А.О. Семкин, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание диалоговых окон программы»

2015

Министерство образования и науки Российской федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

(ТУСУР)

Кафедра сверхвысокочастотной и квантовой радиотехники
(СВЧиКР)

А.О. Семкин, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание диалоговых окон программы»

2015

УДК 004.4+519.6

Рецензент:
профессор каф..СВЧиКР, А.Е. Мандель

А.О. Семкин, С.Н. Шарангович

Информатика: Руководство к лабораторной работе «Библиотека Qt. Создание диалоговых окон программы» / А.О. Семкин, С.Н. Шарангович. – Томск: ТУСУР, 2015. – 30 с.

В данном руководстве изложены принципы работы в средах разработки *Qt Creator* и *Qt Designer*. Приведено описание встроенного в *Qt* класса *QDialog*. Представлены методические материалы компьютерной лабораторной работы, посвященной созданию диалоговых окон программ. Дополнительно для повторения описан механизм сигналов и слотов.

Предназначено для студентов очной, заочной, и вечерней форм обучения по направлению подготовки бакалавриата «Инфокоммуникационные технологии и системы связи». – 11.03.02.

УДК 004.4+519.6

Оглавление

1. Цель работы

Целью данной работы является изучение принципов работы и возможностей сред *Qt Creator* и *Qt Designer* для создания диалоговых окон программ с графическим пользовательским интерфейсом.

2. Введение

Qt представляет собой комплексную рабочую среду, предназначенную для разработки на C++ межплатформенных приложений с графическим пользовательским интерфейсом по принципу «написал программу – компилируй ее в любом месте». Qt позволяет программистам использовать дерево классов с одним источником в приложениях, которые будут работать в системах Windows, Mac OS X, Linux. Solaris, HP-UX и во многих других.

Графический пользовательский интерфейс (GUI — Graphical User Interface) это средства позволяющие пользователям взаимодействовать с аппаратными составляющими компьютера комфортным и удобным для себя образом.

В рамках данной лабораторной работы будут рассмотрены принципы создания диалоговых окон программ с графическим пользовательским интерфейсом с использованием возможностей *Qt Creator* и *Qt Designer*.

3. Подклассы QDialog

Первым примером будет диалоговое окно Find (найти) для поиска заданной пользователем последовательности символов, и оно будет полностью написано на C++. Мы реализуем это диалоговое окно в виде его собственного класса. Причем мы сделаем его независимым и самодостаточным компонентом, со своими сигналами и слотами.

Рис. 1 – Диалоговое окно поиска

Исходный код программы содержится в двух файлах: finddialog.h and finddialog.cpp. Сначала приведем файл finddialog.h.

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H
3 #include <QDialog>
4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;
```

Строки 1 и 2 (а также строка 27) предотвращают многократное включение в программу этого заголовочного файла.

В строке 3 в программу включается определение QDialog – базового класса для диалоговых окон в Qt. Класс QDialog наследует свойства класса QWidget.

В строках с 4 но 7 даются предварительные объявления классов Qt, использующихся для реализации диалогового окна. Предварительное объявление (forward declaration) указывает компилятору C++ только на существование класса, не давая подробного определения этого класса (обычно определение класса содержится в его собственном заголовочном файле). Чуть позже мы поговорим об этом более подробно.

Затем мы определяем FindDialog как подкласс QDialog:

```
 8 class FindDialog : public QDialog
 9 {
10     Q_OBJECT
11 public:
12     FindDialog(QWidget *parent = 0);
```

Макрос Q_OBJECT необходимо задавать в начале определения любого класса, содержащего сигналы или слоты.

Конструктор FindDialog является типичным для классов виджетов в QL В параметре parent (родитель) указывается родительский виджет. По умолчанию задается нулевой указатель, указывающий на то, что у данного диалога нет родительского виджета.

```
13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
15     void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

В секции signals объявляется два сигнала, которые генерируются диалоговым окном при нажатии пользователем кнопки Find (найти). Если установлен флажок поиска в обратном направлении (Search backward), генерируется сигнал findPrevious(); в противном случае генерируется сигнал findNext().

Ключевое слово signals на самом деле является макросом. Препроцессор C++ преобразует его в стандартные инструкции языка C++ и затем передает их компилятору. Qt::CaseSensitivity является перечислением и может принимать значение Qt::CaseSensitive или Qt::CaseInsensitive.

```
16 private slots:
17     void findClicked();
18     void enableFindButton(const QString &text);
19 private:
20     QLabel *label;
21     QLineEdit *lineEdit;
22     QCheckBox *caseCheckBox;
23     QCheckBox *backwardCheckBox;
24     QPushButton *findButton;
25     QPushButton *closeButton;
26 };
```

```
27 #endif
```

В закрытой (private) секции класса мы объявляем два слота. Для реализации слотов нам потребуется большинство дочерних виджетов диалогового окна, поэтому мы резервируем для них соответствующие переменные-указатели. Ключевое слово slots так же как и signals, является макросом, который преобразуется в последовательность инструкций, понятных компилятору C++.

Для закрытых переменных мы использовали предварительные объявления их классов. Это допустимо, потому что все они являются указателями, и мы не используем их в заголовочном файле - поэтому компилятору не требуется иметь полные определения классов. Мы могли бы воспользоваться соответствующими заголовочными файлами (<QCheckBox>, <QLabel> и так далее), но при использовании предварительных объявлений компилятор работает немного быстрее.

Теперь рассмотрим файл finddialog.cpp, в котором находится реализация класса FindDialog.

```
1  #include <QtGui>
2  #include "finddialog.h"
```

Во-первых, мы включаем <QtGui> – заголовочный файл, который содержит определения классов графического интерфейса Qt. Qt состоит из нескольких модулей, каждый из которых находится в своей собственной библиотеке. Наиболее важными модулями являются QtCore, QtGui, QtNetwork, QtOpenGl, QtSql, QtSvg и QtXmL Заголовочный файл <QtGui> содержит определение всех классов, входящих в модули *QtCore* и QtGui. Включив этот заголовочный файл, мы можем не беспокоиться о включении каждого отдельного класса.

В finddialog.h вместо включения <QDialog> и использования предварительных объявлений для классов QCheckBox, QLabel, QLineEdit и QPushButton мы могли бы просто включить <QtGui>. Однако включение такого большого заголовочного файла, взятого из другого заголовочного файла, обычно свидетельствует о плохом стиле кодирования, особенно при разработке больших приложений.

```
 3 FindDialog::FindDialog(QWidget *parent)
 4     : QDialog(parent)
 5 {
 6     label = new QLabel(tr("Find &what:"));
 7     lineEdit = new QLineEdit;
 8     label->setBuddy(lineEdit);
 9     caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));
11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);
14     closeButton = new QPushButton(tr("Close"));
```

В строке 4 конструктору базового класса перелается указатель на родительский виджет (параметр parent). Затем мы создаем дочерние внджеты. Функция tr() переводит строковые литералы на другие языки. Она объявляется в классе QObject и в каждом подклассе, содержащем макрос Q_OBJECT. Любое строковое значение, которое пользователь будет видеть на экране, полезно преобразовывать функцией tr(), даже если вы не планируете в настоящий момент переводить ваше приложение на какой-нибудь другой язык.

Мы используем знак амперсанда ('&') для задания клавиш быстрого доступа. Например, в строке 11 создается кнопка Find, которая может быть активирована нажатием пользователем сочетания клавиш Alt+F на платформах, поддерживающих клавиши быстрого доступа. Амперсанды могут также применяться для управления фокусом: в строке 6 мы создаем текстовую метку с клавишей быстрого доступа (Alt+W), а в строке 8 мы устанавливаем строку редактирования в качестве «партнера» этой текстовой метки. *Партнером (buddy)* называется виджет, на который передается фокус при нажатии клавиши быстрого доступа текстовой метки. Поэтому при нажатии пользователем сочетания клавиш Alt+W (клавиша быстрого доступа текстовой метки) фокус переходит на строку редактирования (которая является партнером текстовой метки).

