

CSCI222

Exercise 2

Mostly Qt

4 marks

(must be demonstrated in a laboratory class in week 4, 5, or 6)

Aims:

This exercise introduces the Qt packages that can be used to build graphical user interfaces for C++ programs.

Objectives:

On completion of this exercise, students should be able to:

- Create a simple GUI for a C++ program using procedural code to build an interface using basic Qt GUI classes;
- Use QtBuilder to compose interfaces within a graphical editor that generates the boiler-plate Qt/C++ code needed to populate windows with GUI widgets;
- Use the boost XML serialization libraries to save complex data structures to disk;
- Implement simple interactive GUI based programs in C++.

Overview

Text based interfaces (menu-select etc) are so very 1980s. Nowadays, essentially all programs that have any need for direct user interaction will require graphical interfaces. Web-based interfaces are often required. For cases where web use is inappropriate, modern languages like C# and Java supply standard GUI classes and frameworks. It's a bit more problematical with C++.

Microsoft does its best to make Microsoft defined GUI libraries the “standard” for Windows applications built with Visual Studio C++. In the Linux world, there are several competing graphics libraries, but the most successful appears to be Qt (which can also be used on Windows).

Qt started as a project by a few Norwegians in need of a good C++ graphical user interface toolkit. The original TrollTech company that created Qt was taken over by Nokia. For several years, Nokia did most of the development work on this toolkit. Nokia's problems competing with iToys lead to the sale of the Qt software to a services company Digia. Most of Qt is free; there are some commercial extensions; Digia can act as a consultancy for commercial developers.

Qt involves extensions to the standard C++ language. Qt classes contain new syntactic elements such as “slots” and “signals”. Qt code has to pass through a pre-compiler that converts the code into conventional C++. Editing, compilation, and linkage of Qt programs can be a little complex when done at the command line. Fortunately, IDEs like NetBeans provide automation tools that simplify the entire build process.

Qt has many libraries. There are libraries for basic windows, for “widgets” (the standard text-boxes, check-boxes, radio-buttons etc as are used in all GUI systems), its own collection classes, classes that simplify the construction of multi-threaded programs, its own wrappers for TCP/IP communication, XML parsers, and a whole lot more. In addition to the libraries, a Qt installation will generally include a number of helper applications such as **QtAssistant** (demonstrated in exercise 1, this is essentially a tool for viewing reference documentation), **QtDesigner** (a GUI editor for building GUI interfaces for your programs), and **QtLinguist** (which helps create “resources”, e.g. sets of error messages and prompts, needed if you have to create a program that supports multiple languages English/French/German/...).

The QtAssistant program has a considerable amount of tutorial material (see below) along with the API documentation for Qt classes. You should work through some of these tutorials in addition to completing the tasks defined for this exercise. The actual code is all provided in the Qt tutorials, so getting the Qt examples to run is simply a matter of cutting and pasting code from the tutorial into files that you create within NetBeans projects. The tutorials take a very careful incremental approach, providing lots of explanatory commentary on each new feature introduced.

The same old ...

You should already have built GUIs.

In CSCI110, you constructed GUI interfaces by composing HTML “form pages”. Your “code” was written in HTML, but it was still code. The code consisted of instructions for the browser to layout the GUI, handle most of the GUI events automatically, and call your Javascript code for a few specially chosen events. You will have used HTML <table> or HTML <fieldset> components to layout the elements of your GUI; maybe you will have used CSS as an alternative layout approach. You defined your event handling by adding *onEvent* (e.g. onclick, onmouseover, onsubmit, onchange) attributes to chosen interactive HTML elements. The attribute values would have been Javascript calls to functions that you had written to handle selection of an element, to show some highlighting, or to check entered data before sending the data to a remote web-server.

In CSCI213, you will have used the swing libraries and written Java code to create JFrame windows. You would have used instances of Java layout managers (GridLayout, GridBagLayout etc) to organise the placement of interactive components such as instances of JButton. You would have handled events by first defining classes that implement interfaces such as ActionListener (something that will handle action events such as those generated by clicking a JButton); your class would have had an actionPerformed() method that did whatever processing was required (comparable to your Javascript functions that handled events in a web page). In your overall Java program, you would have created an instance of your actionlistener class and then added that instance as an “event listener” on some element (JButton or other) in your GUI.

Hopefully, CSCI213 will also have covered the modern way of building a Java GUI using a GUI builder such as the one that is a part of NetBeans. Modern Java code relies entirely on auto-generated GUI handling elements that employ quite different layout managers from the long out-dated GridBagLayout and others. In modern Java GUI building, an automatic code generator is used to create the code for all the low level mechanics of instantiating GUI widgets, placing them in a window, registering event listeners and so forth. The code generator leaves you with stub functions that you must complete by writing code to actually perform some action when the event occurs.

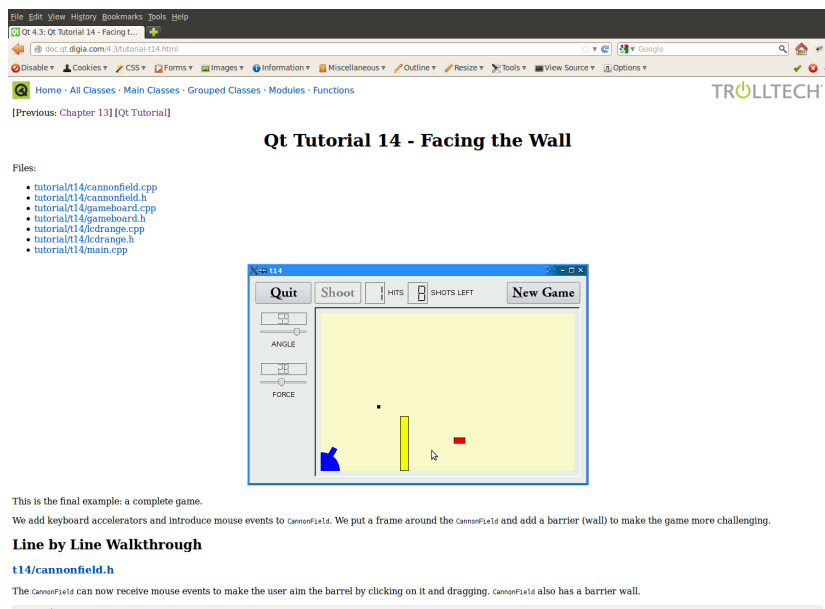
Qt is just the same:

- A few differences in terminology.
- A slightly different way of linking event sources (like action buttons) to handler functions.

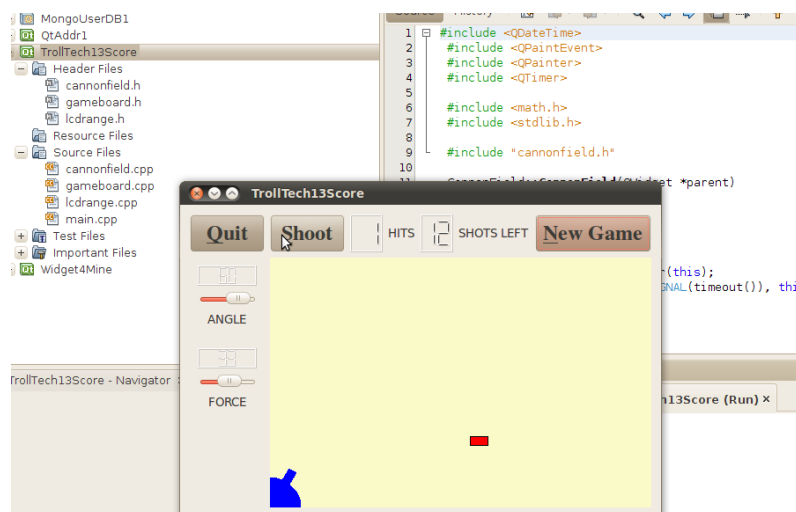
Do try some of the Qt supplied tutorials!

The tasks given below for this exercise cover only a small aspect of Qt and don't really delve far into its event handling. The tasks focus on the kinds of GUI that you might need to build in a simple CSCI222 assignment or in a very basic CSCI321 project.

The [initial Qt tutorial](http://doc.qt.digia.com/4.3/tutorial-t14.html) (<http://doc.qt.digia.com/4.3/tutorial-t14.html>) shows how to build an interactive game – built in 14 steps (!) each adding a tiny bit more functionality, each step being fully explained in lengthy commentary. It's a great way to get a basic understanding of Qt's event handling, and to get a feel for some of its widget classes.



It is worth completing that tutorial, all of its steps! As the code is supplied complete, file-by-file, its just a matter of cutting and pasting into a series of NetBeans Qt projects. It will take you about one hour; you should try it in your own time (not in the CSCI222 labs).



Next, look at the tutorials in QtAssistant (look for Qt Assistant in 'Dash Home' as Qt 4 Assistant):

Contents Index Bookmarks Search Qt 4.6: Qt Examples Qt 4.6: Tutorials

Qt Assistant Manual
Qt Designer Manual
Qt Linguist Manual
QMake Manual
Qt Reference Documentation
Classes
Tutorials and Examples
Overviews

Home · All Classes · All Functions · Overviews

[How to Learn Qt] [Next: Qt Examples]

Tutorials

A collection of tutorials and "walkthrough" guides are provided with Qt to help new users get started. These documents cover a range of topics, from basic use of widgets to step-by-step tutorials that show how an application is put together.

Widgets
A beginner's guide to getting started with widgets and layouts to create GUI applications.

Address Book
A simple application that demonstrates how to create a simple application.

Qt Designer
A quick guide through Qt Designer showing the basic steps to create a form with this interactive tool.

Qt Linguist
A guide to translating the tool development release.

The AddressBook tutorial should be attempted, again in your own time, as it illustrates different approaches from those taken in the tasks below and gives a feel for how to build editing and correction aspects into an application.

The Widgets examples contain many simple exercises:

Qt comes with a large range of standard widgets.
You can also develop your own custom widgets.
It is even possible to provide custom styles for standard widgets and appropriately write

- [Analog Clock*](#)
- [Calculator*](#)
- [Calendar Widget*](#)
- [Character Map*](#)
- [Code Editor*](#)
- [Digital Clock*](#)
- [Group Box*](#)
- [Icons*](#)
- [Image Viewer*](#)
- [Line Edits*](#)
- [Movie*](#)
- [Scribble*](#)
- [Shaped Clock*](#)
- [Sliders*](#)
- [Soft Keys*](#)
- [Spin Boxes*](#)
- [Styles*](#)
- [Style Sheet*](#)
- [Tablet*](#)
- [Tetrix*](#)
- [Tooltips*](#)
- [Validators*](#)
- [Wiggly*](#)
- [Window Flags*](#)

More and more Qt tutorials -

Search Qt 4.6: Qt Examples Qt 4.6: Widgets Examples

Main windows

Layouts

Item Views

Graphics View

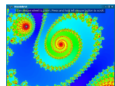
All the standard features of application main windows are provided by Qt. Main windows can have pull down menus, tool bars, and dock windows. These separate forms of user input are unified in an integrated action system that also supports keyboard shortcuts and accelerator keys in menu items.

Qt uses a layout-based approach to widget management. Widgets are arranged in the optimal positions in windows based on simple layout rules, leading to a consistent look and feel. Custom layouts can be used to provide more control over the positions and sizes of child widgets.

Item views are widgets that typically display data sets. Qt 4's model/view framework lets you handle large data sets by separating the underlying data from the way it is represented to the user, and provides support for customized rendering through the use of delegates.

Qt is provided with a comprehensive canvas through the [GraphicsView](#) classes.

Threading and Concurrent Programming



Qt 4 makes it easier than ever to write multithreaded applications. More classes have been made usable from non-GUI threads, and the signals and slots mechanism can now be used to communicate between threads.

The [QtConcurrent](#) namespace includes a collection of classes and functions for straightforward concurrent programming.

Tools



Qt is equipped with a range of capable tool classes, from containers and iterators to string handling and manipulation.

Other classes provide application infrastructure support, handling plugin loading and managing configuration files.

Network



Qt is provided with an extensive set of network classes to support both client-based and server-side network programming.

Inter-Process Communication



Simple, lightweight inter-process communication can be performed using shared memory or local sockets.

WebKit



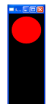
Qt provides an integrated Web browser component based on [WebKit](#), the open source browser engine.

Help System



Support for interactive help is provided by the Qt Assistant application. Developers can take advantage of the facilities it offers to display specially-prepared documents in their applications.

State Machine



Qt provides a powerful hierarchical finite state machine through the Qt State Machine class.

Animation Framework



These examples show how to use the [animation framework](#) to build highly performant GUIs.

Multi-Touch Framework



Support for multi-touch input makes it possible for developers to create extremely intuitive user interfaces.

