МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

Кафедра радиоэлектронных устройств и систем

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ №9-10
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Воронеж 2024

УДК 681.3.06(07)
ББК 32.97я7

Составитель А. И. Сукачев

Информационные технологии: методические указания к выполнению лабораторных работ №9-10 для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост.: А. И. Сукачев. – Воронеж: Изд-во ВГТУ, 2024. – 37 с.

В соответствии с рабочими учебными программами дисциплин приведены описания методов измерений и методик выполнения лабораторных работ, изложены теоретические сведения, лежащие в основе программирования на языке С++. По каждой лабораторной работе в описание включены: цель, основные теоретические сведения, порядок подготовки и проведения работы, перечень положений, которые необходимо отразить в выводах.

Предназначены для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения.

Методические указания подготовлены в электронном виде и содержатся в файле МУ_ИТ_ЛР9-10.pdf.

Ил. 46. Табл. 3. Библиогр.: 3 назв.

УДК 681.3.06(07)
ББК 32.97я7

Рецензент – А. В. Останков, д-р техн. наук, профессор кафедры радиотехники ВГТУ

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

ЛАБОРАТОРНАЯ РАБОТА № 9
РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКОВ QML И C++

1.1. ОБЩИЕ УКАЗАНИЯ

1.1.1. ЦЕЛЬ РАБОТЫ

Получить базовые навыки и умения проектирования программного обеспечения с использованием языков QML и C ++.

1.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является создание приложения на языке QML, поддерживающее взаимодействие с C++. В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание.

1.2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.2.1. ИНТЕГРАЦИЯ QML И C ++

QML разработан так, чтобы его можно было легко дополнять с помощью кода C ++. Классы в модуле Qt QML позволяют загружать объекты QML и манипулировать ими из C ++, а характер интеграции движка QML с системой мета-объектов Qt позволяет вызывать функционал C ++ непосредственно из QML. Это позволяет разрабатывать гибридные приложения, которые реализуются с использованием смеси кода QML, JavaScript и C ++.

Интеграция QML и C ++ предоставляет множество возможностей, включая:

- Отделение кода пользовательского интерфейса от кода логики приложения путём реализации первого на QML и JavaScript в документах QML, а второго на C ++
- Использование и вызов некоторых функций C ++ из QML (например, чтобы вызвать код логики приложения, можно использовать модель данных, реализованную в C ++ или вызвать некоторые функции в сторонней библиотеке C ++)
- Доступ к функциональности в Qt QML или Qt Quick C ++ API (например, для динамической генерации изображений с использованием QQuickImageProvider)
- Реализация своих собственных типов объектов QML из C ++ - как для использования в своем конкретном приложении, так и для распространения среди других

Чтобы предоставить некоторые данные или функциональные возможности C ++ для QML, необходимо получить к ним доступ из класса QObject. Благодаря интеграции механизма QML с системой мета-объектов свойства, методы и сигналы любого класса, унаследованного от QObject, доступны из QML. Ко-

гда требуемый функционал предоставлен таким классом, он может быть представлен в QML различными способами:

- Класс может быть зарегистрирован как инстанцируемый тип QML , так что его можно создавать и использовать как любой обычный тип объекта QML из кода QML
- Класс может быть зарегистрирован как Singleton Type, так что отдельный экземпляр класса может быть импортирован из кода QML, что позволяет получить доступ к свойствам, методам и сигналам экземпляра из QML.
- Экземпляр класса может быть встроен в код QML как свойство контекста или объект контекста, позволяя получить доступ к свойствам, методам и сигналам экземпляра из QML.

Это наиболее распространенные методы доступа к функционалу C ++ из кода QML. Помимо возможности доступа к функционалу C ++ из QML, модуль Qt QML также предоставляет возможность манипулировать объектами QML из кода C ++.

1.2.2. ВЫБОР ПРАВИЛЬНОГО МЕТОДА ИНТЕГРАЦИИ МЕЖДУ C++ И QML

Чтобы быстро определить подходящий метод интеграции, можно использовать следующую блок-схему:

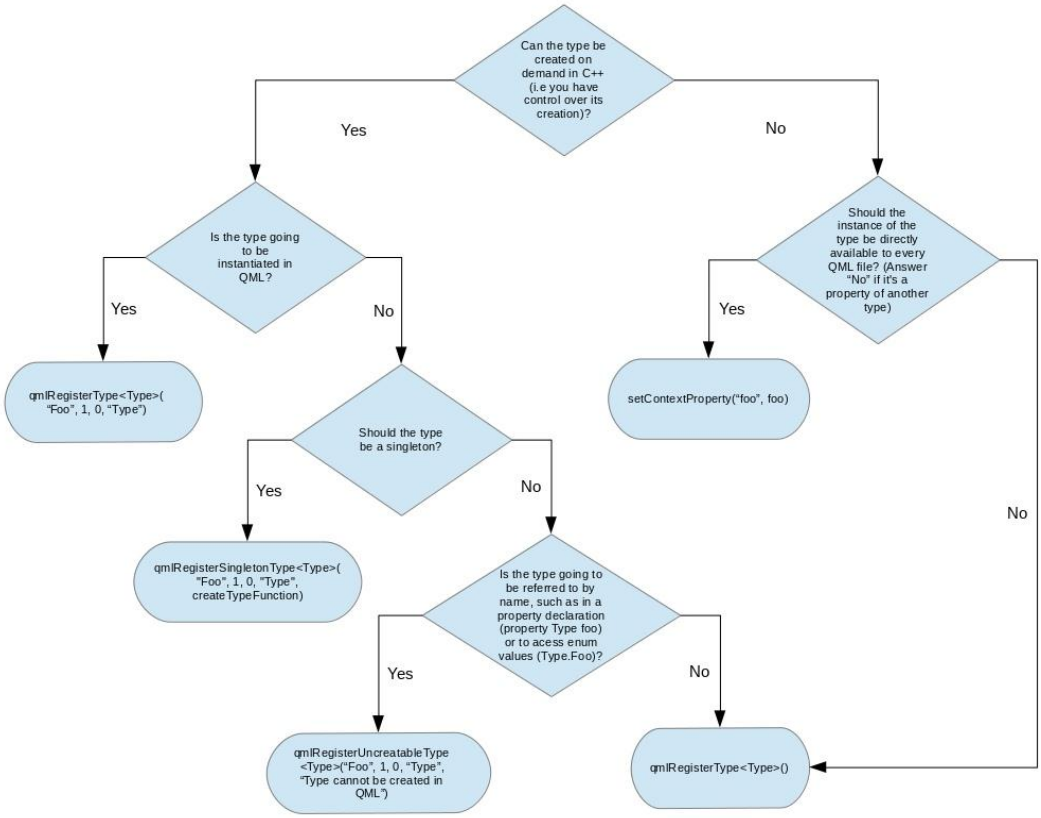

Рис. 1. Блок-схема

1.2.3. ПРЕДОСТАВЛЕНИЕ АТРИБУТОВ КЛАССОВ C ++ В QML

QML можно легко расширить с помощью функционала, определенного в коде C ++. Из-за тесной интеграции механизма QML с мета-объектной системой Qt любые функциональные возможности, которые соответствующим образом предоставляются классом QObject, доступны из кода QML. Это позволяет C ++ данным и функциям быть доступными непосредственно из QML с небольшими изменениями или без них.

Движок QML обладает способностью анализировать экземпляры QObject через систему мета-объектов. Это означает, что любой код QML может получить доступ к следующим элементам класса QObject:

- свойства
- методы (при условии, что они являются открытыми слотами или помечены Q_INVOKABLE)
- сигналы

Любые данные, которые передаются из C++ в QML должны иметь тип, поддерживаемый механизмом QML.

По умолчанию движок поддерживает несколько типов Qt C++ и может автоматически преобразовывать их при использовании из QML. Кроме того, классы C++, которые зарегистрированы в системе типов QML, могут использоваться в качестве типов данных. Кроме того, правила владения данными учитываются при переносе данных из C++ в QML.

Свойство может быть задано для любого класса QObject, используя макрос Q_PROPERTY(). Свойство - это элемент данных класса с обязательной функцией чтения и необязательной функцией записи.

Все свойства класса, полученного из QObject , доступны из QML.

1.2.4. ОПРЕДЕЛЕНИЕ ТИПОВ QML ИЗ C++

При расширении QML с помощью кода C++ класс C++ можно зарегистрировать в системе типов QML, чтобы этот класс можно было использовать в качестве типа данных в коде QML. Хотя свойства, методы и сигналы любого класса, полученного из QObject, доступны из QML, как обсуждалось в разделе «Предоставление атрибутов типов C++ для QML» , такой класс нельзя использовать в качестве типа данных из QML до тех пор, пока он не будет зарегистрирован в системе типов. Кроме того, регистрация может предоставлять другие функции, такие как разрешение использования класса в качестве экземпляра объекта класса QML или возможность импорта и использования одноэлементного экземпляра класса из QML.

Кроме того, модуль Qt QML предоставляет механизмы для реализации специфических для QML функций, таких как вложенные свойства и свойства по умолчанию в C++.

QObject – класс, который может быть зарегистрирован в системе типа QML быть зарегистрирован в системе типа QML для того, чтобы использоваться в качестве типа данных внутри QML кода.

Движок позволяет регистрировать как инстанцируемые, так и не инстанцируемые типы. Регистрация экземпляра типа позволяет использовать класс C++ в качестве определения типа объекта QML, что позволяет использовать его в объявлениях объектов из кода QML для создания объектов этого типа. Регистрация также предоставляет движку дополнительные метаданные типа, позволяя использовать тип (и любые перечисления, объявленные классом) в качестве типа данных для значений свойств, параметров методов и возвращаемых значений, а также параметров сигналов, которыми обмениваются QML и C++.

Таким образом, регистрация неинстанцируемого типа также регистрирует класс как тип данных, но этот тип нельзя использовать в качестве экземпляра типа объекта QML. Это полезно, например, если тип имеет перечисления, которые должны быть представлены QML, но сам тип не должен быть инстанцируемым.

Любой класс C++, полученный из QObject, может быть зарегистрирован как определение типа объекта QML. Как только класс зарегистрирован в системе типов QML, класс может быть объявлен и создан как любой другой тип объекта из кода QML. После создания экземпляром класса можно управлять из QML. Свойства, методы и сигналы любого класса, полученного из QObject, доступны из кода QML.

Чтобы зарегистрировать класс QObject в качестве экземпляра объекта типа QML, который можно создать, вызовите qmlRegisterType(), чтобы зарегистрировать класс как тип QML в пространстве имен определенного типа. Клиенты могут затем импортировать это пространство имен для использования типа.

