Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Промышленная электроника»

А. В. Сахарук

ОСНОВНЫЕ КЛАССЫ QT И ИХ ПРИМЕНЕНИЕ

ПРАКТИКУМ
по выполнению лабораторных работ
по дисциплине «Технология разработки программного обеспечения систем управления»
для студентов специальности
1-53 01 07 «Информационные технологии и управление в технических системах»
дневной формы обучения

Гомель 2021

УДК 681.518(075.8)
ББК 32.973-018.2я73
С22

Рекомендовано научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 10 от 03.06.2019 г.*)

Рецензент: доц. каф. «Автоматизированный электропривод» ГГТУ им. П. О. Сухого
канд. техн. наук *В. С. Захаренко*

Сахарук, А. В.

С22 Основные классы Qt и их применение : практикум по выполнению лаборатор. работ по дисциплине «Технология разработки программного обеспечения систем управления» для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» днев. формы обучения / А. В. Сахарук. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 147 с. – Системные требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://elib.gstu.by. – Загл. с титул. экрана.

Является продолжением цикла лабораторных работ. Содержит основные теоретические сведения, порядок выполнения лабораторных работ, а также задания для самостоятельной работы и контрольные вопросы.

Для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» дневной формы обучения.

УДК 681.518(075.8)
ББК 32.973-018.2я73

Лабораторная работа № 12

Элементы ввода, управления и выбора

Часть 1

1. Цель работы

Изучить основные элементы ввода и методы их использования.

2. Основные теоретические сведения

Группа виджетов элементов ввода представляет собой основу для ввода и редактирования данных (текста и чисел) пользователем. Большая часть элементов ввода может работать с буфером обмена и поддерживает технологию перетаскивания (drag & drop), что избавляет разработчика от дополнительной реализации. Текст можно выделять с помощью мыши, клавиатуры и контекстного меню.

2.1. Однострочное текстовое поле (QlineEdit)

Этот виджет является самым простым элементом ввода. Класс QLineEdit однострочного текстового поля определен в заголовочном файле QLineEdit.

```
#include <QLineEdit>
.
.
QLineEdit *editText = new QLineEdit();
```

Текстовое поле состоит из прямоугольной области для ввода строки текста, поэтому не следует использовать этот виджет в тех случаях, когда требуется вводить более одной строки. Для ввода многострочного текста имеется класс QTextEdit.

Текст, находящийся в виджете, возвращает метод text(). Для того чтобы извлечь текст из виджета и занести его в переменную типа Qstring, необходимо выполнить следующую команду:

```
QString text = editText->text();
```

Т.к. QLineEdit является наследником классов QObject и QWidget, он имеет все те же слоты, что и классы-родители. Однако есть и собственные. Рассмотрим собственные открытые слоты, которые содержит данный класс. Они описаны в таблице 12.1.

Таблица 12.1. Собственные открытые слоты

Слот	**Описнаие**
void QLineEdit::clear()	Очистить текстовое поле
void QLineEdit::copy() const	Скопировать содержимое в буфер обмена
void QLineEdit::cut()	Вырезать содержимое в буфер обмена
void QLineEdit::paste()	Вставить текст из буфера обмена
void QLineEdit::redo()	Вернуться на шаг вперед (при редактировании)
void QLineEdit::selectAll()	Выделить весь текст
void QLineEdit::setText(const QString &)	Установить текст
void QLineEdit::undo()	Откатиться на шаг назад (при редактировании)

Аналогичная ситуация и с сигналами. Данный класс наследует все сигналы от родителей, а также имеет и свои собственные. Рассмотрим собственные сигналы, описанные в таблице 12.2.

Таблица 12.2. Собственные сигналы

Сигнал	**Описание**
void QLineEdit::cursorPositionChanged(int old, int new)	Изменена позиция курсора. Old – предыдущая позиция, new – новая позиция
void QLineEdit::editingFinished()	Редактирование завершено.
void QLineEdit::returnPressed()	Нажата клавиша Enter.
void QLineEdit::selectionChanged()	Выделение изменено.
void QLineEdit::textChanged(const QString &text)	Текст изменен. Text – текст из виджета
void textEdited(const QString &text)	Текст отредактирован

Так же рассмотрим собственные открытые методы данного класса, описанные в таблице 12.3.

Таблица 12.3. Открытые собственные методы

Метод	**Описание**
QLineEdit::QLineEdit(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на родителя, по умолчанию равен нулю

QLineEdit::QLineEdit(const QString &contents, QWidget *parent = Q_NULLPTR)	Так же конструктор, но в отличие от предыдущего, имеет параметр contents, которым можно задать текст в компоненте
void QLineEdit::backspace()	Эмулирует нажатие клавиши backspace. Если нет выделения, то удаляет левый символ от курсора, если есть выделение, то оно полностью удаляется
QMenu * QLineEdit::createStandardContextMenu()	Создает контекстное меню и возвращает на него указатель
int QLineEdit::cursorPosition() const	Возвращает текущую позицию курсора
void QLineEdit::del()	Эмулирует нажатие клавиши del. Удаляет правый от курсора символ
QString QLineEdit::inputMask() const	Возвращает текущую маску ввода
bool QLineEdit::isReadOnly() const	Возвращает состояние «только для чтения»
QString QLineEdit::selectedText() const	Возвращает выделенный текст
void QLineEdit::setInputMask(const QString &inputMask)	Установить маску ввода
void QLineEdit::setReadOnly(bool)	Установить состояние «только для чтения»
void QLineEdit::setSelection(int start, int length)	Выделить часть текста. Start – начальная позиция выделение, length – количество символов

2.2. Счетчик QSpinBox

Виджет QSpinBox предоставляет пользователю доступ к ограниченному диапазону чисел.

```
#include < QSpinBox>
.
.
QSpinBox *spinEdit = QSpinBox(this);
```

Для предотвращения выхода за пределы установленного диапазона, который устанавливается методом setRange (), все вводимые значения проверяются. Значения можно устанавливать с помощью метода setvalue (), а получать — методом value ().

```
spinEdit->setRange(0,10); //Устанавливаем интервал от 0 до 10
spinEdit-> setValue (5); //Устанавливаем значение 5
int Value = spinEdit-> value (); //Забираем значение и заносим его в переменную
```

При изменении значений посылаются сразу два сигнала valueChanged (): один с параметром типа int, а другой – const QString&.

Можно изменить способ отображения с помощью методов setPrefix() и setSuffix(). Например, вызов следующих методов приведет к тому, что число будет отображаться в скобках:

```
spinEdit ->setPrefix("(") ;
spinEdit ->setSuffix(")") ;
```

Можно изменить изображение стрелок на символы + (плюс) или – (минус), передав методу setButtonSymbols () флаг PlusMinus. Собственные сигналы данного класса описаны в таблице 12.4, собственные слоты в таблице 12.5, а собственные методы в таблице 12.6.

Таблица 12.4. Собственные сигналы QSpinBox

Сигнал	**Описание**
void QSpinBox::valueChanged(int i)	Значение изменено. i – Текущее значение типа int
void QSpinBox::valueChanged(const QString &text)	Значение изменено. text – Текущее значение типа QString
void QSpinBox::editingFinished()	Редактирование завершено

Таблица 12.5. Собственные слоты QSpinBox

Слот	**Описание**
virtual void QSpinBox::clear()	Очистить
void QSpinBox::selectAll()	Выделить все
void QSpinBox::stepDown()	Шаг вниз (уменьшить значение на один шаг)
void QSpinBox::stepUp()	Шаг вверх (увеличить значение на один шаг)

Таблица 12.6. Собственные методы QSpinBox

Метод	**Описание**
QSpinBox::QSpinBox(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на объект-родитель. По умолчанию равен 0
QString QSpinBox::cleanText() const	Этот метод возвращает текст виджета, без суффиксов и префиксов
int QSpinBox::displayIntegerBase() const	Метод возвращает систему счисления, в которой отображаются значения(например 10-ая, 2-ая и т. д.)
int QSpinBox::maximum() const	Возвращает текущее максимальное значение
int QSpinBox::minimum() const	Возвращает текущее минимальное значение
QString QSpinBox::prefix() const	Возвращает текущий префикс
void QSpinBox::setDisplayIntegerBase(int base)	Устанавливает систему счисления для отображения чисел
void QSpinBox::setMaximum(int max)	Устанавливает максимум
void QSpinBox::setMinimum(int min)	Устанавливает минимум
void QSpinBox::setPrefix(const QString &prefix)	Устанавливает префикс (отображается перед значением). prefix – текстовый префикс для отображения в виджете

void QSpinBox::setRange(int minimum, int maximum)	Устанавливает рабочий интервал виджета (максимальное и минимальное значение). minimum – минимальное значение, maximum – максимальное значение
void QSpinBox::setSingleStep(int val)	Устанавливает размер единичного шага значений
void QSpinBox::setSuffix(const QString &suffix)	Устанавливает текстовый суффикс (отображается после значения)
int QSpinBox::singleStep() const	Возвращает единичный шаг
QString QSpinBox::suffix() const	Возвращает суффикс
int QSpinBox::value() const	Возвращает текущее значение

2.3. Элемент ввода даты и времени

Существует три виджета ввода даты и времени:

- QDateTimeEdit;
- QDateEdit;
- QTimeEdit.

Два последних являются наследниками QDateTimeEdit, поэтому рассматривать их в отдельности нет смысла. В свою очередь, QDateTimeEdit является потомком от следующих классов:

- QAbstractSpinBox;
- QWidget;
- QObject;
- QPaintDevice.

И, соответственно, наследует их сигналы, слоты и методы. Более подробно рассмотрим собственные открытые сигналы, соты и методы класса QdateTimeEdit. Сигналы, слоты и методы класса QDateTimeEdit представлены в таблица 12.7–12.9.

Таблица 12.7. Собственные сигналы QDateTimeEdit

Сигнал	**Описание**
void QDateTimeEdit::dateChanged(const QDate &date)	Дата изменена. Date – установленная дата
void QDateTimeEdit::dateTimeChanged(const QDateTime &datetime)	Дата и время изменены. Datetime – установленные дата и время
void QDateTimeEdit::timeChanged(const QTime &time)	Время изменено. time – установленное время

Таблица 12.8. Собственные слоты QDateTimeEdit

Слот	**Описание**
void QDateTimeEdit::setDate(const QDate &date)	Установить дату. date – дата
void QDateTimeEdit::setDateTime(const QDateTime &dateTime)	Установить дату и врямя. dateTime – дата и время
void QDateTimeEdit::setTime(const QTime &time)	Установить время. time – время

Методов у данного класса намного больше, чем у предыдущих. Поэтому рассмотрим только основные из них. Более подробную информацию по данному компоненту можно получить из официальной документации на сайте http://qt.io и из встроенного справочника Qt Assistant.

Таблица 12.9. Собственные открытые методы QDateTimeEdit

Метод	**Описание**
QDateTimeEdit::QDateTimeEdit(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на родителя, по умолчанию равен 0.
QDateTimeEdit::QDateTimeEdit(const QDateTime &datetime, QWidget *parent = Q_NULLPTR)	Так же конструктор класса, однако имеет дополнительный параметр datetime, который задает время и

	дату, устанавливаемые в виджете
QDateTimeEdit::QDateTimeEdit(const QDate &date, QWidget *parent = Q_NULLPTR)	Аналогично предыдущему имеет дополнительный входной параметр date, который устанавливает дату
QDateTimeEdit::QDateTimeEdit(const QTime &time, QWidget *parent = Q_NULLPTR)	Аналогичный предыдущему, только входной параметр time – устанавливает время в виджете
void QDateTimeEdit::clearMaximumDate()	Сбросить максимальную дату
void QDateTimeEdit::clearMaximumDateTime()	Сбросить максимальные дату и время
void QDateTimeEdit::clearMaximumTime()	Сбросить максимальное время
void QDateTimeEdit::clearMinimumDate()	Сбросить минимальную дату
void QDateTimeEdit::clearMinimumDateTime()	Сбросить минимальные дату и время
void QDateTimeEdit::clearMinimumTime()	Сбросить минимальное время
QDate QDateTimeEdit::date() const	Возвращает установленную дату
QDateTime QDateTimeEdit::dateTime() const	Возвращает текущие дату и время
QTime QDateTimeEdit::time() const	Возвращает установленное время
void QDateTimeEdit::setDisplayFormat(const QString &format)	Установить формат отображения

Как видно из выше перечисленных методов работа с датой и временем производится через классы QDateTime, QTime и QDate. Поэтому стоит рассмотреть их отдельно. Основные открытые методы класса QdateTimeEdit можно увидеть в таблице 12.10.