Painting



Qt's painting system is able to render vector graphics, images, and outline font-based text with sub-pixel accuracy using anti-aliasing to improve rendering quality.

Rich Text



Qt provides a powerful document-oriented rich text engine that supports Unicode and right-to-left scripts. Documents can be manipulated using a cursor-based API, and their content can be imported and exported as both HTML and in a custom XML format.

Desktop



Qt provides features to enable applications to integrate with the user's preferred desktop environment.

Features such as system tray icons, access to the desktop widget, and support for desktop services can be used to improve the appearance of applications and take advantage of underlying desktop facilities.

Drag and Drop



Qt supports native drag and drop on all platforms via an extensible MIME-based system that enables applications to send data to each other in the most appropriate formats.

Drag and drop can also be implemented for internal use by applications.

OpenGL and OpenVG Examples



Qt provides support for integration with OpenGL implementations on all platforms, giving developers the opportunity to display hardware accelerated 3D graphics alongside a more conventional user interface.

Qt provides support for integration with OpenVG implementations on platforms with suitable drivers.

Multimedia Framework



Qt provides low-level audio support on Linux, Windows, and Mac platforms by default and an audio plugin API to allow developers to implement their own audio support for custom devices and platforms.

The Phonon Multimedia Framework brings multimedia support to Qt applications.

SQL



Qt provides extensive database interoperability, with support for products from both open source and proprietary vendors.

SQL support is integrated with Qt's model/view architecture, making it easier to provide integration for your database applications.

XML



XML parsing and handling is supported through SAX and DOM compliant APIs as well as streaming classes.

The XQuery/XPath and XML Schema engines in the [QtXmlPatterns](#) modules provide a framework for querying XML files and custom data models.

If you do end up using C++ in a CSCI321 project, you will almost certainly have to delve into some of those tutorial examples.

In addition, QtAssistant contains an “Overview” section. This contains detailed explanations of the conceptual structure of GUI applications. For example, you will have heard a little about the “Model-View-Controller” paradigm in CSCI204/CSCI205/CSCI222. The commentary in QtAssistant is much more complete:

Contents

- Qt Assistant Manual
- Qt Designer Manual
- Qt Linguist Manual
- QMake Manual
- Qt Reference Documentation
 - Classes
 - Tutorials and Examples
 - Overviews
 - An Introduction to Model/V...
 - Using Models and Views
 - Model Classes
 - Creating New Models
 - View Classes
 - Handling Selections in It...
 - Delegate Classes
 - Item View Convenience Cl...
 - Using Drag and Drop with ...
 - Proxy Models
 - Model Subclassing Refere...

Qt

Home · All Classes · All Functions · Overviews

[Previous: [Model/View Programming](#)] [Next: [Using Models and Views](#)]

An Introduction to Model/View Programming

- The Model/View Architecture
 - Models
 - Views
 - Delegates
 - Sorting
 - Convenience Classes
- The Model/View Components

Qt 4 introduces a new set of item view classes that use a model/view architecture to manage the relation data and the way it is presented to the user. The separation of functionality introduced by this architecture developers greater flexibility to customize the presentation of items, and provides a standard model interface wide range of data sources to be used with existing item views. In this document, we give a brief introduction to the model/view paradigm, outline the concepts involved, and describe the architecture of the item view system. The components in the architecture is explained, and examples are given that show how to use the classes

The Model/View Architecture

Model-View-Controller (MVC) is a design pattern originating from Smalltalk that is often used when building interfaces. In [Design Patterns](#), Gamma et al. write:

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

If the view and the controller objects are combined, the result is the model/view architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles. This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures. To allow flexible handling of user input, we introduce the concept of the *delegate*. The advantage of having a delegate in this framework is that the way items of data are rendered and edited can be customized.

Data

Model

View

Delegate

Rendering

Editing

Rendering

The model/view architecture

The model communicates with a source of data, providing an *interface* to the data. The nature of the communication depends on the type of data source, and the way the model is implemented.

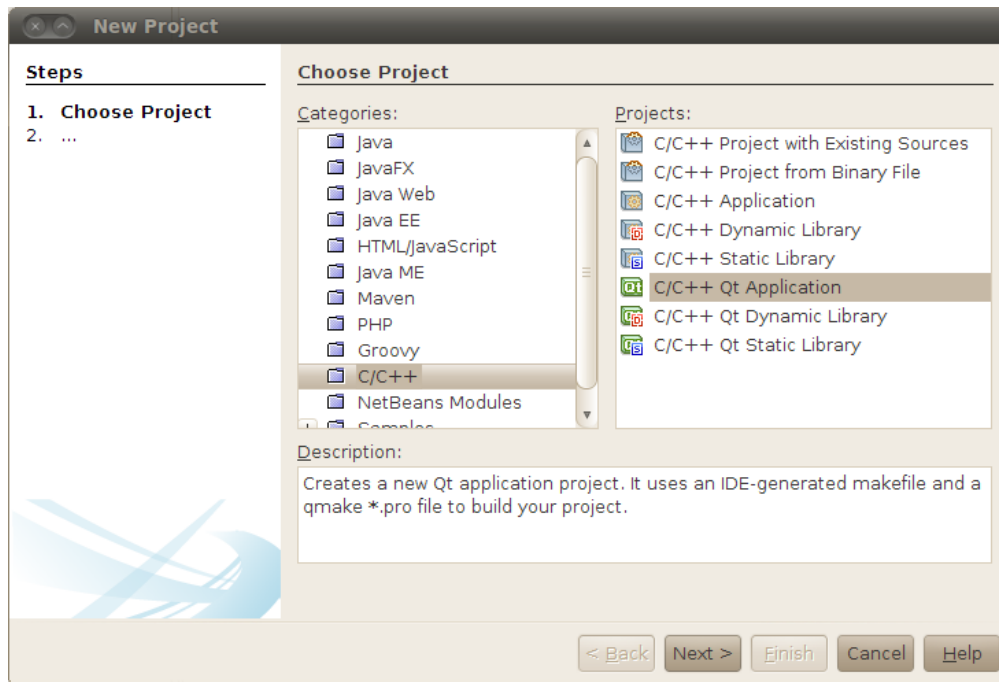
The view obtains *model indexes* from the model; these are references to the data. By supplying model indexes to the model, the view can retrieve the data from the data source.

In standard views, a *delegate* renders the items of data. When an item is selected, the delegate communicates with the model directly using model indexes.

2015

Task 1: A Simple Qt version of AddressBook

Create a new NetBeans project – this one is a C/C++ Qt Application:



This QtAddr1 project will use procedural code to generate a set of MyRecord objects, storing them in a STL vector. It then builds a simple Qt GUI that presents a tabular view of the data in the records.

Qt's approach to getting a tabular view of data is very similar to the approach in Java. The library (C++ Qt, or Java swing) provides a **generic table class**. The code for this class handles things like displaying column headers and, usually, working with a scrollbar mechanism if the number of data rows exceeds the number that can be displayed in the window. The generic table class must of course be able to determine how many columns will be needed, and must be able to access the data that are to go in any specific (row, column) cell of the table. The generic table class will work with an application defined **TableModel class** (the library will typically provide some AbstractTableModel base class). The programmer will define a suitable TableModel class; methods will supply the number of columns, the column headers, the values of cells (and, if editing is permitted on the table, there will be methods for changing values).

The actual data are going to be instances of some application defined record structure (they will be instances of MyRecord in this task). They will be held in some collection – e.g. STL vector, STL map, Java ArrayList, or whatever. The TableModel will have an instance member to store this collection. (My old [CSCI213 lab exercises](#) contain an example of how to build table models etc in Java.)

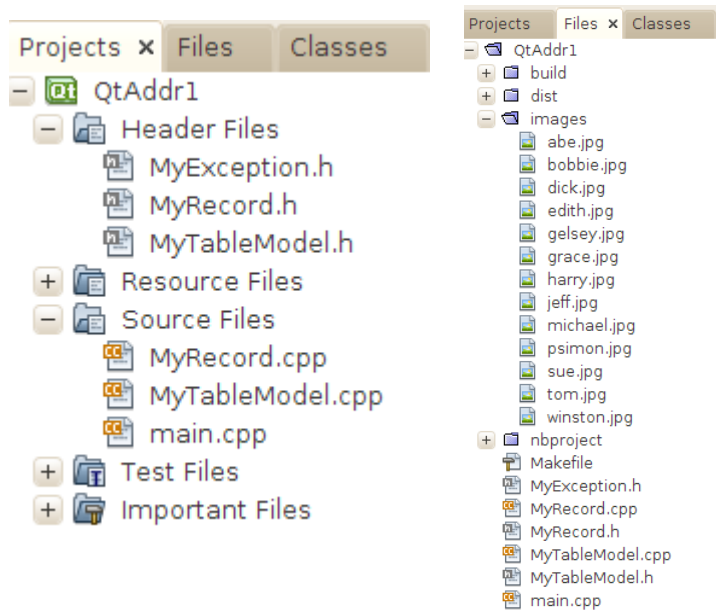
So you have:

- Table – a GUI library widget for display of data. It owns an instance of an application defined TableModel.
- TableModel – an auxiliary helper class that supplies the table class with information

such as the number of rows and columns in the table, the column headers, the values of cells. The TableModel will have an instance member that is some collection class that holds the actual data records.

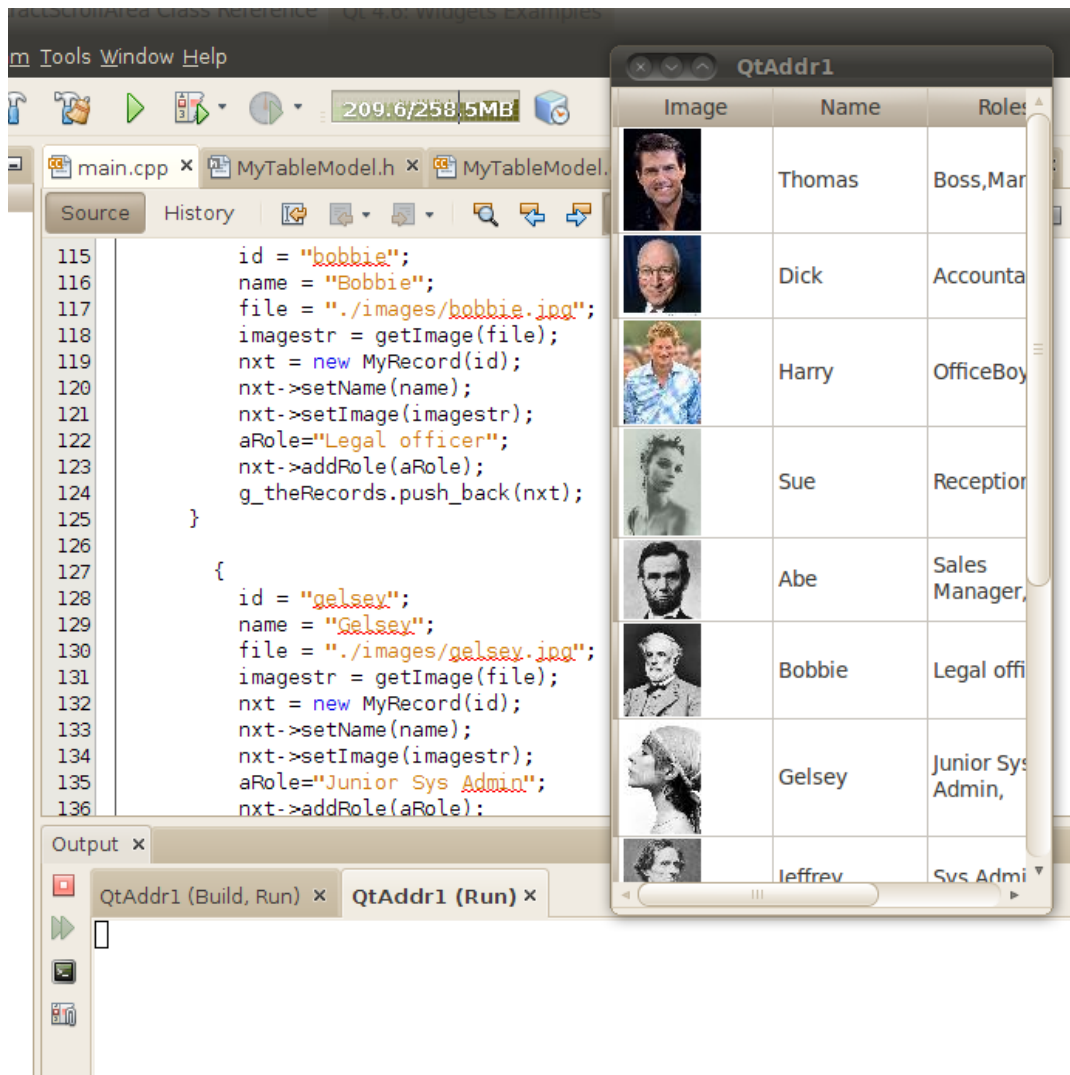
- Collection class for the data.
- Many instances of an application defined data record.

