Tutorial letter 203/1/2018

Programming: Contemporary Concepts

COS2614

Semester 1

School of Computing

Discussion of Solutions to Assignment 3

BAR CODE

CONTENTS

Page

Afrikaanse studente: Die studiemateriaal vir hierdie module is slegs in Engels beskikbaar. As daar enigiets onduidelik is daarin is u welkom om die dosente vir COS2614 te kontak.

1 INTRODUCTION

The code of the solutions to the programming questions of Assignment 3 is placed under Additional Resources of COS2614 on *my*Unisa. A separate folder is created for each question, in which you will find all the relevant files for its solution. Please note that the solutions provided on *my*Unisa are only suggested solutions and are not the best or only solutions.

This tutorial letter contains a short discussion of the solutions to Assignment 3 of COS2614 made available on *my*Unisa. Hence, this tutorial letter should be used in conjunction with the solutions available on *my*Unisa.

The marking schedule used for marking Assignment 3 is also available on *my*Unisa under Additional Resources. It is impossible to follow a marking schedule strictly for a programming assignment. Hence, the given marking schedule should be used only as a rough guideline.

2 TUTORIAL MATTER AVAILABLE SO FAR

COS2614/101/2018*	First tutorial letter
COS2614/MO001/2018^	Contents of Learning Units on *my*Unisa
COS2614/102/2018^	Additional Notes
COS2614/201/2018^	Discussion of solutions to assignment 1
COS2614/202/2018^	Discussion of solutions to assignment 2
COS2614/203/2018^	Discussion of solutions to assignment 3 (This tutorial letter)

If you did not receive all the tutorial letters please download them from *my*Unisa.
*available under Official Study Material on *my*Unisa
^available in the folder Tutorial Letters under Additional Resources on *my*Unisa

3 MARKING OF ASSIGNMENT 3

Questions 1 to 2 were marked out of 50 marks each. Assignments that contained solutions copied from other students' assignments were penalized heavily. Submissions that didn't compile were also awarded 0 marks.

4 DISCUSSION OF SOLUTIONS TO ASSIGNMENT 3

4.1 Question 1

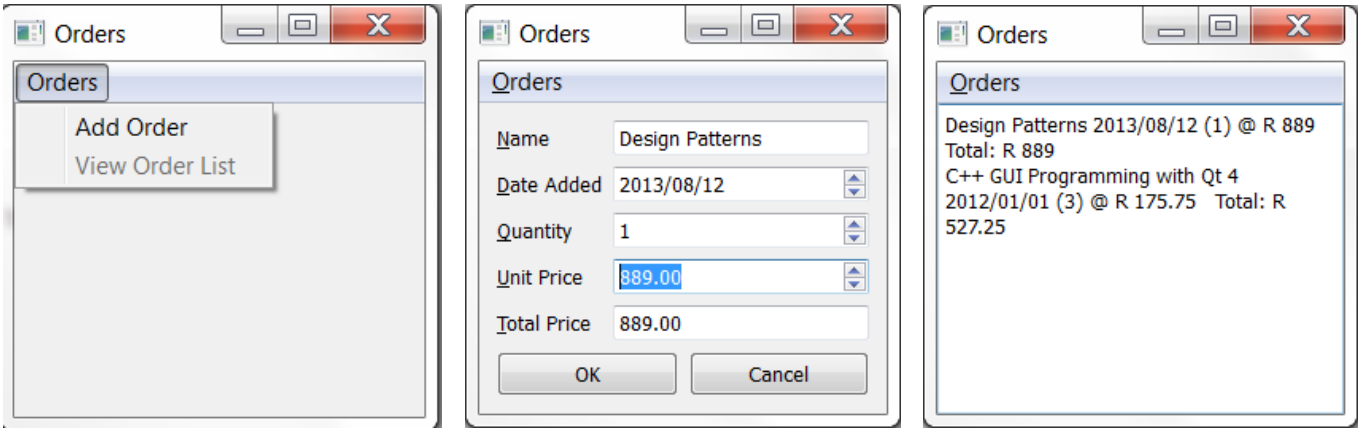
The aim of this question is to expand your knowledge of Graphical User Interface programming by including menus on the user interface as well as using `QAction`s for event handling.