В строке 12 мы делаем кнопку Find используемой по умолчанию, вызывая функцию setDefault(true). Кнопка, для которой задан режим использования по умолчанию, будет срабатывать при нажатии пользователем клавиши Enter (ввод). В строке 13 мы устанавливаем кнопку Find в неактивный режим. В неактивном режиме виджет обычно имеет серый цвет и не реагирует на действия пользователя.

```
15      connect(lineEdit, SIGNAL(textChanged(const QString &)),
16              this, SLOT(enableFindButton(const QString &)));
17      connect(findButton, SIGNAL(clicked()),
18              this, SLOT(findClicked()));
19      connect(closeButton, SIGNAL(clicked()),
20              this, SLOT(close()));
```

Закрытый слот enableFindButton(const QString &) вызывается при всяком изменении значения в строке редактирования. Закрытый слот findClicked() вызывается при нажатии пользователем кнопки Find. Само диалоговое окно закрывается при нажатии пользователем кнопки Close (закрыть). Слот close() наследуется от класса QWidget, и по умолчанию он делает виджет невидимым (но не удаляет его). Программный код слотов enableFindButton() и findClicked() мы рассмотрим позднее.

Поскольку QObject является одним из прародителей FindDialog, мы можем не указывать префикс QObject:: перед вызовами connect().

```
21      QHBoxLayout *topLeftLayout = new QHBoxLayout;
22      topLeftLayout->addWidget(label);
23      topLeftLayout->addWidget(lineEdit);
24      QVBoxLayout *leftLayout = new QVBoxLayout;
25      leftLayout->addLayout(topLeftLayout);
```

```
26      leftLayout->addWidget(caseCheckBox);
27      leftLayout->addWidget(backwardCheckBox);
28      QVBoxLayout *rightLayout = new QVBoxLayout;
29      rightLayout->addWidget(findButton);
30      rightLayout->addWidget(closeButton);
31      rightLayout->addStretch();
32      QHBoxLayout *mainLayout = new QHBoxLayout;
33      mainLayout->addLayout(leftLayout);
34      mainLayout->addLayout(rightLayout);
35      setLayout(mainLayout);
```

Затем для размещения виджетов в окне мы используем менеджеры компоновки (layout managers). Менеджеры компоновки могут содержать как виджеты, так и другие менеджеры компоновки. Используя различные вложенные комбинации менеджеров компоновки QHBoxLayout, QVBoxLayout, и QGridLayout, можно построить очень сложные диалоговые окна.

Для диалогового окна поиска мы используем два менеджера горизонтальной компоновки QHBoxLayout и два менеджера вертикальной компоновки QVBoxLayout (см. рис 2). Внешний менеджер компоновки является главным; он устанавливается в FindDialog в строке 35 и ответственен за всю область, занимаемую диалоговым окном. Остальные три менеджера компоновки являются внутренними. Показанная в нижнем правом углу на рис 2 маленькая «пружинка» является пустым промежутком («распоркой»). Она применяется для образования ниже кнопок Find и Close пустого пространства, обеспечивающего перемещение кнопок в верхнюю часть своего менеджера компоновки.

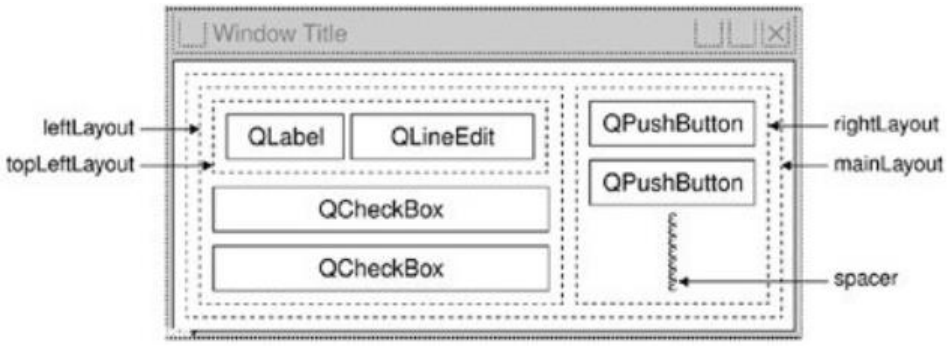

Рис. 2 – Менеджеры компоновки диалогового окна поиска данных

Одна из особенностей классов менеджеров компоновки заключается в том, что они не являются виджетами. Взамен этого они наследуют свойства класса QLayout, который, в свою очередь, является наследником класса QObject. На данном рисунке виджеты выделены сплошными линиями, а менеджеры компоновки очерчены пунктирными линиями, чтобы подчеркнуть их различие. При работе приложения менеджеры компоновки невидимы.

При добавлении внутренних менеджеров компоновки к родительскому менеджеру компоновки (строки 25, 33 и 34) для них автоматически устанавливается родительская связь. Затем, когда главный менеджер компоновки устанавливается для диалога (строка 35), он становится дочерним

элементом диалога и все виджеты в менеджерах компоновки становятся дочерними элементами диалога. Иерархия полученных родословных связей представлена на рис 3.

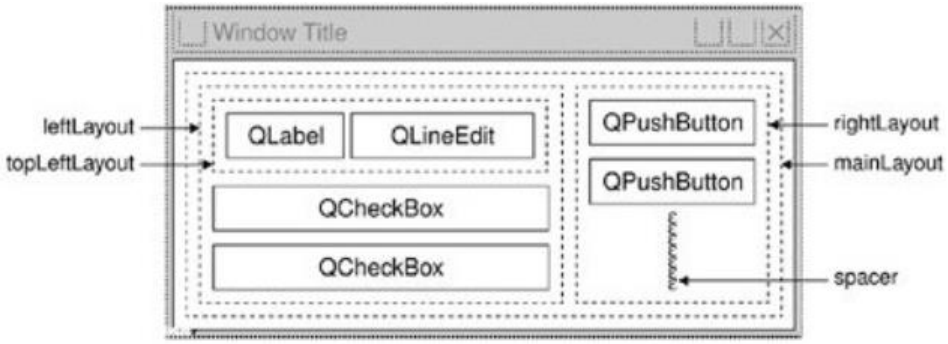

Рис. 3 – Родословная объектов диалогового окна поиска данных

```
36     setWindowTitle(tr("Find"));
37     setFixedHeight(sizeHint().height());
38 }
```

Наконец, мы задаем название диалогового окна и устанавливаем фиксированной его высоту, поскольку в диалоговом окне нет виджетов, которым может понадобиться дополнительное пространство по вертикали. Функция QWidget::sizeHint() возвращает «идеальный» размер виджета.

На этом завершается рассмотрение конструктора FindDialog. Поскольку нами использован оператор new при создании виджетов и менеджеров компоновки, нам, по-видимому, придется написать деструктор, где будут предусмотрены операторы delete для удаления каждого созданного нами виджета и менеджера компоновки. Но поступать так не обязательно, поскольку Qt автоматически удаляет дочерние объекты при разрушении родительского объекта, а все дочерние виджеты и менеджеры компоновки являются потомками FindDialog.

Теперь мы рассмотрим слоты диалогового окна:

```
39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43             caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                       : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
47     } else {
48         emit findNext(text, cs);
49     }
50 }
51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }
```

Слот findClicked() вызывается при нажатии пользователем кнопки Find. Он генерирует сигнал findPrevious() или findNext() в зависимости от состояния

флажка Search backward (поиск в обратном направлении). Ключевое слово emit (генерировать сигнал) имеет особый смысл в Qt; как и другие расширения Qt, оно преобразуется препроцессором C++ в стандартные инструкции C++.

Слот enableFindButton() вызывается при любом изменении значения в строке редактирования. Он устанавливает активный режим кнопки, если в редактируемой строке имеется какой-нибудь текст; в противном случае кнопка устанавливается в неактивный режим.

Эти два слота завершают написание программы диалогового окна. Теперь мы можем создать файл main.cpp и протестировать наш виджет FindDialog:

```
1 #include <QApplication>
2 #include "finddialog.h"
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }
```

Теперь выполните программу. Если клавиши быстрого доступа доступны на вашей платформе, убедитесь в правильной работе клавиш Alt+W, Alt+C, Alt+B и Alt+F. Для перехода с одного виджета на другой используйте клавишу табуляции Tab. По умолчанию последовательность таких переходов соответствует порядку создания виджетов. Эту последовательность можно изменить с помощью функции QWidget::setTabOrder().