Таблица 12.10. Основные открытые методы QDateTime

Метод	**Описание**
QDate QDateTime::date() const	Возвращает установленную дату в формате QDate
qint64 QDateTime::daysTo(const QDateTime &other) const	Возвращает количество дней от установленной даты до даты указанной в параметре other
void QDateTime::setDate(const QDate &date)	Устанавливает дату
void QDateTime::setTime(const QTime &time)	Устанавливает время из формата QTime
void QDateTime::setTime_t(uint seconds)	Устанавливает время из формата UTC
QTime QDateTime::time() const	Возвращает установленное время в формате QTime
QDateTime QDateTime::toLocalTime() const	Возвращает системные дату и время в формате QDateTime
QString QDateTime::toString(const QString &format) const	Преобразовывает установленное время в строку с указанным форматом format
uint QDateTime::toTime_t() const	Преобразовывает установленные дату и время в формат UTC

UTC – Всемирное координированное время, является целым количеством секунд 1 января 1970 года. Основные открытые методы класса Qdate представлены в таблице 12.11.

Таблица 12.11. Основные открытые методы QDate

Метод	**Описание**
QDate::QDate(int y, int m, int d)	Конструктор класса. в качестве входных параметров задается дата: y – год, m – месяц, d - день
int QDate::day() const	Возвращает день установленной даты
int QDate::dayOfWeek() const	Возвращает день недели установленной даты
int QDate::dayOfYear() const	Возвращает номер установленного дня в году
int QDate::daysInMonth() const	Возвращает количество дней в

	установленном месяце
int QDate::daysInYear() const	Возвращает количество дней в установленном году
void QDate::getDate(int *year, int *month, int *day) const	Возвращает дату в виде чисел. В качестве параметров передаются указатели на целочисленные переменные, в которые будет помещена дата
bool QDate::setDate(int year, int month, int day)	Устанавливает дату из целых чисел
QString QDate::toString(const QString &format) const	Преобразовывает текущую дату в текстовую строку, в соответствии с форматом
int QDate::year() const	Возвращает установленный год

Основные открытые методы класса Qtime можно найти в таблице 12.12.

Таблица 12.12. Основные открытые методы QTime

Метод	**Описание**
QTime::QTime(int h, int m, int s = 0, int ms = 0)	Конструктор класса. В качестве параметра может принимать время в числовом формате
int QTime::hour() const	Возвращает установленные часы
int QTime::minute() const	Возвращает установленные минуты
int QTime::msec() const	Возвращает установленные миллисекунды
int QTime::second() const	Возвращает установленные секунды
bool QTime::setHMS(int h, int m, int s, int ms = 0)	Устанавливает время.
QString QTime::toString(const QString &format) const	Преобразовывает установленное время в текстовом формате.
QTime QTime::currentTime()	Возвращает установленное время
QTime QTime::fromString(const QString &string, const QString &format)	Преобразовывает время из текстового формата в формат QTime

2.4. Редактор текста

Класс QTextEdit позволяет осуществлять просмотр и редактирование как простого текста, так и текста в формате HTML. Он унаследован от класса QAbstractScrollArea, что дает возможность автоматически отображать полосы прокрутки, если текст не может быть полностью отображен в отведенной для него области.

Класс QTextEdit содержит следующие методы:

- setReadOnly () — устанавливает или снимает режим блокировки изменения текста;
- text () — возвращает текущий текст.

Слоты:

- setPlainText () — установка обычного текста;
- setHtml () — установка текста в формате HTML;
- copy (), cut () и paste () — работа с буфером обмена (копировать, вырезать и вставить соответственно);
- selectAll () или deselect () — выделение или снятие выделения всего текста;
- clear () — очистка поля ввода.

И сигналы:

- textChanged () — отправляется при изменении текста;
- selectionChanged () — отправляется при изменениях выделения текста.

Для работы с выделенным текстом служит класс QTextCursor, и объект этого класса содержится в классе QTextEdit. Класс QTextCursor предоставляет методы для создания участков выделенння текста, получения содержимого выделенного текста и его удаления. Указатель на объект класса QTextCursor можно получить вызовом метода QTextEdit:: textCursor ().

Виджеты класса QTextEdit также содержат в себе объект QTextDocument, указатель на который можно получить посредством метода QTextEdit:: document (). Можно также присвоить ему другой документ при помощи метода QTextEdit:: setDocument (). Класс QTextDocument предоставляет слот undo() (для отмены) или redo() (для повтора действий). При вызове слотов undo() и redo() посылаются сигналы undoAvailable(bool) и redoAvailable(bool), сообщающие об успешном (или безуспешном) проведении операции.

Эти сигналы отправляются как из класса QTextDocument, так и из класса QTextEdit. В большинстве случаев удобнее использовать сигналы класса QTextEdit.

Большинство методов класса QTextEdit являются делегирующими для класса QTextDocument. Например, как уже было сказано ранее, класс QTextEdit способен отображать файлы с кодом на языке HTML, содержащие таблицы и растровые изображения. Для его размещения и показа можно воспользоваться либо методом setHtml(), в который передается строка, содержащая в себе текст в формате HTML, либо слотом insertHtml(). Эти методы определены в обоих классах, и их вызов из объекта класса QTextEdit приведет к тому, что будет вызван аналогичный метод из объекта класса QTextDocument.

Для помещения обычного текста в область виджета можно воспользоваться методом setPlainText() или слотом insertPlainText(). При помощи слота append() осуществляется добавление текста, причем добавленный текст не вносится в список операций, действие которых можно вернуть с помощью слота undo(), что делает этот слот быстрым и не требующим дополнительных затрат памяти. Метод find() может быть использован для поиска и выделения заданной строки в тексте.

3. Порядок выполнения работы

3.1 Использование QLineEdit

Рассмотрим пример использования QLineEdit. В данной программе используется два текстовых поля. Первое текстовое поле предназначено для ввода обычного текста, при вводе текст дублируется в отладочную консоль. Второе текстовое поле предназначено для ввода пароля, вводимые символы скрываются маской.

Programm12_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_1
TEMPLATE = app
```

```
SOURCES += main.cpp widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
   QApplication a(argc, argv);
   Widget w;
   w.show();

   return a.exec(); }
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QGridLayout>
#include <QLineEdit>
#include <QDebug>
#include <QLabel>

class Widget : public Qwidget {
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   QGridLayout *mainLayout=NULL;
   QLabel *labelText=NULL;
   QLineEdit *editText=NULL;
   QLabel *labelPassword=NULL;
   QLineEdit *editPassword=NULL;
```

```
private slots:
   void slotTextChanged(QString text); };
#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : Qwidget(parent) {
   mainLayout = new QGridLayout(this);

   labelText = new QLabel(tr("Enter text"),this);
   mainLayout->addWidget(labelText,0,0,1,1);
   editText = new QLineEdit(tr("Hello edit"),this);

connect(editText,SIGNAL(textChanged(QString)),SLOT(slotTextChanged
(QString)));
   mainLayout->addWidget(editText,0,1,1,1);

   labelPassword = new QLabel(tr("Password"),this);
   mainLayout->addWidget(labelPassword,1,0,1,1);
   editPassword = new QLineEdit(this);
   editPassword->setEchoMode(QLineEdit::Password);
   mainLayout→addWidget(editPassword,1,1,1,1); }

void Widget::slotTextChanged(QString text) {
   qDebug()<<text; }
```

3.2 Использование счетчика QSpinBox.

Рассмотрим пример использования QSpinBox. При выводе чисел применен суффикс и префикс. При изменении числа оно дублируется в отладочную консоль.

Programm12_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_2
```

```
TEMPLATE = app
SOURCES += main.cpp widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec(); }
```

widget.h

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec(); }
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : Qwidget(parent) {
    mainLayout = new QHBoxLayout(this);

    spinValue = new QSpinBox(this);
    spinValue->setMinimum(0);
    spinValue->setMaximum(20);
    spinValue->setSingleStep(2);
    spinValue->setPrefix("(") ;
    spinValue->setSuffix(")") ;
```

```
connect(spinValue,SIGNAL(valueChanged(int)),SLOT(slotValueChanged
(int)));
   mainLayout→addWidget(spinValue); }

void Widget::slotValueChanged(int I) {
   qDebug()<<QString::number(i); }
```

3.3. Использование QDateTimeEdit

Рассмотрим пример вывода текущего времени и даты. В QDateTimeEdit выводятся текущие дата и время, в формате "hh:mm:ss dd-MM-yyyy". За счет использования таймера, каждую секунду производится обновление даты и времени.

Programm12_3.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_3
TEMPLATE = app
SOURCES += main.cpp widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
   QApplication a(argc, argv);
   Widget w;
   w.show();
   return a.exec(); }
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QVBoxLayout>
#include <QDateTime>
#include <QDateTimeEdit>
#include <QTimer>

class Widget : public Qwidget {
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);

private:
   QVBoxLayout *mainLayout=NULL;
   QDateTimeEdit *editDateTime=NULL;
   QTimer *timerClock=NULL;

private slots:
   void slotTimeOut(); };
#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : Qwidget(parent) {
   mainLayout = new QVBoxLayout(this);

   editDateTime = new
QDateTimeEdit(QDateTime::currentDateTime(),this);
   editDateTime->setDisplayFormat("hh:mm:ss dd-MM-yyyy");
   mainLayout->addWidget(editDateTime);

   timerClock = new QTimer;
   connect(timerClock,SIGNAL(timeout()),SLOT(slotTimeOut()));
   timerClock→start(1000); }

Widget::~Widget() {
```

```
    timerClock->stop();
    delete timerClock; }

void Widget::slotTimeOut() {
    editDateTime→setDateTime(QDateTime::currentDateTime()); }
```

3.4. Использование QTextEdit.

Простой пример использования QTextEdit. В текстовое поле выводится текст, который возможно редактировать. При изменении размеров формы, выводимый текст подстраивается под новые размеры текстового поля.

Programm12_4.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_4
TEMPLATE = app
SOURCES += main.cpp widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
```

```
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QTextEdit>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   QVBoxLayout *mainLayout=NULL;
   QTextEdit *textEdit=NULL;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QVBoxLayout(this);

   textEdit = new QTextEdit(this);
   textEdit->setPlainText("Qt кроссплатформенный инструментарий\n "
                          "разработки ПО на языке программирования C++.\n");
   mainLayout->addWidget(textEdit);
}
```

3.5. Задание для самостоятельного выполнения

3.5.1. Программа «Пароль»

Реализовать программу, которая производит вычисление функции, указанной в таблице (функция выбирается в соотвествии с порядковым номером в журнале). При этом вход в программу осуществляется по паролю. После запуска программы появляется окно ввода пароля. Символы в поле ввода пароля должны быть скрыты. Если пароль введен верно, появляется основное окно программы с формой для расчёта функции. Если пароль введен неверно, то выдается ошибка и программа завершается. Значения переменных вводятся в QLineEdit на основной форме. Индивидуальные формулы для расчета приведены в таблице 12.13.

Таблица 12.13. Формулы для расчета

№	Задание	№	Задание
1	y = 15a +45b – 123c	11	y = (88a+5)*b/c
2	y = 10sin(a) – 12b + c^2	12	y = /5c + 15b
3	y = 56sin(a) – 10cos(b) + 13c	13	y = sin(a)*cos(b) + 5c
4	y = 11a – 3sin(b) +16c	14	y = 192-4a+5c/b
5	y = 2a^2 + 15b – 150/c	15	y = 4ab/c
6	y = 2(a+b)^2-16c	16	y = 11a^2+b^3+22c
7	y = 21/a + 14b = c^2	17	y = 11(a+b) + c/10
8	y = 4a/b + c^6	18	y = 77(a^2+2b) – 3c/4
9	y = (11a^4 +b)*c	19	y = 11(a+b+c)
10	y = a^2+b^2+c^2	20	y = 90a – ¾(b-c^2)

3.5.2. Программа «Текстовый редактор»

Реализовать программу, позволяющую открыть текстовый файл, провести редактирование (добавление, удаление текста) и затем сохранить его.

4. Содержание отчета

Наименование и цель работы. Краткие теоретические сведения, которые применялись при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые

отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Для чего предназначены виджеты-элементы ввода?
2. Для чего предназначен класс QLineEdit?
3. Область применения класса QSpinBox?
4. Область применения класса QDateTimeEdit?
5. Область применения класса QTextEdit?

Часть 2

1. Цель работы

Изучить основные элементы ввода и методы их использования.

2. Основные теоретические сведения

2.1. Кнопки

Класс QAbstractButton – базовый для всех кнопок. В приложениях применяются три основных вида кнопок: нажимающиеся кнопки (QPushButton), которые обычно называют просто кнопками, флажки (QCheckBox) и переключатели (QRadioButton). В классе QAbstractButton реализованы методы и возможности, присущие всем кнопкам. Сначала мы обсудим основные из этих возможностей, а потом поговорим о каждом виде в отдельности.

Все кнопки могут содержать текст, который можно передать как в конструкторе первым параметром, так и установить с помощью метода setText (). Для получения текста в классе QAbstractButton определен метод text ().

Растровое изображение устанавливается на кнопке при помощи метода seticon(). После установки изображения вызовом метода seticonSize() можно изменить его максимальный размер, который занимает изображение на кнопке (изображения меньшего размера не растягиваются). Для получения текущего максимального размера изображения определен метод iconSize (). И, наконец, для того чтобы кнопка возвратила установленное в ней изображение, нужно вызвать метод icon ().

Для взаимодействия с пользователем класс QAbstractButton предоставляет сигналы, указанные в таблице 12.14.

Таблица 12.14 Сигналы класса QAbstractButton

Сигнал	**Описание**
clicked ()	отправляется при щелчке кнопкой мыши
pressed ()	отправляется при нажатии на кнопку мыши
released ()	отправляется при отпускании кнопки мыши
toggled ()	отправляется при изменении состояния кнопки, имеющей статус выключателя

Для опроса текущего состояния кнопок в классе QAbstractButton определены три метода (таблица 12.15).

Таблица 12.15 Методы опроса состояния

Метод	**Описание**
isDown()	возвращает значение true, если кнопка находится в нажатом состоянии. Изменить текущее состояние может либо пользователь, нажав на кнопку, либо вызов метода setDown ()
isChecked ()	возвращает значение true, когда кнопка находится во включенном состоянии. Изменить текущее состояние может либо пользователь, нажав на кнопку, либо вызов метода setChecked ()
setEnabled()	кнопка доступна, то есть реагирует на действия пользователя, если метод isEnabled() возвращает значение true. Изменить текущее состояние можно вызовом метода setEnabled()

Класс QPushButton виджета нажимающейся кнопки определен в заголовочном файле QPushButton.

Создать нажимающуюся кнопку можно следующим образом:

```
QPushButton* buttonA = new QPushButton(tr("My Button"),this);
```

Первый параметр (типа строка) задает надпись кнопки. Второй параметр – указатель на родителя, т.е. виджет, на котором будет располагаться кнопка. При щелчке по кнопке отправляется сигнал clicked(). Кнопка считается нажатой, если пользователь щелкнул на ней кнопкой мыши или нажал на клавишу <Enter>, при условии, что кнопка была текущей (была в фокусе). Для того чтобы кнопку сделать таковой, можно воспользоваться методом setDefault (). Возможные варианты кнопок приведены в таблице 12.16. Внешний вид кнопок приведен на рисунке 12.1.

Таблица 12.16. Варианты кнопок

Кнопка	**Описание**
Normal Button	(Обычная кнопка) соответствует той самой кнопке, которую мы привыкли видеть в большинстве случаев. После отпускания кнопка всегда возвращается в свое исходное положение

Toggle Button	(Выключатель) может пребывать в двух состояниях: нажатом или не нажатом, которые соответствуют положениям «включено» или «выключено». Логика действия этой кнопки идентична, например, логике обычного комнатного электровыключателя
Flat Button	Flat Button (Плоская кнопка) по своим функциональным особенностям идентична обычной кнопке. Разница лишь во внешнем виде. Например, благодаря тому, что конту- ры этой кнопки не видны, ею можно воспользоваться для размещения «секретной кнопки» диалогового окна
Pixmap Button	(Кнопка с изображением) представляет собой кнопку, содержащую растровое изображение

Рис. 12.1. Внешний вид различных вариантов кнопок

Для управления внешним видом кнопок используются методы указанные в таблице 12.17.

Таблица 12.17 Методы для управления внешним видом кнопок

Метод	**Описание**
setCheckable()	Включает режим переключателя
setChecked ()	Устанавливает кнопку в нажатое состояние
setFlat ()	Переводит кнопку в плоский вид
seticonSize()	Установка размера иконок
setIcon()	Устанавливает иконку на кнопке

Существует возможность использования в нажимающихся кнопках всплывающего меню. Подобные кнопки можно встретить, например, кнопка Start (Пуск) панели задач ОС Windows 7. Добавить меню можно, вызвав метод setMenu() и передав указатель на объект всплывающего меню. Внещний вид кнопки с контестным меню можно увидеть на рисунке 12.2.

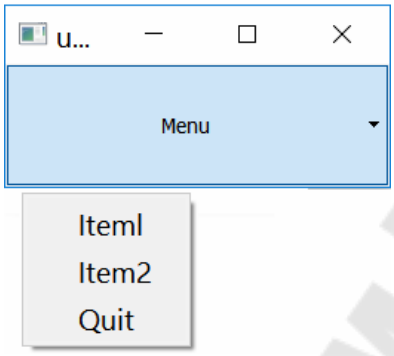

Рис. 12.2. Всплывающее меню на кнопке

2.2 Виджет флажка QCheckBox

`

Большинство программ представляют наборы настроек, дающих возможность изменять поведение программы. Для этих целей может пригодиться виджет флажка, который позволяет пользователю выбирать сразу несколько опций. Класс QCheckBox виджета кнопки флажка определен в заголовочном файле QCheckBox.

Флажок представляет собой маленький прямоугольник и может содержать поясняющий текст или картинку. При щелчке на виджете в прямоугольнике появится отметка. Этого же можно добиться нажатием клавиши <Пробел>, когда виджет находится в фокусе. Виджет флажка устанавливается в положение «включено» или «выключено» и является, по логике действия, кнопкой-выключателем (toggle button). Но, в отличие от последней, флажок может иметь еще и третье состояние — неопределенное.

Пример использования такого состояния можно увидеть в диалоговом окне Properties (Свойства) Проводника в ОС Windows при выборе нескольких файлов, имеющих разные атрибуты. Внещний вид класса флажка представлен на рисунке 12.3, а основные методы данного класса перечислены в таблице 12.18.

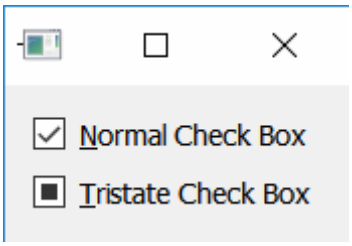

Рис. 12.3. Внешний вид

Таблица 12.18. Основные методы QCheckBox

Метод	**Описание**
setChecked ()	Устанавливает/убирает флажок
setTristate ()	Включение неопределенного состояния
setCheckState ()	Установка состояния. Если передать параметр Qt:: PartiallyChecked устанавливается третье состояние

2.3 Переключатели QRadioButton

Свое английское название — radiobutton — виджет переключателя получил благодаря схожести с кнопками переключения диапазонов радиоприемника, на панели которого может быть нажата только одна из таких кнопок. Нажатие на кнопку другого диапазона приводит к тому, что автоматически отключается кнопка диапазона, нажатая до этого.

Переключатель представляет собой виджет, который должен находиться в одном из двух состояний: включено (on) или выключено (off). Эти состояния пользователь может устанавливать с помощью мыши или клавиши <Пробел>, когда кнопка находится в фокусе. Класс QRadioButton виджета переключателя определен в заголовочном файле QRadioButton.

Виджеты переключателей должны предоставлять пользователю, по меньшей мере, выбор одной из двух альтернатив, не могут использоваться в отдельности и должны быть сгруппированы вместе. Их группировку можно выполнить, например, при помощи класса QGroupBox. Внешний вид QradioButton приведен на рисунке 12.4.

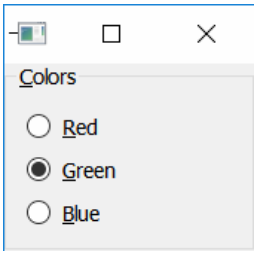

Рис. 12.4. Внешний вид переключателей

2.4. Ползунок QSlider

Ползунок позволяет довольно комфортно выполнять настройки некоторых параметров приложения. Класс QSlider ползунка определен в заголовочном файле QSlider.

Класс QSlider содержит метод, управляющий размещением рисок (шкалы) ползунка. Риски очень важны при отображении ползунка. Они дают пользователю визуально более четкое представление о его местонахождении и показывают шаг.

Возможные значения, которые можно передать в метод setTickPosition (), приведены в таблице 12.19.

Таблица 12.19. Значения setTickPosition

Значение	**Описание**
NoTicks	Ползунок без рисок
TicksAbove	Отображение рисок на верхней стороне ползунка
TicksBelow	Отображение рисок на нижней стороне ползунка
TicksBothSides	Отображение рисок на верхней и нижней сторонах ползунка

Метод setTickinterval () задает шаг рисования рисок. Не следует задавать большое количество рисок, так как это приведет к появлению сплошной серой линии и не принесет никакой пользы.

При помощи метода setvalue () можно задавать стартовое значение, используемое для синхронизации с другими элементами управления, работающими вместе с ползунком. С его помощью можно сделать так, чтобы величины виджетов совпадали друг с другом, он также может служить просто для задания начального значения при первом показе элемента.

3. Порядок выполнения работы

3.1. Использование QPushButton.

Рассмотрим пример использования кнопок. В данной программе создается форма с четырьмя кнопками. Каждая кнопка имеет свой стиль. При нажатии на кнопку в отладочную консоль выдается сообщение о текущем состоянии выбранной кнопки.

Programm12_5.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_5
TEMPLATE = app
SOURCES += main.cpp widget.cpp
HEADERS  += widget.h
RESOURCES += icon.qrc
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QVBoxLayout>
```

```
#include <QPushButton>
#include <QDebug>

class Widget : public QWidget {
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);

private:
   QVBoxLayout *mainLayout=NULL;
   QPushButton *buttonA=NULL;
   QPushButton *buttonB=NULL;
   QPushButton *buttonC=NULL;
   QPushButton *buttonD=NULL;

private slots:
   void slotPushButtonA();
   void slotButtonBToggled(bool input);
   void slotPushButtonC();
   void slotPushButtonD(); };
#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QVBoxLayout(this);

   buttonA = new QPushButton(tr("Button A"),this);
   connect(buttonA,SIGNAL(pressed()),this,SLOT(slotPushButtonA()));
   mainLayout->addWidget(buttonA);

   buttonB = new QPushButton(tr("BUtton B"),this);
   buttonB->setCheckable(true);
```

```
connect(buttonB,SIGNAL(toggled(bool)),this,SLOT(slotButtonBToggled(
bool)));
   mainLayout->addWidget(buttonB);

   buttonC = new QPushButton(tr("Button C"),this);
   buttonC->setFlat(true);
   connect(buttonC,SIGNAL(pressed()),this,SLOT(slotPushButtonC()));
   mainLayout->addWidget(buttonC);

   buttonD = new QPushButton(tr("Button D"),this);
   QPixmap pix(":/battery.png");
   buttonD->setIcon(pix);
   buttonD->setIconSize(QSize(64,64));
   connect(buttonD,SIGNAL(pressed()),this,SLOT(slotPushButtonD()));
   mainLayout->addWidget(buttonD);
}

void Widget::slotPushButtonA()
{
   qDebug()<<"Button A pressed";
}

void Widget::slotButtonBToggled(bool input)
{
   qDebug()<<"Button B toggled "<<input;
}

void Widget::slotPushButtonC()
{
   qDebug()<<"Button C pressed";
}

void Widget::slotPushButtonD()
{
   qDebug()<<"Button D pressed";
}
```

icon.qrc

```
<RCC>
   <qresource prefix="/">
      <file>battery.png</file>
   </qresource>
</RCC>
```

Стоит отметить, что данный пример включается в себя файл ресурсов. Ресурсы – дополнительные материалы, такие как картинки, текстовые документы и т.д., которые при сборке вносятся в исполняемый файл и могут использоваться в процессе выполнения программы. В данном случае файл ресурсов содержит картинку, которая отображается на последней кнопке. Для добавлени файла ресурсов необходимо вызвать выпадающее меню, на имени проекта, выбрать пункт «Добавить новый…» и, в появившимся окне, выбрать файл ресурсов. Затем открыть его и добавить необходимые материалы.

3.2. Использование QCheckBox

Рассмотрим пример использования QCheckBox. В данной программе используется 4 виджета флажка и один виджет группировки. Первый QCheckBox разрешает или запрещает виджет группировки элементов. Совтояние второго виджета флажка зависит от двух последних. Если два последних флажка выставлены в состояние true, то и второй принимает аналогичное состояние. Если они выставлены оба в состояние false, то и второй принимает это состояние. Если состояние последнего и предпоследнего различно, то второй виджет флажка принимает неопределенное состояние.