The application structure should be as shown here:



You don't need to specify any extra include directories, nor do you need to add link libraries. These issues are handled by the “qmake” build process.

(You will need to create a folder for images, and add ~10 images of people of your own choice.)



Classes:

MyException and MyRecord:

- no change from previous.

MyTableModel:

Class MyTableModel is a subclass of, extension of the Qt library's QAbstractTableModel. It's definition includes a Qt feature – Q_OBJECT:

Q_OBJECT?

Q_OBJECT is like a macro; it adds lots of code to this class – code that allows an instance of class MyTableModel to work with Qt signals (events). It's a necessary part of the class – but we don't (at this level of usage) need to understand anything about how it works.

QAbstractTableModel has numerous methods but we only need to change 4 in this simple example; we add an extra data member – recordsCollection – and provide a method to set this member.

```

main.cpp x MyTableModel.h x MyTableModel.cpp x MyRecord.h x MyRecord.cpp x
Source History
5  * Created on 3 December 2012, 12:24 PM
6  */
7
8  #ifndef MYTABLEMODEL_H
9  #define MYTABLEMODEL_H
10 #include <QAbstractTableModel>
11 #include <QtGui>
12 #include <vector>
13 using namespace std;
14 class MyRecord;
15 typedef MyRecord* RecordPtr;
16
17 class MyTableModel : public QAbstractTableModel {
18     Q_OBJECT
19 public:
20     MyTableModel(QObject *parent);
21     void addTheData(vector<RecordPtr> *data) { this->recordsCollection = data; }
22     int rowCount(const QModelIndex &parent = QModelIndex()) const;
23     int columnCount(const QModelIndex &parent = QModelIndex()) const;
24     QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
25     QVariant headerData(int section, Qt::Orientation orientation, int role) const;
26 private:
27     // Don't assume ownership, don't delete on destruct
28     vector<RecordPtr> *recordsCollection;
29     // Disallow value operations
30     MyTableModel& operator=(const MyTableModel&);
31     MyTableModel(const MyTableModel& orig);
32 };
33
34 #endif /* MYTABLEMODEL_H */

```

The use of the rowCount() and columnCount() methods is obvious – it allows the generic table display mechanism to format the table correctly.

```

* Author: naba
*
* Created on 3 December 2012, 12:24 PM
*/

#include <boost/foreach.hpp>
#include <string>
#include "MyTableModel.h"
#include "MyRecord.h"
using namespace std;

MyTableModel::MyTableModel(QObject *parent)
    :QAbstractTableModel(parent)
{ }

int MyTableModel::rowCount(const QModelIndex & /*parent*/) const
{
    return recordsCollection->size();
}

int MyTableModel::columnCount(const QModelIndex & /*parent*/) const
{
    return 3;
}

```

The `headerData()` method will return `QString` data objects that represent the headers for the table columns.

```
59  QVariant MyTableModel::headerData(int section, Qt::Orientation orientation, int role) const
61  {
62      if (role == Qt::DisplayRole) {
63          if (orientation == Qt::Horizontal) {
64              switch (section) {
65                  case 0:
66                      return QString("Image");
67                  case 1:
68                      return QString("Name");
69                  case 2:
70                      return QString("Roles");
71              }
72          }
73      }
74      return QVariant();
75  }
```

The `data()` method is used to retrieve cell data for a particular cell identified by the row and column arguments.

Qt differs a little here from the otherwise very similar Java `JTable` class. In Java, the display code makes a single request for cell data and determines how to display the data from the type of object returned. Qt makes multiple requests for cell data – the `DisplayRole` argument identifies what the display code requires. It can be text data, or it maybe decorations like images that are required. The implementation of the `data()` function must resolve these different requests. The code should return a suitable data object or, in some cases, an empty `QVariant` object.

```

QVariant MyTableModel::data(const QModelIndex &index, int role) const
{
    if (role == Qt::DisplayRole) {
        // role = DisplayRole : return any string data
        if (index.column() == 0) {
            // column 0 is for images, so just return placeholder
            return QVariant();
        }

        if (index.column() == 1)
            return QString(recordsCollection->at(index.row())->getName().c_str());

        if (index.column() == 2) {
            vector<string> grps = recordsCollection->at(index.row())->getRoles();
            string grplist;

            BOOST_FOREACH(string s, grps) {
                grplist.append(s).append(",");
            }
            return QString(grplist.c_str());
        }

    } else
        if ((role == Qt::DecorationRole) && (index.column() == 0)) {
            string imagedata = recordsCollection->at(index.row())->getImage();
            QByteArray imagebytes = QByteArray::fromBase64(imagedata.c_str());
            QImage img = QImage::fromData(imagebytes);
            return img;
        }
    return QVariant();
}

```

One point to note in my implementation of the data() method is the use of the Boost library's “for-each” construct. Here, I need to construct a single string that has strung together all the entries in the vector<string> roles attribute of a MyRecord. I could use STL iterators (vector<string>::const_iterator it=grps.begin() etc); but I'm really not a fan of STL iterators which I regard as clumsy and intrusive. I use them where I have to. I prefer a Perl style (or C#, or Java style) for-each loop. The Boost library supplies it.

Mainline:

The mainline code, nothing too hard:

```

#include <iostream>
#include <QtGui/QApplication>

#include "MyRecord.h"
#include "MyTableModel.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

static string getImage(string filename) {...21 lines }

static void createData() {...164 lines }

static void displayData(int argc, char *argv[]) {...11 lines }

int main(int argc, char *argv[]) {
    // initialize resources, if needed
    // Q_INIT_RESOURCE(resfile);
    createData();

    displayData(argc, argv);

    return EXIT_SUCCESS;
}

```

I have an auxiliary function `getImage()` that uses code similar to the little Qt Image loading example from exercise 1; it converts the image into a STL string:

```

static string getImage(string filename) {
    // Qt library has its own string class, convert STL string
    QString qtfilename(filename.c_str());
    QImage animage;
    bool readimage = animage.load(qtfilename);
    if (!readimage) {
        cout << "Image load failed for " << filename << endl << "Bye" << endl;
        exit(1);
    }

    QImage resized = animage.scaledToWidth(50, Qt::FastTransformation);

    QByteArray ba;
    QBuffer buf(&ba);
    resized.save(&buf, "JPG");

    QByteArray coded = ba.toBase64();
    string result(coded); // relying on QByteArray operator char*
    return result;
}

```

My `createData()` method simply creates a series of `MyRecord` data structures and adds the to the global `g_theRecords` collection:

```

static void createData() {
    // Hard code procedural creation of a few records so that can
    // have some data to show in the Qt based GUI
    RecordPtr nxt;

    string id;
    string name;
    string aRole;
    string imagestr;
    string file;

    // You will need to adjust filenames etc to match the image files
    // that you provide
    {
        id = "tom";
        name = "Thomas";
        file = "./images/tom.jpg";
        imagestr = getImage(file);
        nxt = new MyRecord(id);
        nxt->setName(name);
        nxt->setImage(imagestr);
        aRole="Boss";
        nxt->addRole(aRole);
        aRole="Manager";
        nxt->addRole(aRole);
        g_theRecords.push_back(nxt);
    }
    {
        id = "dick";
        name = "Dick";
        file = "./images/dick.jpg";
        imagestr = getImage(file);
    }
}

```

The really complex Qt display code:

All that is left to do is implement the code that builds the GUI – a window, containing a table, with scrollbars.

```

36  static void createData() {...}
198
199  static void displayData(int argc, char *argv[]) {
200      QApplication app(argc, argv);
201      QTableView tableView;
202
203      MyTableModel myModel(0);
204      myModel.addTheData(&g_theRecords);
205      tableView.setModel(&myModel);
206      tableView.resizeRowsToContents();
207      tableView.show();
208      int returnCode = app.exec();
209  }
210
211  int main(int argc, char *argv[]) {




```

Simple GUIs in Qt are really simple.

The code here:

- Create an instance of QApplication – essentially this means initialise the Qt graphics system.
- Create an instance of the QTableView, an instance of the table model class, and load the table model with some data.
The 0 argument to the model is passed to its base class constructor. What it is actually doing here is saying that there will be no parent GUI window – this model is to be displayed in a tableview that is the entire window.
- The tableview is linked to the model, and one of its display options is adjusted (I didn't want it using the default row size – which defaults to the height of a line of text – because I have those images; so it's to determine the height of each row before displaying the row);
- The table is shown.
- The Qt application's exec() method is called – this starts all the interactive event handling, like responding to the scrollbars.
The default behaviour is for app.exec()'s event-handling loop to terminate and for the function to return if the window's close box is clicked.



Image	Name	
	Thomas	Boss
	Dick	Acco
		

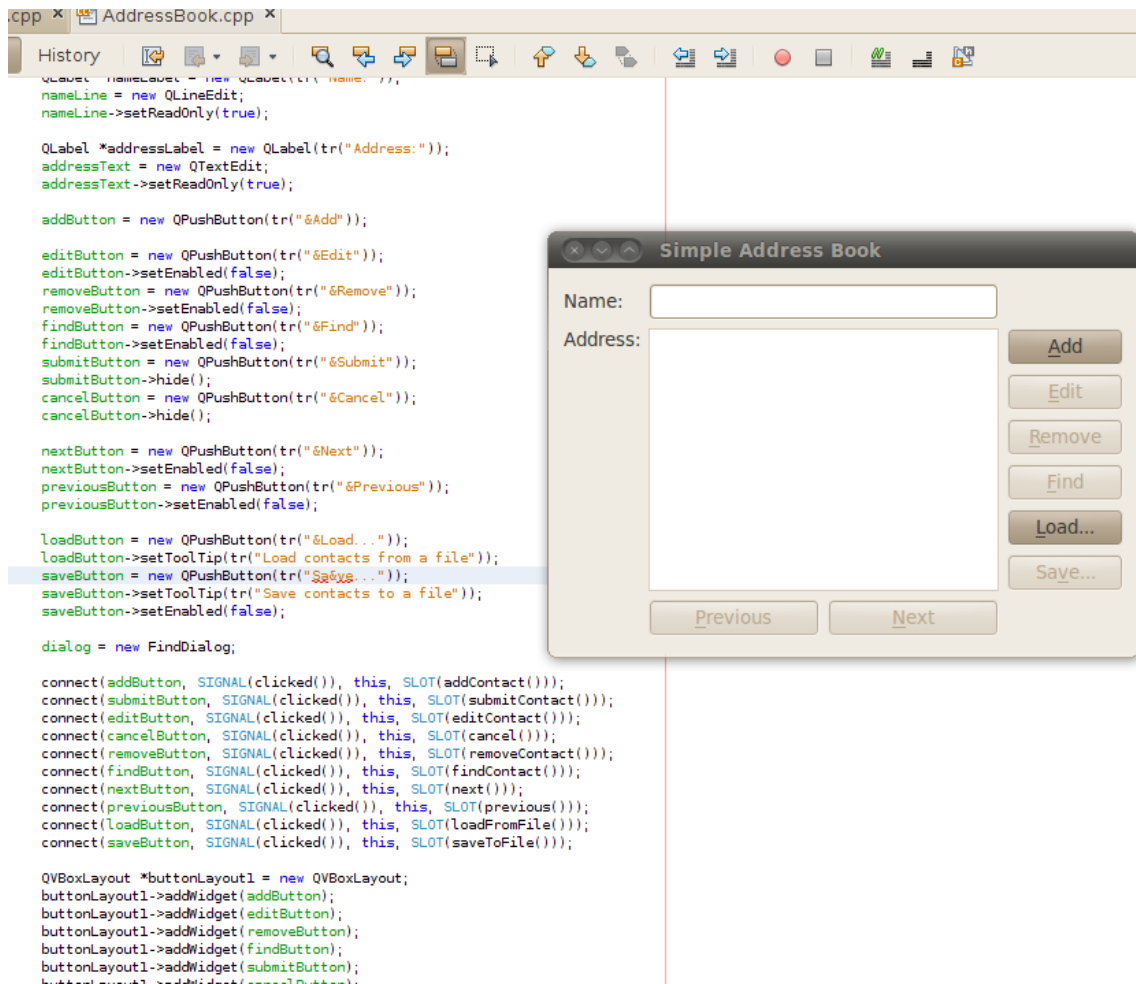
Task 1 – completion (1 mark)

Demonstrate your working QtAddr1 project.

Task 2: A more complex Qt GUI

Simple GUIs are simple in Qt. But by the time you are getting to something more elaborate, the procedural code to create all the widgets and link them together is getting a bit tiresome.

