+getTitle(): QString
+getAuthors(): QStringList
+getIsbn(): QString
+getPublicationDate(): QDate
+obtainBookInfo()
+saveBook(b: Book&)
```

The `Book` class is burdened with two additional responsibilities:

1. `obtainbookInfo()` to get user input on book details (title, authors, ISBN, and date of publication) via a console, and
2. `saveBook()` to save the state of a `Book` instance into a file.

Redesign the `Book` class so that the concept/process currently included in it is achieved using three different classes:

1. The `Book` class to represent a book with the necessary functions (as given in the above UML diagram). Remember that a book may have more than one author.
2. A `BookWriter` class which saves the state of a `Book` instance (values of the data members of that instance) in a file (you may save it in any sensible format).
3. A Graphical User Interface (GUI) class `BookInput`, which allows users to enter book information to create `Book` instances, which should also be saved to a

file (using the `BookWriter`). *Note*: You can decide when `Book` instance(s) should be saved.

Question 2

Make necessary changes to the `Book` class so that you can access the data members of `Book` instances using `QMetaObject`. Rewrite the `BookWriter` class so that the state of a `Book` instance can be accessed using the generated `QMetaObject`.
Test the modified `Book` and `BookWriter` using `BookInput`.

Question 3

Different systems use different rules for choosing passwords. Here are some examples of such rules:

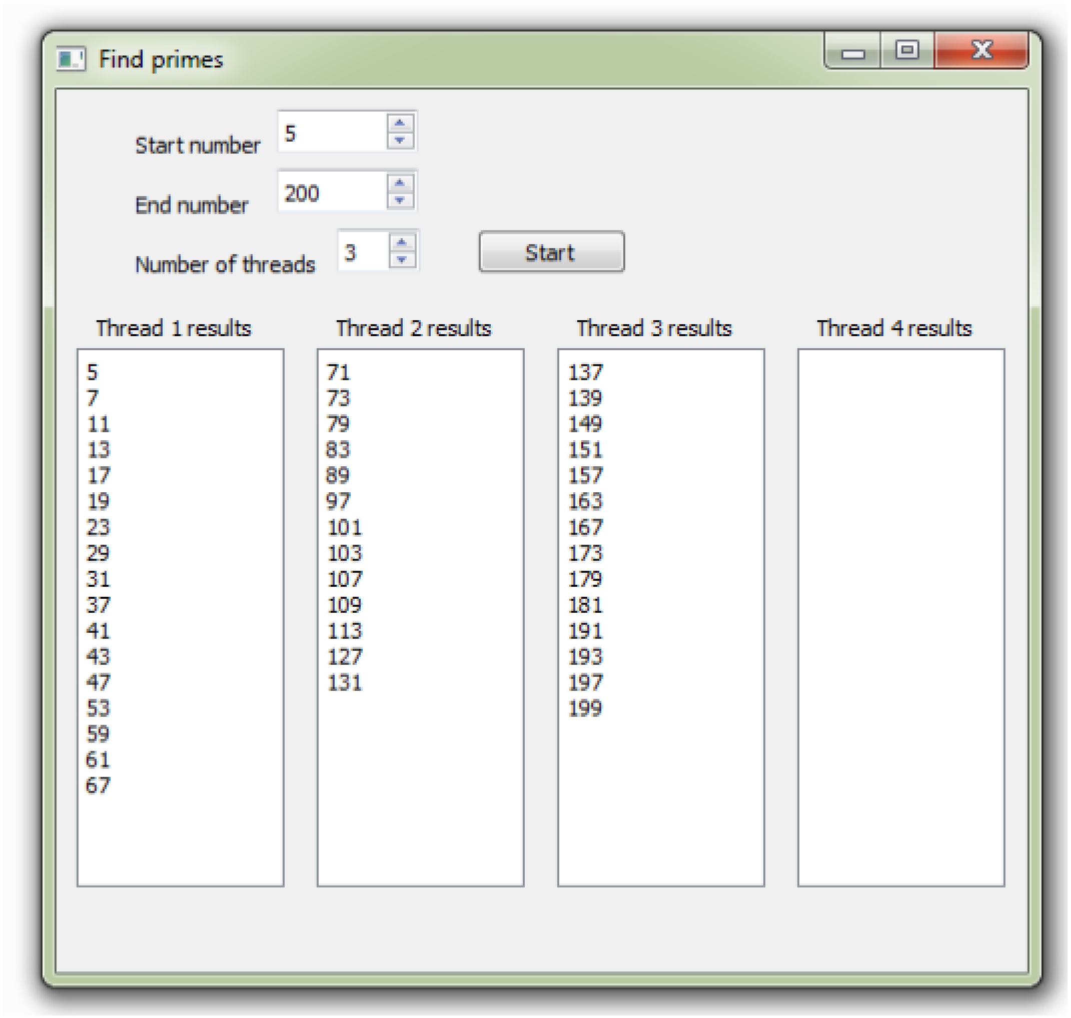
Rule 1: It must consist of a minimum of five characters and no spaces are allowed.
Rule 2: It must consist of 5 characters, where the third character can be either a digit or an alphabetic character (a-f or A-F).
Rule 3: It must consist of a minimum of four and a maximum of six digits and the starting digit cannot be 0.

Write a program that asks the user to select **one** validation technique stated above using `QCheckBox` and then to enter a password in a `QLineEdit`. The program should evaluate whether the provided password satisfies the selected validation rule.

Question 4

Write a class named `PrimeFinder`, capable of running as a thread, to find prime numbers between two given numbers.
Write a GUI application that allows a user to enter start and end numbers as well as the number of threads to use (maximum 4). When Start is clicked, the task should be divided into the required number of parts, the required number of `PrimeFinder` threads started, and the primes displayed in an appropriate window (per thread) as they are found.
This is to be done without using `QtConcurrent` – you are to manage the threads yourself, and clean up after the threads have completed their work. This means that you

should be able to change the start, end, and number of thread numbers and click Start again without the program crashing or producing strange results. Also, you should handle a user closing the application (`closeEvent`) before the threads have completed running so that the threads can tidy up after themselves before the application is closed.



Start number 5

End number 200

Number of threads 3

Thread 1 results	Thread 2 results	Thread 3 results	Thread 4 results
5	71	137	
7	73	139	
11	79	149	
13	83	151	
17	89	157	
19	97	163	
23	101	167	
29	103	173	
31	107	179	
37	109	181	
41	113	191	
43	127	193	
47	131	197	
53		199	
59			
61			
67			

The above screenshot is only an example, and you can design the graphical user interface as you wish as long as it satisfies the above requirements.