Programm12_6pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_6
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   Widget w;
   w.show();

   return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QCheckBox>
#include <QGroupBox>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);

private:
   QVBoxLayout *mainLayout=NULL;
   QCheckBox *checkEnableSettings=NULL;
   QGroupBox *groupSetings=NULL;
   QVBoxLayout *layoutGroupSettings=NULL;
   QCheckBox *checkParam1=NULL;
   QCheckBox *checkParam2=NULL;
   QCheckBox *checkParam3=NULL;
```

```
private slots:
    void slotToggle(bool input);
};

#endif // WIDGET_H
```

Widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    checkEnableSettings = new QCheckBox(tr("Enable settings"),this);
    mainLayout->addWidget(checkEnableSettings);
    groupSetings = new QGroupBox(tr("Settings"),this);
    groupSetings->setEnabled(checkEnableSettings->isChecked());

connect(checkEnableSettings,SIGNAL(clicked(bool)),groupSetings,SLOT
(setEnabled(bool)));
    mainLayout->addWidget(groupSetings);
    layoutGroupSettings = new QVBoxLayout(groupSetings);

    checkParam1 = new QCheckBox(tr("Option 1"),groupSetings);
    checkParam1->setCheckState(Qt::PartiallyChecked);
    layoutGroupSettings->addWidget(checkParam1);

    checkParam2 = new QCheckBox(tr("Option 2"),groupSetings);

connect(checkParam2,SIGNAL(clicked(bool)),this,SLOT(slotToggle(bool
)));
    layoutGroupSettings->addWidget(checkParam2);

    checkParam3 = new QCheckBox(tr("Option 3"),groupSetings);

connect(checkParam3,SIGNAL(clicked(bool)),this,SLOT(slotToggle(bool
)));
    layoutGroupSettings->addWidget(checkParam3);
```

```
}

void Widget::slotToggle(bool input)
{
   if (checkParam2->isChecked()&&checkParam3->isChecked())
   {
      checkParam1->setChecked(false);
      checkParam1->setChecked(true);

   } else if ((!checkParam2->isChecked()) && (!checkParam3-
>isChecked()))
   {
      checkParam1->setChecked(false);
   } else
   {
      checkParam1->setCheckState(Qt::PartiallyChecked);
      checkParam1->setTristate(true);
   }
}
```

3.3. Использование QRadioButton

Рассмотрим пример использования QRadioButton. В данной программе используется три объекта типа QRadioButton. Они находятся на виджете группировки. При выборе одного из трех элементов – цвет формы меняется на соответствующий.

Programm12_7.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_7
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QRadioButton>
```

```
#include <QGroupBox>
#include <QPalette>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);

private:
   QVBoxLayout *mainLayout=NULL;
   QGroupBox *groupColors=NULL;
   QVBoxLayout *layoutGroupColor=NULL;
   QRadioButton *radioRed=NULL;
   QRadioButton *radioGreen=NULL;
   QRadioButton *radioBlue=NULL;

private slots:
   void slotClickRadioRed(bool input);
   void slotClickRadiokGreen(bool input);
   void slotClickRadioBlue(bool input);
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QVBoxLayout(this);
   groupColors = new QGroupBox(tr("Colors"),this);
   mainLayout->addWidget(groupColors);
   layoutGroupColor = new QVBoxLayout(groupColors);
```

```
    radioRed = new QRadioButton(tr("Red"),groupColors);

connect(radioRed,SIGNAL(clicked(bool)),this,SLOT(slotClickRadioRed(
bool)));
    layoutGroupColor->addWidget(radioRed);

    radioGreen = new QRadioButton(tr("Green"),groupColors);

connect(radioGreen,SIGNAL(clicked(bool)),this,SLOT(slotClickRadiokGr
een(bool)));
    layoutGroupColor->addWidget(radioGreen);

    radioBlue=new QRadioButton(tr("Blue"),groupColors);

connect(radioBlue,SIGNAL(clicked(bool)),this,SLOT(slotClickRadioBlue
(bool)));
    layoutGroupColor->addWidget(radioBlue);
}

void Widget::slotClickRadioRed(bool input)
{
    QPalette palette;
    palette.setColor(QPalette::Background,QColor( Qt::red ) );
    this->setPalette( palette );
}

void Widget::slotClickRadiokGreen(bool input)
{
    QPalette palette;
    palette.setColor(QPalette::Background,QColor( Qt::green ) );
    this->setPalette( palette );
}

void Widget::slotClickRadioBlue(bool input)
{
    QPalette palette;
    palette.setColor(QPalette::Background,QColor( Qt::blue ) );
    this->setPalette( palette );
}
```

3.4. Использование QSlider

Рассмотрим пример использования QSlider. В данной программе на форме расположено два элемента: QSlider и QpushButton. Размер и ширина кнопки зависит от положения ползунка.

Programm12_8.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_8
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec(); }
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QSlider>
#include <QPushButton>

class Widget : public QWidget {
```

```
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QVBoxLayout *mainLayout=NULL;
    QSlider *sliderSize=NULL;
    QPushButton *buttonA=NULL;

private slots:
    void slotValueChanged(int value);
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : Qwidget(parent) {
    mainLayout = new QVBoxLayout(this);
    sliderSize = new QSlider(Qt::Horizontal,this);

connect(sliderSize,SIGNAL(valueChanged(int)),this,SLOT(slotValueChan
ged(int)));
    sliderSize->setMinimum(10);
    sliderSize->setMaximum(200);
    mainLayout->addWidget(sliderSize);

    buttonA = new QPushButton(tr("Button A"),this);
    buttonA->setMaximumWidth(sliderSize->value());
    mainLayout→addWidget(buttonA); }

void Widget::slotValueChanged(int value) {
    if (buttonA!=NULL)
buttonA→setMaximumWidth(sliderSize→value()); }
```

3.5. Задание для самостоятельного выполнения

Программа «Пароль» модернизированная

Реализовать программу, которая производит вычисление трех функциий, выбранных из таблицы (первая функция выбирается в соотвествии с порядковым номером в журнале, две других берутся по номеру в журнале +1 и -1). При этом вход в программу осуществляется по паролю, после запуска программы появляется окно ввода пароля. Символы в поле ввода пароля должны быть скрыты. Диалог ввода пароля должен содержать флажок, при нажатии на котором пароль, введенный в соответствующее поле, открывается. Если пароль введен верно, появляется основное окно программы с формой для расчёта функции. Если пароль введен неверно, то выдается ошибка и программа завершается. Значения переменных задаются с помощью QSlider, выбор вычисляемой функции осуществляется с помощью QradioButton. Индивидуальные формулы для рассчета приведены в таблице 12.20.

Таблица 12.20. Формулы для расчета

№	Задание	№	Задание
1	y = 15a +45b – 123c	11	y = (88a+5)*b/c
2	y = 10sin(a) – 12b + c^2	12	y = 10a/5c + 15b
3	y = 56sin(a) – 10cos(b) + 13c	13	y = sin(a)*cos(b) + 5c
4	y = 11a – 3sin(b) +16c	14	y = 192-4a+5c/b
5	y = 2a^2 + 15b – 150/c	15	y = 4ab/c
6	y = 2(a+b)^2-16c	16	y = 11a^2+b^3+22c
7	y = 21/a + 14b = c^2	17	y = 11(a+b) + c/10
8	y = 4a/b + c^6	18	y = 77(a^2+2b) – 3c/4
9	y = (11a^4 +b)*c	19	y = 11(a+b+c)
10	y = a^2+b^2+c^2	20	y = 90a – ¾(b-c^2)

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Для чего предназначены виджеты-управления?
2. Для чего предназначен класс QCheckBox?
3. Область применения класса QSlider*?*
4. Область применения класса QRadioButton?
5. Область применения класса QpushButton?

Лабораторная работа № 13

Управление событиями

1. Цель работы

Изучить основные события и их использование при построении графического интерфейса.

2. Основные теоретические сведения

Обработка событий лежит в основе работы каждого приложения, имеющего пользовательский интерфейс. Событие можно охарактеризовать, как механизм оповещения приложения о каком-либо происшествии. Например, нажатие пользователем кнопки мыши или клавиши клавиатуры приведет к созданию события мыши или клавиатуры Событие будет создаваться и при необходимости перерисовывать содержимое окна и т. п. Очевидно, что основная масса событий тесно связана с действиями, предпринимаемыми пользователем. Но есть и события, создаваемые самой операционной системой, — например, событие таймера. Все события помещаются в соответствующую очередь для их дальнейшей обработки.

Механизм сигналов и слотов, по сравнению с событиями, представляет собой механизм более высокого уровня, предназначенный для связи объектов. Хотя и то, и другое является уведомлением о происходящем. Например, нажатие кнопки приводит к оповещению о происходящем всех подключенных к сигналу объектов. События оповещают объекты о действиях пользователя общего и детального характера (например, о перемещении указателя мыши или нажатии какой-либо клавиши клавиатуры). Другими словами, воспользовавшись стандартными сигналами, вы можете сделать заключение о том, что кнопка была нажата, но узнать координаты указателя мыши в момент нажатия не представляется возможным. Для получения подобного рода информации понадобятся объекты событий, часто содержащие дополнительную информацию, которой может воспользоваться объект, получающий события. В частности, в объекте события мыши QMouseEvent передаются координаты и код нажатой кнопки.

2.1. События клавиатуры QKeyEvent

Класс QKeyEvent содержит данные о событиях клавиатуры. С его помощью можно получить информацию о клавише, вызвавшей событие, а также ASCII-код отображенного символа (American Standard Code for Information Interchange, американский стандартный код для обмена информацией). Объект события передается в методы QWidget::keyPressEvent() и QWidget:: keyReleaseEvent (), определенные в классе QWidget. Событие может вызываться нажатием любой клавиши на клавиатуре, включая <Shift>, <Ctrl>, <Alt>, <Esc> и <F1>- <F12>. Исключение составляют клавиша табулятора <Tab> и ее совместное нажатие с клавишей <Shift>, которые используются методом обработки QWidget:: event () для передачи фокуса следующему виджету.

Метод keyPressEvent () вызывается каждый раз при нажатии одной из клавиш на клавиатуре, а метод keyReleaseEvent () — при ее отпускании.

В методе обработки события с помощью метода QKeyEvent:: key () можно определить, какая из клавиш его инициировала. Этот метод возвращает значение целого типа, которое можно сравнить с константами клавиш, определенными в классе Qt (таблица 13.1).

Таблица 13.1. Некоторые значения перечислений кеупространства
имен Qt

Константа	**Значение (HEX)**	**Константа**	**Значение (HEX)**	**Константа**	**Значение (HEX)**
Key_Space	20	Key_B	42	Key_Insert	1000006
Key_NumberSign	23	Key_C	43	Key_Delete	1000007
Key_Dollar	24	Key_D	44	Key_Pause	1000008
Key_Percent	25	Key_E	45	Key_Print	1000009
Ke y_Ampe r s	26	Key_F	46	Key_Home	1000010
Key_Apostrophe	27	Key_G	47	Key_End	1000011
Key_ParenLeft	28	Key_H	48	Key_Left	1000012
Key_ParenRight	29	Key_I	49	Key_Up	1000013
Key_Asterisk	2A	Key_J	4A	Key_Right	1000014
Key_Plus	2B	Key_K	4B	Key_Down	1000015
Key_Comma	2C	Key_L	4C	Key_PageUp	1000016
Key_Minus	2D	Key_M	4D	Key_PageDown	1000017

Key_Period	2E	Key_N	4E	Key_Shift	1000020
Key_Slash	2F	Key_O	4F	Key_Control	1000021
Key_0	30	Key_P	50	Key_Meta	1000022
Key_l	31	Key_Q	51	Key_Alt	1000023
Key_2	32	Key_R	52	Key_CapsLock	1000024
Key_3	33	Key_S	53	Key_NumLock	1000025
Key_4	34	Key_T	54	Key_ScrollLock	1000026
Key_5	35	Key_U	55	Key_Fl	1000030
Key_6	36	Key_V	56	Key_F2	1000031
Key_7	37	Key_W	57	Key_F3	1000032
Key_8	38	Key_X	58	Key_F4	1000033
Key_9	39	Key_Y	59	Key_F5	1000034
Key_Colon	ЗА	Key_Z	5A	Key_F6	1000035
Key_Semicolon	ЗВ	Key_Backslash	5C	Key_F7	1000036
Key_Less	ЗС	Key_Escape	1000000	Key_F8	1000037
Key_Equal	3D	Key_Tab	1000001	Key_F9	1000038
Key_Greater	ЗЕ	Key_Backspace	1000003	Key_F10	1000039
Key_Question	3F	Key_Return	1000004	Key_Fll	100003A
Key_A	41	Key_Enter	1000005	Key_F12	100003B

2.2. Событие обновления контекста рисования

Qt поддерживает *двойную буферизацию* (double buffering). Ее можно отключить вызовом метода QWidget::setAttribute(Qt::WA_PaintOnScreen). Вполне возможно, последствия вас удивят: дело в том, что некогда выведенная в окно графическая информация вдруг исчезнет при изменении размеров окна приложения или после перекрытия его окном другого приложения. Чтобы этого не произошло, необходимо получать и обрабатывать событие QPaintEvent. В объекте класса QPaintEvent передается информация для перерисовки всего изображения или его части. Событие возникает тогда, когда виджет впервые отображается на экране явным или неявным вызовом метода show (), а также в результате вызова методов repaint () и update (). Объект события передается в метод paintEvent (), в котором реализуется отображение самого виджета. В большинстве случаев, этот метод используется для полной перерисовки виджета. Для маленьких виджетов такой подход вполне приемлем, но, для виджетов больших размеров, рациональнее

перерисовывать только отдельпую область, действительно нуждающуюся в этом. Для получения координат и размеров такого участка вызывается метод region(). Вызовом метода contains() можно проверить, находится ли объект в заданной области.