Some comments on this solution:

- We have two model classes named `Order` and `OrderList`. `Order` represents an order and `OrderList` represents a list of `Order`s. `OrderList` uses a data member `QList<Order>` to handle a number of `Order`s.
- We have created a separate class named `OrderForm`, which is a specialized `QWidget` that creates and manages user input for an order.
- In the main window, we used a `QStackedWidget` as the central widget, where we switch between either no widget or an `OrderForm` or the `QTextEdit` based on the menu option selected by the user.

- Note how we have used signals and slots to update `OrderList` in `MyMainWindow` class. When the user clicks on `OK` button on the order form, an `OrderForm` reads data from the input widgets, creates an instance of `Order` and subsequently emits a signal `orderCreated()` with the `Order` created. An instance of `MyMainWindow` class listens to this signal and when emitted executes the slot `addOrder()`. This slot adds the `Order` to an `OrderList` and activates the menu option `View Order List`.

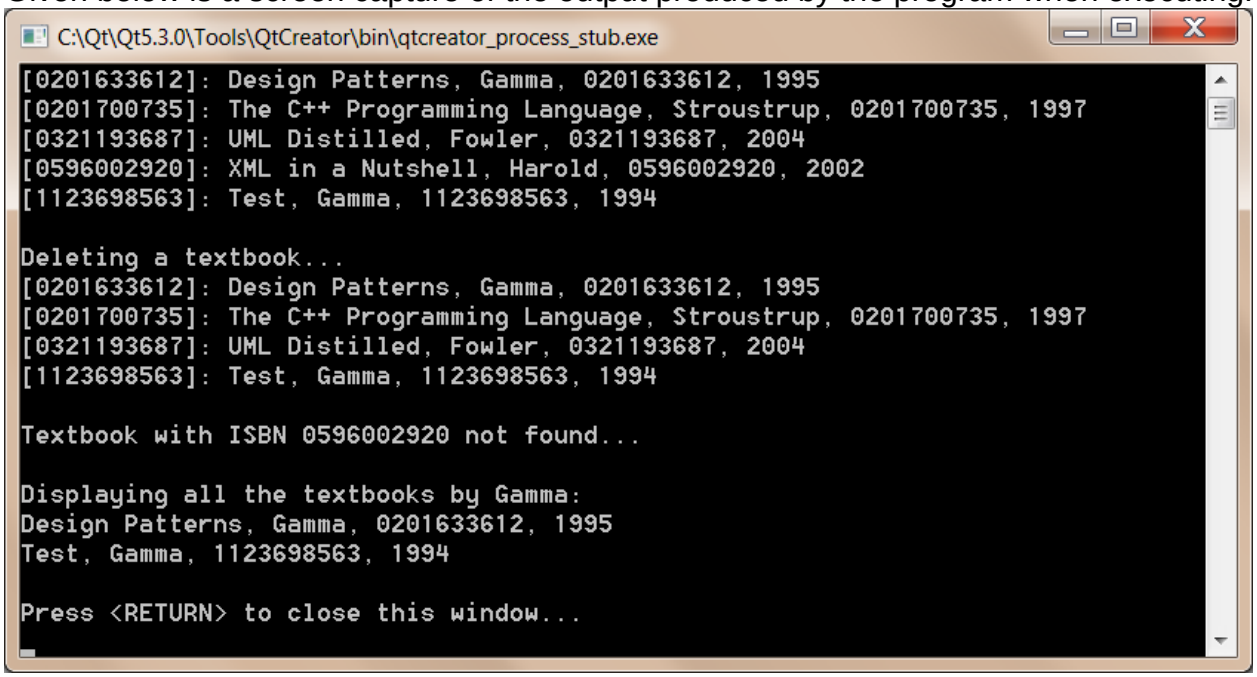
Given below are three sample screen prints of the output generated by the program:



4.2 Question 2

The aim of this question is to introduce you to the Qt class `QMap`.