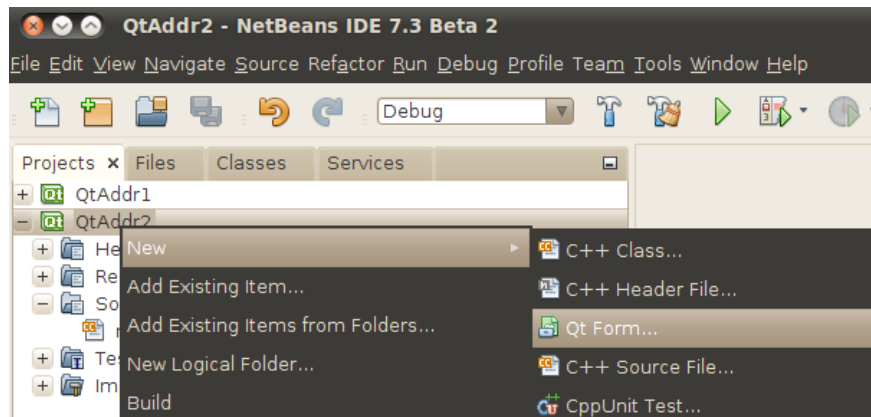
The following is from Trolltech's Qt AddressBook tutorial (it shows the interface and a part of the code needed to generate it):



It's not the kind of code that is fun to write.

So, try using QtBuilder – an interactive visual editor that lets you build a GUI by selecting and placing widgets in a symbolic window. QtBuilder generates the messy boiler-plate code.

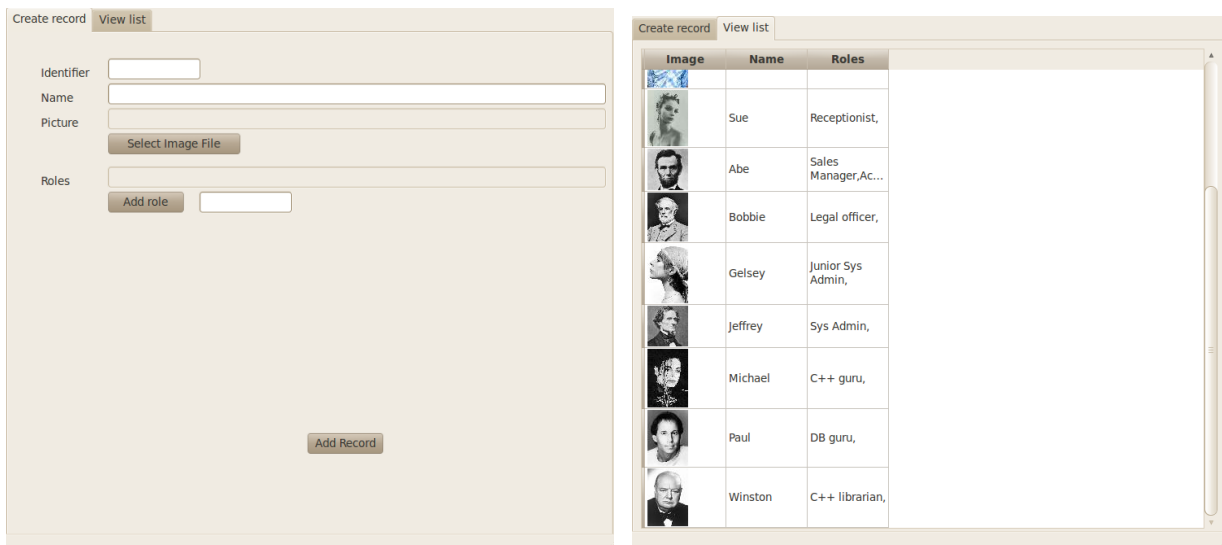
QtBuilder is integrated with NetBeans – you start a Qt project and then ask for a “new Qt form”. NetBeans temporarily passes control to QtBuilder; when you exit from QtBuilder, NetBeans picks up the generated files and adds them to your project.



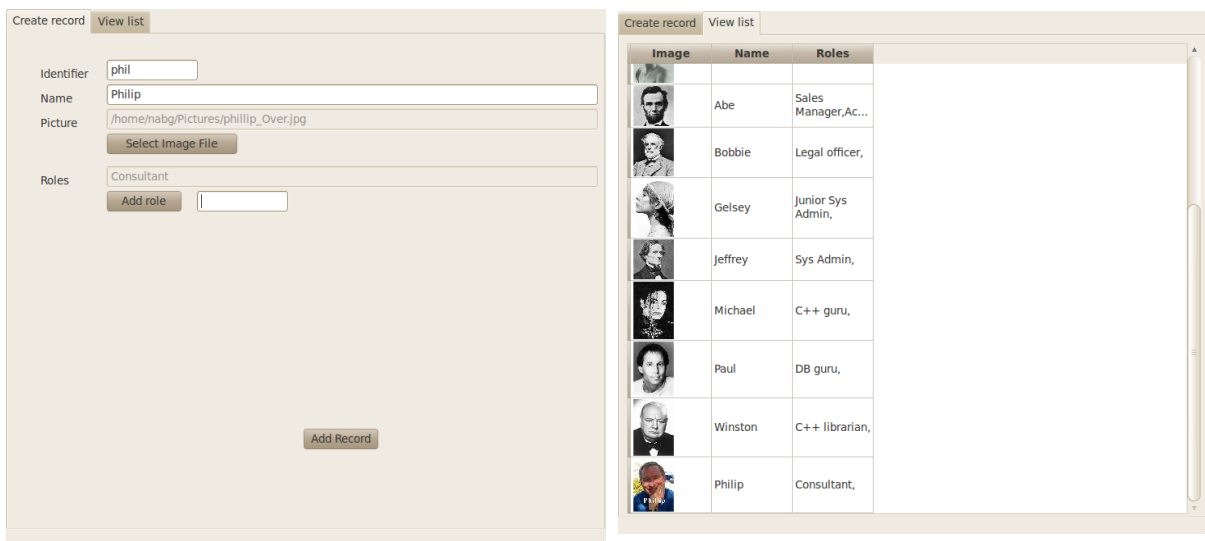
So, what is the application for which we want a GUI?

Another variation on the ongoing MyRecord exercise – now we want the ability to create instances of MyRecord via a data entry form, and have the list of records displayed.

A common idiom for applications that have data entry forms, list displays, record displays etc is the “Tabbed Pane Interface”. The main window has tabs that will approximate different “Use Cases” of the program – a tab for record creation, a tab to view a list of records etc. These tab panes will hold instances of standard GUI widgets.



Fill in the form, add the new record, scroll down the display:



The interface consists of

- A main window that holds a “tab” widget; the tab widget has two tabs (container widgets) – with “currentTabText” fields holding the names “Create record” and “View list”;
- The first tab QWidget has a number of labels, line edit fields, and push buttons;
- The second tab has a tableview similar to that illustrated in the last task.

(It is all done at a fairly naïve level; things like the table do not grow if the overall window is enlarged.)

This interface can be built using the QtBuilder application invoked from within NetBeans.

As shown above, you start by adding a new “Qt Form”; there is a choice for the basic style (dialog, mainwindow etc), in this case it's best to start as a main window:

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Form Name:

Form Type:

☒ Create C++ wrapper class

Project:

Folder:

Form File:

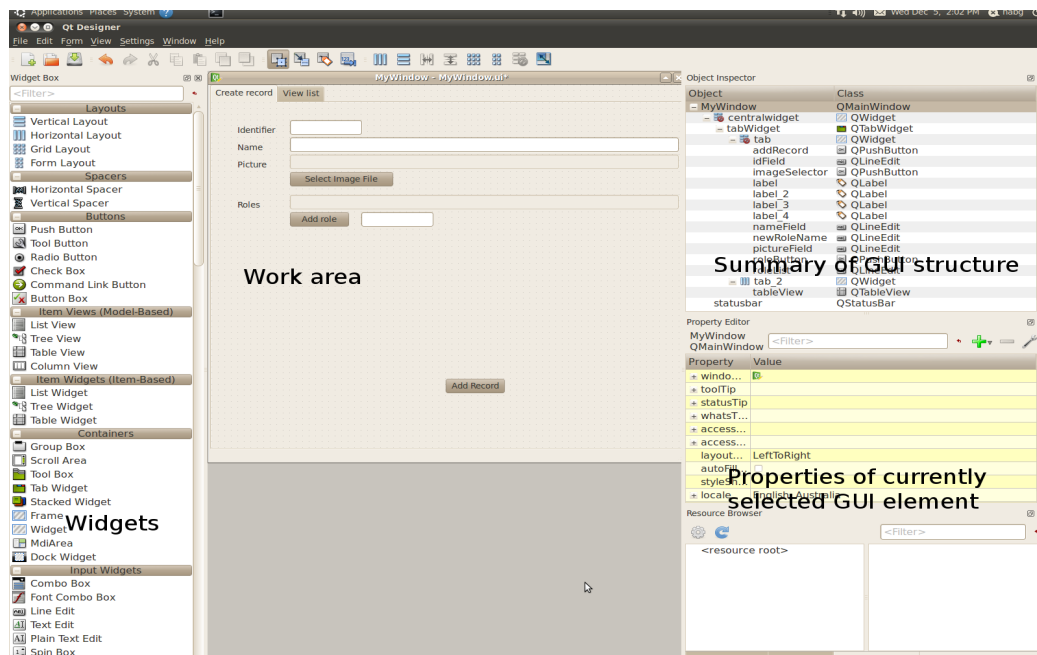
Class File:

Header File:

NetBeans creates three files – MyWindow.h, MyWindow.cpp, and MyWindow.ui; and starts QtBuilder. You can then edit the user interface in QtBuilder; on exit, it generates some more files that get hidden in your NetBeans project. If you need to modify your GUI, e.g. add some more input elements, you can later pick the MyWindow.ui file in NetBeans and “open” it – QtBuilder

again starts (sometimes it's a bit slow starting).

QtBuilder has a display with a work area, a palette of Widgets, and panes summarising the overall structure of the GUI and showing detailed properties of the currently selected GUI element.



The new Main window that you start with will be shown in the work area as blank apart from a “Menu Bar” at the top. We don't need the menu bar, so the first step should be to select it (right-click) and delete it.

Next, select the “Tab Widget” entry (in the Containers section of the Widget Box palette) and add a tab widget to the work area, resizing it to fill the window. A Tab Widget starts with two tabs; you should immediately change the display text to appropriate titles (“Create record” and “View List”).

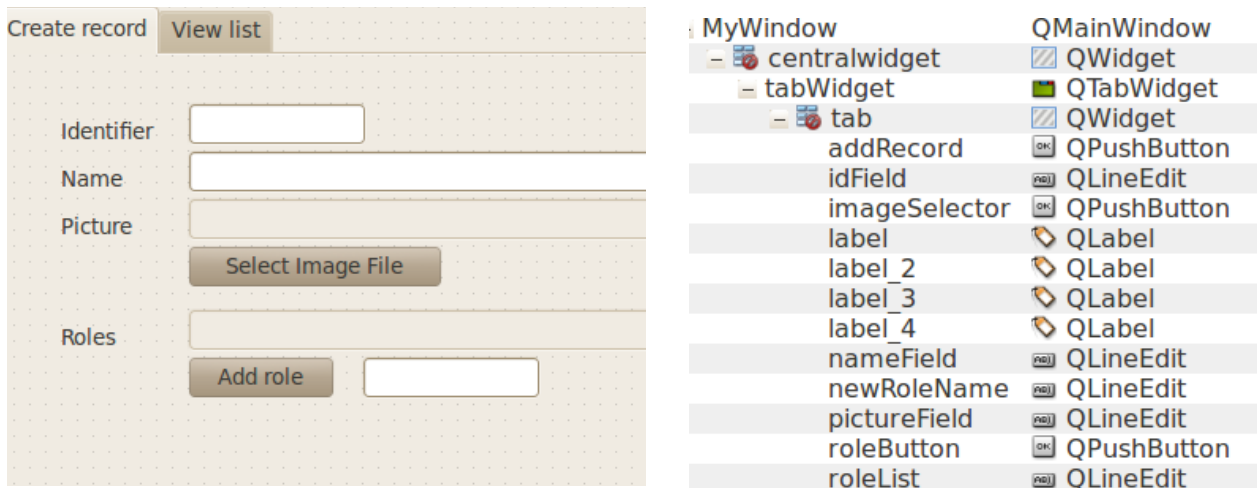
Then it is a matter of adding labels (from Widget Box/Display Widgets), “line edits” (from Widget Box/Input Widgets) and “push buttons” (from Widget Box/Buttons) to the first tab. When you add a widget to a GUI, QtBuilder adds some code to the C++ files it is composing. Its actually defining a new C++ class to represent your GUI; each added widget becomes a new (public) data member of that class; code is added to the constructor for the class that will instantiate an instance of the appropriate Qt widget class and adjust its coordinates to match the placement on the work area. QtBuilder assigns names to the elements as they are added - “label”, “label_2”, “label_3”, “lineEdit”, “lineEdit_2” etc. You should rename fields that your program will be manipulating; names like “idField” and “nameField” make the code much easier to understand than “lineEdit” and “lineEdit_2” (you don't manipulate things like labels in your own code, so you don't have to rename them).