Обеспечение осмысленного порядка переходов с одного виджета на другой с помощью клавиши табуляции и применение клавиш быстрого доступа позволяют использовать все возможности приложений тем пользователям, которые не хотят (или не могут) пользоваться мышкой. Тот, кто быстро работает с клавиатурой, также предпочитает иметь возможность полного управления приложением посредством клавиатуры.

4. Механизм сигналов и слотов

Механизм сигналов и слотов играет решающую роль в разработке программ Qt. Он позволяет прикладному программисту связывать различные объекты, которые ничего не знают друг о друге. Мы уже соединяли некоторые сигналы и слоты, объявляли наши собственные сигналы и слоты, реализовывали наши собственные слоты и генерировали наши собственные сигналы. Рассмотрим этот механизм более подробно.

Слоты почти совпадают с обычными функциями, которые объявляются внутри классов C++ (функции-члены). Они могут быть виртуальными, они могут быть перегруженными, они могут быть открытыми (public), защищенными (protected) и закрытыми (private), они могут вызываться непосредственно, как и любые другие функции-члены C++ и их параметры, могут быть любого типа. Однако слоты (в отличие от обычных функций-членов) могут подключаться к сигналам, и в результате они будут вызываться

при каждом генерировании соответствующего сигнала.

Оператор connect() выглядит следующим образом:

```
connect(отправитель, SIGNAL(сигнал), получатель, SLOT(слот));
```

где отправитель и получатель являются указателями на объекты QObject и где сигнал и слот являются сигнатурами функций без имен параметров. Макросы SIGNAL() и SLOT() фактически преобразуют свои аргументы в строковые переменные.

В приводимых ранее примерах мы всегда подключали разные слоты к разным сигналам. Существует несколько вариантов подключения слотов к сигналам.

К одному сигналу можно подключать много слотов:

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

При генерировании сигнала последовательно вызываются все слоты, причем порядок их вызова неопределен.

Один слот можно подключать ко многим сигналам:

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

Данный слот будет вызываться при генерировании любого сигнала.

Один сигнал может соединяться с другим сигналом:

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

При генерировании первого сигнала будет также генерироваться второй сигнал, остальном связь «сигнал-сигнал» не отличается от связи «сигнал-слот».

Связь можно аннулировать:

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

Это редко приходится делать, поскольку **Qt автоматически убирает все связи при удалении объекта.**

При успешном соединении сигнала со слотом (или с другим сигналом) их параметры должны задаваться в одинаковом порядке и иметь одинаковый тип:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

Имеется одно исключение, а именно: если у сигнала больше параметров, чем у подключенного слота, то дополнительные параметры просто игнорируются:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

Если параметры имеют несовместимые типы либо будет отсутствовать сигнал или слот, то Qt выдаст предупреждение во время выполнения программы, если сборка программы проводилась в отладочном режиме. Аналогично Qt выдаст предупреждение, если в сигнатуре сигнала или слота будут указаны имена параметров.

До сих пор мы использовали сигналы и слоты только при работе с виджетами. Но сам по себе этот механизм реализован в классе QObject, и его не обязательно применять только в пределах программирования графического пользовательского интерфейса. Этот механизм можно использовать в любом подклассе QObject:

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }
public slots:
    void setSalary(int newSalary);
signals:
    void salaryChanged(int newSalary);
private:
    int mySalary;
};
void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

Обратите внимание на реализацию слота setSalary(). Мы генерируем сигнал salary-Changed() только при выполнении условия newSalary != mySalary. Это позволяет предотвратить бесконечный цикл генерирования сигналов и вызовов слотов.

Мета объектная система Qt

Одним из главных преимуществ средств разработки Qt является расширение языка C++ механизмом создания независимых компонентов программного обеспечения, которые можно соединять вместе, несмотря на то что они могут ничего не знать друг о друге.

Этот механизм называется метаобъектной системой, и он обеспечивает две основные служебные функции: взаимодействие сигналов и слотов и анализ внутреннего состояния приложения (introspection). Анализ внутреннего состояния необходим для реализации сигналов и слотов и позволяет прикладным программистам получать «метаинформацию» о подклассах QObject во время выполнения программы, включая список поддерживаемых объектом сигналов и слотов и имена их классов. Этот механизм также поддерживает свойства (для Qt Designer) и перевод текстовых значений (для интернационализации приложений), а также создает основу для системы сценариев в Qt (Qt Script for Applications – QSA).

В стандартном языке C++ не предусмотрена динамическая поддержка метаданных, необходимых системе метаобъектов Qt. В Qt эта проблема решена за счет применения специального инструментального средства компилятора moc, который просматривает определения классов с макросом Q_OBJECT и делает соответствующую информацию доступной функциям C++. Поскольку все функциональные возможности moc обеспечиваются только с помощью «чистого» C++, мета-объектная система Qt будет работать с любым компилятором C++.

Этот механизм работает следующим образом:

- макрос `Q_OBJECT` объявляет некоторые функции, которые необходимы для анализа внутреннего состояния и которые должны быть реализованы в каждом подклассе `QObject:metaObject()`, `TR()`, `qt_metacall()` и некоторые другие.
- компилятор `moc` генерирует реализации функций, объявленных макросом `Q_OBJECT`, и всех сигналов.
- такие функции-члены класса `QObject`, как `connect()` и `disconnect()`, во время своей работы используют функции анализа внутреннего состояния.

Все это выполняется автоматически при работе qmake, moc, и при компиляции QObject, и поэтому у вас крайне редко может возникнуть необходимость вспомнить об этом механизме. Однако если вам интересны детали реализации этого механизма, вы можете воспользоваться документацией по классу QMetaObject и просмотреть файлы исходного кода C++, сгенерированные компилятором moc.

5. Быстрое проектирование диалоговых окон

Многие программисты предпочитают применять визуальные средства проектирования форм, поскольку этот метод представляется более естественным и позволяет получать конечный результат быстрее, чем при программировании «вручную», и такой подход дает возможность программистам быстрее и легче экспериментировать и изменять дизайн.

Qt Designer расширяет возможности программистов, предоставляя визуальные средства проектирования. *Qt Designer* может использоваться для разработки всех или только некоторых форм приложения. Формы, созданные с помощью *Qt Designer*, в конце концов представляются в виде программного кода на C++, поэтому *Qt Designer* может использоваться совместно с обычными средствами разработки, и он не налагает никаких специальных требований на компилятор.

В данном разделе мы применяем Qt Designer для создания диалогового окна (см. рис 4.), которое управляет переходом на заданную ячейку таблицы (Go-to-Cell dialog). Создание диалогового окна как при ручном кодировании, так и при использовании Qt Designer предусматривает выполнение следующих шагов:

- Создание и инициализация дочерних виджетов.
- Размещение дочерних виджетов в менеджерах компоновки.
- Определение последовательности переходов по клавише табуляции.
- Установка соединений «сигнал - слот».
- Реализация пользовательских слотов диалогового окна.

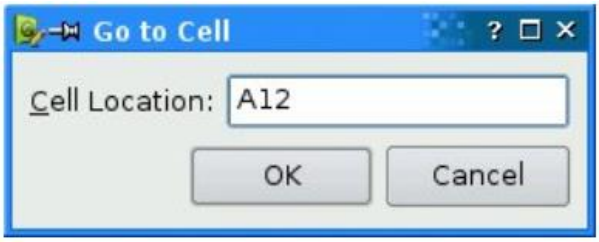

Рис. 4 – Диалоговое окно для перехода на заданную ячейку таблицы

На рис. 5 приведен интерфейс программы *Qt Designer*.