Например, предположим, что есть класс Message с свойствами author и creationDate:

```
class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
    Q_PROPERTY(QDateTime creationDate READ creationDate WRITE setCreationDate NOTIFY creationDateChanged)
public:
    // ...
};
```

Рис. 2. Пример класса

Этот тип можно зарегистрировать, вызвав qmlRegisterType() с соответствующим пространством имен и номером версии. Например, чтобы сделать тип доступным в пространстве имен com.mycompany.messaging с версией 1.0:

```
qmlRegisterType < Message > ( "com.mycompany.messaging" , 1 , 0 , "Message" );
```

Рис. 3. Код регистрации типа

Тип можно использовать в объявлении объекта из QML, а его свойства можно прочитать и записать, как показано в следующем примере:

```
import com.mycompany.messaging 1.0

Message {
    author: "Amelie"
    creationDate: new Date()
}
```

Рис. 4. Запись типа при объявлении объекта

1.2.5. ВСТРАИВАНИЕ ОБЪЕКТОВ C ++ В QML С ИСПОЛЬЗОВАНИЕМ СВОЙСТВ КОНТЕКСТА

При загрузке объекта QML в приложение C++ может быть полезно встроить некоторые данные C++, которые можно использовать из кода QML. Это позволяет, например, вызывать метод C++ для встроенного объекта или использовать экземпляр объекта C++ в качестве модели данных для представления QML.

Внедрение данных C++ в объект QML стало возможным благодаря классу QQmlContext . Этот класс предоставляет данные контексту объекта QML, чтобы к ним можно было обращаться непосредственно из области действия кода QML.

Например, вот элемент QML, который ссылается на значение currentDateTime, которое не существует в текущей области:

```
// MyItem.qml
import QtQuick 2.0

Text { text: currentDateTime }
```

Рис. 5. Пример ссылки на значение вне области

Это значение currentDateTime может быть установлено непосредственно приложением C++, которое загружает компонент QML, используя QQmlContext::setContextProperty():

```
        QQuickView view;
        view.rootContext()->setContextProperty("currentDateTime",        QDateTime::current-
DateTime());
        view.setSource(QUrl::fromLocalFile("MyItem.qml"));
        view.show();
```

Рис. 6. Установка значения контекстом

Примечание. Поскольку все выражения, оцениваемые в QML, оцениваются в определенном контексте, если контекст изменяется, все привязки в этом контексте будут переоцениваться. Таким образом, свойства контекста следует использовать с осторожностью вне инициализации приложения, так как это может привести к снижению производительности приложения.

1.2.6. ВЗАИМОДЕЙСТВИЕ С ОБЪЕКТАМИ QML ИЗ C ++

Все типы объектов QML являются типами, производными от QObject , независимо от того, реализованы они внутренне или определены сторонними источниками. Это означает, что механизм QML может использовать систему мета-объектов Qt для динамического создания экземпляров любого типа объектов QML и проверки созданных объектов.

Это полезно для создания объектов QML из кода C ++, как для отображения объекта QML, который может быть визуализирован, так и для интеграции невизуальных данных объекта QML в приложение C ++. После создания объекта QML его можно проверить в C ++ для чтения и записи свойств, вызова методов и получения уведомлений о сигналах.

Документ QML можно загрузить с помощью QQmlComponent или QQuickView. QQmlComponent загружает документ QML как объект C ++, который затем можно изменить из кода C ++. QQuickView также делает это, но, поскольку QQuickView является классом QWindow, загруженный объект также будет визуализирован; QQuickView обычно используется для интеграции отображаемого объекта QML в пользовательский интерфейс приложения.

Например, предположим, что есть файл MyItem.qml, который выглядит так:

```
import QtQuick 2.0

Item {
    width: 100; height: 100
}
```

Рис. 7. Содержимое файла MyItem.qml

Этот документ QML может быть загружен с QQmlComponent или QQuickView со следующим кодом C ++. Использование QQmlComponent требует вызова QQmlComponent::create () для создания нового экземпляра компонента, в то время как QQuickView автоматически создает экземпляр компонента, который доступен через QQuickView::rootObject():

```
// Using QQmlComponent
QQmlEngine engine;
QQmlComponent component(&engine,
        QUrl::fromLocalFile("MyItem.qml"));
QObject *object = component.create();
...
delete object;
```

```
// Using QQuickView
QQuickView view;
view.setSource(QUrl::fromLocalFile("MyItem.qml"));
view.show();
QObject *object = view.rootObject();
```

Рис. 8. Загрузка документа при помощи QQmlComponent(слева) и QQuickView(справа)

Этот объект – экземпляр компонента MyItem.qml, который был создан. Теперь вы можете изменить свойства элемента, используя QObject::setProperty() или QQmlProperty::write():

```
object->setProperty("width", 500);
QQmlProperty(object, "width").write(500);
```

Рис. 9. Пример кода для изменения свойств

Разница между QObject::setProperty() и QQmlProperty::write() заключается в том, что последний также удалит привязку в дополнение к установке значения свойства. Например, предположим, что вышеприведенное присваивание width было обязательным для height:

```
width: height
```

Рис. 10. Пример присваивания

Если height из Item изменилось после вызова object->setProperty("width", 500), то width будет обновляться, так как привязка остается активной. Однако, если height изменения после вызова QQmlProperty(object, "width").write(500) width не будут изменены, так как привязка больше не существует.

Кроме того, вы можете привести объект к его фактическому типу и вызвать методы с безопасностью во время компиляции. В этом случае базовым объектом MyItem.qml является Item, который определяется классом QQuickItem:

```
QQuickItem *item = qobject_cast<QQuickItem*>(object);
item->setWidth(500);
```

Рис. 11. Определение базового объекта

Вы также можете подключиться к любым сигналам или вызвать методы, определенные в компоненте, используя QMetaObject::invokeMethod() и QObject::connect().

1.2.7 ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ МЕЖДУ QML И C++

Когда происходит обмен данными между QML и C++, они преобразуются механизмом QML для получения правильных типов данных, подходящих для использования в QML или C++. Это требует, чтобы обмениваемые данные были того типа, который распознается механизмом.

Механизм QML обеспечивает встроенную поддержку большого количества типов данных Qt C++. Кроме того, пользовательские типы C++ могут быть зарегистрированы в системе типов QML, чтобы сделать их доступными для механизма.

Когда данные переносятся из C++ в QML, право собственности на данные всегда остается за C++. Исключением из этого правила является случай, когда QObject возвращается из явного вызова метода C++: в этом случае механизм QML предполагает владение объектом, если явно не установлено, что владение объектом остается в C++.

Кроме того, механизм QML учитывает обычную семантику владения родительского объекта QObject для объектов Qt C++ и никогда не удаляет экземпляр QObject, у которого есть родительский объект.

По умолчанию QML распознает следующие типы данных Qt, которые автоматически преобразуются в соответствующий базовый тип QML при передаче из C++ в QML и наоборот(табл. 1):

Таблица 1

Типы данных

Qt Type	QML Basic Type
bool	bool
unsigned int, int	int
double	double
float, qreal	real
QString	string
QUrl	url
QColor	color
QFont	font
QDateTime	date
QPoint, QPointF	point
QSize, QSizeF	size
QRect, QRectF	rect
QMatrix4x4	matrix4x4
QQuaternion	quaternion
QVector2D, QVector3D, QVector4D	vector2d, vector3d, vector4d
Enums declared with Q_ENUM() or Q_ENUMS()	enumeration

Примечание. Классы, предоставляемые модулем Qt GUI, такие как QColor, QFont, QQuaternion и QMatrix4x4, доступны только в QML, если включен модуль Qt Quick.

Для удобства многие из этих типов могут быть определены в QML строковыми значениями или связанным методом, предоставленным объектом QtQml::Qt. Например, свойство Image::sourceSize имеет тип size, который автоматически преобразуется в тип QSize, и может быть задано строковым значением, отформатированным как «widthxheight», или функции Qt.size():

```
Item {
    Image { sourceSize: "100x200" }
    Image { sourceSize: Qt.size(100, 200) }
}
```

Рис. 12. Определение типа

Механизм QML имеет встроенную поддержку для преобразования ряда типов Qt в связанные типы JavaScript и наоборот при передаче данных между QML и C++. Это позволяет использовать эти типы и получать их в C++ или JavaScript без необходимости реализации пользовательских типов, которые обеспечивают доступ к значениям данных и их атрибутам.

1.2.8. АТРИБУТЫ СИГНАЛА

Сигнал - это уведомление от объекта о том, что произошло какое-то событие: изменилось свойство, была запущена или остановлена анимация, было загружено изображение. Например, тип MouseArea имеет сигнал clicked, который посылается, когда пользователь нажимает в пределах области мыши.

Объект может быть уведомлен обработчиком сигнала всякий раз, когда излучается конкретный сигнал. Обработчик сигнала объявляется с синтаксисом <Signal>, где <Signal> - имя сигнала с первой заглавной буквой. Обработчик сигнала должен быть объявлен в определении объекта, который излучает сигнал, и обработчик должен содержать блок кода JavaScript, который будет выполняться при вызове обработчика сигнала.

Например, описанный ниже обработчик сигнала onClicked объявляется в определении объекта MouseArea и вызывается при нажатии MouseArea, в результате чего выводится консольное сообщение:

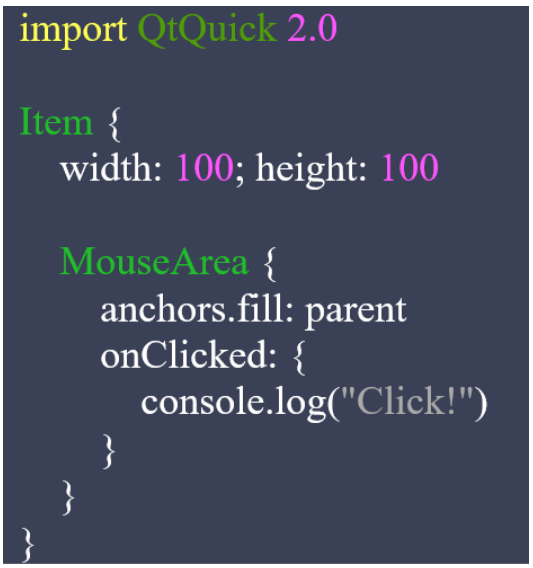

Рис. 13. Определение сигнала

Сигнал может быть определен для типа в C++ путем регистрации Q_SIGNAL класса, который затем регистрируется в системе типов QML. В качестве альтернативы, пользовательский сигнал для типа объекта может быть определен в объявлении объекта в документе QML со следующим синтаксисом:

```
signal <signalName>[([<type> <parameter name>[, ...]])]
```

Рис. 14. Определение сигнала в QML

Попытка объявить два сигнала или метода с одинаковым именем в одном блоке типа является ошибкой. Однако новый сигнал может повторно использовать имя существующего сигнала в типе. (Это следует делать с осторожностью, поскольку существующий сигнал может быть скрыт и стать недоступным.)

