RMIT UNIVERSITY

Advanced Programming Techniques
COSC1076
24 hour Programming Exercise

Assessment Type	24 hour Programming Exercise
Start Date	Please check Canvas
Due Date	Please check Canvas
Weight	25% of the final course mark
Submission	Online via Canvas.
Learning Outcomes	This assignment contributes to CLOs: 1, 2, 3, 4

General Information

- You must not share your questions or answers. These are individual questions. It is **academic misconduct** to share your questions or work with any individual.
- Please follow the specific instructions for each question that describe where to place your answers in your submission.
- Submission instructions are located in the Appendix on the last page.
- Any code you write must comply with the Course Style Guide.
- You may only use C++ language features and C++ STL elements that have been taught in the course. The reason for this is to limit the scope of what you are expected to answer to the contents of the course and therefore be fair to all students.

Question 1: Abstract Data Types (12 marks)

Instructions:

- Place all of your code files for this question in a sub-folder called `question1`.

In this question you will *design* and *implement* a set of ADTs for representing simple information about a hospital. You should *carefully consider* your design and implementation as this question contains a number of details. You are marked on the *quality* of your design and implementation. Just "making the code work" is insufficient to receive full marks. Rushed, quick or ill-considered work will often results in poor marks.

Design & Functionality

You will need to design and implement 3 ADTs:

1. `Patient`
2. `PatientRegister`
3. `WaitingList`

You may implement additional classes to support these ADTs. However, remember that you are marked on the quality of your design. Excessive classes may be considered poor design.

The `Patient` ADT represents simple information about a patient. This information must include:

1. The name, such as "John Doe" or "Jane Doe".
2. Date of birth. such as 10-2-1965.
3. A registration number, such as 1234567. You may assume that registration numbers are unique for all patients (but you do not need to check this).
4. Admission history. This would be a ordered list of entries where each entry contain the information: admission date, discharge date (if applicable), admitted ward.
5. Support the use of the operator `operator<<`, that enables the data stored about a patient to be written to an output stream.

The `PatientRegister` ADT holds the *set* of all patients that has received (or is receiving) treatment at the hospital. The ADT should support the following functionality:

1. Add a patient to the register.
2. Remove a patient from the register.
3. Check if a patient is in the register.
4. Retrieve an individual patient stored within the register.
5. Ensure that there are no duplicate patients in the register, which is determined by the patient's registration numbers.
6. Support the use of the operator `operator[]`, to retrieve a patients by his/her registration number.

You will need to think about the parameters you need for the methods of the ADT to support this functionality.

The `WaitingList` ADT represents a *queue* of patients scheduled to have elective surgery on a particular day. WaitingList for elective surgery use a predefined order (or queue) in which patients are taken in for surgery. That is, a patient cannot be taken into his/her surgery until the patient ahead have already started. The ADT should support the following functionality:

1. Initially the WaitingList is empty until patients are added to the queue.
2. Add a patients to the starting queue, and also assign a patients a unique *queue number*. Note this queue number is different to the patient's registration number.
3. Remove patients from the queue as they "start" the surgery
4. Get the next patient to start the surgery.

Continued . . .

5. Get the unique queue number for a patient.
6. Get the number of patients remaining in the queue yet to start the surgery

Note, a patient cannot appear multiple time in the WaitingList. You will need to think about the parameters you need for to the methods of the ADT to support this functionality.

You may choose a suitable underlying representation for each ADT including:

- Single or double-ended linked list
- Binary search tree
- C++14 STL ordered container (array, vector, list, deque, stack, queue)
- C++14 STL associative container (set, map)

Design Justification & Explanation

As comments in your ADTs, you should provide short description(s) the justify any choices that you have made in your design. Remember that you are marked on the *quality* of your design and implementation. Thus, you should provide *short* justifications of your design. Good design should be clear and only require a short explanation.

In your design you should consider:

- **Abstraction**, such as the scope (public/protected/private) of variables and methods to ensure that only the desired functionality is available in the public ADT interface.
- **Encapsulation**, by keeping code to within the most relevant ADT and class.
- Use of **Inheritance** to maximise code-reuse, and inheritance principles, such as virtual methods, and calling constructors and destructors.
- Use of **Polymorphism** to maximise code-reuse.
- **Computational efficiency**, such that you should avoid computational inefficient operations.
- **Minimising excess copying** of classes through the use of pointers or move semantics.
- Minimising code duplication through **Code-Reuse**.

Defensive Programming Paradigm

In the *implementation* of your ADTs and classes, you must use a fully *defensive programming* paradigm. That is, the methods on your classes should account for all inputs where reasonably possible. You should also consider, where reasonably possible, all program states of your classes when methods are called.

Recall that as part of the defensive programming paradigm is that you methods should:

1. Not crash or result in a segmentation fault
2. Should always perform a "reasonable" operation and always ensure that the class remains in a fully functional state

Marking *(12 marks)*

The marks are distributed as follows:

- Design *(4 marks)*
- Implementation *(6 marks)*
- Conformance to Defensive Programming Paradigm *(2 marks)*

Continued . . .