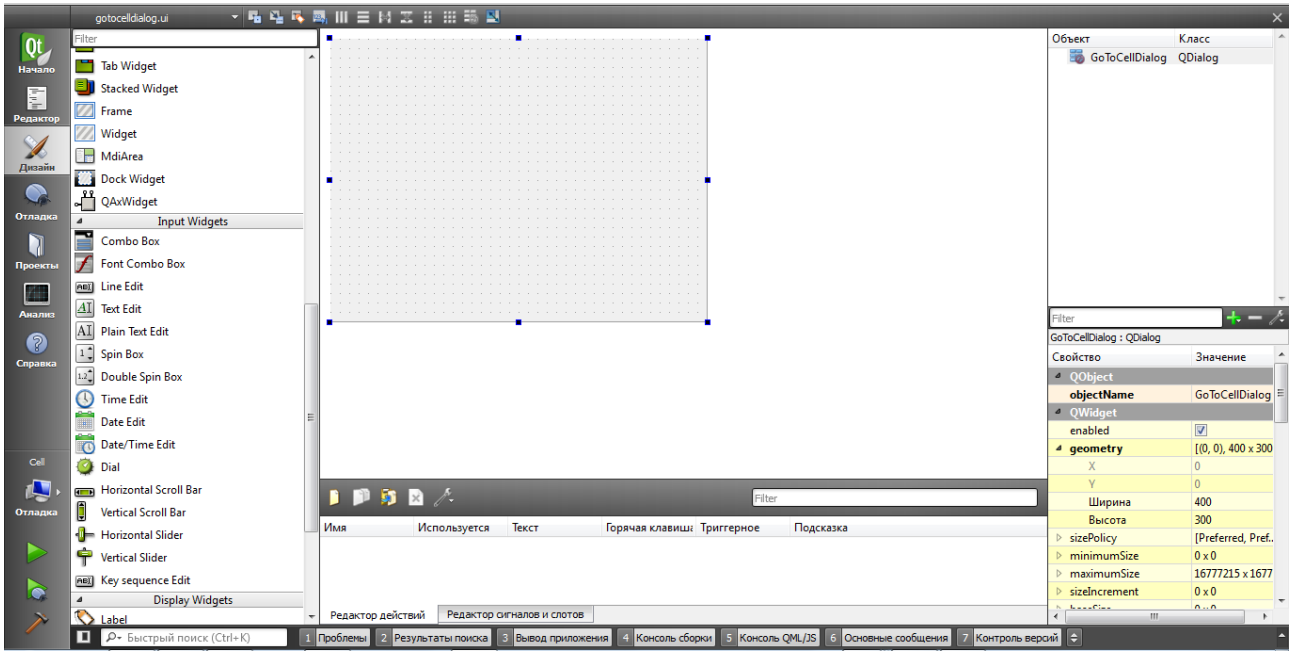

Рис. 5 – Интерфейс *Qt Designer* в версиях Qt 5.0.1 и старше

На первом этапе создайте дочерние виджеты и поместите их в форму. Создайте одну текстовую метку, одну строку редактирования, одну (горизонтальную) распорку (spacer) и две кнопки. При создании любого элемента перенесите его название или пиктограмму из окна виджетов Qt Designer на форму приблизительно в то место, где он должен располагаться. Элемент распорка, который не будет видим при работе формы, в Qt Designer показан в виде синей пружинки.

Затем передвиньте низ формы вверх, чтобы она стала короче. В результате вы получите форму, похожую на показанную на рис. 6.

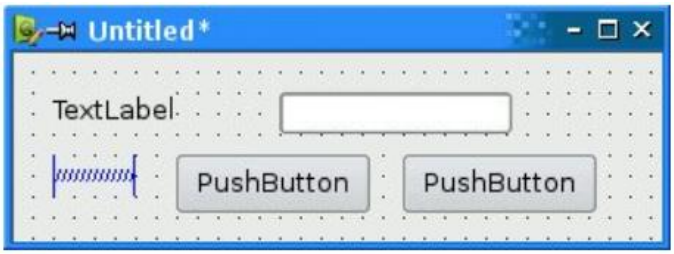

Рис. 6 – Форма с несколькими виджетами

Задайте свойства каждого виджета, используя редактор свойств Qt Designer:

1. Щелкните по текстовой метке. Убедитесь, что свойство objectName (имя объекта) имеет значение «label» (текстовая метка), а свойство text (текст) установите на значение «&Cell Location» (расположение ячейки).
2. Щелкните по строке редактирования. Убедитесь, что свойство objectName имеет значение «lincEdit» (строка редактирования).
3. Щелкните по первой кнопке. Установите свойство objectName на значение «окButton» (кнопка подтверждения), свойство enabled (включена) на

значение «false» (ложь), свойство default (режим умолчания) на «true» (истина), свойство text на значение «ОК» (подтвердить).

4. Щелкните по второй кнопке. Установите свойство objectName на значение «cancel Button» (кнопка отмены) и свойство text назначение «Cancel» (отменим.).
5. Щелкните по свободному месту формы для выбора самой формы. Установите objectName на значение «GoToCellDialog» (диалоговое окно перехода на ячейку) и windowTitle (заголовок окна) на значение «Go to Cell» (перейти на ячейку).

Рис. 7 – Вид формы после установки свойств виджетов

Выберите Edit|Edit Buddies (Правка|Редактировать партнеров) для входа в специальный режим, позволяющий задавать партнеров. Щелкните по этой метке и перенесите красную стрелку на строку редактирования а, затем отпустите кнопку мышки. Теперь эта метка будет выглядеть как Cell Location и иметь строку редактирования в качестве партнера. Выберите Edit|Edit Widgets (Правка|Редактировать виджеты) для выхода из режима установки партнеров.

На следующем этапе виджеты размещаются в форме требуемым образом:

1. Щелкните по текстовой метке Cell Location и нажмите клавишу Shift одновременно со щелчком по полю редактирования, обеспечив одновременный выбор этих виджетов. Выберите в меню Form|Lay Out Horizontally (Форма|Горизонтальная компоновка).
2. Щелкните по растяжке, затем, удерживая клавишу Shift, щелкните по клавишам ОК и Cancel. Выберите в меню Form|Lay Out Horizontally.
3. Щелкните по свободному месту формы, аннулируя выбор любых виджетов, затем выберите в меню функцию Form|Lay Out Vertically (Форма|Вертикальная компоновка).
4. Выберите в меню функцию Form|Adjust Size для установки предпочитаемого размера формы.

Красными линиями на форме обозначаются созданные менеджеры компоновки. Они невидимы при выполнении программы.

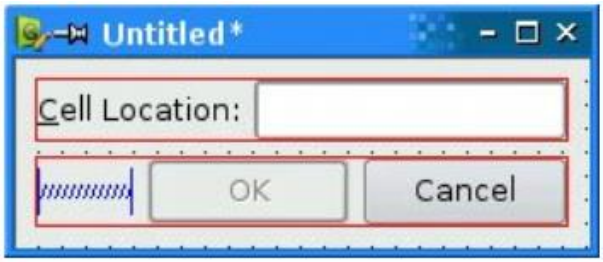

Рис. 8 – Форма с менеджерами компоновки

Теперь выберите в меню функцию Edit|Edit Tab Order (Правка|Редактировать порядок перехода по клавише табуляции). Рядом с каждым виджетом, которому может передаваться фокус, появятся синие прямоугольники. Щелкните по каждому виджету, соблюдая необходимую вам последовательность перевода фокуса, затем выберите в меню функцию Edit|Edit Widgets для выхода из режима редактирования переходов по клавише табуляции.

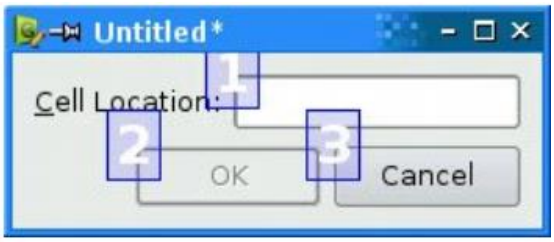

Рис. 9 – Установка последовательности перевода фокуса по виджетам формы

Для предварительного просмотра спроектированного диалогового окна выберите в меню функцию Form|Preview (Форма|Предварительный просмотр). Проверьте последовательность перехода фокуса, нажимая несколько раз клавишу табуляции. Нажмите одновременно клавиши Alt+C для перевода фокуса на строку редактирования. Нажмите на кнопку Cancel для прекращения работы.

Создайте файл main.cpp:

```
#include <QApplication>
#include <QDialog>
#include "ui_gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Ui::GoToCellDialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();
    return app.exec();
}
```

Теперь выполните сборку и компиляцию программы. Средства Qt обнаружат файл пользовательского интерфейса gotocelldialog.ui и сгенерирут соответствующие команды для вызова uic-компилятора пользовательского интерфейса, входящего в состав средств разработки Qt. Компилятор

uic преобразует gotocelldialog.ui в инструкции C++ и помещает результат в ui_gotocelldialog.h.