Вот три примера объявлений сигналов:

```
import QtQuick 2.0

Item {
    signal clicked
    signal hovered()
    signal actionPerformed(string action, var actionResult)
}
```

Рис. 15. Примеры объявления сигналов

Если сигнал не имеет параметров, скобки "()" являются необязательными. Если используются параметры, типы параметров должны быть объявлены, как аргументы string и var сигнала actionPerformed выше.

Чтобы испустить сигнал, вызовите его как метод. Любые соответствующие обработчики сигналов будут вызываться при испускании сигнала. Обработчики могут использовать определенные имена аргументов сигнала для доступа к соответствующим аргументам.

1.3. ЛАБОРАТОРНЫЕ ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ

1. Открыть проект, созданный в лабораторной работе №2

2. Добавить в интерфейс объект ListView, заменив один из объектов исходного проекта

```
19      ListView {
20          id: lvw
21          x: 20
22          y: 40
23          height: parent.height -60
24          width: parent.width -20
25          delegate: Rectangle {height: 100
26              width: 200
27              color: "#00ff00"
28              Label {
29                  text: edit
30                  anchors.centerIn: parent
31              }
32          }
33          spacing: 10
34      }
35  }
36
```

Рис. 16. Добавление объекта ListView

3. Добавить в проект новый файл QML с названием MyWindow

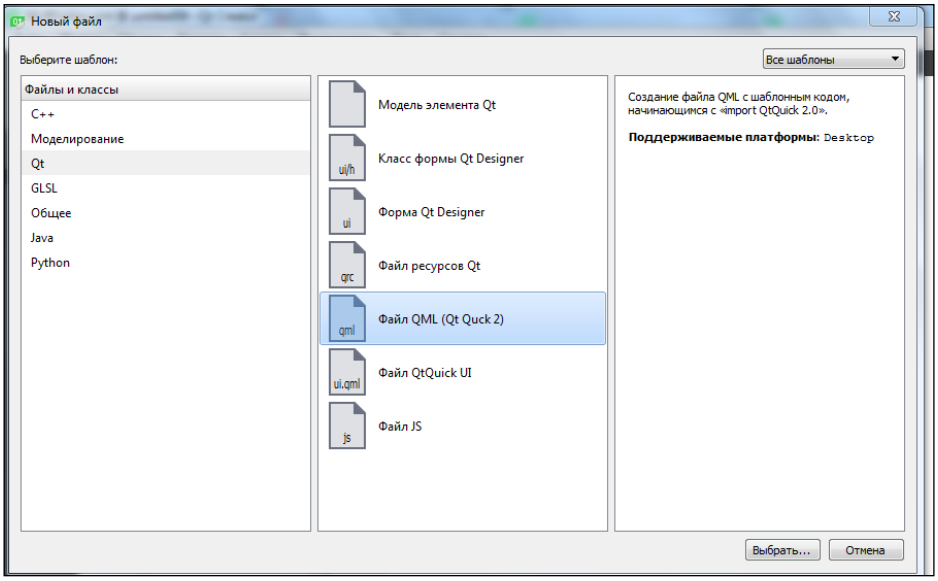

Рис. 17. Настройка проекта

В созданном файле создать поле для ввода текста с id: txt.

Создать сигнал create с аргументом str (строка 10).

Создать слот onClicked (строки 20-23). Теперь при нажатии на кнопку Ок испускается сигнал create с текстом, введённым в поле, а затем закрывается окно *wnd*.

Рис. 18. Файл MyWindow.qml

5. Создать в файле main.qml объект нового окна MyWindow (рис. 19, строки 35-37)

6. В свойствах кнопки «+» создать слот onClicked (рис. 19, строка 15)

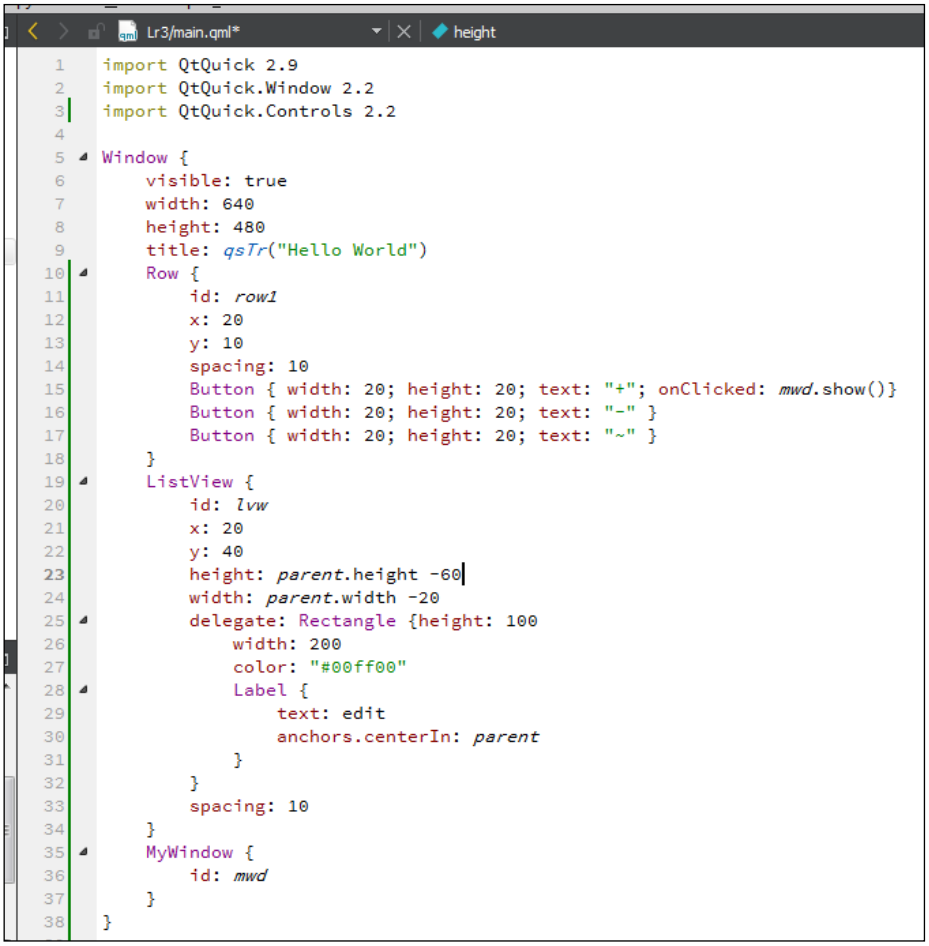

Рис. 19. Файл main.qml

7. Добавить в проект новый класс C++, задав базовый класс QObject

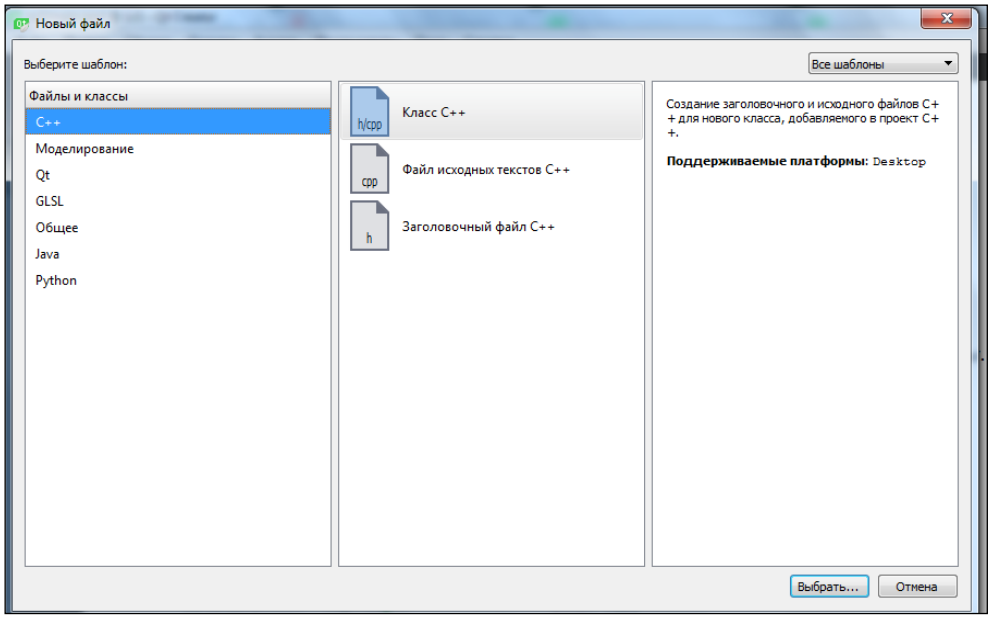

Рис. 20. Настройка класса

8. Создать свою модель данных

- Подключить библиотеки QObject и QStandardItemModel
- Объявить свойства объекта класса QStandardItemModel
- Создать с ключом private объект класса QStandardItemModel
- Создать методы getMyModel и setMyModel сключом public, а также метод appenditem
- Создать сигнал myModelChanged, который будет сообщать о том, что модель изменилась


```
#ifndef WORKER_H
#define WORKER_H

#include <QObject>
#include <QStandardItemModel>

class worker : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QStandardItemModel*myModel
               READ getMyModel
               WRITE setMyModel
               NOTIFY myModelChanged)
public:
    explicit worker(QObject *parent = nullptr);
    QStandardItemModel* getMyModel() {
        return myModel;
    }
    void setMyModel(QStandardItemModel* m) {
        myModel = m;
    }
    Q_INVOKABLE void appenditem(QString);

private:
    QStandardItemModel* myModel;

signals:
    void myModelChanged();
public slots:
};

#endif // WORKER_H
```

Рис. 21. Файл worker.h

9. Перейти к реализации объявленных методов

- В конструкторе класса создать модель
- Вставить в неё несколько значений

- Реализовать слот appenditem. При вызове этого метода будет осуществляться вставка в модель новой строки с текстом, введённым в пользователем в поле ввода и испускаться сигнал myModelChanged

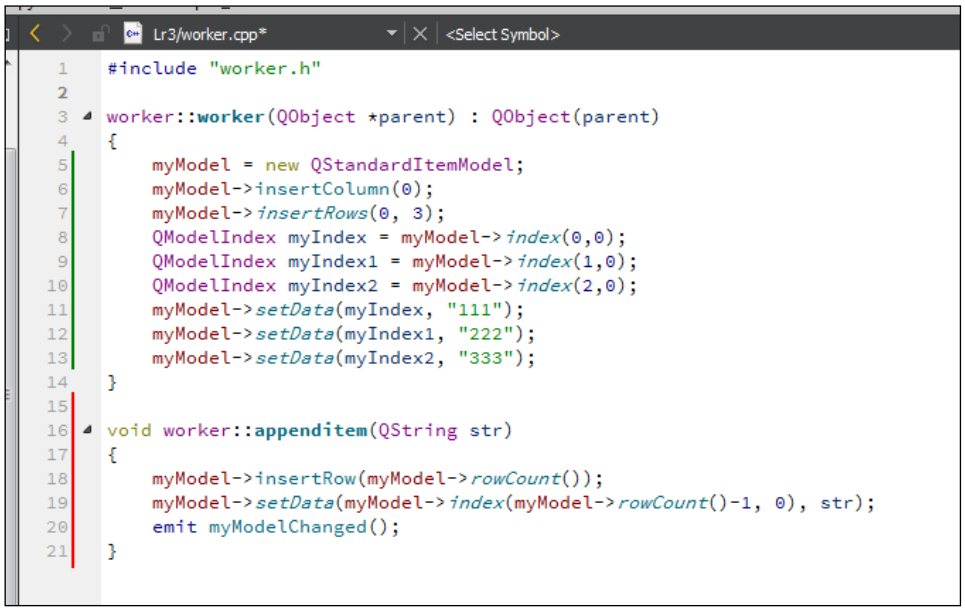

Рис. 22. Файл worker.cpp

10. Создать в файле main.qml объект класса worker (рис. 18, строки 11-13).