Layout is somewhat crude. QtBuilder does work with a pixel grid that helps align widgets, but things are simply positioned at absolute (x, y) coordinates. You should follow my design (you could try to be more ambitious – but then sort out your own problems!) and have the following:

- A label “Identifier” and a line edit (idField);
- A label “Name” with another line edit (nameField)
- A label “Picture” with a third line edit (pictureField); this line edit is not to allow direct

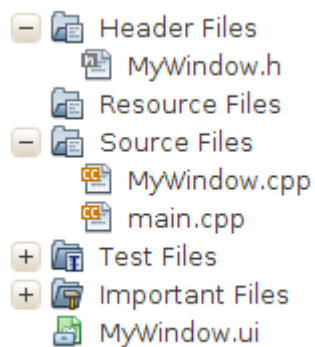
editing (it will be for the name of an input file selected by a dialog); so go to the properties pane and deselect the “Enabled” checkbox;

- A Push Button with text “Select Image File”, renamed as imageSelector;
- A label “Roles” and another of those disabled input fields (renamed as roleList);
- Another Push Button with text “Add Role” renamed as roleButton;
- Another line edit, renamed as newRoleName;
- A final Push Button, addRecord.



The other tab is simpler – it just holds a Table View (from Widget Box/Item Views (Model Based)). (Note that there is another 'Table Widget' – don't pick that.)

When you have finished creating the GUI in QtBuilder, save and exit. NetBeans will resume; your NetBeans project should now be something like the following:



There doesn't seem to be much there:

```

* File:    MyWindow.h
* Author:  nabq
*
* Created on 5 December 2012, 2:48 PM
*/

#ifndef _MYWINDOW_H
#define _MYWINDOW_H

#include "ui_MyWindow.h"

class MyWindow : public QMainWindow {
    Q_OBJECT
public:
    MyWindow();
    virtual ~MyWindow();
private:
    Ui::MyWindow widget;
};

```

```

* File:    MyWindow.cpp
* Author:  nabq
*
* Created on 5 December 2012, 2:48 PM
*/

#include "MyWindow.h"

MyWindow::MyWindow() {
    widget.setupUi(this);
}

MyWindow::~MyWindow() {
}

```

But, we can edit main:

```

#include <QtGui/QApplication>
#include "MyWindow.h"
int main(int argc, char *argv[]) {
    // initialize resources, if needed
    // Q_INIT_RESOURCE(resfile);

    QApplication app(argc, argv);

    // create and show your widgets here

    return app.exec();
}

```

and add the code to create and show our widgets:

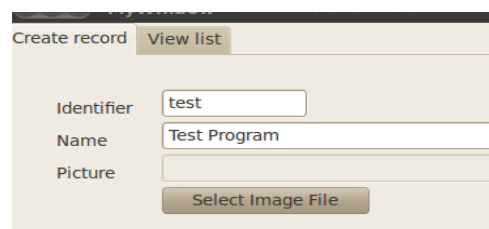
```

    QApplication app(argc, argv);
    MyWindow win;
    win.show();

    return app.exec();
}

```

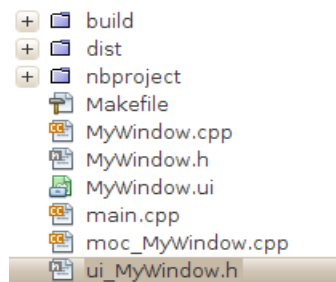
and it runs (sort of):



The line edit fields can be selected, and data can be edited; but of course, the buttons do nothing and the table view is empty.

But how does it work when there is no code there?

The code is hidden – if you switch from NetBeans project view to file view you can see the files:



The file ui_MyWindow.h contains the class declaration and constructor generated by QtBuilder (don't try editing this class file, just view it when you need the names of the widgets):

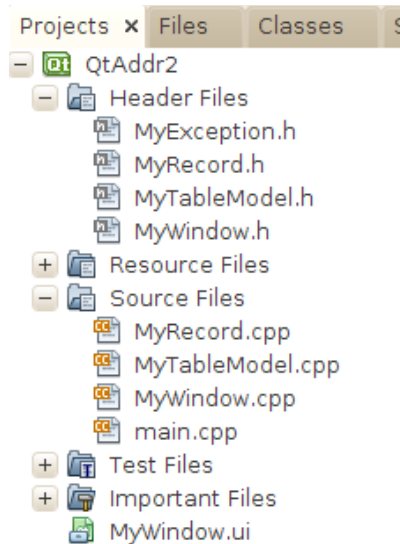
```
23 #include <QtGui/QStatusBar>
24 #include <QtGui/QTabWidget>
25 #include <QtGui/QTableView>
26 #include <QtGui/QWidget>
27
28 QT_BEGIN_NAMESPACE
29
30 @
31 class Ui_MyWindow
32 {
33 public:
34     QWidget *centralwidget;
35     QTabWidget *tabWidget;
36     QWidget *tab;
37     QLabel *label;
38     QLineEdit *idField;
39     QLabel *label_2;
40     QLineEdit *nameField;
41     QLabel *label_3;
42     QLineEdit *pictureField;
43     QPushButton *imageSelector;
44     QLabel *label_4;
45     QLineEdit *roleList;
46     QPushButton *roleButton;
47     QPushButton *addRecord;
48     QLineEdit *newRoleName;
49     QWidget *tab_2;
50     QHBoxLayout *horizontalLayout;
51     QTableView *tableView;
52     QStatusBar *statusbar;
53
54 void setupUi(QMainWindow *MyWindow)
55 {
56     if (MyWindow->objectName().isEmpty())
57         MyWindow->setObjectName(QString::fromUtf8("MyWindow"));
58     MyWindow->resize(733, 655);
59     centralwidget = new QWidget(MyWindow);
60     centralwidget->setObjectName(QString::fromUtf8("centralwidget"));
61     QSizePolicy sizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
62     sizePolicy.setHorizontalStretch(0);
63     sizePolicy.setVerticalStretch(0);
64     sizePolicy.setHeightForWidth(centralwidget->sizePolicy().hasHeightForWidth());
65     centralwidget->setSizePolicy(sizePolicy);
66     tabWidget = new QTabWidget(centralwidget);
67     tabWidget->setObjectName(QString::fromUtf8("tabWidget"));
68     tabWidget->setGeometry(QRect(0, 0, 731, 641));
69     tab = new QWidget();
```

The generated MyWindow class has an instance data member “widget” of type Ui_MyWindow. Since all the data members in class Ui_MyWindow are public, code that will be written for class MyWindow will be able to directly manipulate the widgets.

So, we will be able to get a MyWindow object to handle a click on the QPushButton imageSelector by putting up an appropriate file dialog that asks the user to select an image file. Of course, we have to write that code – QtBuilder cannot guess what it is that we want the imageSelector button to do.

The working version of the program will also require classes and functions from the earlier

exercise. The classes MyException, MyRecord, MyTableModel should all be recreated in this new project. MyException and MyRecord should be unchanged; MyTableModel will require a few additions that will allow for editing of data (editing is only partially implemented in this example task).



The main line code should create some records as done in the last task so that there are data to display:

```
#include <QApplication>
#include "MyWindow.h"
#include <iostream>

#include "MyRecord.h"
#include "MyTableModel.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

string getImage(string filename) {...21 lines }

static void createData() {...164 lines }

int main(int argc, char *argv[]) {
    // initialize resources, if needed
    // Q_INIT_RESOURCE(resfile);

    createData();
    QApplication app(argc, argv);
    MyWindow win(&g_theRecords);
    win.show();

    return app.exec();
}
```

Next, the MyWindow class must be more completely defined and some additions must be made to MyTableModel.

MyTableModel

The changes here are fairly limited – they involve the addition of functions that will allow data to be edited (actually, not much editing support is provided in this task – it was just the right time to add such functions even if their implementations are just stubs).

The class must now include versions of a few more public virtual functions defined in class AbstractTableModel:

```
class MyTableModel : public QAbstractTableModel {
    Q_OBJECT
public:
    MyTableModel(QObject *parent);

    void addTheData(vector<RecordPtr> *data) {
        this->recordsCollection = data;
    }
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    int columnCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole);
    bool insertRows(int position, int rows, const QModelIndex &index = QModelIndex());
    bool removeRows(int position, int rows, const QModelIndex &index = QModelIndex());
    void addRecord(RecordPtr newone);
}
```

(Method addRecord() is application specific; for this simple example, it's a more limited but more convenient way of adding records than use of insertRows().)

The flags() method will return an indicator that elements in the table can be selected but cannot actually be edited in situ (you could be more ambitious and allow some editable columns). The other overridden methods are just stubs. The addRecord() method puts a pointer to new record in the vector<RecordPtr> used to store data.

```
bool MyTableModel::insertRows(int /* position */, int /* rows */, const QModelIndex & /* index */) {
    // Q_UNUSED(index);
    // beginInsertRows(QModelIndex(), position, position+rows-1);
    //
    // // Add a row
    // endInsertRows();
    // return true;
    //cout << "In insertRows - which isn't implemented" << endl;
    return true;
}

bool MyTableModel::removeRows(int /*position */, int /* rows */, const QModelIndex & /* index */) {
    //...7 lines
    return true;
}

bool MyTableModel::setData(const QModelIndex & /* index */, const QVariant & /* value */, int /* role */) {
    //...9 lines
    return false;
}
```

Three of the overridden methods from QAbstractTableModel are just stubs – maybe later one might add more complete editing functionality:

The flags() method for this application simply returns an value that indicates that elements are selectable – we want to be able to click on a view of the table and determine which row was selected.

```

Qt::ItemFlags MyTableModel::flags(const QModelIndex &index) const {
    if (!index.isValid())
        return Qt::ItemIsEnabled;

    //return QAbstractTableModel::flags(index) | Qt::ItemIsEditable;

    return QAbstractTableModel::flags(index) | Qt::ItemIsSelectable;
}

```

The application defined addRecord() method is:

```

void MyTableModel::addRecord(RecordPtr newone) {

    //...

    int position = recordsCollection->size();
    beginInsertRows(QModelIndex(), position, position);
    recordsCollection->push_back(newone);
    endInsertRows();
    emit(dataChanged(QModelIndex(), QModelIndex()));
}

```

The beginInsertRows() and endInsertRows() methods (from the base class) are used by Qt to coordinate updates; the *emit* operation (part of Qt's event-handling system) sends a signal to the table view object that is displaying the data in this model. On receipt of the signal, the table view object will redraw its display.

MyWindow

This is where the work of the application gets done. A few more methods (most are “slots” - i.e. event handling functions) will have to be added to those generated automatically:

<pre> 1 // file: MyWindow.h 2 * Author: nabaq 3 * 4 * Created on 5 December 2012, 2:48 PM 5 */ 6 7 #ifndef _MYWINDOW_H 8 #define _MYWINDOW_H 9 10 #include "ui_MyWindow.h" 11 12 class MyWindow : public QMainWindow { 13 Q_OBJECT 14 public: 15 MyWindow(); 16 virtual ~MyWindow(); 17 private: 18 Ui::MyWindow widget; 19 }; </pre>		<pre> using namespace std; class MyRecord; typedef MyRecord* RecordPtr; class MyTableModel; class MyWindow : public QMainWindow { Q_OBJECT public: MyWindow(vector<RecordPtr> *theData); virtual ~MyWindow(); public slots: void chooseFile(); void addRole(); void addRecord(); void itemSelection(const QModelIndex & index); private: string getImage(QString filename); bool checkForId(string anid); Ui::MyWindow widget; vector<RecordPtr> *data; MyTableModel *tablemodel; }; </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The constructor for the class will take an argument that is a pointer to the vector<RecordPtr> collection for the data records; this gets used when constructing the instance of MyTableModel. The code in the constructor will initialise its instance Ui::MyWindow and then complete the configuration of the interface.

There are a few minor adjustments, e.g. the “selection mode” for the table view should allow for row selection rather than selection of individual cells.