Полученный файл ui_gotocelldialog.h содержит определение класса Ui::GoToCellDialog, который содержит инструкции C++, эквивалентные файлу gotocelldialog.ui. В этом классе объявляются переменные-члены, в которых содержатся дочерние виджеты и менеджеры компоновки формы, а также функция setupUi(), которая инициализирует форму. Сгенерированный класс выглядит следующим образом:

```
class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...
    void setupUi(QWidget *widget) {
        ...
    }
};
```

Сгенерированный класс не наследует никакой Qt-класс. При использовании формы в main.cpp мы создаем QDialog и передаем его функции setupUi().

Если вы станете выполнять программу в данный момент, она будет работать, но не совсем так, как требуется:

- Кнопка OK всегда будет в неактивном состоянии.
- Кнопка Cancel не выполняет никаких действий.
- Поле редактирования будет принимать любой текст, а оно должно принимать только допустимое обозначение ячейки.

Правильную работу диалогового окна мы можем обеспечить, написав некоторый программный код. Лучше всего создать новый класс, который наследует QDialog и Ui::GoToCellDialog и реализует недостающую функциональность (подтверждая известное утверждение, что любую проблему программного обеспечения можно решить, просто добавив еще один уровень представления объектов). По нашим правилам мы даем этому новому классу такое же имя, которое генерируется компилятором uic, но без префикса Ui::.

Используя текстовый редактор, создайте файл с именем gotocelldialog.h, который будет содержать следующий код:

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H
#include <QDialog>
#include "ui_gotocelldialog.h"
class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
    Q_OBJECT
```

```
public:
    GoToCellDialog(QWidget *parent = 0);
private slots:
    void on_lineEdit_textChanged();
};
#endif
```

Реализация методов класса делается в файле gotocelldialog.cpp:

```
#include <QtGui>
#include "gotocelldialog.h"
GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}
void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

В конструкторе мы вызываем setupUi() для инициализации формы. Благодаря множественному наследованию мы можем непосредственно получить доступ к членам класса Ui::GoToCellDialog. После создания пользовательского интерфейса setupUi() будет также автоматически подключать все слоты с именами типа on_objectName_signalName() к соответствующему сигналу signalName() виджета objectName. В нашем примере это означает, что setupUi() будет устанавливать следующее соединение «сигнал-слот»:

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));
```

Также в конструкторе мы задаем ограничение на допустимый диапазон вводимых значений. Qt обеспечивает три встроенных класса по проверке правильности значений: QIntValidator, QDoubleValidator и QRegExpValidator. В нашем случае мы используем QRegExpValidator, задавая регулярное выражение «[A-Za-z][1-9][0-9]{0,2}», которое означает следующее: допускается одна маленькая или большая буква, за которой следует одна цифра в диапазоне от 1 до 9; затем идут ноль, одна или две цифры в диапазоне от 0 до 9. (Введение в регулярные выражения вы можете найти в документации по классу QRegExp.)

Указывая в конструкторе QRegExpValidator значение this, мы его делаем дочерним злсментом объекта GoToCellDialog. После этого нам можно не беспокоиться об удалении в будущем QRegExpValidator; этот объект будет удален автоматически после удаления его родигельского элемента.

Механизм взаимодействия объекта с родительскими и дочерними элементами реализован в QObject. Когда мы создаем объект (виджет, функцию по проверке правильности значений или любой другой объект) и он имеет

родительский объект, то к списку дочерних элементов этого родителя добавится и данный объект. При удалении родительского элемента будет просмотрен список его дочерних элементов и все они будут удалены. Эти дочерние элементы, в свою очередь, сами удалят все свои дочерние элементы, и эта процедура будет выполняться до тех пор, пока ничего не останется.

Механизм взаимодействия объекта с родительскими и дочерними элементами значительно упрощает управление памятью, снижая риск утечек памяти. Явным образом мы должны удалять только объекты, которые созданы оператором new и которые не имеют родительского элемента. А если мы удаляем дочерний элемент до удаления его родителя, то Qt автоматически удалит этот объект из списка дочерних объектов этого родителя.

Для виджетов родительский объект имеет дополнительный смысл: дочерние виджеты размешаются внутри области, которую занимает родительский объект. При удалении родительского виджета не только освобождается занимаемая дочерними объектами память - он исчезает с экрана.

В конце конструктора мы подключаем кнопку ОК к слоту accept() виджета QDialog и кнопку Cancel к слоту reject(). Оба слота закрывают диалог, но accept() устанавливает результат диалога на значение QDialog::Accepted (которое равно 1), а reject() устанавливает значение QDialog::Rejected (которое равно 0). При использовании этого диалога мы можем использовать значение результата, чтобы узнать, была ли нажата кнопка ОК, и действовать соответствующим образом.

Слот on_lineEdit_textChanged() устанавливает кнопку ОК в активное или неактивное состояние в зависимости от наличия в строке редактирования допустимого обозначения ячейки. QLineEdit::hasAcceptableInput() использует функцию проверки допустимости значений, которую мы задали в конструкторе.

На этом завершается построение диалога. Теперь мы можем переписать main.cpp следующим образом:

```
#include <QApplication>
#include "gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
    return app.exec();
}
```

Выполните приложение. Наберите в строке редактирования значение «A12» и обратите внимание на то, как кнопка ОК становится активной. Попытайтесь ввести какой-нибудь произвольный текст и посмотрите, как сработает функция по проверке допустимости значения. Нажмите кнопку Cancel для закрытия диалогового окна.

Особенностью применения Qt Designer является возможность для программиста действовать достаточно свободно при изменении дизайна формы, причем при этом исходный код программы не будет нарушен. При

разработке формы с непосредственным написанием операторов C++ на изменение дизайна уходит много времени. При использовании Qt Designer не будет тратиться много времени, поскольку uic просто заново генерирует исходный код программы для форм, которые были изменены. Пользовательский интерфейс диалога сохраняется в файле .ui (который имеет формат XML), а соответствующая функциональная часть реализуется путем создания подкласса, сгенерированного компилятором uic класса.

6. Методические указания по выполнению работы

Лабораторную работу необходимо выполнить в следующей последовательности:

1. Изучите теоретические разделы 2-5, выполните на компьютере все приведенные в данных разделах примеры, скомпилируйте и запустите на исполнение все записанные программы. В качестве дополнительных источников информации, воспользуйтесь сведениями раздела 7, а так же указанными в нем пособиями, а также открытыми источниками в сети Интернет.
2. Самостоятельно выполните задание по разработке изменяющегося диалогового окна, описанное ниже в разделе 6.1.
3. Подготовьте исходный код программы в Qt Creator и файл формы в Qt Designer и покажите их преподавателю.
4. В случае отсутствия видимых ошибок в коде, скомпилируйте программу и запустите ее на выполнение.
5. Получите оценку.

6.1. Изменяющиеся диалоговые окна

Нами были рассмотрены способы формирования диалоговых окон, которые всегда содержат одни и те же виджеты. В некоторых случаях требуется иметь диалоговые окна, форма которых может меняться. Наиболее известны два типа изменяющихся диалоговых окон: расширяемые диалоговые окна (are extension dialogs) и многостраничные диалоговые окна (multi-page dialogs). Оба типа диалоговых окон можно реализовать в Qt либо с помощью непосредственного кодирования, либо посредством применения Qt Designer.

Расширяемые диалоговые окна, как правило, имеют обычное (нерасширенное) представление и содержат кнопку для переключения между обычным и расширенным представлениями этого диалогового окна. Расширяемые диалоговые окна обычно применяются в тех приложениях, которые предназначаются как для неопытных, так и для опытных пользователей и скрывают дополнительные опции до тех пор, пока пользователь явным образом не захочет ими воспользоваться. Мы будем использовать Qt Designer для создания расширяемого диалогового окна, показанного на рис. 10.

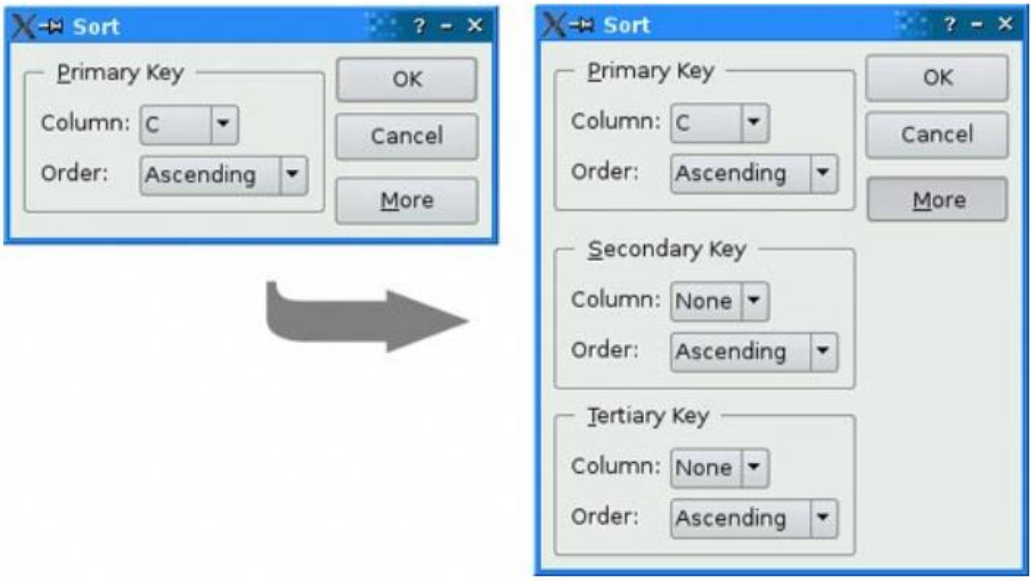

Рис. 10 – Обычный и расширенный виды окна сортировки данных

Данное диалоговое окно является окном сортировки в приложении Электронная таблица, позволяющим пользователю задавать один или несколько столбцов сортировки. В обычном представлении этого окна пользователь может ввести один ключ сортировки, а в расширенном представлении он может ввести дополнительно еще два ключа сортировки. Кнопка More (больше) позволяет пользователю переключаться с обычного представления на расширенное и наоборот.

Мы создадим в Qt Designer расширенное представление виджета, второй и третий ключи сортировки которого не будут видны при выполнении программы, когда они не нужны. Этот виджет кажется сложным, однако он очень легко строится в Qt Designer. Сначала нужно создать ту часть, которая относится к первичному ключу, затем сдублировать ее дважды, получая вторичный и третичный ключи:

1. Выберите функцию меню File|New Form и затем шаблон «Dialog with Buttons Right» (диалог с кнопками, расположенными справа).
2. Создайте кнопку More (больше) и перенесите ее в вертикальный менеджер компоновки ниже вертикальной распорки. Установите свойство text кнопки More на значение «&Moге», а свойство checkable - на значение «true». Задайте свойство default кнопки OK на значение «true».
3. Создайте объект «группа элементов (group box)», две текстовые метки, два поля с выпадающим списком (comboboxes) и одну горизонтальную распорку и разместите их где-нибудь на форме.
4. Передвиньте нижний правый угол элемента группа, увеличивая его. Затем перенесите другие виджеты внутрь элемента группа и расположите их приблизительно так, как показано на рис. 11 (а).

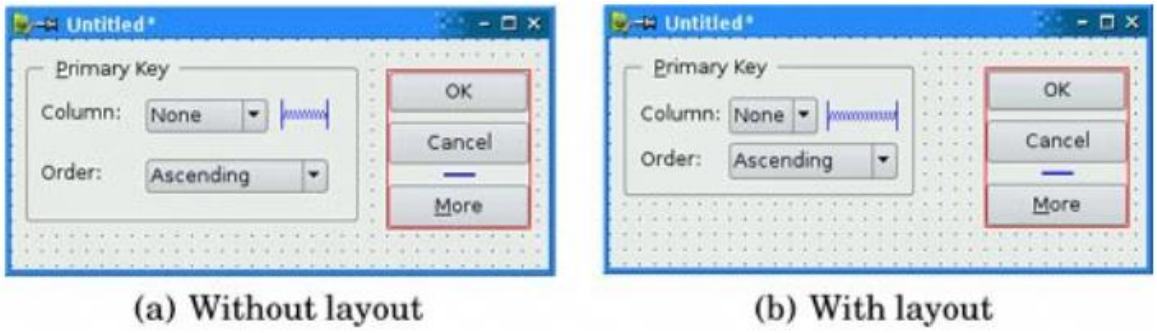

Рис. 11 – Размещение дочерних виджетов группового элемента в табличной сетке

5. Перетащите правый край второго поля с выпадающим списком так, чтобы оно было в два раза шире первого поля.
6. Свойство title (заголовок) группы установите на значение «&Primary Key» (первичный ключ), свойство text первой текстовой метки установите на значение «Column:» (столбец), а свойство text второй текстовой метки установите на значение «Order:» (порядок сортировки).
7. Щелкните правой клавишей мышки по первому полю с выпадающим списком и выберите функцию Edit Items (редактировать элементы) в контекстном меню для вызова в Qt Designer редактора списков. Создайте один элемент со значением «None» (нет значений).