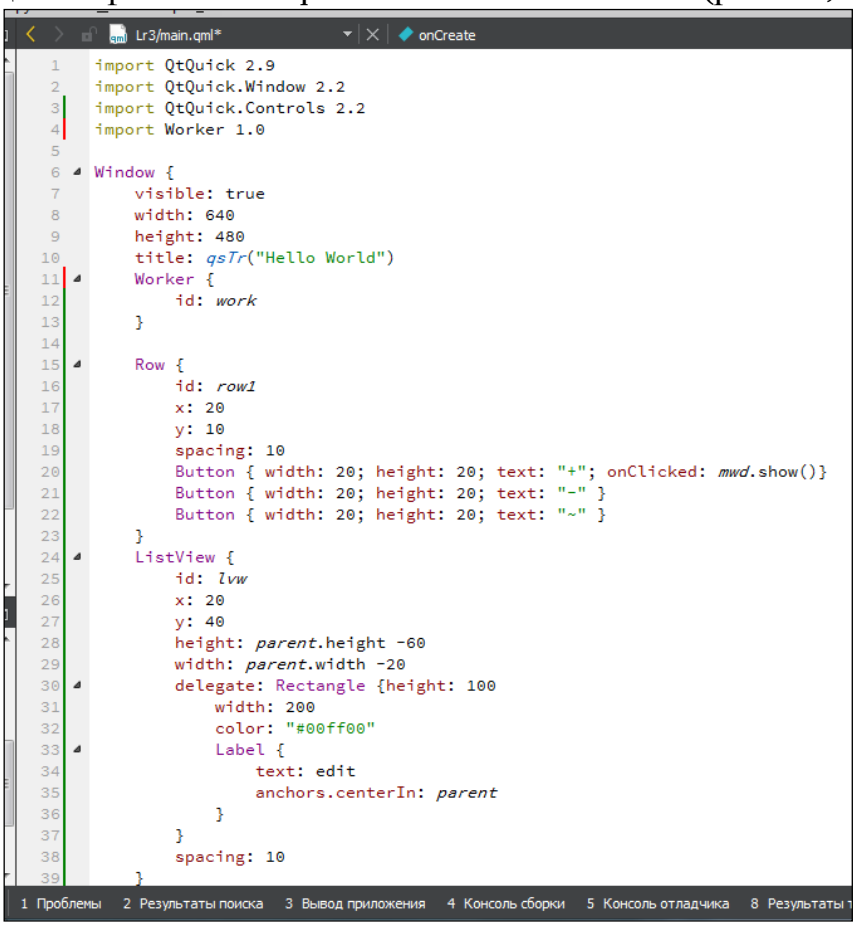

Рис. 23. Файл main.qml

11. В свойствах объекта MyWindow создать слот onCreate
12. В созданном слоте вызвать метод appenditem класса worker

```
    }
    MyWindow {
        id: mwd
        onCreate: {
            work.appenditem(str)
        }
    }
}

```

1 Проблемы 2 Результаты поиска 3 Вывод приложения 4 Консоль

Рис. 24. Файл main.qml (продолжение)

13. При запуске приложения на экране будет появляться представление созданной вами модели

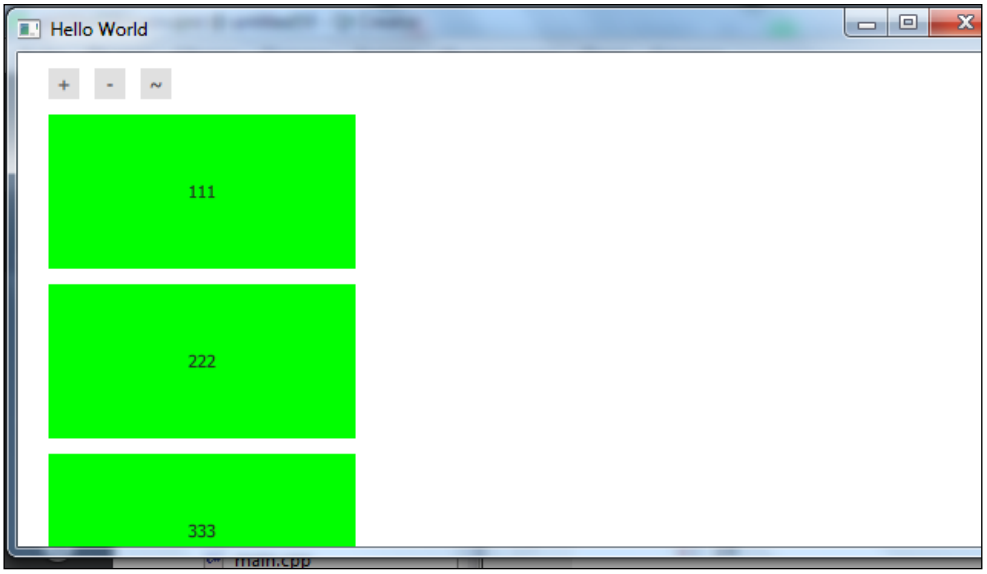

Рис. 25. Представление заданной модели

14. При нажатии на клавишу «+» открывается окно для ввода текста

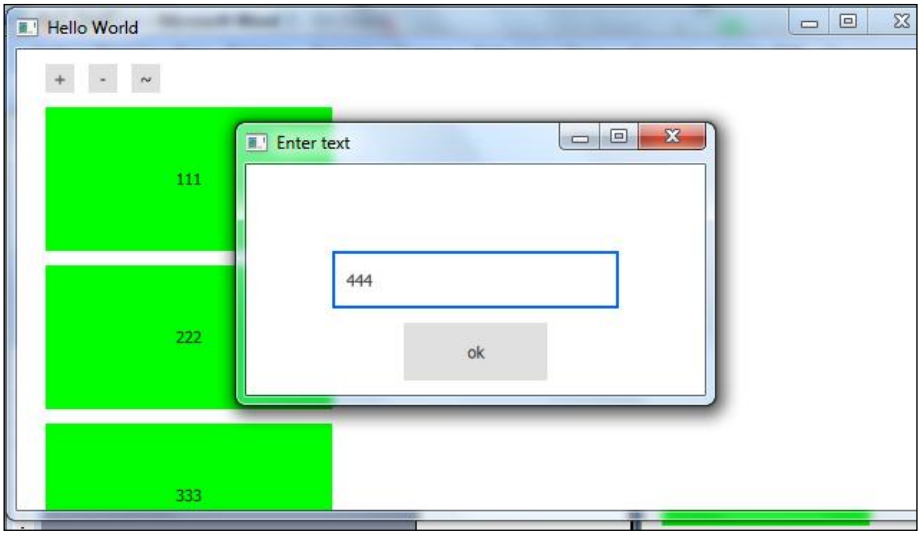

Рис. 26. Окно ввода текста

15. При нажатии на кнопку «Ок» в представлении появляется новый ряд

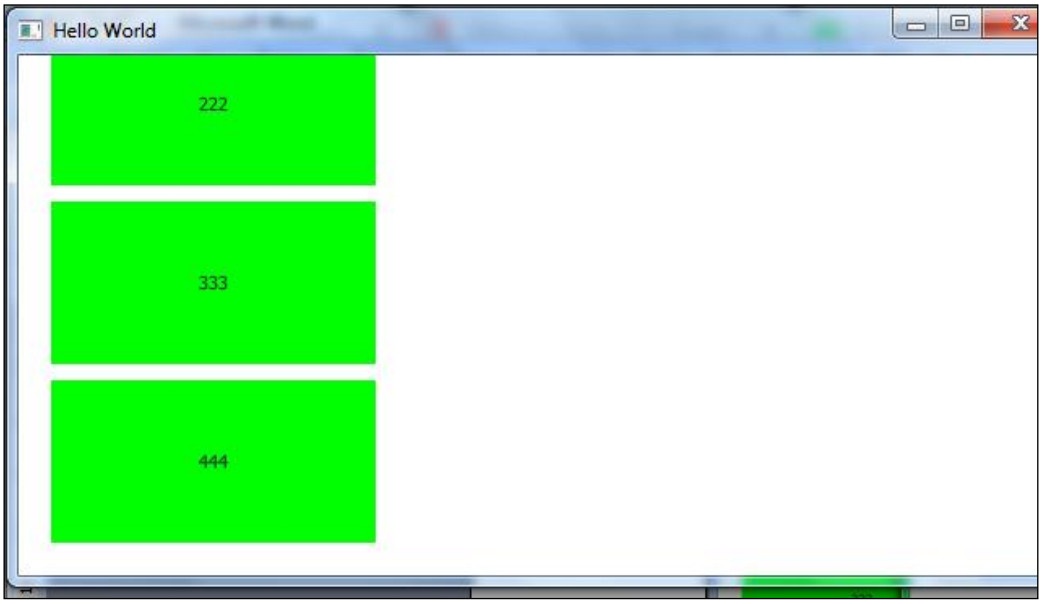

Рис. 27. Добавленный новый ряд

1.4. ВАРИАНТЫ ЛАБОРАТОРНЫХ ЗАДАНИЙ

Реализовать в своём варианте интерфейса:
Варианты 1, 4, 7, 10 – кнопку добавления записей
Варианты 2, 5, 8 – кнопку удаления записей
Варианты 3, 6, 9 – кнопку редактирования записей

1.5. ДОМАШНЕЕ ЗАДАНИЕ

Реализовать две оставшиеся кнопки.