2.3. События мыши Класс QMouseEvent

Объект этого класса содержит информацию о событии, вызванном действием мыши, и хранит в себе информацию о позиции указателя мыши в момент вызова события, статус кнопок мыши и даже некоторых клавиш клавиатуры. Этот объект передается в методы mousePressEvent(), mouseMoveEvent(), mouseReleaseEvent() И mouseDoubleClickEvent().

Метод mousePressEvent () вызывается тогда, когда произошло нажатие на одну из кнопок мыши в области виджета. Если нажать кнопку мыши и, не отпуская ее, переместить указатель мыши за пределы виджета, то он будет получать события мыши, пока кнопку не отпустят. При движении мыши станет вызываться метод mouseMoveEvent (), а при отпускании кнопки произойдет вызов метода mouseReleaseEvent ().

По умолчанию метод mouseMoveEvent () вызывается при перемещении указателя мыши, только если одна из ее кнопок нажата. Это позволяет не создавать лишних событий во время простого перемещения указателя. Если же необходимо отслеживать все перемещения указателя мыши, то нужно воспользоваться методом setMouseTracking() класса QWidget, передав ему параметр true.

Метод mouseDoubleClickEvent () вызывается при двойном щелчке кнопкой мыши в области виджета.

Для определения местоположения указателя мыши в момент возникновения события можно воспользоваться методами globalx (), globalY (), x () и y (), которые возвращают целые значения. Также можно воспользоваться методами pos () или globalPos (). Метод pos () класса QMouseEvent возвращает позицию указателя мыши в момент наступления события относительно левого верхнего угла виджета. Если нужна абсолютная позиция (относительно левого верхнего угла экрана), то ее получают с помощью метода globalPos ().

Вызвав метод button (), можно узнать, какая из кнопок мыши была нажата в момент наступления события (табл. 13.2). Метод buttons() возвращает битовую комбинацию из приведенных в табл.

13.2 значений. Как видно из этой таблицы, значения не пересекаются, поэтому можно применять операцию | (ИЛИ) для их объединения.

Таблица 13.2. Некоторые значения перечисления MouseButton пространства имен Qt

Константа	**Значение**	**Описание**
NoButton	0	Кнопки мыши не нажаты
LeftButton	1	Нажата левая кнопка мыши
RightButton	2	Нажата правая кнопка мыши
MidButton	4	Нажата средняя кнопка мыши

3. Порядок выполнения работы

3.1. Работы с событиями клавиатуры QKeyEvent

В данном примере реализован перехват нажатия клавиш на клавиатуре. Т.е. для виджета QNUmberLineEdit переопределен метод обработки событий от клавиатуры. В данном методе производится определение какая клавиша нажата, и если она принадлежит цифровой клавиатуре, то событие передается дальше.

Programm13_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm13_1
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp \
    qnumberlineedit.cpp
HEADERS  += widget.h \
    qnumberlineedit.h
```

main.cpp

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
```

```
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include "qnumberlineedit.h"

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    QVBoxLayout *mainLayout;
    QNUmberLineEdit *editNumber;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    editNumber = new QNUmberLineEdit(this);
```

```
    mainLayout->addWidget(editNumber);
}

Widget::~Widget()
{

}
```

qnumberlineedit.h

```
#ifndef QNUMBERLINEEDIT_H
#define QNUMBERLINEEDIT_H
#include <QLineEdit>
#include <QDebug>
#include <QKeyEvent>

class QNUmberLineEdit : public QLineEdit
{
public:
    QNUmberLineEdit(QWidget *parent);

private:
    virtual void keyPressEvent(QKeyEvent *event);
};

#endif // QNUMBERLINEEDIT_H
```

qnumberlineedit.cpp

```
#include "qnumberlineedit.h"

QNUmberLineEdit::QNUmberLineEdit(QWidget *parent)
:QLineEdit(parent)
{

}

void QNUmberLineEdit::keyPressEvent(QKeyEvent *event)
```

```
{
   qDebug()<<"Key pressed";

   if ((event->key()>0x2F)&&((event->key()<0x3A)))
         QLineEdit::keyPressEvent(event);
}
```

3.2. Работа с событиями обновления контекста рисования

В данном примере реализовано переопределение обработчика события обновления контекста рисования. При вызове обработчика в отладочную консоль выводится текущие размеры окна.

Programm31_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm13_2
TEMPLATE = app
SOURCES += main.cpp\
       widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>
int main(int argc, char *argv[]) {
   QApplication a(argc, argv);
   Widget w;
   w.show();

   return a.exec(); }
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QDebug>
#include <QPaintEvent>
#include <QPainter>
#include <QRect>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   void paintEvent(QPaintEvent *event);
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
}

Widget::~Widget()
{

}

void Widget::paintEvent(QPaintEvent *event)
{
   qDebug()<<"Paint event "<<event->rect().size();
}
```

3.3. Работа с событиями мыши QMouseEvent

Данный пример показывает перехват основных событий мыши. При возникновении события данные о нем выводятся на основную форму.

Programm11_3.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm13_3
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QLabel>
#include <QMouseEvent>
```

```
class Widget : public QLabel
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

protected:
   virtual void mousePressEvent  (QMouseEvent* event);
   virtual void mouseReleaseEvent(QMouseEvent* event);
   virtual void mouseMoveEvent   (QMouseEvent* event);

   void    dumpEvent     (QMouseEvent* event, const QString&
strMessage);
   QString modifiersInfo (QMouseEvent* event                  );
   QString buttonsInfo   (QMouseEvent* event                  );
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : QLabel(parent)
{
   setAlignment(Qt::AlignCenter);
   setText("Mouse interactions\n(Press a mouse button)");
}


Widget::~Widget()
{

}

void Widget::mousePressEvent(QMouseEvent* event)
{
```

```
    dumpEvent(event, "Mouse Pressed");
}

void Widget::mouseReleaseEvent(QMouseEvent* event)
{
    dumpEvent(event, "Mouse Released");
}

void Widget::mouseMoveEvent(QMouseEvent* event)
{
    dumpEvent(event, "Mouse Is Moving");
}


void Widget::dumpEvent(QMouseEvent* event, const QString& strMsg)
{
    setText(strMsg
            + "\n buttons()=" + buttonsInfo(event)
            + "\n x()=" + QString::number(event->x())
            + "\n y()=" + QString::number(event->y())
            + "\n globalX()=" + QString::number(event->globalX())
            + "\n globalY()=" + QString::number(event->globalY())
            + "\n modifiers()=" + modifiersInfo(event)
           );
}


QString Widget::modifiersInfo(QMouseEvent* event)
{
    QString strModifiers;

    if(event->modifiers() & Qt::ShiftModifier) {
        strModifiers += "Shift ";
    }
    if(event->modifiers() & Qt::ControlModifier) {
        strModifiers += "Control ";
    }
    if(event->modifiers() & Qt::AltModifier) {
        strModifiers += "Alt";
```

```
    }
    return strModifiers;
}


QString Widget::buttonsInfo(QMouseEvent* event)
{
    QString strButtons;

    if(event->buttons() & Qt::LeftButton) {
        strButtons += "Left ";
    }
    if(event->buttons() & Qt::RightButton) {
        strButtons += "Right ";
    }
    if(event->buttons() & Qt::MidButton) {
        strButtons += "Middle";
    }
    return strButtons;
}
```

3.5. Задание для самостоятельного выполнения

3.5.1. Программа «Виджет журнала»

На основе виджета текстового поля реализовать виджет журнала (лога). Данный виджет должен иметь слот, с входным параметром типа QString, при вызове которого текст, находящийся в параметре, заносится как новая строка в текстовое поле. В данном виджете должны быть отключены возможности ввода и удаления текста с клавиатуры, а также возможность выделения текста (как мышкой, так и клавиатурой) и копирование.

3.5.2. Программа «виджет консоли»

На основе текстового поля реализовать виджет консоли (командной строки). Строка приглашения должна содержать символы >> . В данный виджет должны вводиться только числа в 16-ной системе. При нажатии клавиши ввод должен вызываться сигнал, в

который передается последняя команда. А также сама команда должна дублироваться в поле виджета. Так же реализовать возможность выбора последних введенных команлд с помощью клавиш «стрелка вверх» и «стрелка вниз».

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Что такое события и для чего они предназначены?
2. События клавиатуры?
3. События мышки?
4. События перерисовки контекста?

Лабораторная работа № 14

Компоненты для работы с мультимедиа

Часть 1

1. Цель работы

Изучить основные компоненты для работы с графикой, а также методы их применения.

2. Основные теоретические сведения

Графика — одна из наиболее быстро развивающихся отраслей компьютерной индустрии. Известно, что 70 % информации человек воспринимает визуально, поэтому графика — это важнейший компонент для взаимодействия человека с компьютером.

Для программирования компьютерной графики часто используются такие классы геометрии, как точки, двумерные размеры, прямоугольники, а также специальные классы для хранения цветовых значений.

2.1. Классы геометрии

Группа классов геометрии ничего не отображает на экране. Основное их назначение состоит в задании расположения, размеров и в описании формы объектов.

Точка

Для задания точек в двумерной системе координат служат два класса: QPoint и QPointF. В двумерной системе координат точка обозначается парой чисел *X* и *Y*, где *X* — горизонтальная, а *Y* — вертикальная координаты. В отличие от обычного расположения координатных осей, при задании координат точки в Qt обычно подразумевается, что ось *Y* смотрит вниз (рис. 14.1).

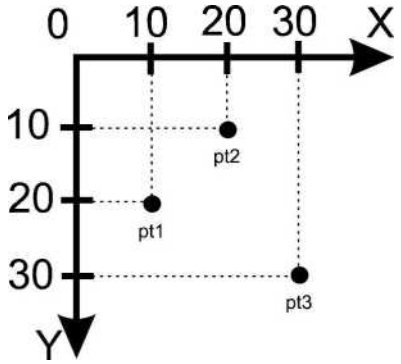

Рис. 14.1. Создание и сложение точек

Для получения координат точки *{X, Y)* реализованы методы x () и y () соответственно. Изменяются координаты точки с помощью методов setx () и setY ().

Можно получать ссылки на координаты точки, чтобы измеиять их значения.

Например:

```
QPoint pt(10, 20);
pt.rx() += 10; // pt = (20, 20)
```

Объекты точек можно сравнивать друг с другом при помощи операторов == (равно) и ! = (не равно).

Например:

```
QPoint ptl(10, 20) ;
QPoint pt2(10, 20);
bool b = (ptl == pt2); // b = true
```

Если необходимо проверить, равны ли координаты *X* и *Y* нулю, то вызывается метод isNull ().

Например:

```
QPoint pt; // (0, 0)
bool b = pt.isNull(); // b = true
```

Метод manhattanLength () возвращает сумму абсолютных значений координат *X* и *Y*.

Например:

```
QPoint pt(10, 20);
```

int n = pt.manhattanLength(); // n = 10 + 20 = 30

Этот метод был назван в честь улиц Манхэттена, расположенных перпендикулярно друг к другу.

Двумерный размер

Классы QSize и QSizeF служат для хранения целочисленных и вещественных размеров. Оба класса обладают одинаковыми интерфейсами. Структура их очень похожа на QPoint, так как хранит две величины, над которыми можно проводить операции сложения/вычитания и умножения/деления.

Классы QSize и QSizeF, как и классы QPoint, QPointF, предоставляют операторы сравнения ==, ! = и метод isNull (), возвращающий значение true в том случае, если высота и ширина равны нулю.

Для получения ширины и высоты вызываются методы width () и height (). Изменить эти параметры можно с помощью методов setwidth () и setHeight (). При помощи методов rwidth () и rheight () получают ссылки на значения ширины и высоты. Например:

QSize size(10, 20) ;
int n = size.rwidth()++; // n = 11; size = (11, 20).

Помимо них, класс предоставляет метод scale (), позволяющий изменять размеры оригинала согласно переданному в первом параметре размеру. Второй параметр этого метода управляет способом изменения размера, а именно:

- Qt:: ignoreAspectRatio — изменяет размер оригинала на переданный в него размер;
- Qt:: KeepAspectRatio — новый размер заполняет заданную площадь, насколько это будет возможно с сохранением пропорций оригинала;
- Qt:: KeepAspectRatioByExpanding — новый размер может находиться за пределами переданного в scale (), заполняя всю его площадь.