The main setting up is the establishment of the event-handling links. As always, we have to connect the element that emits an event to the code that handles that event (so it's really just the same as adding an *onEvent* to a HTML tag). There are three buttons in the first tab pane of the interface – the imageSelector, the roleButton, and the addRecordButton. Buttons emit clicked signals. Such signals are to be routed to “slot” functions – the imageSelector's click should result in a call to the chooseFile() slot defined in this class.

```
MyWindow::MyWindow(vector<RecordPtr> *theData) {
    data = theData;
    widget.setupUi(this);

    tablemodel = new MyTableModel(0);
    tablemodel->addTheData(theData);
    widget.tableView->setModel(tablemodel);
    widget.tableView->resizeRowsToContents();
    widget.tableView->setSelectionBehavior(QAbstractItemView::SelectRows);

    // We have to put in the event handling links
    connect(widget.imageSelector, SIGNAL(clicked()), this, SLOT(chooseFile()));
    connect(widget.roleButton, SIGNAL(clicked()), this, SLOT(addRole()));
    connect(widget.addRecord, SIGNAL(clicked()), this, SLOT(addRecord()));
    connect(
        widget.tableView,
        SIGNAL(clicked(const QModelIndex &)), this,
        SLOT(itemSelection(const QModelIndex &))
    );
}
```

Code like `connect(widget.imageSelector, ...)` looks like C++, but really it isn't. It's part of the Qt language extensions. It gets converted into genuine C++ by the pre-compiler. The SIGNAL and SLOT “macros” take the names of signals and slot methods (the argument list for a slot must match that defined for the signal).

The first connect statement is interpreted as follows:

- The emitter of the event will be the QPushButton object represented by the imageSelector data member in the widget (instance of Ui::MyWindow);
- The signature of the signal function is clicked();
- The object with the slot to handle the signature is *this* object (i.e. the MyWindow object);

- The slot method of that handler object is called chooseFile().

Of course, you must know the declarations of the signal functions; you find these in the documentation in QtAssistant.

The fourth of the connect actions sets up a handler for a mouse click in the table view.

The clicked() signal is actually defined in an ancestor class – QAbstractItemView:

Signals

```
void activated ( const QModelIndex & index )
void clicked ( const QModelIndex & index )
void doubleClicked ( const QModelIndex & index )
void entered ( const QModelIndex & index )
void pressed ( const QModelIndex & index )
void viewportEntered ()
```

The slot method that handles this “clicked” signal must have a signature that matches the clicked function – hence void MyWindow::itemSelection(const QModelIndex & index).

Don't include variable names in the argument lists when specifying the connect statement. The following will not compile:

```
connect (
    widget.tableView,
    SIGNAL(clicked(const QModelIndex& index)), this,
    SLOT(itemSelection(const QModelIndex& index))
);
```

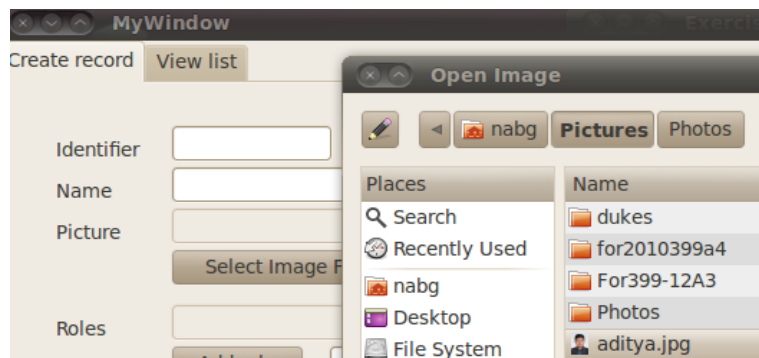
the statement must have the form:

```
connect (
    widget.tableView,
    SIGNAL(clicked(const QModelIndex& )), this,
    SLOT(itemSelection(const QModelIndex& ))
);
```

Selecting a file with an image

```
void MyWindow::chooseFile() {
    // Display a standard file dialog and let user select a file
    // Dialog to open with MyWindow as parent, the messages is Open Image
    // Start in current directory, restrict to standard image file types
    QString fileName = QFileDialog::getOpenFileName(this,
        "Open Image", ".", "Image Files (*.png *.jpg *.bmp)");
    if (!fileName.isEmpty()) {
        // A file was picked, set the pictureField to match
        widget.pictureField->setText(fileName);
    }
}
```

The program should open a standard file dialog that allows a user to select jpg, png, or bmp files. If a file is selected, the dialog will return with a non-empty filename. This name is to be copied into the disabled line edit element “picture field”:



Selecting a row in the table view

This operation doesn't really do anything in this example. It shows how you would identify a selection in the table view; in a more complex example, there could be a third tab in the tab pane interface that gets used to display all details of a selected record. But for now, it simply prints a log message identifying the row selected:

```
void MyWindow::itemSelection(const QModelIndex & index) {
    cout << "Selected row " << index.row() << "\t";
    RecordPtr p = data->at(index.row());
    cout << p->getName() << endl;
}
```

Adding a role

When the “add role” button is clicked, the program should check for a string in the new role line edit widget. If there is a string, it is to be appended to the string currently in the role list (disabled) line edit:

```

void MyWindow::addRole() {

    QString aRole = widget.newRoleName->text().trimmed();
    if (aRole.isEmpty()) return;

    if (widget.roleList->text().isEmpty())
        widget.roleList->setText(aRole);
    else {
        QString roles = widget.roleList->text();
        roles.append(",");
        roles.append(aRole);
        widget.roleList->setText(roles);
    }
    widget.newRoleName->clear();
}

```

Note the use of QString variables. Qt defines its own string class. (It seems that almost every C++ library defines its own string class.) You will usually end up with code converting instances of class String from library X into class String from library Y (or to STL string, and sometimes even to const char*!). Qt's string class at least has a trimmed() method built in!

(There are problems with using std::string and QString and char* etc; firstly, you keep having to convert formats; secondly, you increase the chance of memory leaks when you keep allocating and reassigning objects.)

Adding a record

This entails some real work via addRecord().

The various input fields must be checked – has the user provided all of: identifier, name, roles, and image file? Can the image be loaded (using the function in file main.cpp as an extern function)? Is the identifier already in use? If there are any problems with the data, the program should display an error dialog and then return without adding the record.

If the data are good, a new instance of MyRecord should be created and added to the collection, and then the input fields should be tidied up so that a subsequent record can be added.


```

void MyWindow::addRecord() {
    bool ok = true;
    string problems;

    QString idstr = widget.idField->text().trimmed();
    QString namestr = widget.nameField->text().trimmed();
    QString filestr = widget.pictureField->text().trimmed();
    QString rolestr = widget.roleList->text();
    if (idstr.isEmpty()) {
        ok = false;
        problems.append(" You didn't supply a record id.");
    }
    if (namestr.isEmpty()) {
        ok = false;
        problems.append(" You didn't supply a name.");
    }
    if (filestr.isEmpty()) {
        ok = false;
        problems.append(" You didn't supply a picture file.");
    }

    if (rolestr.isEmpty()) {
        ok = false;
        problems.append(" You didn't define any roles.");
    }
    string filename = filestr.toStdString();
    string stlchars = getImage(filename);
    QString imgchars = stlchars.c_str();
    if (imgchars == NULL) {
        ok = false;
        problems.append(" Unable to load image.");
    }
    string recordid = idstr.toStdString();
    bool idPresent = checkForId(recordid);
    if (idPresent) {
        ok = false;
        problems.append("You already have a record with that id.");
    }
}

```

```

if (!ok) {
    QMessageBox msgBox;
    msgBox.setWindowTitle("Error!");
    msgBox.setText("Invalid record");
    msgBox.setDetailedText(problems.c_str());
    msgBox.exec();
    return;
}

```

Report errors

```

RecordPtr newrec = new MyRecord(recordid);
string nstr = namestr.toStdString();
newrec->setName(nstr);
string imgstr = imgchars.toStdString();
newrec->setImage(imgstr);
QStringList list = rolestr.split(",", QString::SkipEmptyParts);
QStringList::const_iterator constIterator;
for (constIterator = list.constBegin(); constIterator != list.constEnd();
    ++constIterator) {
    string str = (*constIterator).toStdString();
    newrec->addRole(str);
}

tablemodel->addRecord(newrec);
widget.tableView->resizeRowsToContents();

```

**Create new MyRecord
instance and fill in its fields**

**Add new record to
collection**

(The code filling in the fields of the MyRecord object is making further use of Qt's string libraries. A QString has regex methods – such as split(); there are iterators that work through collections of strings, QStringList, etc. Qt has many

other collection classes, e.g. QMap, QList. Generally, I find the classes in the Qt libraries more convenient than those in STL. They have greater functionality, clearer semantics, and are far less syntactically fussy than the STL templates. Of course, given your experience with STL in CSCI204 you may well prefer the STL classes.)

```
tablemodel->addRecord(newrec);
widget.tableView->resizeRowsToContents();

QMessageBox report;
report.setWindowTitle("Success!");
report.setText("Record added");
report.exec();

widget.idField->clear();
widget.nameField->clear();
widget.pictureField->clear();
widget.newRoleName->clear();
widget.roleList->clear();
}
```

Finally, add the new MyRecord to the data collection and clear up the input fields.

The helper function checkForId() is used when checking whether the identifier for a new record is unique:

```
bool MyWindow::checkForId(string anid) {

    vector<RecordPtr>::iterator it;
    for (it = data->begin(); it != data->end(); it++) {
        RecordPtr p = *it;
        if (p->getID() == anid) return true;
    }

    return false;
}
```

Task 2 – completion (2 marks)

Demonstrate your working QtAddr2 project.

(Note: when building applications that involve a “QT form” class you may find that your code in the NetBeans editor is marked by numerous error flags but you cannot see anything wrong. The problem is related to the way the “qmake” script works. It has to generate C++ files with the definitions of code for displaying the widgets along with associated header files. These files get deleted and recreated each build. If one of the generated headers files is missing when the NetBeans' editor loads one of your class files the editor will end up flagging spurious errors. Usually these disappear when you build the application; occasionally, an odd error flag or two may get left. Don't worry. If the application builds it should be OK.)

Task 3: Saving the records

If we are going to the trouble of constructing a collection of address records, we would probably want to save them to a file on disk when the program ended, and reload them when the program was again run.

If you look at Qt's own address book example, you will see saving and loading files as one-liners. The Qt example has an address book that is a simple `QMap<QString, QString>`; both Qt classes `QMap` and `QString` have overloaded operator `<<` output functions (and input functions) making it easy to serialize the data and send them to a text file.

The `MyRecord` objects are a bit more complex. There are variable numbers of roles, and there are those map data elements (OK, these haven't used those since exercise 1 but they are meant to be there and are meant to be used). The coding will inevitably be a little more complex.

Some approaches are discussed below – *the one I favour (and which you should implement) is last.*

How to save? Define `ostream& operator<<(ostream&, const MyRecord&)?`

How would you save the address book collection?

One approach would be to use text files. You would need output code that would start with an integer value for the number of `MyRecord` entries and then would have a series of `MyRecords` all serialized.

It would be a really bad idea to have a function that took a `MyRecord` argument and interrogated it for each data element and then wrote this out as text. A better approach would rely on definition of friend functions for the `MyRecord` class that used a private print function. The `ostream& operator<<` and `istream& operator>>` functions would get added to the `MyRecord` class declaration:

```
// private (and no implementations will be defined). I choose to
// disallow such operations.
MyRecord(const MyRecord& orig);
MyRecord& operator=(const MyRecord&);
// Further there is no virtual destructor, I do not intend MyRecord
// to be the base class in some hierarchy.

friend ostream& operator<<(ostream& out, const MyRecord&);
void printOn(ostream&) const;

};

inline ostream& operator<<(ostream& out, const MyRecord& rec) {
    rec.printOn(out);
    return out;
}
```

(The definition of the corresponding input function is left as an exercise!)