Given below is a screen capture of the output produced by the program when executing:



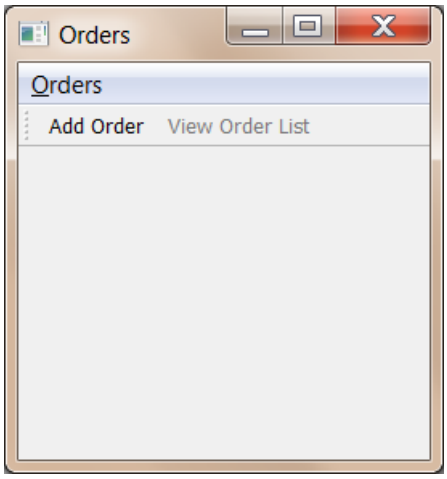
Some comments on this solution:

- We have separated the class definitions and implementations of `Textbook` and `TextbookMap` into separate header and cpp files respectively.
- Ezust did not provide an implementation of the `Textbook` class. So you had to implement `Textbook` according the given definition.

4.3 Question 3

Similar to question 1, the aim of this question is to expand your knowledge of Graphical User Interface programming by including menus and toolbars on the user interface as well as using `QAction`s for event handling.

Given below is a sample screen print of the output generated by the program:



4.4 Question 4

Ezust Chapter 10, Section 10.8: Review Question 1
The main features of `QMainWindow` are a menu bar, tool bars, dock regions, a status bar and a central widget. A `QMainWidow` has a default menu bar, but merely provides facilities for tool bars, dock widgets, a status bar and a central widget, which have to be added explicitly.

Ezust Chapter 10, Section 10.8: Review Question 2
Invoking the function `menuBar()` on the `QMainWindow` instance returns its default `QMenuBar` instance to which you can add a menu using the `addMenu()` function. A different menu bar can be installed by using `setMenuBar()`.

Ezust Chapter 10, Section 10.8: Review Question 3
The current state of a GUI app (in terms of its size, position and arrangements of its widgets) is maintained in a `QByteArray`. You can save and later restore the state by using the `QSettings` class. The `setValue()` member function can be used to save the state, and `value()` to restore it. The `QSettings` class writes and reads the values to and from the system registry (in Windows).

Ezust Chapter 10, Section 10.8: Review Question 4
A central widget is a `QWidget`, i.e. either a Qt widget such as `QTextEdit` or `QGraphicsView` or a programmer-defined widget which inherits from `QWidget`, which is placed in the center of a `QMainWindow`.

Ezust Chapter 10, Section 10.8: Review Question 5
A `QWidget` can be made the central widget by invoking the function `setCentralWidget(QWidget* w)` on the `QMainWindow`.

Ezust Chapter 10, Section 10.8: Review Question 6

Dock widgets are for attaching secondary windows, usually containing other widgets, to a `QMainWindow`.

Ezust Chapter 10, Section 10.8: Review Question 7
You can have any number of dock widgets in an application even though they should be contained within the four docking areas. There are different settings available for making different dock widgets in a single region accessible to the user.

Ezust Chapter 10, Section 10.8: Review Question 8
Dock widgets are useful to display multiple views of the data or different tools to manipulate data in the same application.

Ezust Chapter 10, Section 10.8: Review Question 9
A `QAction` is an object representing something that the user can do to the program. It can be triggered through menu items, button clicking, keyboard presses, etc.

Ezust Chapter 10, Section 10.8: Review Question 10
By creating a unique `QAction` for each action, you can add it to different menus and toolbars, and from one place, disable/enable it, or change other properties (such as its label or tool tip) and it will reflect in all views (menu items, toolbar buttons, etc) that refer to it.