Прямоугольник

Классы QRect и QRectF служат для хранения целочисленных и вещественных координат прямоугольных областей (точка и размер) соответственно. Задать прямоугольную область можно, например, передав в конструктор точку (верхний левый угол) и размер. Область, приведенная на рисунке 14.2, создается при помощи следующих строк:

```
QPoint pt(10, 10);
QSize size(20, 10);
QRect r(pt, size).
```

Получить координаты *X* левой грани прямоугольника или *Y* верхней можно при помощи методов x () или y () соответственно. Для изменения этих координат нужно воспользоваться методами setx () и setY ().

Размер получают с помощью метода size(), который возвращает объект класса QSize. Можно просто вызвать методы, возвращающие составляющие размера: ширину width () и высоту height(). Изменить размер можно методом setSize(), а каждую его составляющую — методами setwidth () и setHeight ().

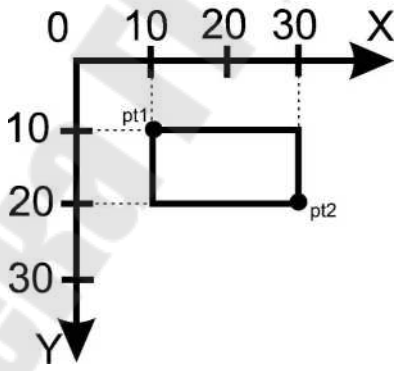

Рис. 14.2 Прямоугольная область

Прямая линия

Классы QLine и QLineF описывают прямую линию или, правильней сказать, отрезок на плоскости в целочисленных и вещественных координатах соответственно. Позиции начальной точки можно получить при помощи методов xi () и yi (), а конечной — x2 () и y2 (). Аналогичного результата можно добиться вызовами pi ()

и p2 (), которые возвращают объекты класса QPoint/QPointF, описанные ранее. Методы dx() и dy() возвращают величины горизонтальной и вертикальной проекций прямой на оси *X* и *Y* соответственно. Прямую, показанную на рисунке 14.3, можно создать при помощи одной строки кода:

```
QLine line(10, 10, 30, 20);
```

Оба класса — QLine и QLineF — предоставляют операторы сравнения ==, != и метод isNull(), возвращающий логическое значение true в том случае, когда начальная и конечная точки не установлены.

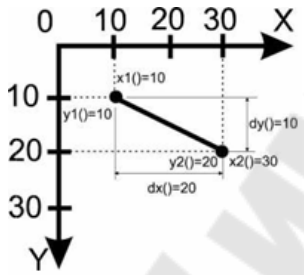

Рис. 14.3. Прямая линия

Многоугольник

Многоугольник (или полигон) — это фигура, представляющая собой замкнутый контур, образованный ломаной линией. В Qt эту фигуру реализуют классы QPolygon и QPolygonF, в целочисленном и вещественном представлении соответственно. По своей сути эти классы являются массивами точек QVector<QPoint> и QVector<QPointF>. Самый простой способ инициализации объектов класса полигона — это использование оператора потока вывода «. Треугольник представляет собой самую простую форму многоугольника (рисунок 14.4), а его создание выглядит следующим образом:

```
QPolygon polygon;
polygon « QPoint(10, 20) « QPoint(20, 10) « QPoint(30, 30);
```

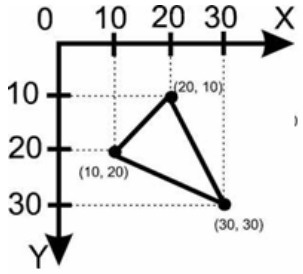

Рис. 14.4. Задание многоугольника

2.2. Цвет (класс QColor)

Цвета, которые вы видите, есть не что иное, как свойство объектов материального мира, воспринимаемое нами как зрительное ощущение от воздействия света, имеющего различные электромагнитные частоты. На самом деле, человеческий глаз в состоянии воспринимать очень малый диапазон этих частот. Цвет с наибольшей частотой, которую в состоянии воспринять глаз, – фиолетовый, а с наименьшей – красный. Но даже в таком небольшом диапазоне находятся миллионы цветов. Одновременно человеческий глаз может воспринимать около 10 тысяч различных цветовых оттенков.

Цветовая модель — это спецификация в трехмерной или четырехмерной системе координат, которая задает все видимые цвета. В Qt поддерживаются три цветовые модели: *RGB* (Red, Green, Blue— красный, зеленый, голубой), *CMYK* (Cyan, Magenta, Yellow и Key color— голубой, пурпурный, желтый и «ключевой» черный цвет) и *HSV* (Hue, Saturation, Value — оттенок, насыщенность, значение).

С помощью класса QColor можно сохранять цвета моделей RGB и HSV. Его определение находится в заголовочном файле QColor. Объекты класса QColor можно сравнивать при помощи операторов == и ! =, присваивать и создавать копии.

Наиболее чувствителен глаз к зеленому цвету, потом следует красный, а затем – синий. На этих трех цветах и построена модель RGB (Red, Green, Blue – красный, зеленый, синий). Пространство цветов задает куб, длина ребер которого равна 255 в целочисленном числовом представлении (либо единице в вещественном представлении).

Первый параметр задает оттенок красного, второй — зеленого, а третий — оттенок синего цвета. Диагональ куба, идущая от черного

цвета к белому, – это оттенки серого цвета. Диапазон каждого из трех значений может изменяться в пределах от 0 до 255 (либо от 0 до 1 в вещественном представлении), где 0 означает полное отсутствие оттенка цвета, а 255 – его максимальную насыщенность.

Эта модель является «аддитивной», то есть посредством смешивания базовых цветов в различном процентном соотношении можно создать любой нужный цвет. Смешав, например, синий и зеленый, мы получим голубой цвет.

Для создания цветового значения RGB нужно просто передать в конструктор класса QCoior три значения. Каналы цвета класса QCoior могут содержать необязательный уровень прозрачности, значение для которого можно передавать в конструктор четвертым параметром. В первый параметр передается значение красного цвета, во второй – зеленого, в третий – синего, а в четвертый – уровень прозрачности. Например:

```
QColor colorBlue(0, 0, 255, 128);
```

Получить из объекта QColor каждый компонент цвета возможно с помощью методов red (), green (), blue () и alpha (). Эти же значения можно получить и в вещественном представлении, для чего нужно вызвать: redF (), greenF (), blueF () и alphaF (). Можно вообще обойтись одним методом getRgb (), в который передаются указатели на переменные для значений цветов, например:

```
QColor color(100, 200, 0); int r, g, b;
color.getRgb(&r, &g, &b);
```

Для записи значений RGB можно, по аналогии, воспользоваться методами, похожими на описанные, но имеющими префикс set (вместо префикса get, если он есть), после которого идет заглавная буква. Также для этой цели можно прибегнуть к структуре данных QRgb, которая состоит из четырех байтов и полностью совместима с 32-битным значением. Эту структуру можно создать с помощью функции qRgb () или qRgba (), передав в нее параметры красного, зеленого и синего цветов. Но можно присваивать переменным структуры QRgb и 32-битное значение цвета. Например, синий цвет устанавливается сразу несколькими способами:

```
QRgb rgbBluel = qRgba(0, 0, 255, 255); // С информацией о
        прозрачности
QRgb rgbBlue2 = qRgb(0, 0, 255);
QRgb rgbBlue3 = 0x000000FF;
```

При помощи функций qRed (), qGreen (), qBlue () и qAlpha () можно получить значения цветов и информацию о прозрачности соответственно.

Значения типа QRgb можно передавать в конструктор класса QColor или в метод setRgb ():

```
QRgb rgbBlue = 0x000000FF;
QColor colorBluel(rgbBlue);
QColor colorBlue2; colorBlue2.setRgb(rgbBlue);
```

Также можно получать значения структуры QRgb от объектов класса QColor вызовом метода rgb ().

Цвет можно установить, передав значения в символьном формате, например:

```
QColor colorBluel("#0000FF") ;
QColor colorBlue2;
colorBlue2.setNameColor("#0000FF");
```

Модель HSV (Hue, Saturation, Value — оттенок, насыщенность, значение) не смешивает основные цвета при моделировании нового цвета, как в случае с RGB, а просто измеияет их свойства. Это очень напоминает принцип, используемый художниками для получения, новых цветов, — подмешивая к чистым цветам белую, черную или серую краски.

Пространство цветов этой модели задается пирамидой с шестиконечным основанием, так называемым Hexcone. Координаты в этой модели имеют следующий смысл:

- оттенок (Hue)— это «цвет» в общеупотребительном смысле этого слова, например, красный, оранжевый, синий и т. д., который задается углом в цветовом круге, изменяющемся от 0 до 360 градусов;
- насыщенность (Saturation) обозначает наличие белого цвета в оттенке. Значение насыщенности может изменяться в

диапазоне от 0 до 255. Значение, равное 255 в целочисленном числовом представлении либо единице в вещественном представлении, соответствует полностью насыщенному цвету, который не содержит оттенков белого. Частично насыщенный оттенок светлее – например, оттенок красного с насыщенностью, равной 128, либо 0,5 в вещественном представлении, соответствует розовому;

- значение (Value) или яркость – определяет интенсивность цвета. Цвет с высокой интенсивностью – яркий, а с низкой – темный. Значение этого параметра может изменяться в диапазоне от 0 до 255 в целочисленном числовом представлении (либо от 0 до 1 в вещественном представлении).

Установку значения цвета в координатах HSV можно выполнить с помощью метода QColor::setHsv() ИЛИ QColor::setHsvF():

```
color.setHsv(233, 100, 50);
```

Для того чтобы получить цветовое значение в цветовой модели HSV, нужно передать в метод getHsv () адреса трех целочисленных значений (или вещественных, если это getHsvF ()). Следующий пример устанавливает RGB-значение и получает в трех переменных его HSV-эквивалент:

```
QColor color(100, 200, 0); int h, s, v;
color.getHsv(&h, &s, &v);
```

2.3. Класс QPainter

Класс QPainter, определенный в заголовочном файле QPainter, является исполнителем команд рисования. Он содержит массу методов для отображения линий, прямоугольников, окружностей и др. Рисование осуществляется на всех объектах классов, унаследованных от класса QPaintDevice. Это означает, что то, что отображается контекстом рисования одного объекта, может быть точно так же отображено контекстом и другого объекта.

Чтобы использовать объект QPainter, необходимо передать ему адрес объекта контекста, на котором должно осуществляться рисование. Этот адрес можно передать как в конструкторе, так и с помощью метода QPainter:: begin (). Смысл метода begin () состоит в

том, что он позволяет рисовать на одном контексте несколькими объектами класса QPainter. При использовании метода begin() нужно по окончании работы с контекстом вызвать метод QPainter:: end (), чтобы отсоединить установленную этим методом связь с контекстом рисования, давая возможность для рисования другому объекту.

```
QPainter painterl;
QPainter painter2;
painterl.begin(this);
// Команды рисования painterl.end();
painter2.begin(this);
// Команды рисования painter2.end();
```

Но чаще всего используется рисование одним объектом QPainter в разных контекстах:

```
QPainter painter;
painter.begin(this);
//контекст виджета
// Команды рисования painter.end();
QPixmap pix(rect());
painter.begin(&pix);
//контекст растрового изображения
// Команды рисования
painter.end();
```

Объекты QPainter содержат установки, влияющие на их рисование. Это могут быть трансформация координат, модификация кисти и пера, установка шрифта, установка режима сглаживания и т. д. Поэтому, для того чтобы оставить старые настройки без изменений, рекомендуется перед началом изменения сохранить их с помощью метода QPainter:: save (), а по окончании работы — восстановить с помощью метода QPainter: :restore ().

3. Порядок выполнения работы

3.1. Рисование геометрических объектов

В данном листинге приведен пример рисования простых геометрических объектов на Qwidget.

PainterPath.pro

```
TEMPLATE     = app
QT           += widgets
SOURCES      = main.cpp
windows:TARGET = ../PainterPath
```

main.cpp

```
#include <QtWidgets>

class PainterPathWidget : public QWidget {
protected:
   virtual void paintEvent(QPaintEvent*)
   {
      QPainterPath path;
      QPointF      pt1(width(), height() / 2);
      QPointF      pt2(width() / 2, -height());
      QPointF      pt3(width() / 2, 2 * height());
      QPointF      pt4(0, height() / 2);
      path.moveTo(pt1);
      path.cubicTo(pt2, pt3, pt4);

      QRect rect(width() / 4, height() / 4, width() / 2, height() / 2);
      path.addRect(rect);
      path.addEllipse(rect);

      QPainter painter(this);
      painter.setRenderHint(QPainter::Antialiasing, true);
      painter.setPen(QPen(Qt::blue, 6));
      painter.drawPath(path);
   }
```

```
public:
   PainterPathWidget(QWidget* pwgt = 0) : QWidget(pwgt)
   {
   }
};

int main(int argc, char** argv)
{
   QApplication app(argc, argv);

   PainterPathWidget wgt;

   wgt.resize(200, 200);
   wgt.show();

   return app.exec();
}
```

3.2. Наложение объектов

В данной программе реализован алгоритм отрисовки геометрических фигур с наложением друг на друга в различных режимах.