1.6. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать название и цель работы, краткие теоретические сведения, а также код с комментариями. В конце отчета должен быть вывод.

1.7. КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНЫМ ЗАДАНИЯМ

1. Какие возможности предоставляет интеграция QML и C ++?
2. Какой класс C++ может быть зарегистрирован в системе типа QML?
3. Какой макрос нужно использовать для задания свойств класса QObject?
4. Приведите пример объявления сигнала
5. Приведите пример обработчика сигнала

2. ЛАБОРАТОРНАЯ РАБОТА № 10
ПРОЦЕССЫ И ПОТОКИ. РАБОТА С ПОТОКАМИ. ОДНОПОТОЧНОСТЬ И МНОГОПОТОЧНОСТЬ

2.1. ОБЩИЕ УКАЗАНИЯ

2.1.1. ЦЕЛЬ РАБОТЫ

Познакомиться с понятиями процесс, поток, однопоточность, многопоточность. На основе полученных знаний выполнить индивидуальное лабораторное задание.

2.1.2 ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является создание многопоточного приложения, с использованием класса QThread. В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание по проектированию приложения.

2.2 ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.2.1 ОБЩИЕ СВЕДЕНИЕ

В современных компьютерах одновременно выполняется множество программ. Одни из них запускаются при старте системы, другие – пользователем, а третьи – другими программами.

Но и многие приложения сами по себе часто выполняют несколько действий одновременно. Для организации подобного поведения служат *потоки*.

Процессы

В том случае, когда пользователь или программа выполняют запуск другой программы, операционная система всегда создаёт новый процесс. *Процесс* – это экземпляр программы, загруженной для выполнения в память компьютера.

По своей сути, процессы – это независимые друг от друга программы, обладающие своими собственными данными. Коротко процесс можно охарактеризовать как совокупность кода, данных и ресурсов, необходимых для его работы. Под ресурсами подразумеваются объекты, запрашиваемые и используемые процессами в период их работы.

Создание процесса может оказаться полезным для использования функциональных возможностей программ, не имеющих графического интерфейса и

работающих с командной строкой. Особенно это имеет смысл при запуске команд или программ, время работы которых не продолжительно.

Потоки

Потоки становятся всё более популярным средством программирования. Для управления потоком Qt предоставляет класс QThread.

Поток – это независимая задача, которая выполняется внутри процесса и разделяет с ним общее адресное пространство. Код и глобальные данные.

Процесс сам по себе не исполняется, поэтому для выполнения программного кода он должен иметь хотя бы один поток. Конечно, можно создавать и более одного потока. Вновь созданные потоки начинают выполняться сразу же, параллельно с главным потоком, при этом их количество может изменяться – одни создаются, другие завершаются. Завершение основного потока приводит к завершению процесса, независимо от того, существуют другие потоки или нет. Создание нескольких потоков в процессе получило название многопоточность.

Многопоточность требуется для выполнения действий в фоновом режиме, параллельно с действиями основной программы, и позволяет разбить выполнение программы на параллельные потоки, которые могут быть абсолютно независимы друг от друга. А если приложение выполняется на компьютере с несколькими процессорами, то разделение на потоки может значительно ускорить работу всей программы. Используя многопоточностьможно решить эту проблему, запустив подобные операции в отдельно созданных потоках. Тем самым при зависании одного из потоков функционирование основной программы не будет нарушено.

2.2.2. КЛАСС QTHREAD

Класс QThread предоставляет независимый от платформы способ управления потоками.

Таблица 2

Описание класса QThread

Заголовок:	#include <QThread>
QMAKE:	QT + = ядро
Наследуется:	QObject

Объект QThread управляет одним потоком управления в программе. QThreads начинает выполняться в run(). По умолчанию run() запускает цикл обработки событий, вызывая exec() и запускает цикл обработки событий Qt внутри потока.

Вы можете использовать рабочие объекты, перемещая их в поток с помощью QObject::moveToThread().

Код внутри рабочего слота будет выполняться в отдельном потоке. Однако вы можете подключить рабочие слоты к любому сигналу, от любого объекта,

в любом потоке. Это безопасно для подключения сигналов и слотов через различные потоки, благодаря механизму, называемому queued connections (рис. 28).

```
class Worker : public QObject
{
    Q_OBJECT

public slots:
    void doWork(const QString &parameter) {
        QString result;
        /* ... here is the expensive or blocking operation ... */
        emit resultReady(result);
    }

signals:
    void resultReady(const QString &result);
};

class Controller : public QObject
{
    Q_OBJECT
    QThread workerThread;
public:
    Controller() {
        Worker *worker = new Worker;
        worker->moveToThread(&workerThread);
        connect(&workerThread, &QThread::finished, worker, &QObject::deleteLater);
        connect(this, &Controller::operate, worker, &Worker::doWork);
        connect(worker, &Worker::resultReady, this, &Controller::handleResults);
        workerThread.start();
    }
    ~Controller() {
        workerThread.quit();
        workerThread.wait();
    }
public slots:
    void handleResults(const QString &);
signals:
    void operate(const QString &);
};
```

Рис. 28. Пример реализации многопоточности при помощи queued connections

Другой способ заставить код выполняться в отдельном потоке, это создать подкласс QThread и переопределить run() (рис. 29). Например:

```
class WorkerThread : public QThread
{
    Q_OBJECT
    void run() override {
        QString result;
        /* ... here is the expensive or blocking operation ... */
        emit resultReady(result);
    }
signals:
    void resultReady(const QString &s);
};

void MyObject::startWorkInAThread()
{
    WorkerThread *workerThread = new WorkerThread(this);
    connect(workerThread, &WorkerThread::resultReady, this, &MyObject::handleResults);
    connect(workerThread, &WorkerThread::finished, workerThread, &QObject::deleteLater);
    workerThread->start();
}
```

Рис. 29. Пример реализации многопоточности при помощи переопределения run()

В этом примере поток завершит работу после возврата из функции run. В потоке не будет никакого цикла обработки событий, если вы не вызовете exec ().

Важно помнить, что экземпляр QThread живет в старом потоке, который его создал, а не в новом потоке, который вызывает run (). Это означает, что все очереди и вызываемые методы QThread будут выполняться в старом потоке. Таким образом, разработчик, желающий вызывать слоты в новом потоке, должен использовать подход рабочий объект; новые слоты не должны быть реализованы непосредственно в подклассе QThread.

2.2.3. ДОКУМЕНТАЦИЯ ПО ФУНКЦИЯМ

QThread :: QThread (QObject * parent = nullptr)

Создает новый QThread для управления новым потоком. Родитель становится владельцем QThread. Поток не начинает выполняться, пока не будет вызвана start ().

[signal] void QThread :: terminate ()

Этот сигнал испускается из связанного потока непосредственно перед его завершением.

Когда этот сигнал испускается, цикл событий уже остановлен. Больше событий не будет обрабатываться в потоке, кроме отложенных событий удаления. Этот сигнал может быть подключен к QObject :: deleteLater (), чтобы освободить объекты в этом потоке.

Примечание. Если связанный поток был прерван с помощью terminate (), то из какого потока выдается этот сигнал, не определено.

Примечание: это частный сигнал. Он может использоваться в сигнальных соединениях, но не может излучаться пользователем.

[slot] void QThread :: quit ()

Сообщает о выходе из цикла потока с кодом возврата 0 (успех). Эквивалентно вызову QThread :: exit (0).

Эта функция ничего не делает, если у потока нет цикла обработки событий.

[slot] void QThread :: start (QThread :: Priority priority = InheritPriority)

Начинает выполнение потока с вызова run (). Операционная система будет планировать поток в соответствии с параметром приоритета. Если поток уже запущен, эта функция ничего не делает.

Влияние параметра priority зависит от политики планирования операционной системы. В частности, приоритет будет игнорироваться в системах, которые не поддерживают приоритеты потоков.

void QThread :: start ()

Этот сигнал испускается из связанного потока, когда он начинает выполняться, до вызова функции run ().

Примечание: это частный сигнал. Он может использоваться в сигнальных соединениях, но не может излучаться пользователем.

[slot] void QThread :: terminate ()

Завершает выполнение потока. Поток может быть или не быть немедленно прекращен, в зависимости от политик планирования операционной системы. Используйте QThread :: wait () после terminate (), чтобы быть уверенным.

Когда поток завершается, все потоки, ожидающие завершения потока, будут разбужены.

Завершение может быть явно включено или отключено путем вызова QThread :: setTerminationEnabled (). Вызов этой функции, когда завершение отключено, приводит к тому, что завершение откладывается до повторного включения завершения.

[virtual] QThread :: ~ QThread ()

Уничтожает QThread. Обратите внимание, что удаление объекта QThread не остановит выполнение потока, которым он управляет. Удаление запущенного QThread (т.е. возвращает isFinished () false) приведет к сбою программы. Дождитесь сигнала завершения () перед удалением QThread.

[static] QThread * QThread :: create (Функция && f , Args && ... args)

Создает новый объект QThread, который будет выполнять функцию f с аргументами args.

Новый поток не запущен - он должен быть запущен явным вызовом start(). Это позволяет вам подключаться к его сигналам, перемещать объекты QObject в поток, выбирать приоритет нового потока и так далее. Функция f будет вызываться в новом потоке.

Возвращает вновь созданный экземпляр QThread.

Примечание: вызывающая сторона получает право собственности на возвращенный экземпляр QThread.

Предупреждение: не вызывайте start () для возвращенного экземпляра QThread более одного раза; это приведет к неопределенному поведению.

[static] QThread * QThread :: create (Function && f)

Создает новый объект QThread, который будет выполнять функцию f.

Новый поток не запущен - он должен быть запущен явным вызовом start (). Это позволяет вам подключаться к его сигналам, перемещать объекты QObject в поток, выбирать приоритет нового потока и так далее. Функция f будет вызываться в новом потоке.

Возвращает вновь созданный экземпляр QThread.

Примечание: вызывающая сторона получает право собственности на возвращенный экземпляр QThread.

Предупреждение: не вызывайте start () для возвращенного экземпляра QThread более одного раза; это приведет к неопределенному поведению.

[static] QThread * QThread :: currentThread ()

Возвращает указатель на QThread, который управляет текущим выполняющимся потоком.

[static] Qt :: HANDLE QThread :: currentThreadId ()

Возвращает дескриптор потока текущего выполняющегося потока.

Предупреждение: дескриптор, возвращаемый этой функцией, используется для внутренних целей и не должен использоваться ни в одном коде приложения.

[override virtual]bool QThread :: event (QEvent * event)

Переопределения: QObject :: event (QEvent * e).

QAbstractEventDispatcher * QThread :: eventDispatcher () const

Возвращает указатель на объект диспетчера событий для потока. Если для потока не существует диспетчера событий, эта функция возвращает nullptr.