Ezust Chapter 10, Section 10.8: Review Question 11
A `QActionGroup` is an object representing a group of `QAction`s in an application. It is useful to use an action group when multiple events triggered from multiple views on the user interface should result in the execution of the same event handling code. You can also use `QActionGroup` when only one action in the action group should be active at a time.

Ezust Chapter 10, Section 10.8: Review Question 12
One of the easiest features that can be included to support internationalization is to use the `QObject::tr()` function with any literal string. The string specified using the `tr()` function is then extracted for translation purposes when a translation is requested.

4.5 Question 5

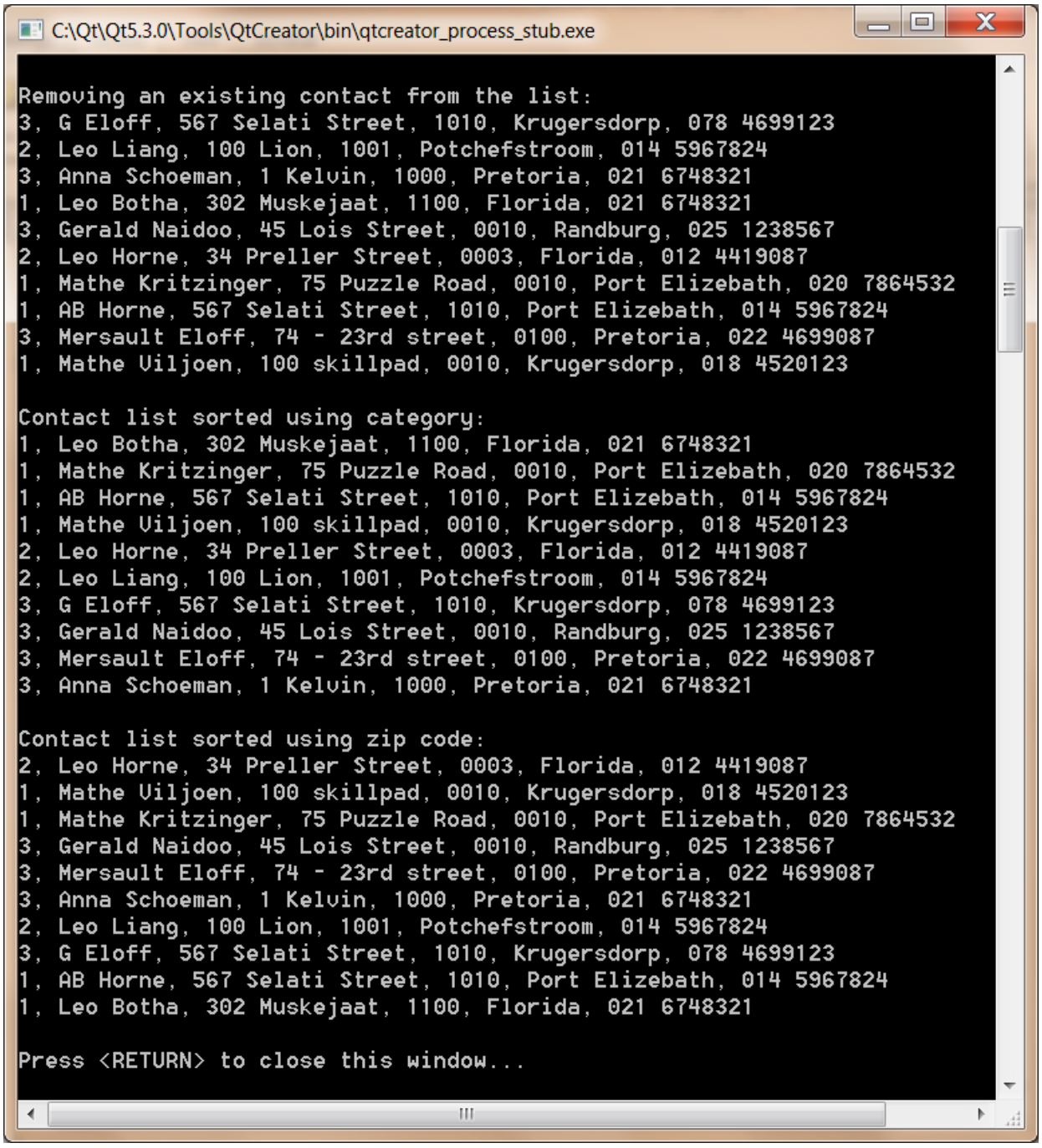
The aim of this exercise is to deepen your understanding of `QList`. You were expected to add four more features to the `ContactList` class.