Programm31_2.pro

```
TEMPLATE   = app
QT         += widgets
SOURCES              = main.cpp
windows:TARGET     = ../CompositionModes
```

main.cpp

```
#include <QtWidgets>

QLabel* lbl(const QPainter::CompositionMode& mode)
{
```

```
    QLabel* plbl = new QLabel;
    plbl->setFixedSize(100, 100);

    QRect    rect(plbl->contentsRect());
    QPainter painter;

    QImage sourceImage(rect.size(),
QImage::Format_ARGB32_Premultiplied);
    painter.begin(&sourceImage);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(QColor(0, 255, 0)));
    painter.drawPolygon(QPolygon() << rect.bottomLeft()
                           << QPoint(rect.center().x(), 0)
                           << rect.bottomRight()
                    );
    painter.end();

    QImage resultImage(rect.size(),
QImage::Format_ARGB32_Premultiplied);
    painter.begin(&resultImage);
    painter.setRenderHint(QPainter::Antialiasing, true);

painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
    painter.setPen(QPen(QColor(0, 255, 0), 4));
    painter.setBrush(QBrush(QColor(255, 0, 0)));
    painter.drawEllipse(rect);
    painter.setCompositionMode(mode);
    painter.drawImage(rect, sourceImage);
    painter.end();

    plbl->setPixmap(QPixmap::fromImage(resultImage));
    return plbl;
}


int main(int argc, char** argv)
{
  QApplication app(argc, argv);
  QWidget      wgt;
```

```
  //Layout Setup
  QGridLayout* pgrd = new QGridLayout;
  pgrd->addWidget(lbl(QPainter::CompositionMode_Source), 0, 0);
  pgrd->addWidget(new QLabel("<CENTER>Source</CENTER>"), 1,
0);
  pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOver), 0, 1);
  pgrd->addWidget(new QLabel("<CENTER>SourceOver</CENTER>"),
1, 1);
  pgrd->addWidget(lbl(QPainter::CompositionMode_SourceIn), 0, 2);
  pgrd->addWidget(new QLabel("<CENTER>SourceIn</CENTER>"), 1,
2);
  pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOut), 0, 3);
  pgrd->addWidget(new QLabel("<CENTER>SourceOut</CENTER>"),
1, 3);
  pgrd->addWidget(lbl(QPainter::CompositionMode_SourceAtop), 0, 4);
  pgrd->addWidget(new QLabel("<CENTER>SourceAtop</CENTER>"),
1, 4);
  pgrd->addWidget(lbl(QPainter::CompositionMode_Clear), 0, 5);
  pgrd->addWidget(new QLabel("<CENTER>Clear</CENTER>"), 1, 5);
  pgrd->addWidget(lbl(QPainter::CompositionMode_Destination), 2, 0);
  pgrd->addWidget(new QLabel("<CENTER>Destination</CENTER>"),
3, 0);
  pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationOver), 2,
1);
  pgrd->addWidget(new
QLabel("<CENTER>DestinationOver</CENTER>"), 3, 1);
  pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationIn), 2, 2);
  pgrd->addWidget(new
QLabel("<CENTER>DestinationIn</CENTER>"), 3, 2);
  pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationOut), 2,
3);
  pgrd->addWidget(new
QLabel("<CENTER>DestinationOut</CENTER>"), 3, 3);
  pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationAtop), 2,
4);
  pgrd->addWidget(new
QLabel("<CENTER>DestinationAtop</CENTER>"), 3, 4);
  pgrd->addWidget(lbl(QPainter::CompositionMode_Xor), 2, 5);
```

```
  pgrd->addWidget(new QLabel("<CENTER>Xor</CENTER>"), 3, 5);
  wgt.setLayout(pgrd);

  wgt.show();

  return app.exec();
}
```

3.3. Задание для самостоятельного выполнения

Разработать программу, которая на основной форме рисует государственные флаги разных стран. Выбор страны осуществляется в Qcomobox над областью рисования. При изменении размера формы, размер рисунка так же должен изменяться. Количество стран не должно быть меньше 5.

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Для чего предназначен класс Qpoint?
2. В чем отличие класса Qpoint от QpointF?
3. Какие классы предназначены для хранения размера?
4. Для чего предназначен класс QrectF?
5. В чем отличие класса QrectF от класса QRect?
6. Какой класс предназначен для отрисовки прямой линии?
7. Какой класс предназначен для отрисовки многоугольника?
8. Какие две системы кодирования цветов используются в библиотеках Qt?

Часть 2

1. Цель работы

Изучить основы работы со звуковыми устройствами. Освоить на практике алгоритмы записи и воспроизведения звуковых сигналов.

2. Основные теоретические сведения

Звуковая карта компьютера по своей сути является устройствао с АЦП и ЦАП. АЦП – аналогоцифровой преобразователь, предназначенный для оцифровки (записи) звука с микрофона или линейного входа. ЦАП – цифроаналоговый преобразователь и предназначен для преобразования цифрового сигнала в аналоговый, т. е. воспроизводить звук. В следствии чего работа со звуковой картой аналогична работе с АЦП и ЦАП. Звук в цифровом формате представляет собой массив отсчетов заданной битности и с заданной частотой дескритизации.

Отсчет по своей сути является моментальным значением напряжения аналогового сигнала. Однако данное значение не является напряжением в прямом понимании. Фактически это количество шагов квантования, которое наиболее близко по своему значению к напряжению аналогового сигнала. Размер шага квантования зависит от разрядности устройства и значения опорного напряжения. Процесс оцифровки сигнала приведен на рисунке 14.5.

К основным параметрам оцифровки сигнала можно отнести:

- Разрядность;
- Частота дискретизации;
- Порядок следования байтов.

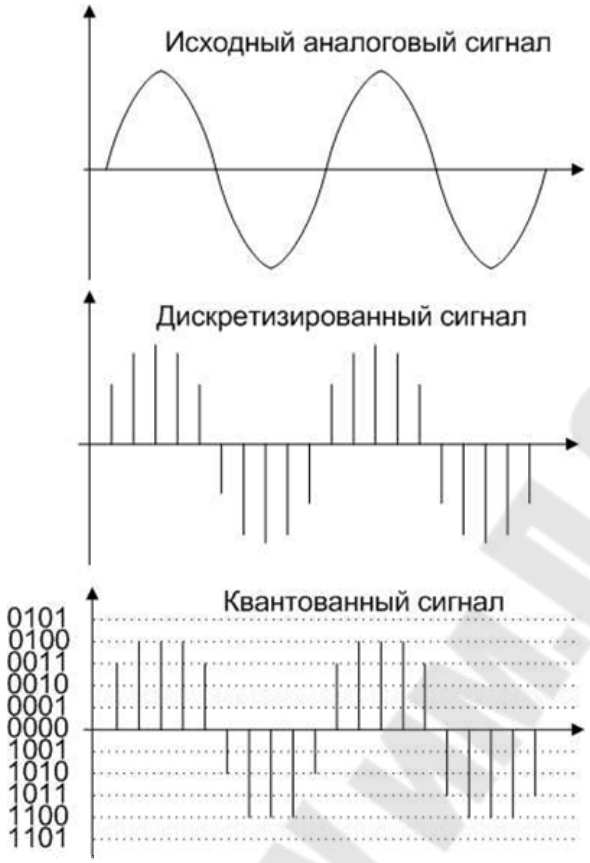

Рис. 14.5. Процесс оцифровки сигнала

2.1. Класс QIODevice

Данный класс является АТД (Абстрактным Типом Данных, т.е. на основе данного класса нельзя создать объект непосредственно. Данный класс предназначен для наследования и создания на основе него собственных классов.

Имя класса расшифровывается как Qt Input Output Device и предназначен для создания различных классов для работы с устройствами ввода и вывода. Такой подход применен для того чтобы разлчиные готовые классы могли взаимодействовать с устройствами ввода вывода через унифицированный интерфейс.

Данный класс содержит множество виртуальных методов, которые могут быть переопределены в зависимости от той или иной задачи. Устройства могут быть как блочными, так и последовательными. К блочным устройствам можно отнести файл, т. е. блочное устройство позволяет получать произвольный доступ к данным, например, можно прочитать сначала первый байт, потом пятый, а потом третий. Серийные устройства не позволяют в

произвольном порядке читать и записывать данные. В таких устройствах все данные располагаются последовательно и после чтения удаляются из буфера.

К наиболее распространенным наследникам QIODevice можно отнести такие классы как:

- QTCPSocket;
- QFile;
- QProcess;
- QSerialPort;
- QBuffer.

Виртуальные открытые методы класса предназначены для реализации общего интерфейса для всех устройств ввода вывода и могут переопределяться в зависимости от задачи. Основные открытые виртуальные методы приведены в таблице 14.1.

Таблица 14.1. Основные виртуальные методы QIODevice

Метод	**Описание**
~QIODevice()	Деструктор класса, как и всех абстрактных типах данных он виртуальный
bool atEnd() const	Данный метод проверяет, остались ли еще данные для чтения. Если данные есть, то возвращается true
qint64 bytesAvailable() const	Возвращает количество байт доступных для чтения
qint64 bytesToWrite() const	Возвращает количество байт, которое еще может быть записано
void close()	Закрывает устройство
bool is Sequential() const	Возвращает true если устройство является последовательным
bool open(QIODevice::OpenMode mode)	Открывает устройство
qint64 pos() const	Возвращает текущую позицию в данныхС
bool reset()	Сбрасывает все параметры к исходным значениям

bool seek(qint64 pos)	Устанавливает позицию в данных
qint64 size() const	Возвращает размер устройства
bool waitForBytesWritten(int msecs)	Ожидание окончания записи данных
bool waitForReadyRead(int msecs)	Ожидание чтения данных
bool canReadLine() const	Возвращает true если может быть прочитана строка данных

Помимо открытых виртуальных методов существуют и защищенные виртуальные методы. Они обеспечивают внутреннюю логику работы класса. Данные методы рассмотрены в таблице 14.2.

Таблица 14.2. Защищенные виртуальные методы класса QIODevice

Метод	**Описание**
qint64 readData(char *data, qint64 maxSize)	Данный метод предназначен для считывания данных. При вызове данного метода данные должны помещаться в буфер, адрес которого располагается в указателе data. Объем считываемых данных не должен превышать размер maxSize в байтах. Так же данный метод должен возвращать фактическое количество байт данных, помещенных в буфер
qint64 readLineData(char *data, qint64 maxSize)	Данный метод предназначен для считывания строки данных и по своему функционалу аналогичен предыдущему
qint64 writeData(const char *data, qint64 maxSize)	Данный метод предназначен для записи данных в класс. Данные располагаются во внешнем буфере, на который ссылается указатель data. Объем данных в буфере равен maxSize. Данный метод должен

	возвращать фактическое количество байт данных которые были записаны в устройство, т.к. все данные находящиеся в буфере могут фактически не поместиться в устройство.

Данные методы должны быть обязательно переопределены для реализации своего класса устройства, иначе ваш класс устройства будет неработоспособен.

Применение механизма наследования от класса QIODevice при реализации классов различных устройств ввода вывода хорошо тем, что ваш класс автоматически будет совместим с классами, которые уже реализованы. Например, классы QaudioInput и QAudioOutput, которые будут рассмотрены ниже, могут записывать свои данные в класс типа QIODevice, а следоватеьно без различных доработок могут взаимодействовать со стандартными классами наследниками QIODevice (QTCPSocket, QFile и т. д.), так и с пользовательскими классами.

2.2 Класс QAudioFormat

Данный класс предназначен для установки параметров звукового устройства, как для записи звука, так и для его воспроизведения. В данном классе есть методы, позволяющие задавать как основные параметра оцифровки звука, так и дополнительные. В таблице 14.3 приведены основные методы класса QAudioFormat.

Таблица 14.3. Основные методы класса QAudioFormat

Метод	**Описание**
QAudioFormat::Endian byteOrder()	Возвращает установленный порядок следования байтов
QString codec()	Возращает имя установленного кодека
int channelCount()	Возвращает количество каналов
int sampleRate()	Возвращает частоту дискретизации
int sampleSize()	Возвращает разрядность
QAudioFormat::SampleType sampleType()	Возвращает тип отсчета
void setByteOrder(QAudioFormat::Endian byteOrder)	Устанавливает порядок следования байтов
void setChannelCount(int channels)	Устанавливает количество каналов
void setCodec(const QString &codec)	Устанавливает кодек
void setSampleRate(int samplerate)	Устанавливает частоту дескритизации
void setSampleSize(int sampleSize)	Устанавливает разрядность
void setSampleType(QAudioFormat::SampleType sampleType)	Устанавливает тип отсчета

Пример использования:

```
QAudioFormat format;
format.setSampleRate(8000);
format.setSampleSize(16);
format.setChannelCount(1);
format.setCodec("audio/pcm");
format.setByteOrder(QAudioFormat::LittleEndian);
format.setSampleType(QAudioFormat::SignedInt).
```

В данном примере частота дискретизации равна 8000 Гц, разрядность – 16 бит, количество каналов – 1, имя кодека –

audio/pcm, порядок следования байтов – от младшего к старшему, тип отсчета – знаковый целочисленный тип.