8. Щелкните правой клавишей мышки по второму полю с выпадающим списком и выберите функцию Edit Items. Создайте элементы «Ascending» (по возрастанию) и «Descending» (по убыванию).
9. Щелкните по группе и выберите в меню функцию Form|Lay Out in a Grid (Форма|Размещение в сетке). Еще раз щелкните по группе и выберите в меню функцию Form|Adjust Size (Форма|Настроить размер). В результате получите изображение, представленное на рис. 11 (b).

Если изображение оказалось не совсем таким или вы ошиблись, то всегда можно выбрать в меню функцию Edit|Undo (Правка|Отменить) или Form|Break Layout (Форма|Прервать компоновку), затем изменить положение виджетов и снова повторить все действия.

Теперь мы добавим групповые элементы для второго и третьего ключей сортировки:

1. Увеличьте высоту диалогового окна, чтобы можно было в нем разместить дополнительные части.
2. При нажатой клавише Ctrl щелкните по элементу группы Primary Key (первичный ключ) для создания копии элемента группа (и его содержимого) над оригинальным элементом. Перетащите эту копию ниже оригинального элемента группа, по-прежнему нажимая клавишу Ctrl (или Alt). Повторите этот процесс для создания третьего элемента группа, размещая его ниже второго элемента группа.
3. Измените их свойство title на значения «&Secondary Key» (вторичный

ключ) и «&Tertiary Key» (третичный ключ).

4. Создайте одну вертикальную растяжку и расположите ее между элементом группы первичного ключа и элементом группы вторичного ключа.
5. Расположите виджеты в сетке, как показано на рис. 12 (a).

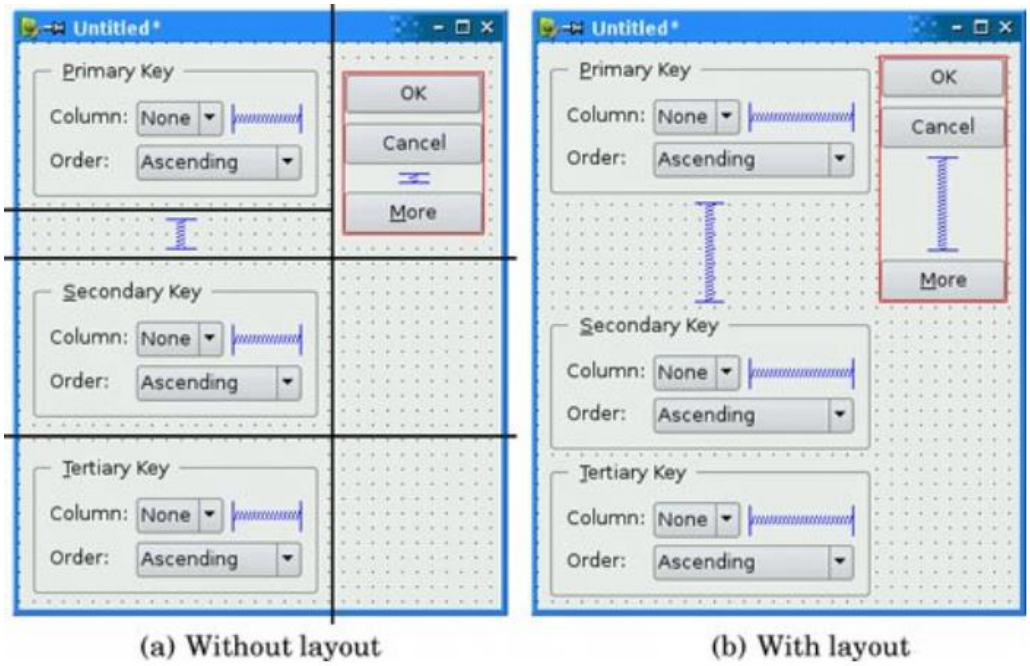

Рис. 12 – Расположение дочерних элементов формы в сетке

6. Щелкните по форме, чтобы отменить выбор любых виджетов, затем выберите функцию меню Form|Lay Out in a Grid (Форма|Расположить в сетке). Форма должна иметь вид, показанный на рис. 12 (b).
7. Свойство sizeHint («идеальный» размер) двух вертикальных растяжек установите на значение [20, 0].

В результате менеджер компоновки в ячейках сетки будет иметь два столбца и четыре строки - всего восемь ячеек. Элемент группа первичного ключа, левая вертикальная распорка, элемент группа вторичного ключа и элемент группа третичного ключа - каждый из них занимает одну ячейку. Менеджер вертикальной компоновки, содержащий кнопки OK, Cancel и More, занимает две ячейки. Справа внизу диалогового окна будет две свободные ячейки. Если у вас получилась другая картинка, отмените компоновку, измените положение виджетов и повторите все сначала.

Переименуйте форму на «SortDialog» (диалоговое окно сортировки) и измените заголовок на «Sort» (сортировка). Задайте имена дочерним виджетам, как показано на рис. 13.

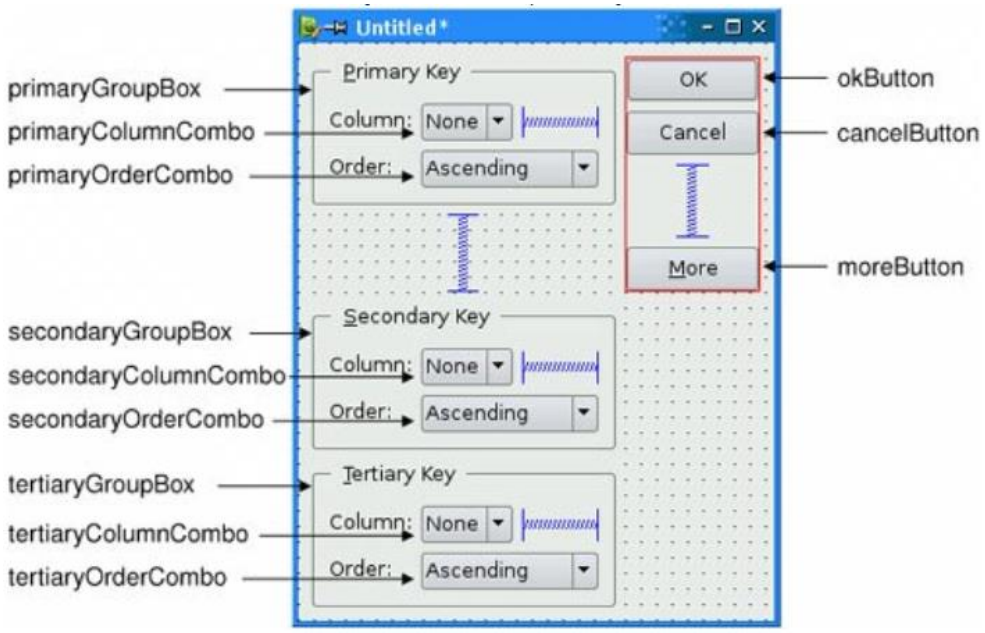

Рис. 13 – Имена виджетов формы

Выберите функцию меню Edit|Edit Tab Order. Щелкайте поочередно по каждому выпадающему списку, начиная с верхнего и заканчивая нижним, затем щелкайте по кнопкам OK, Cancel и More, которые расположены справа. Выберите функцию меню Edit|Edit Widgets для выхода из режима установки переходов по клавише табуляции.

Теперь, когда форма спроектирована, мы готовы обеспечить ее функциональное наполнение, устанавливая некоторые соединения «сигнал-слот». Qt Designer позволяет устанавливать соединения между виджетами одной формы. Нам требуется обеспечить два соединения.

Выберите функцию меню Edit|Edit Signals/Slots (Правка|Редактировать сигналы и слоты) для входа в режим формирования соединений в Qt Designer. Соединения представлены синими стрелками между виджетами формы. Поскольку нами выбран шаблон «Dialog with Buttons Rights», кнопки OK и Cancel уже подключены к слотам accept() и reject() виджета QDialog. Эти соединения также указаны в окне редактора сигналов и слотов Qt Designer.

Для установки соединения между двумя виджетами щелкните по виджету, передающему сигнал, соедините красную стрелку с виджетом - получателем сигнала и отпустите клавишу мышки. В результате будет выдано диалоговое окно, позволяющее выбрать для соединения сигнал и слот.

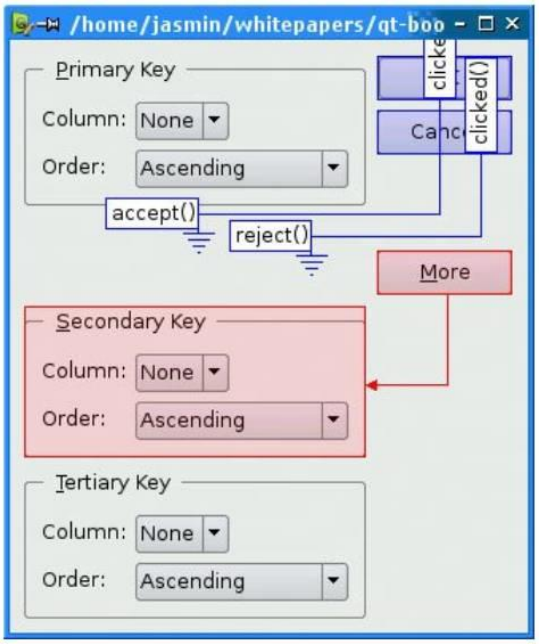

Рис. 14 – Соединение виджетов формы

Сначала устанавливается соединение между moreButton и secondary-GroupBox. Соедините эти два виджета красной стрелкой, затем выберите toggled(bool) в качестве сигнала и setVisible(bool) в качестве слота. По умолчанию Qf Designer не имеет в списке слотов setVisible(bool), но он появится, если вы включите режим «Show all signals and slots» (Показывать все сигналы и слоты).

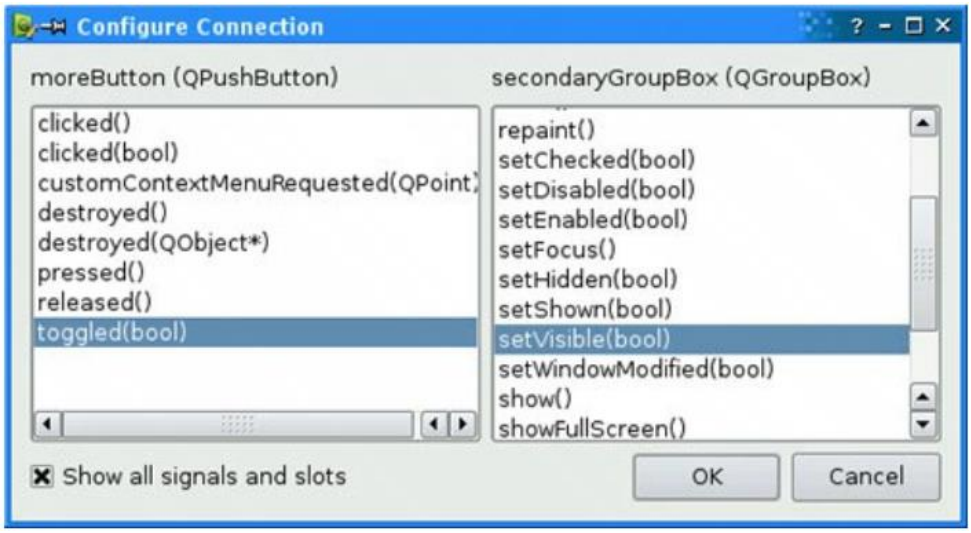

Рис. 15 – Редактор соединений в Qt Designer

Второе соединение устанавливается между сигналом toggled(bool) виджета moreButton и слотом setVisible(bool) виджета tertiaryGroupBox. После установки соединения выберите функцию меню Edit|Edit Widgets для выхода из режима установки соединений.

Сохраните диалог под именем sortdialog.ui в каталоге sort. Для добавления программного кода в форму мы будем использовать тот же подход

на основе множественного наследования, который нами применялся в предыдущем разделе для диалога «Go-to-Cell».

Сначала создаем файл sortdialog.h со следующим содержимым:

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H
#include <QDialog>
#include "ui_sortdialog.h"
class SortDialog : public QDialog, public Ui::SortDialog
{
    Q_OBJECT
public:
    SortDialog(QWidget *parent = 0);
    void setColumnRange(QChar first, QChar last);
};
#endif
```