[protected] int QThread :: exec ()

Входит в цикл обработки событий и ожидает вызова метода exit (), возвращая значение, переданное функции exit (). Возвращаемое значение равно 0, если exit () вызывается через quit ().

Эта функция предназначена для вызова изнутри run (). Необходимо вызвать эту функцию, чтобы начать обработку событий.

void QThread :: exit (int returnCode = 0)

Сообщает циклу обработки потока о выходе с кодом возврата.

После вызова этой функции поток покидает цикл обработки событий и возвращается из вызова в QEventLoop :: exec (). Функция QEventLoop :: exec () возвращает returnCode .

По соглашению returnCode 0 означает успех, любое ненулевое значение указывает на ошибку.

Обратите внимание, что в отличие от библиотечной функции C с таким же названием, эта функция делает возврат к абоненту – это обработка события, которое останавливается.

Никакие QEventLoops больше не будут запущены в этом потоке, пока QThread :: exec () не будет вызван снова. Если eventloop в QThread :: exec () не запущен, то следующий вызов QThread :: exec () также немедленно вернется.

[static] int QThread :: idealThreadCount ()

Возвращает идеальное количество потоков, которые могут быть запущены в системе. Это делается путем запроса количества процессорных ядер, как реальных, так и логических, в системе. Эта функция возвращает 1, если число ядер процессора не может быть обнаружено.

bool QThread :: isFinished () const

Возвращает, trueесли поток закончен; в противном случае возвращается false.

bool QThread :: isInterruptionRequested () const

Верните true, если задача, выполняемая в этом потоке, должна быть остановлена. Прерывание может быть запрошено функцией requestInterruption ().

Эта функция может использоваться, чтобы сделать непрерывно выполняемые задачи полностью прерываемыми. Никогда не проверяйте и не изменяйте значение, возвращаемое этой функцией, безопасно, однако рекомендуется делать это регулярно в долго выполняющихся функциях (рис. 30).

```
void long_task() {() {
    forever {{
      if ( ( QThread::currentThread()()->isInterruptionRequested() ) {
         () ) {
         return;
      }}
   }}
}}
```

Рис. 30. Пример использования функции

2.3 ПРИМЕР ЛАБОРАТОРНЫХ ЗАДАНИЙ С МЕТОДИЧЕСКИМИ УКАЗАНИЯМИ ПО ИХ ВЫПОЛНЕНИЮ

1. Запускаем Qt Creator, выбираем пункт меню Файл –> Новый файл или проект. В открывшемся окне Приложение –> Приложение Qt Widgets, далее кнопка Выбрать (рис. 31).

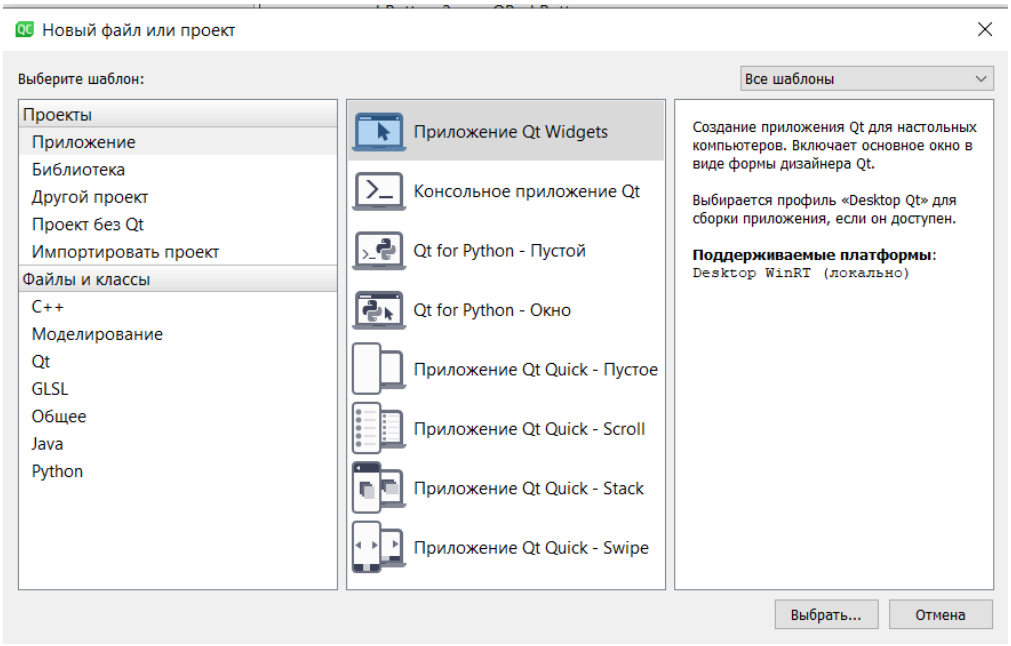

Рис. 31. Настройки проекта

2. Задаём имя и проекта и путь к нему (рис. 32).

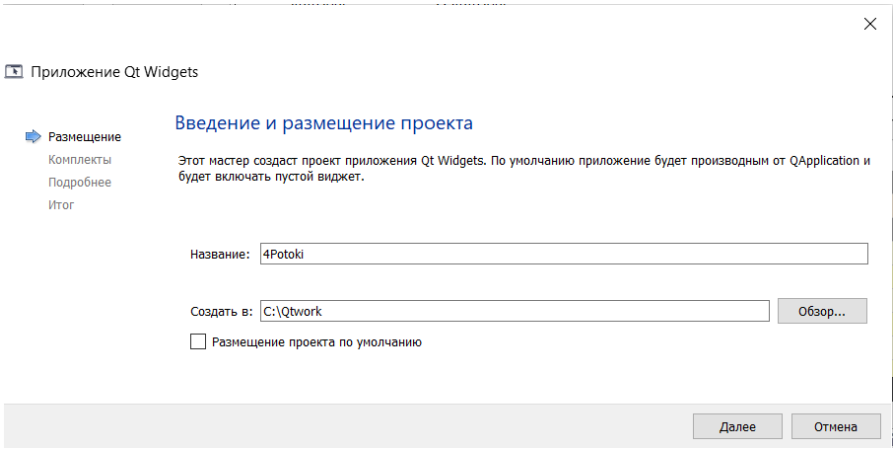

Рис. 32. Задание имени проекта и пути к нему

3. Выбираем комплекты (рис. 33):

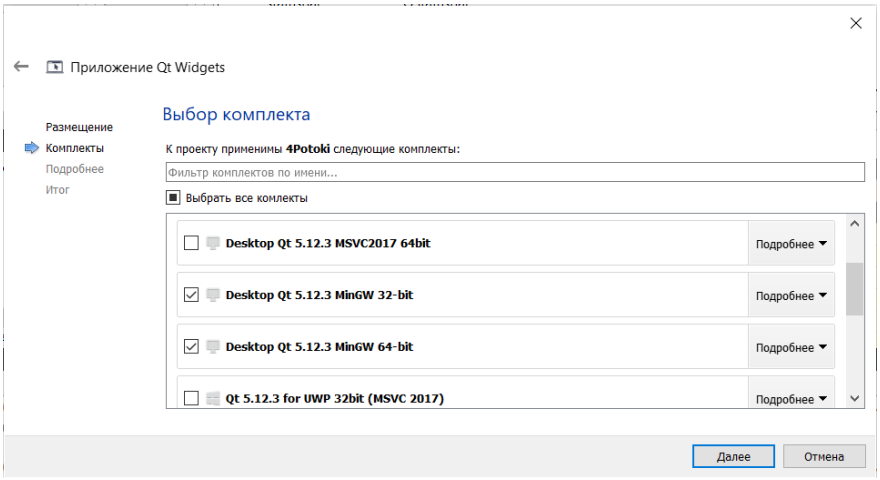

Рис. 33. Выбор комплектов

4. В качестве базового класса задаём QMainWindow (рис. 34).

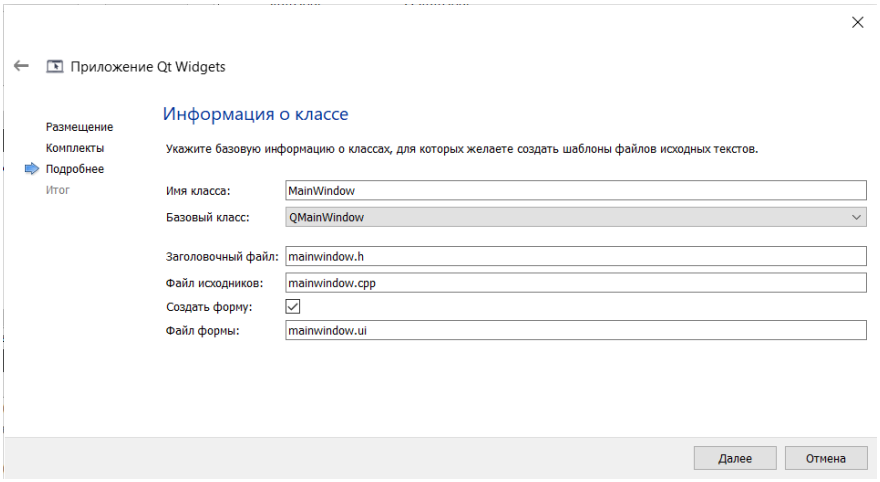

Рис. 34. Задание базового класса

5. Завершаем создание проекта (рис. 35).

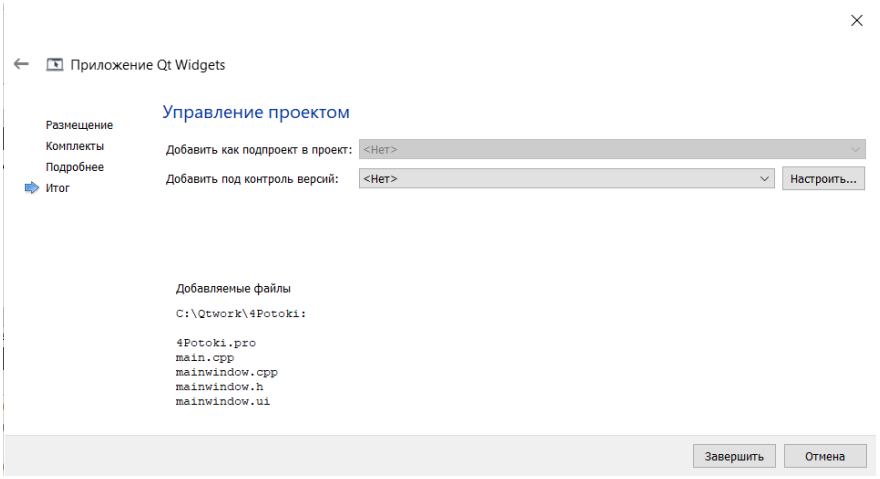

Рис. 35. Завершение создания проекта

6. Сразу же добавим новый класс: нажимаем правой кнопкой мыши на проект, выбирем «Добавить новый…». В «Файлах и классах» выбираем C++ – класс C++ (рис. 36).