Some comments on this code:

- When overloading the `+=` operator, we simply invoke the `add()` function that is already defined and implemented in `ContactList`.
- Similarly when overloading the `-=` operator, we simply invoke the `remove()` function that is already in `ContactList`.
- We have utilized bubble sort to sort contacts using category and zip code.
- We have used two new functions `compareCategory()` and `compareZip()` that each take two contacts and returns whether the category/zip of one contact is ‘less than’ that the category/zip code of the other contact.

Please note that we consider using operator overloading like this to add or remove elements from a container to be bad programming practice. Addition and subtraction operators should be used for numerical addition and subtraction. It is far better to use member functions with meaningful names (like `add()` and `remove()`) to manipulate containers. Our `main()` function therefore uses `add()` and `remove()` rather than `operator+=` and `operator-=`.

Given below is the sample screen print of the output generated by the program:



4.6 Question 6

Ezust Chapter 11, Section 11.7: Review Question 1
A function parameter must be a literal, a constant or a variable, whereas a template parameter can also be (and most commonly is) a *type expression* such as `int`, `QObject` or `Person*`.

Ezust Chapter 11, Section 11.7: Review Question 2
To instantiate a template function, you provide actual template parameters (together with the function arguments) to get the compiler to generate the template code of the function for the specific type you are using. Alternatively, you can simply call the function with arguments of some type, and the compiler infers the types of the template parameters from this, and instantiates the template function accordingly.

Ezust Chapter 11, Section 11.7: Review Question 3
The compiler generates a separate version of a templatised class every time it encounters the class name with a different template parameter type. In order to create a version of the class

with the specified template parameter type, the compiler has to know the complete definition and implementation of the class, The simplest way is to put the implementations of all the member functions into the same header file, although there are tricks that can be used to keep them in a separate file.

Ezust Chapter 11, Section 11.7: Review Question 4
One of the tricks to keep the definition of a templatised class in a header file and its implementation in a source file is to use the `export` keyword.

Ezust Chapter 11, Section 11.7: Review Question 5
Types that are inappropriate to store in value containers are `QObject`s, polymorphic collections in general, and any class that cannot be copied because it has no copy constructor.

Ezust Chapter 11, Section 11.7: Review Question 6
`QMap` and `QHash` are both associative container classes, providing mappings from keys to values. In other words, they both store (*key*, *value*) pairs. For a `QMap`, the type of *key* must support `operator<`, whereas for a `QHash`, the type must support `operator==` and it must implement `qHash()` as a global hash function. Apart from these differences, a `QMap` is slower than a `QHash` for insertions and lookups, particularly for large amounts of data.

Ezust Chapter 11, Section 11.7: Review Question 7
If a container class internally stores its data on the heap by means of pointers and takes responsibility for freeing (deallocating) the heap memory used by this data, it is called a managed container. In other words, when a managed container is destroyed, it calls `delete` on all the pointers to the data which it contains.

Ezust Chapter 11, Section 11.7: Review Question 8
`QString`, `QStringList` and `QVariant` use implicit sharing.

Ezust Chapter 11, Section 11.7: Review Question 9
When defining a class template, the definition and implementation of the templatised class should be combined in the header file so that the pre-compiler can create a copy of the class with a specified template parameter when instantiating an object of the templatised class.