Затем создаем sortdialog.cpp:

```
 1 #include <QtGui>
 2 #include "sortdialog.h"
 3 SortDialog::SortDialog(QWidget *parent)
 4     : QDialog(parent)
 5 {
 6     setupUi(this);
 7     secondaryGroupBox->hide();
 8     tertiaryGroupBox->hide();
 9     layout()->setSizeConstraint(QLayout::SetFixedSize);
10     setColumnRange('A', 'Z');
11 }
12 void SortDialog::setColumnRange(QChar first, QChar last)
13 {
14     primaryColumnCombo->clear();
15     secondaryColumnCombo->clear();
16     tertiaryColumnCombo->clear();
17     secondaryColumnCombo->addItem(tr("None"));
18     tertiaryColumnCombo->addItem(tr("None"));
19     primaryColumnCombo->setMinimumSize(
20             secondaryColumnCombo->sizeHint());
21     QChar ch = first;
22     while (ch <= last) {
23         primaryColumnCombo->addItem(QString(ch));
24         secondaryColumnCombo->addItem(QString(ch));
25         tertiaryColumnCombo->addItem(QString(ch));
26         ch = ch.unicode() + 1;
27     }
28 }
```

Конструктор прячет ту часть диалогового окна, где располагаются поля второго и третьего ключей. Он также устанавливает свойство sizeConstraint менеджера компоновки формы на значение QLayout::SetFixedSize, не позволяя пользователю изменять размер диалогового окна. Для того чтобы размер диалогового окна был оптимальным, менеджер компоновки всю работу по изменению размера диалогового окна берет на себя, изменяя его размер тогда, когда дочерний виджет видим или скрыт.

setColumnRange() слот инициализирует содержимое выпадающих

списков, основываясь на выбранных столбцах в таблице. Вставте значение «None» в содержимое выпадающих списков для (необязательно) вторичного и третичного ключей.

Строки 19 и 20 представляют собой тонкие идиомы расположения. QWidget::sizeHint() возвращает «идеальный» размер виджета, расположение которого система пытается соблюдать. Это объясняет, почему различным видам виджетов или подобным виджетам с различным содержанием могут быть назначены различные размеры системой расположения. Для выпадающих списков это означает, что вторичные и третичные выпадающие списки, которые содержат «None», в конечном счете больше, чем первичный выпадающий список, который содержит только однобуквенные записи. Чтобы избежать этой несогласованности, установите минимальный размер первичного выпадающего списка в идеальный размер вторичного выпадающего списка.

Это main() тест-функция, которая установит диапазон столбцов 'C' и 'F', а затем покажет диалог:

```
#include <QApplication>
#include "sortdialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

7. Справочная документация

Справочная документации по средствам разработки Qt является важным инструментом в руках любого разработчика Qt-программ, поскольку в ней есть все необходимые сведения по любому классу и любой функции Qt. Для более эффективного использования Qt вам необходимо хорошо разбираться в справочной документации.

Эта документация имеется в формате HTML в сети Интернет и ее можно просматривать любым веб-браузером. Вы можете также использовать программу Qt Assistant (помощник Qt) – браузер системы помощи в Qt, который обладает мощными средствами поиска и индексирования информации и поэтому быстрее находит нужную информацию и им легче пользоваться, чем веб-браузером. Программный интерфейс программы Qt Assistant приведен на рис. 16.

Справочная документация для текущей версии Qt и нескольких более старых версий можно найти в сети Интернет по адресу http://doc.trolltech.com/. На этом сайте также находятся избранные статьи из журнала Qt Quarterly (Ежеквартальное обозрение по средствам разработки Qt); этот журнал предназначается для программистов Qt и распространяется по всем коммерческим лицензиям.

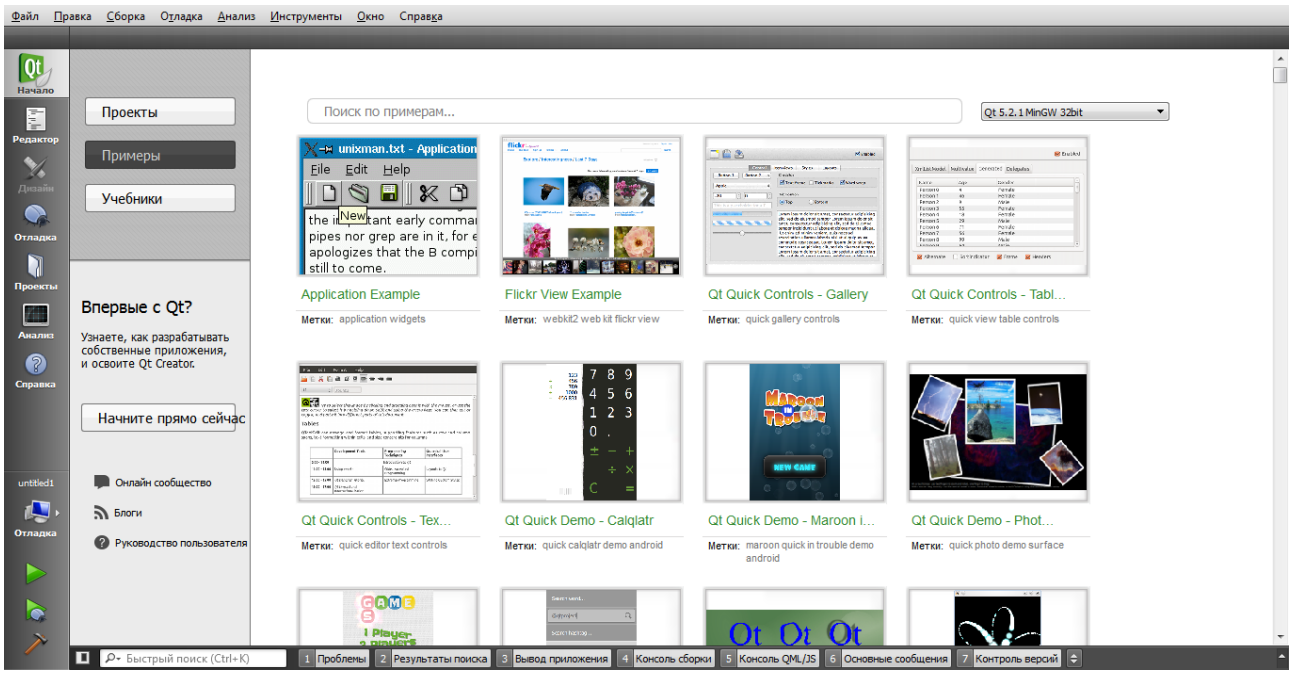

Рис. 6 – Интерфейс программы Qt Assistant в Qt версии 5.0.1 и старше.

7.1. Рекомендуемая литература

1. Qt. Профессиональное программирование на C++: Наиболее полное руководство / М. Шлее. - СПб. : БХВ-Петербург, 2005. – 544 с.
2. Бланшет Ж., Саммерфилд М. Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. – М.: КУДИЦ-ПРЕСС, 2008. – 736 с.

Учебное издание

Семкин Артем Олегович

Шарангович Сергей Николаевич

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание диалоговых окон программы»

Формат 60x84 1/16. Усл. печ. л. _____.
Тираж _____ экз. Заказ_____.
Томскиц государственный университет систем управления и радиоэлектроники.
634050, Томск, пр. Ленина, 40. Тел. (3822) 533018.