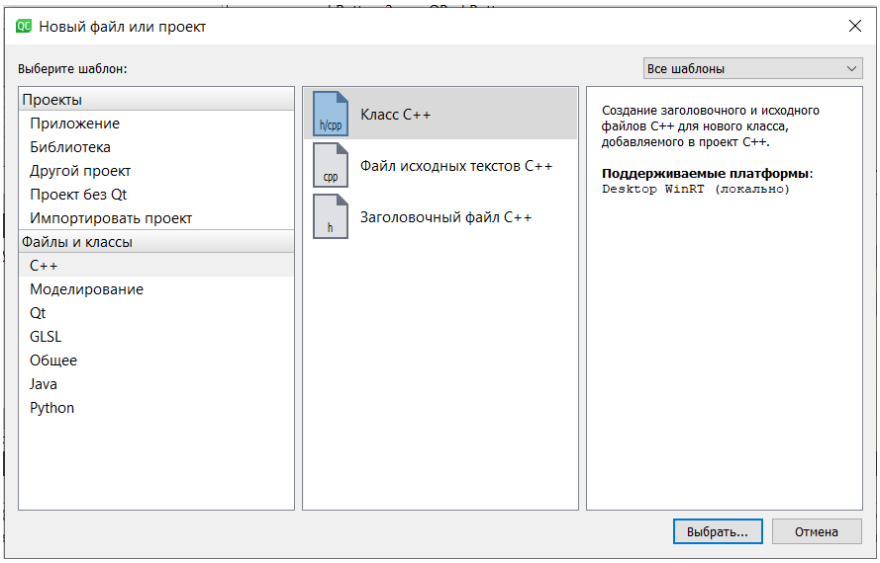

Рис. 36. Добавление нового класса

7. В качестве имени класса прописываем: «mythread». Базовый класс – QWidget (рис. 37).

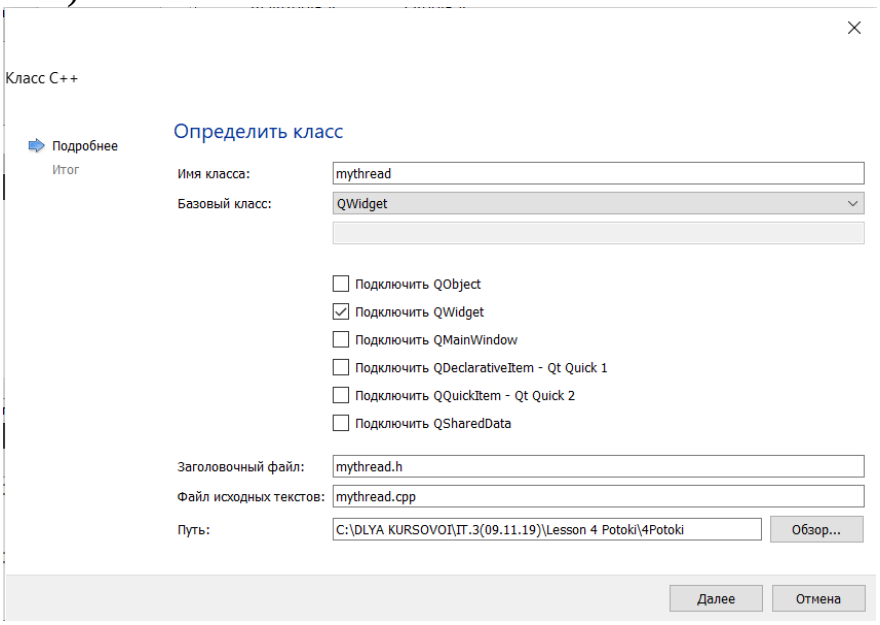

Рис. 37. Настройка нового класса

8. Завершаем создание нового класса (рис. 38).

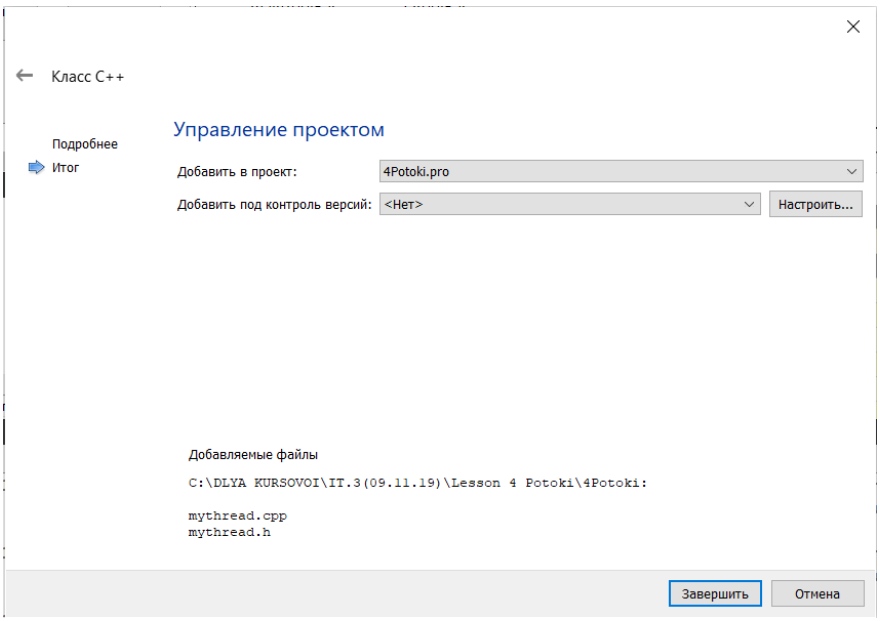

Рис. 39. Завершение создания нового класса

9. В режиме «Дизайн» создаём следующий интерфес с помощью объектов классов QLable и QPushButton (рис. 40).

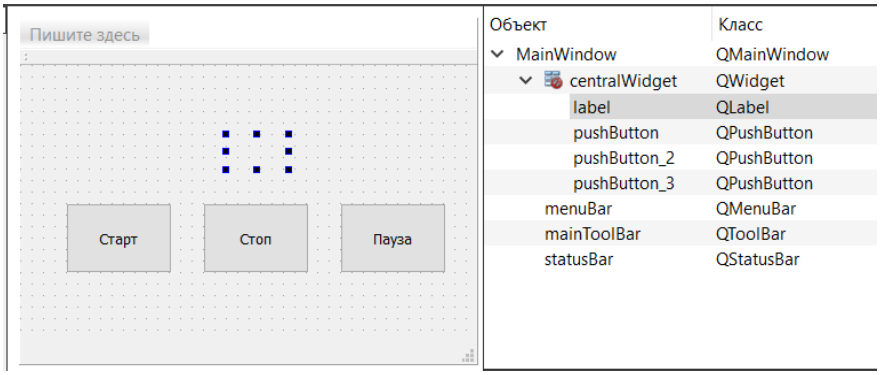

Рис. 40. Создание интерфейса

10. В файле *mainwindow.h* проекта 4Potoki объявить слоты on_pushButton_clicked(), on_pushButton_2_clicked(), on_pushButton_3_clicked() (рис. 41) и подключить библиотеку класса QThread, которая понадобится для их реализации. Далее с ключом доступа private создадим объект данного класса – mythread worker. Макросы QT_BEGIN_NAMESPACE и QT_END_NAMESPACE, которые вы можете переопределить, если хотите создать Qt в определенном пространстве имен (рис. 41).

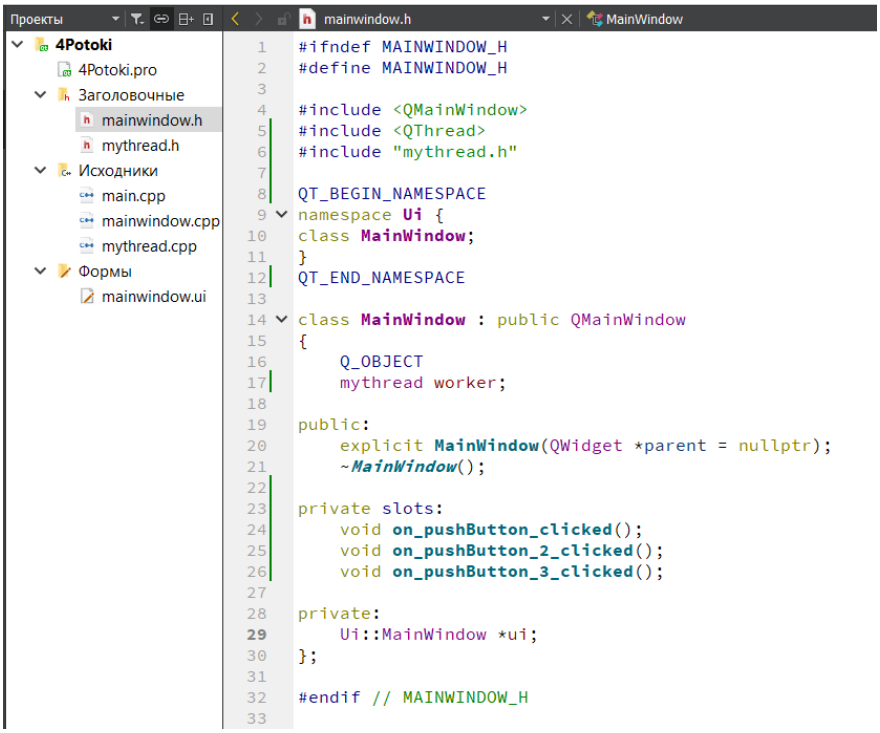

Рис. 41. Листинг файла *mainwindow.h*

11. В файле mythread.h создаём свой класс mythread, который будет отвечать за обработку отдельного потока. В качестве базового класса будем использовать QThread. В этом классе присутствует защищённый метод (protected) – run(), который будет формировать поток. Всё, что описано в этом методе, и будет запускаться в потоке. Создадим свой сигнал void sundering(), в том потоке, который мы будем отслеживать. Изменим сигнал, для того, чтобы QLabel мог принимать в себя его значения – QString, вместо int: void sundering(QString). Создадим публичные методы void stop() и void pause(). Создаём объекты булевых значений: bool pauseFlag, bool stopFlag, current. Затем прописываем публичные слоты void stop() и void pause() (рис. 42).