The `printOn()` function would output a text representation of a `MyRecord`:

```

void MyRecord::printOn(ostream& out) const {

    out << id << endl;
    out << name << endl;
    out << email << endl;
    out << image << endl;
    out << info << endl;
    out << roles.size() << endl;
    vector<string>::const_iterator it1;
    for (it1 = roles.begin(); it1 != roles.end(); it1++) {
        out << (*it1) << endl;
    }
    out << phones.size() << endl;
    map<string, string>::const_iterator it2;
    for (it2 = phones.begin(); it2 != phones.end(); it2++) {
        out << (*it2).first << " " << (*it2).second << endl;
    }
    out << addresses.size() << endl;
    map<string, string>::const_iterator it3;
    for (it3 = addresses.begin(); it3 != addresses.end(); it3++) {
        out << (*it3).first << " " << (*it3).second << endl;
    }
    out << other.size() << endl;
    map<string, string>::const_iterator it4;
    for (it4 = other.begin(); it4 != other.end(); it4++) {
        out << (*it4).first << " " << (*it4).second << endl;
    }

}
}

```

An address book (i.e. vector<RecordPtr>) could be saved as follows:

```

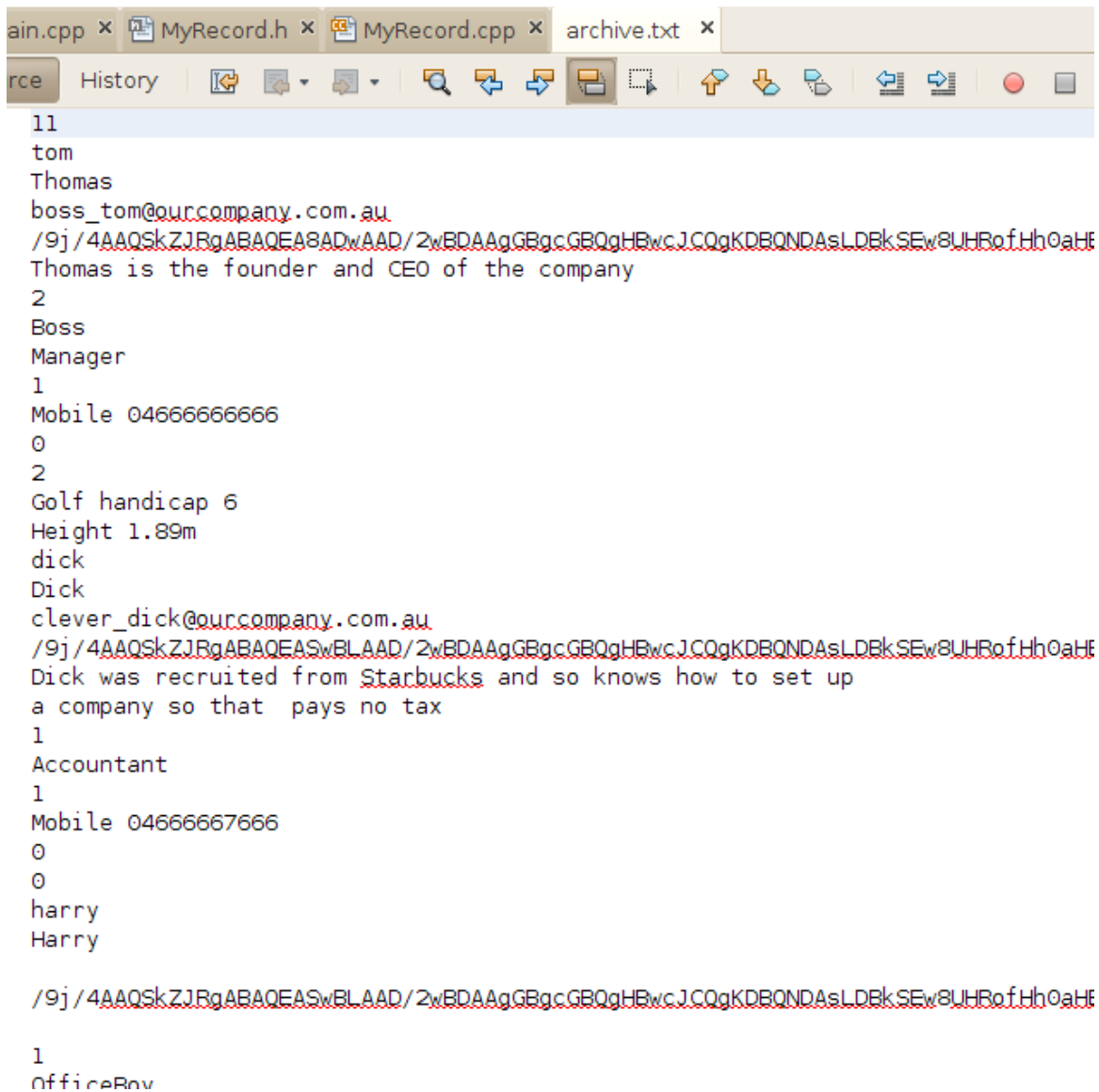
int main(int argc, char *argv[]) {
    // Use the createData from earlier examples to populate the "address book"
    createData();

    ofstream ofile("archive.txt");
    ofile << g_theRecords.size() << endl;
    vector<RecordPtr>::const_iterator it;
    for(it=g_theRecords.begin(); it != g_theRecords.end(); it++) {
        RecordPtr nxt = *it;
        ofile << (*nxt);
    }
    ofile.close();

    return EXIT_SUCCESS;
}

```

This would create a text file such as the following:



```
11
tom
Thomas
boss_tom@ourcompany.com.au
/9j/4AAQSkZJRgABAQEASwBLAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aH
Thomas is the founder and CEO of the company
2
Boss
Manager
1
Mobile 0466666666
0
2
Golf handicap 6
Height 1.89m
dick
Dick
clever_dick@ourcompany.com.au
/9j/4AAQSkZJRgABAQEASwBLAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aH
Dick was recruited from Starbucks and so knows how to set up
a company so that pays no tax
1
Accountant
1
Mobile 0466666766
0
0
harry
Harry

/9j/4AAQSkZJRgABAQEASwBLAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aH

1
OfficeRov
```

The file could be read back by a program given appropriate definitions of `istream& operator>>` and a `readFrom(istream&)` function.

But there would likely be all sorts of “Gotchas” - bugs holding up development.

Strings can have new line characters ---

```

file = "./images/dick.jpg";
imagestr = getImage(file);
nxt = new MyRecord(id);
nxt->setName(name);
nxt->setImage(imagestr);
aRole="Accountant";
nxt->addRole(aRole);
string info = "Dick was recruited from Starbucks and so knows how to set up\ na comp
nxt->setInfo(info);
string email = "clever_dick@ourcompany.com.au";
nxt->setEmail(email);
string phones = "Phones";
string mbl = "Mobile";
string phnnum = "04666667666";

```

The output file will have two lines “Dick ...up” and “a company ... tax”. This would break a naively coded readFrom function that tried something like:

```

void MyRecord::readFrom(istream& inputstream) {
...
string info;
inputstream >> info;
this->setInfo(info);
int rolecount;
inputstream >> rolecount;
...

```

Other problems would occur with things like space in “keys” for the key value collections that form a part of a MyRecord:

```

string email = "boss_tom@ourcompany.com.au";
nxt->setEmail(email);
string phones = "Phones";
string mbl = "Mobile";
string phnnum = "04666666666";
nxt->addKeyValue(phones,mbl,phnnum);

string others = "Other";
string key = "Height";
string value = "1.89m";
nxt->addKeyValue(others, key, value);

key = "Golf handicap";
value = "6";
nxt->addKeyValue(others, key, value);
g_theRecords.push_back(nxt);

```

These data result in lines in the save file like:

```

Boss
Manager
1
Mobile 04666666666
0
2
Golf handicap 6
Height 1.89m

```

which will break a naively coded readFrom function that had something like

```
string key;
inputstream >> key;
string value;
inputstream >> value;
this->addKeyValue(others, key, value);
```

(Tom would end up with an “Other” attribute “Golf” with value “handicap” and there would be some remaining input on the line to disrupt the next read action.)

A correct implementation would be a lot more complex; strings would have to be delimited in some application defined way so that a readFrom() function could consume string data correctly.

Why not change to a scheme that inherently provides delimiters for data elements?

XML!

Do it yourself with XML!

XML data may be verbose but they have the great advantage of being self describing, with properly delimited elements.

It's quite easy to generate an XML file as output – you just remember to add output statements to put the begin tag and end tag XML tokens around each data element.

Output of the address book would go something like:

```
out << "<address_book>" << endl;
vector<RecordPtr>::const_iterator it;
for(it=g_theRecords.begin(); it!= g_theRecords.end(); it++) {
    RecordPtr p = (*it);
    p->writeAsXML(out);
}
out << "</address_book>";
```

The output function writeAsXML() defined for the MyRecord class would similarly tag each data element.

This would result in a file something like:

```
<address_book>
  <myrecord>
    <id>tom</id>
    <name>Thomas</name>
    <email>boss_tom@outcompany.com.au</email>
    <image>....</image>
    <info>Thomas ... </info>
    <roles>
      <role>Boss</role>
      <role>Manager</role>
    </roles>
    <phones>
      <phonerec>
        <type>Mobile</type>
        <number>04666666666</number>
      </phonerec>
    </phones>
    <addresses></addresses>
    <others>
      <otherrec>
```



```

        <key>Golf handicap</key>
        <value>6</value>
    </otherrec>
    <otherrec>
        <key>Height</key>
        <value>1.89m</value>
    </otherrec>
</others>
</myrecord>
<myrecord>
    <id>dick</id>
...

```

The self describing nature of XML makes the file structure much clearer. An entry `<addresses></addresses>` is easier to interpret than a blank line (“no addresses given”).

But how would you read the data file back and recreate a collection of MyRecord objects?

You certainly would not want to write your own XML parser. There are implementations of the W3C DOM parser for all languages, so you could get a C++ DOM parser class from some library and use it to create a “Document Object Model” tree-structure from the data in your save file.

You would then have to write code to traverse the DOM and extract the data needed to recreate a collection of MyRecord structures. Most of you will have written some limited code to do such tasks in CSCI110 where you wrote Javascript code to do things like `document.getElementById(...)`.

But it's hard work.

The Larry Wall approach – *laziness and impatience rule ok*

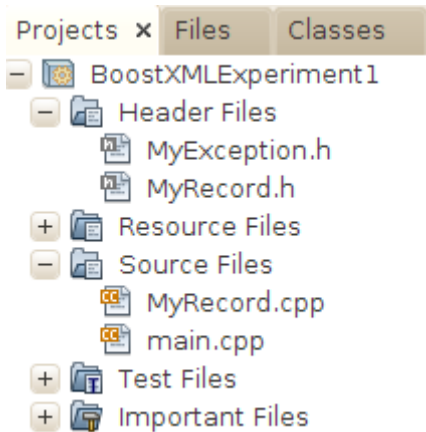
It is only at universities that people write programs from scratch. In the real world, developers create useful working programs by assembling a large number of pre-built components, threading these together, and adding the tiny amount of code that really is unique to their application.

You aren't the first to need to transfer data structures to and from files. You aren't the first to have complex nested data structures with nested collections that will best be saved in XML format. Others have dealt with these problems before and created utilities to help you.

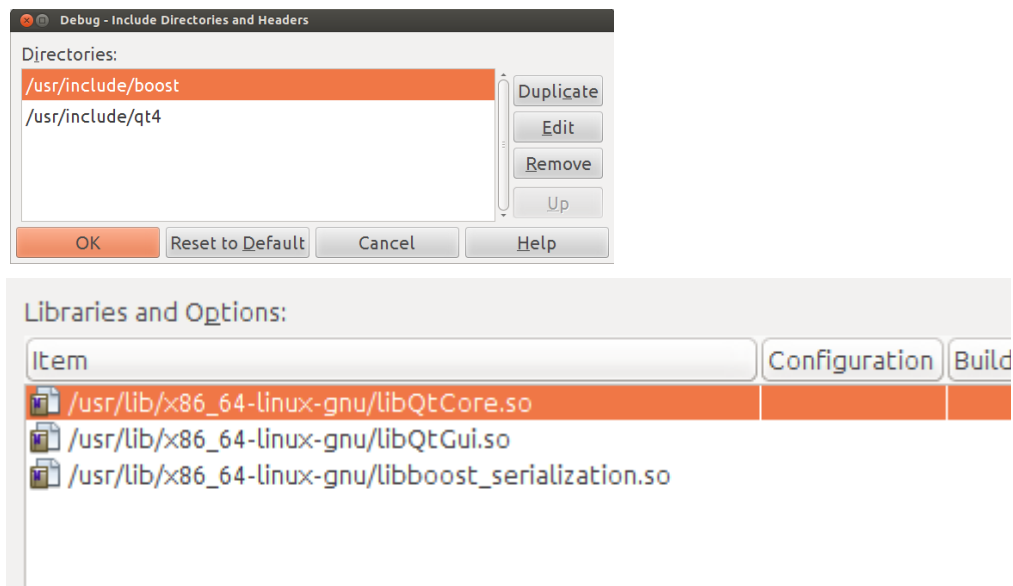
If you need to save and restore data using XML files, then one solution is to use the “serialization” libraries in the “boost” suite. One of the advantages of the boost libraries is that they already incorporate code to handle the STL data collections like vector and map.

The boost serialization libraries make it very easy to save and restore arbitrarily complex collections of records. (A cyclic graph data structure might present some challenges – could be difficult to represent as an XML tree-structure; but there are ways of dealing with such things.)

For this task, you will create two new NetBeans projects, both being made up almost entirely from code written for earlier tasks. The first project “BoostXMLExperiment1” will create an XML file from the standard data that you have been using to initially populate the data collection for your exercises on creating Qt displays. The second project “BoostXMLExperiment2” is a minor reworking of the program that you completed in task 2; instead of having fixed code to initialise the data collection it will start by reading an archive file, and it will write a new version of the archive before terminating.