Question 2: Generics & Inheritance (6 marks)

Instructions:

- Place all of your code files for this question in a sub-folder called `question2`.
- The sub-folder `question2` in "starter code" provides a starting point. The starter code contains errors and you should fix those while developing your solution.
- You should not use libraries that are not already included in the starter code.

In this question you will *implement* a set of classes to represent an image and image pixels. Typically, an image is represented as a matrix (2D) of pixels (In this question, you should represent an image as a two-dimensional vector (vector of vector) of pixels). While there are many representations for a pixel, the most common are the RGB-pixels and grey-scale pixels. A grey-scale pixel has only one attribute which represent the intensity of the pixels. An RGB pixel contain 3 attributes which represent the 3 colour channels ('red', 'green' and 'blue').

You should *carefully consider* implementation as this question contains a number of details. You are marked on the *correctness* and *quality* of your implementation.

The starter code for `question2` consists of the following files:

- `main.cpp`: Contains code that create an RGB image and a Grey-scale image and manipulate them. *This file should not be modified.*
- `Pixel.h`: Abstract class to represent general functionality of a pixel.
- `RGBPixel.h`/`RGBPixel.cpp`: Represent a RGB pixel. Derived from base class Pixel.
- `GreyscalePixel.h`/`GreyscalePixel.cpp`: Represent a grey-scale pixel. Derived from base class Pixel.
- `Image.h`/`Image.cpp`: Represent an image. Has to be able to use either grey-scale or RGB pixels.
- `Makefile`

2.1 Pixel class (1 mark)

The class `Pixel` is an abstract class and it should contain functions: `getBrightness`, `operator[]`.

2.2 RGBPixel & GreyscalePixel classes (3 mark)

Both classes should derive from the `Pixel` class. You should also implement appropriate constructors/deconstructors. The brightness of a grey-scale pixel is the intensity of that pixel and the brightness of RGB pixel is to be computed as $(r + b + g)/3$. Here r, g and b are the values of the colour channels red, green and blue respectively. The `operator[]` should return the value of the appropriate channel based on a character input. See the header files in the starter code for more information.

You **should not modify** the private data members in the two classes RGBPixel & GreyscalePixel.

2.3 Image class (2 mark)

This class should store an image as a two-dimensional vector (vector of vector) of pixels. An image could be initialized to have either grey-scale or RGB pixels. You should also implement a `get` and a `set` method to access the pixels in an image given the row and column.

Programming by Contract Paradigm

In the *implementation* of your classes, you must use *Programming by Contract* paradigm. That is, The programmer should specify a "contract" (where applicable) that must be complied with when using a specific functionality of the class.

Continued ...

Question 3: Operator Overloading (7 marks)

Instructions:

- Place all of your code files for this question in a sub-folder called `question3`.
- The sub-folder `question3` in "starter code" provides a starting point.
- You should not use libraries that are not already included in the starter code.

In this question, you will implement a class to represent a date. The `Date` class has three private data members: year, month and day. Some member functions of the `Date` class is already implemented and you are not required to modify them. However, several member functions are missing. Your task is to complete the `Date` class so that the output of the compiled program matches the expected output as given by `expected_out.txt`.

You should *carefully consider* implementation as this question contains a number of details. You are marked on the *correctness* and *quality* of your implementation.

The starter code for `question3` consists of the following files:

- `main.cpp`: Contains code that manipulate Date objects. *This file* ***should not*** *be modified.*
- `Date.h`/`Date.cpp`: The files for the Date class.
- `expected_out.txt`: Contain the expected output of the program. *This file* ***should not*** *be modified.*
- `Makefile`

A given date can be converted to a "dayNumber" which represent the number of days since *1st march 0000*. Two functions that allow you to go from Date to "dayNumber" and, "dayNumber" to date, is provided in the starter code. It is expected that you will use them to do Arithmetic Operations on `Date`.

Programming by Contract Paradigm

In the *implementation* of your classes, you must use *Programming by Contract* paradigm. That is, The programmer should specify a "contract" (where applicable) that must be complied with when using a specific functionality of the class.

The marks are distributed as follows:

- Implementation *(6 marks)*
- Conformance to Programming by Contract Paradigm *(1 marks)*

Appendix A: Submission Instructions

Combined all of the files for the questions into a single ZIP file called `s123456.zip` (change filename to match your student number). Upload this file to the Canvas assessment module.

Finally you **must** also include *this* PDF files of questions in your ZIP.

Appendix B: Late Submission Policy

Ensure that you give yourself sufficient time to prepare and submit your work. The submission is a hard deadline. That is, there will be no leeway on late submissions. The late submission policy is:

- 20% penalty if submitted up to 1 hour late
- 50% penalty if submitted up to 2 hours late
- Grade of 0 if submitted 2 or more hours late