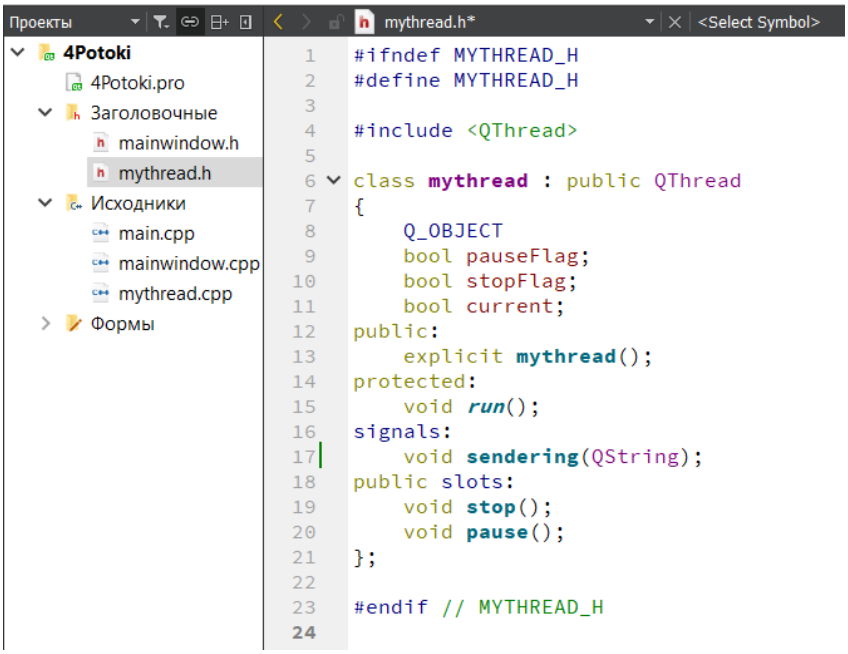

Рис. 42. Листинг файла *mythread.h*

12. Для того, чтобы соединить сигналы и слоты, в конструкторе файла mainwindow.cpp пишем connect. Теперь наш объект worker через слот связался с QLabel.

В слоте on_pushButton_clicked() прописыаем публичный метод worker.start(), который инициализирует создание потока и выполнение кода метода run().

В слоте on_pushButton_2_clicked() прописыаем публичный метод worker.stop(), который инициализирует создание потока и выполнение кода метода run().

В слоте on_pushButton_3_clicked() прописыаем публичный метод worker.pause(), который инициализирует создание потока и выполнение кода метода run().

Далее в конструкторе и слотах on_pushButton_clicked(), on_pushButton_2_clicked() с помощью свойства Enabled, которое определяет, включён ли виджет, устанавливаем включение и отключение кнопок (рис. 43).

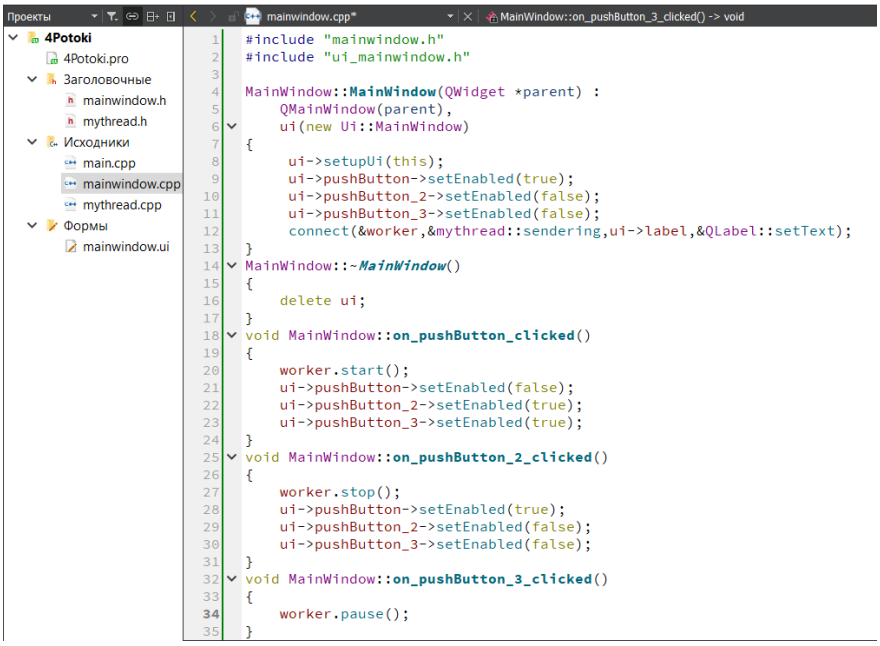

Рис. 43. Листинг файла *mainwindow.cpp*

13. Перейдём к реализации метода run(). Установим по умолчанию значения: current = 0, pauseFlag = false, stopFlag = false. В цикле for с помощью слова emit мы будем формировать сигнал. Метод sleep() заставляет текущий поток «спать» в течение нескольких секунд. В методах stop() и pause() происходит переопределение булевых значений при нажатии на соответствующую кнопку – на true и !pauseFlag соответственно (рис. 44).

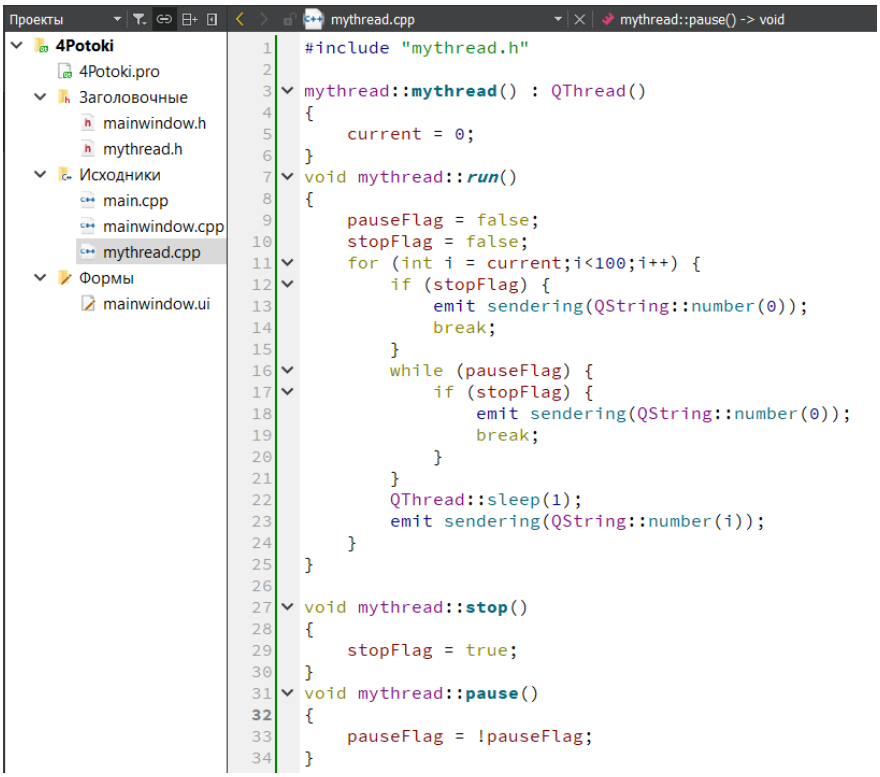

Рис. 44. Листинг файла *mythread.cpp*

14. Выполнение работы программы показано на рис. 45:

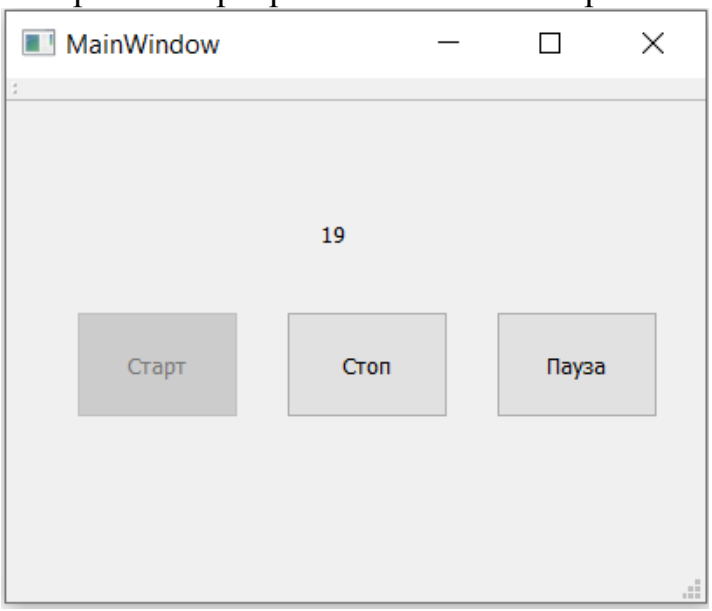

Рис. 46. Выполнение программы

2.4. ЛАБОРАТОРНЫЕ ЗАДАНИЯ

Выполните лабораторное задание в соответствии с полученным вариантом (табл. 3). Вам необходимо разработать приложение с использованием многопоточности:

- Интерфейс приложения разработать по аналогии с приведённым выше примером (кнопки «старт», «стоп», «пауза», строка вывода значений).
- Установить счётчик в обратном направлении с возможностью выбора пользователем начального значения по вариантам (строка ввода значений) – таймер (см. таблицу 3).
- Применить тот функционал, который был рассмотрен в примере (активность/неактивность кнопок).

Таблица 3

Варианты заданий

Вариант	Задание
1	50
2	68
3	84
4	22
5	54
6	23
7	63
8	57
9	64
10	73

2.5. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет оформляется в виде пояснительной записки на листах формата А4 (210 х 297 мм). Необходимо дома подготовить заготовку по всей работе. Заготовка должна содержать цель и содержание работы, основные теоретические сведения, все пункты лабораторных заданий, листинги проектов с комментариями. В конце отчета необходимо привести выводы по лабораторной работе и ответы на контрольные вопросы.

2.6. КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНЫМ ЗАДАНИЯМ

1. Что называют процессом?
2. Что называют потоком?
3. Что такое многопоточность?
4. Охарактеризуйте класс QThread.
5. Для чего предназначена функция run()?
6. Перечислите и кратко охарактеризуйте основные функции класса QThread.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ / М. Саммерфилд – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с.
2. Шлее М. Qt 5.10. Профессиональное программирование на C++ / М. Шлее – СПб: БХВ-Петербург, 2018. – 1072 с.
3. Бланшет Ж. QT 4: программирование GUI на C++ / Ж. Бланшет – М.: КУДИЦ-ПРЕСС, 2007. – 546 с.

ОГЛАВЛЕНИЕ

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ №9-10
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Составитель
Сукачев Александр Игоревич

Издается в авторской редакции

Подписано к изданию 18.03.2024.
Уч.-изд. л. 2,0

ФГБОУ ВО «Воронежский государственный технический университет»
394006 Воронеж, ул. 20-летия Октября, 84