The project requires “includes” for the C++ compiler, and libraries for the C++ linker:



The main() function calls your createData() function (extend your code so that some of the MyRecord instances to have “info”, “phones”, “addresses” and “other” data added to them). It then opens an output file and uses the boost serialization libraries to save the collection:

```

#include "MyRecord.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

string getImage(string filename) { ...21 lines }

static void createData() { ...189 lines }

int main(int argc, char *argv[]) {
    createData();

    ofstream ofile("archive.xml");
    boost::archive::xml_oarchive oa(ofile);

    oa & BOOST_SERIALIZATION_NVP(g_theRecords);

    return EXIT_SUCCESS;
}

```

The line `oa & BOOST_SERIALIZATION_NVP(g_theRecords)` does look weird – but it's simply a matter of the guys who wrote the library defining an overloaded operator `&` function for the `xml_oarchive` class. (Namespaces get a bit complex with boost, so it is easiest if you use fully qualified class names – hence `boost::archive::...`)

That part worked because the boost library has code to handle the output of a STL vector object. But what about the `MyRecords`?

It's easy.

You add a public function to your class (after including a lot more header files):

```

#include "MyException.h"
#include <map>
#include <string>
#include <vector>

#include <boost/archive/xml_iarchive.hpp>
#include <boost/archive/xml_oarchive.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/map.hpp>

using namespace std;

class MyRecord {
public:
    explicit MyRecord(string& id); // I don't want implici
    // Mutator functions - setting of other fields
    // In this version, hardly any of the implementations
    // later validation functions will be added
    void setName(string& aname) throw (MyException);

```

Inline “serialize” function that is both a “save” function and a “reload” function!

```
bool hasRole(string& queryrole) const;
```

```

template<class Archive>
void serialize(Archive& archive, const unsigned int /* version */) {
    archive & BOOST_SERIALIZATION_NVP(id);
    archive & BOOST_SERIALIZATION_NVP(name);
    archive & BOOST_SERIALIZATION_NVP(email);
    archive & BOOST_SERIALIZATION_NVP(info);
    archive & BOOST_SERIALIZATION_NVP(roles);
    archive & BOOST_SERIALIZATION_NVP(phones);
    archive & BOOST_SERIALIZATION_NVP(addresses);
    archive & BOOST_SERIALIZATION_NVP(other);
    archive & BOOST_SERIALIZATION_NVP(image);
}

private:
    string id; // Also known as "nickname", or even "primary key"
    string name; // full name

```

(The BOOST_SERIALIZATION_NVP “macro” standards for “save/restore name-value pair.”)

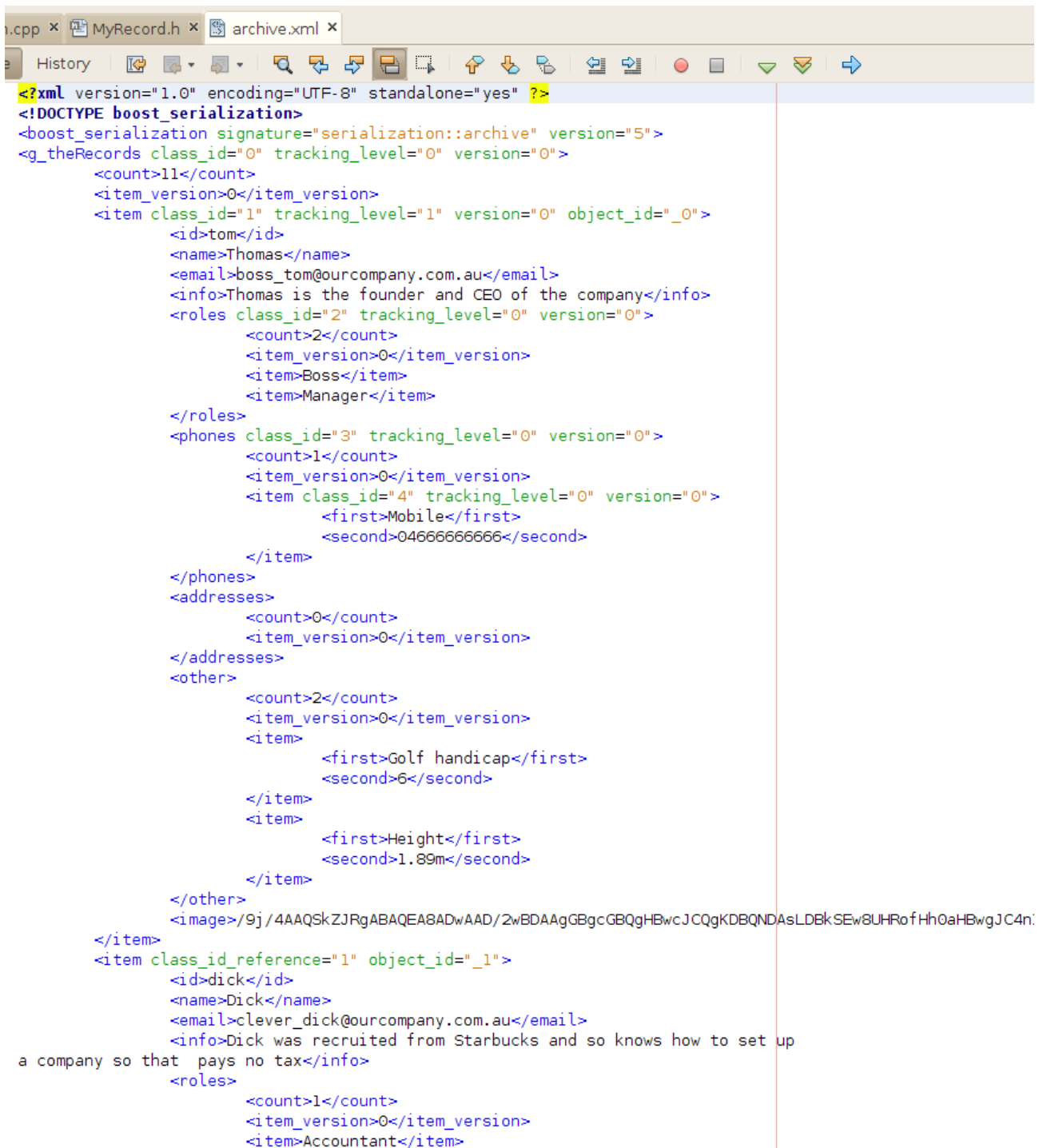
That's all there is to it!

Your serialize function just has transfer statements for each data member that is part of the persistent state of your object. (It would be quite common for a class that you define to have other data

members such as pointers to instances of collaborating classes; you just don't include such “transient” data members in your list of serialization actions.)

The one function handles both input and output. The boost library guys have overloaded the operator & for their xml_oarchive class to mean output, while for their xml_iarchive it means input.

Your program should run and produce an output file like:



```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="5">
<g_theRecords class_id="0" tracking_level="0" version="0">
  <count>11</count>
  <item_version>0</item_version>
  <item class_id="1" tracking_level="1" version="0" object_id="_0">
    <id>tom</id>
    <name>Thomas</name>
    <email>boss_tom@ourcompany.com.au</email>
    <info>Thomas is the founder and CEO of the company</info>
    <roles class_id="2" tracking_level="0" version="0">
      <count>2</count>
      <item_version>0</item_version>
      <item>Boss</item>
      <item>Manager</item>
    </roles>
    <phones class_id="3" tracking_level="0" version="0">
      <count>1</count>
      <item_version>0</item_version>
      <item class_id="4" tracking_level="0" version="0">
        <first>Mobile</first>
        <second>0466666666</second>
      </item>
    </phones>
    <addresses>
      <count>0</count>
      <item_version>0</item_version>
    </addresses>
    <other>
      <count>2</count>
      <item_version>0</item_version>
      <item>
        <first>Golf handicap</first>
        <second>6</second>
      </item>
      <item>
        <first>Height</first>
        <second>1.89m</second>
      </item>
    </other>
    <image>/9j/4AAQSkZJRgABAQEABADwAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEwBUHRofHh0aHBwgJC4n;
  </item>
  <item class_id_reference="1" object_id="_1">
    <id>dick</id>
    <name>Dick</name>
    <email>clever_dick@ourcompany.com.au</email>
    <info>Dick was recruited from Starbucks and so knows how to set up
a company so that pays no tax</info>
    <roles>
      <count>1</count>
      <item_version>0</item_version>
      <item>Accountant</item>
    </roles>
  </item>
</g_theRecords>
</boost_serialization>
```

Finally (!), create another version of your program for editing and viewing the collection. It is to read in an xml file when it starts and write out an updated version when it finishes.

Recreate your QtAddr2 project (Reminder, you cannot simply duplicate the directory at command level or within NetBeans – the configuration files in the nbproject sub-directory will not be updated properly and you will end up trying to build with the versions of header files from the old project.)

Edit the include files for the compiler, and the library files for the linker in the projects properties. You do not need to specify any qt files (this is a Qt project so they are sorted out automatically). You have to add the references to the boost include directory and the boost serialization library.

Add #include <boost...> statements and the declaration of the serialize() method in your MyRecord.h file. **You will also need to add a no-argument constructor – public: MyRecord() { }.**

Change the main.cpp:

```
typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

int main(int argc, char *argv[]) {
    // initialize resources, if needed
    // Q_INIT_RESOURCE(resfile);

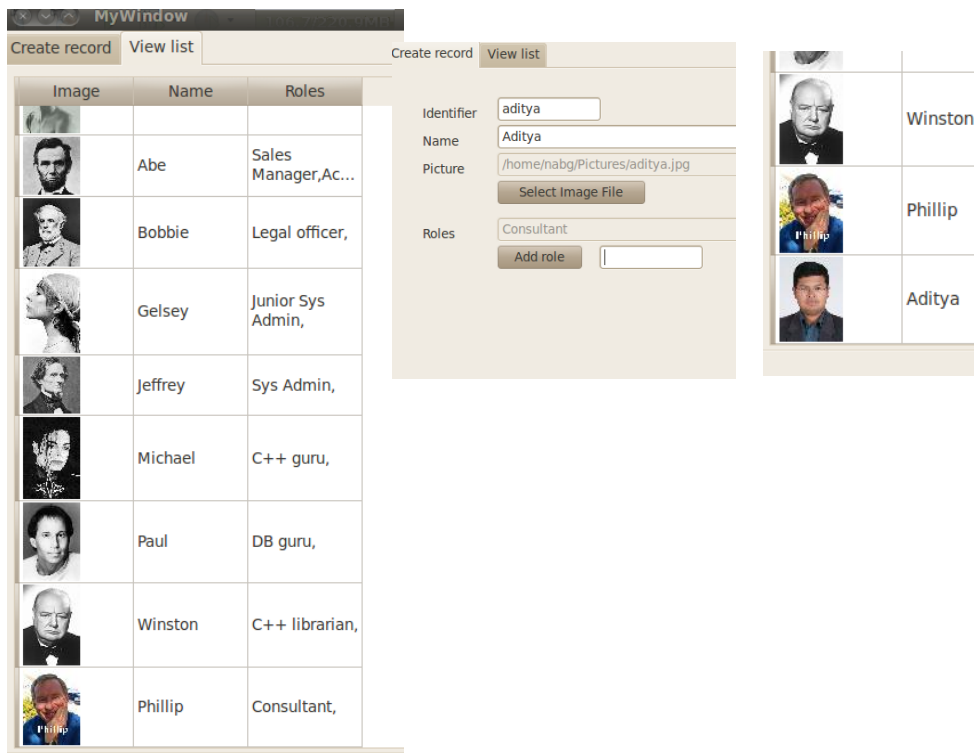
    ifstream inputfile("archive.xml");
    boost::archive::xml_iarchive ia(inputfile);
    ia & BOOST_SERIALIZATION_NVP(g_theRecords);
    inputfile.close();
    QApplication app(argc, argv);
    MyWindow win(&g_theRecords);
    win.show();
    int result = app.exec();
    ofstream outputfile("archive.xml");
    boost::archive::xml_oarchive oa(outputfile);
    oa & BOOST_SERIALIZATION_NVP(g_theRecords);
    outputfile.close();
    return result;
}
```

It should all work!

You never imagined things could be this simple.

You can learn more about the boost serialization libraries at:

<http://www.ibm.com/developerworks/aix/library/au-boostserialization/index.html>
http://www.boost.org/doc/libs/1_52_0/libs/serialization/doc/tutorial.html



Task 3 – completion (1 mark)

Demonstrate that your application can create persistent “address books”.

(Don't forget to “clean” all the projects to recover disk space when you have completed all the tasks in this exercise.)