2.3. Класс QAudioInput

Данный класс предназначен для оцифровки звукового сигнала со входа звуковой карты компьютера. Для работы с данным классом надо обязательно задать параметры звукового устройства и устройство, в которое будет производиться запись звука.

Основные открытые методы класса приведены в таблице 14.4.

Таблица 14.4. Основные методы класса QAudioInput

Метод	**Описание**
QAudioFormat format()	Возвращает установленные параметры звукового устройства
void reset()	Сбрасывает устройство
void setBufferSize(int value)	Устанавливает размер внутреннего буфера
void setVolume(qreal volume)	Устанавливает громкость аудиоустройства
void start(QIODevice *device)	Данный метод запускает работу с устройством, в качестве входного параметра необходимо задать устройство, в которое будет производиться запись данных
QAudio::State state()	Возвращает текущее состояние аудиоустройства
void suspend()	Приостанавливает работу устройства
qreal volume()	Возращает текущее значение громкости

Пример использования:

```
QFile destinationFile;   // Class member
QAudioInput* audio; // Class member
```

```
{
    destinationFile.setFileName("test.raw");
    destinationFile.open( QIODevice::WriteOnly | QIODevice::Truncate );

    QAudioFormat format;
    // Set up the desired format, for example:
    format.setSampleRate(8000);
    format.setChannelCount(1);
    format.setSampleSize(8);
    format.setCodec("audio/pcm");
    format.setByteOrder(QAudioFormat::LittleEndian);
    format.setSampleType(QAudioFormat::UnSignedInt);

    QAudioDeviceInfo info = QAudioDeviceInfo::defaultInputDevice();
    if (!info.isFormatSupported(format)) {
        qWarning() << "Default format not supported, trying to use the
nearest.";
        format = info.nearestFormat(format);
    }

    audio = new QAudioInput(format, this);
    connect(audio, SIGNAL(stateChanged(QAudio::State)), this,
SLOT(handleStateChanged(QAudio::State)));

    QTimer::singleShot(3000, this, SLOT(stopRecording()));
    audio->start(&destinationFile);
    // Records audio for 3000ms
}
```

В данном примере реализован алгоритм, который захватывает звук со входа звуковой карты в течение 3 секунд и записывает его в файл test.raw.

2.4 Класс QAudioOutput

Данный класс выполняет обратную функцию: воспроизводит цифровой сигнал через выход звуковой карты. Так же, как и для предудыщего класса, для работы с QAudioOutput необходимо задать параметры звукового устройства и устройство с данными для воспроизведения. Основные методы данного класса аналогичны методам класса QaudioInput, поэтому нет смысла рассматривать их отдельно.

Пример использования:

```
QFile sourceFile;   // class member.
QAudioOutput* audio; // class member.
{
    sourceFile.setFileName("test.raw");
    sourceFile.open(QIODevice::ReadOnly);

    QAudioFormat format;
    // Set up the format, eg.
    format.setSampleRate(8000);
    format.setChannelCount(1);
    format.setSampleSize(8);
    format.setCodec("audio/pcm");
    format.setByteOrder(QAudioFormat::LittleEndian);
    format.setSampleType(QAudioFormat::UnSignedInt);

    QAudioDeviceInfo info(QAudioDeviceInfo::defaultOutputDevice());
    if (!info.isFormatSupported(format)) {
```

```
        qWarning() << "Raw audio format not supported by backend, cannot play audio.";
        return;
    }
 audio = new QAudioOutput(format, this);
    audio->start(&sourceFile);
 }
```

В данном примере производится воспроизведение цифровой аудиозаписи, расположенной в файле test.raw. Стоит отметить, что для удачного воспроизведения настройки аудиоустройства, через которое происходит воспроизведение звука, должны соответствовать настройкам устройства, с помощью которого производилась запись звука.

2.5 Класс QBuffer

Данный класс является прямым потомком QIODevice. В качестве места хранения информации используется класс QbyteArray, который располагается в оперативной памяти. Таким образом с помощью данного класса реализовано устройство, которое хранит свои данные в оперативной памяти и позволяет другим классам работать с ним через стандартный интерфейс QIODevice. Большинство методов данного класса позаимствованы у родительского класса. Уникальные открытые методы описаны в таблице 14.5.

Таблица 14.5. Основные методы класса QBuffer

Метод	**Описание**
QBuffer(QByteArray *byteArray, QObject *parent = nullptr)	Конструктор класса. В качестве входного параметра можно задать QbyteArray, в котором будут храниться данные
QByteArray &buffer()	Возвращает буфер с данными
void setBuffer(QByteArray *byteArray)	Устанавливает буфер для данных

void setData(const QByteArray &data)	Заполняет буфер данными из data
void setData(const char *data, int size)	Заполняет буфер данными из data

3. Порядок выполнения работы

3.1. Пример программы для работы с аудиоустройствами

Рассмотрим пример программы для работы с аудиоустройствами. В данном примере реализована программа, записывающая 3 секунды аудиосигнала с микрофона, записывает его в буфер, а затем воспроизводит через аудиовыход звуковой карты.

Файл LR14_1.pro

```
QT       += core gui multimedia
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = LR14_1
TEMPLATE = app
DEFINES += QT_DEPRECATED_WARNINGS
deprecated before Qt 6.0.0

SOURCES += main.cpp widget.cpp
HEADERS += widget.h
```

Файл main.cpp

```
#include "widget.h"
#include <QApplication>
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
```

```
    Widget w;
    w.show();
    return a.exec(); }
```

Файл widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QHBoxLayout>
#include <QPushButton>
#include <QAudioInput>
#include <QAudioOutput>
#include <QAudioFormat>
#include <QBuffer>
#include <QByteArray>
#include <QTimer>

class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
private:
    QAudioInput *input;
    QAudioOutput *output;
    QAudioFormat formatAudio;
```

```
    QBuffer *buffer;
    QByteArray data;
    QHBoxLayout *mainLayout;
    QPushButton *buttonRecord;
    QPushButton *buttonPlay;

private slots:
    void slotPushRecord();
    void slotPushPlay();
    void slotTimeOut();
    void slotStateChanged(QAudio::State state); };
#endif // WIDGET_H
```

Файл widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent) : QWidget(parent) {
    formatAudio.setSampleRate(8000);
    formatAudio.setSampleSize(16);
    formatAudio.setChannelCount(1);
    formatAudio.setCodec("audio/pcm");
    formatAudio.setByteOrder(QAudioFormat::LittleEndian);
    formatAudio.setSampleType(QAudioFormat::SignedInt);

    input = new QAudioInput(formatAudio,this);
    output = new QAudioOutput(formatAudio,this);
```

```
connect(output,SIGNAL(stateChanged(QAudio::State)),this,SLOT(slotStat
eChanged(QAudio::State)));
    buffer = new QBuffer(&data,this);
    buffer->open(QIODevice::ReadWrite);


    mainLayout = new QHBoxLayout(this);
    buttonRecord = new QPushButton(tr("Record"),this);

connect(buttonRecord,SIGNAL(pressed()),this,SLOT(slotPushRecord()));
    mainLayout->addWidget(buttonRecord);
    buttonPlay = new QPushButton(tr("Play"),this);
    connect(buttonPlay,SIGNAL(pressed()),this,SLOT(slotPushPlay()));
    mainLayout→addWidget(buttonPlay); }

void Widget::slotPushRecord() {
    buttonPlay->setEnabled(false);
    input->start(buffer);
    Qtimer::singleShot(3000,this,SLOT(slotTimeOut())); }

void Widget::slotPushPlay() {
    buffer->seek(0);
    buttonRecord->setEnabled(false);
    output→start(buffer); }

void Widget::slotTimeOut() {
    buttonPlay->setEnabled(true);
    input→stop(); }
```

```
void  Widget::slotStateChanged(QAudio::State state) {
   if (state == QAudio::State::IdleState) {
      output->stop();
      buttonRecord->setEnabled(true); }
}
```

В конструкторе класса основного окна производиться инициализация аудиоустройств и построение графического интерфейса. Графический интерфейс программы представляет собой основное окно с двумя горизонтально расположенными кнопка. При нажатии на первую кнопку производиться запись звукового сигнала длиной 3 секунды, а при нажатии второй кнопки производится воспроизведение записанного сигнала.

Оцифрованный звуковой сигнал храниться в оперативной памяти с помощью объекта класса QBuffer и динамического массива QbyteArray.

При нажатии кнопки записи срабатывает слот slotPushRecord. В нем производится блокирование кнопки воспроизведения, запуск объекта класса QAudioInput для захвата звука и запуск таймера, который и определяет длительность записываемого звукового сигнала. При срабатывании таймера вызывается слот slotTimeOut, в нем разблокируется кнопка воспроизведения и останавливается запись звука. При нажатии кнокпи воспроизведения срабатывает слот slotPushPlay. В нем производится перемещение позиции буфера в 0, для того чтобы воспроизведение было начато сначала, блокирование кнопки записи и запуск воспроизведения. Окончания воспроизведения определяется по изменению состояния объекта класса QAudioOutput в слоте slotStateChanged. Если состояние воспроизводящего объекта становится равным IdleState (воспроизведение окончено), то останавливается воспроизводящее устройство и разблокируется кнопка записи.

3.2. Задание для самостоятельной работы

В данной лабораторной работе необходимо самостоятельно реализовать две программы. В первой должен быть реализован диктофон, позволяющий делать голосовые записи, сохранять их в файлы и затем прослышивать выбранную запись.

Во второй программе необходимо реализовать генератор аудиосигнала с частотой от 100 Гц до 18000 Гц синусоидальной формы. Частота и громкость сигнала должна задаваться на главном окне программы с помощь слайдеров. Так же должно производиться числовое отображение текущей частоты и громеости.

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Какой класс в Qt предназначен для захвата аудиосигнала?
2. Какой класс в Qt предназначен для вопспроизведения аудиосигнала?
3. Для чего предназначен класс QIODevice и в чем его особенности*?*
4. Какие основные параметры необходимо определять для настройки аудиоустройства?
5. Применение класса QAudioFormat и его основные методы?
6. Для чего предназначен класс QBuffer?
7. Какие методы применяются для запуска и остановки аудиоустройств?

Лабораторная работа № 15

Многопоточность в приложении. Синхронизация потоков

1. Цель работы

Изучить принципы построения многопоточных приложений, а также классы, позволяющие создавать и синхронизировать потоки.

2. Основные теоретические сведения

2.1. Потоки

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 (далее — основной 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 задач 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, так как каждый из процессоров получит отдельный поток для выполнения. В последнее время появляется все больше компьютеров, оснащенных многоядерными процессорами, что делает многопоточное программирование еще более популярным.

Приложения, имеющие один поток, могут выполнять только одну определенную операцию в каждый момент времени, а все

остальные операции ждут ее окончання. Например, такие операции, как вывод на печать, считывание большого файла, ожидание ответа на посланный запрос или выполнение сложных математических вычислений, могут привести к блокировке или зависанию всей программы. Используя многопоточность, можно решить эту проблему, запустив подобные операции в отдельно созданных потоках. Тем самым при зависании одного из потоков функционирование основной программы не будет нарушено.

Создание многопоточного приложения начинается с наследования класса QThread и перезаписи в нем чисто виртуального метода run(), в котором должен быть реализован код, который будет выполняться в потоке. Например:

```
class MyThread : public QThread
{
public:
void run()
{
// Код, вьполняемый в потоке
}
}
```

Второй шаг заключается в создании объекта класса управления потоком и вызове его метода start (), который запустит, в свою очередь, реализованный нами метод run ().

Например:

```
MyThread thread; thread.start();
```

Рассмотрим более подробно интерфейс класс QThread. Основные сигналы класса QThread приведены в таблице 15.1.

Таблица 15.1. Сигналы

Сигнал	**Описание**
void finished()	Вызывается при завершении работы потока
void started()	Вызывается при запуске потока

Основные слоты класса QThread приведены в таблице 15.2.

Таблица 15.2. Слоты

Слот	**Описание**
void quit()	Завершение работы потока
void start(Priority priority = InheritPriority)	Запуск потока, в качестве входного параметра можно установить приоритет потока
void terminate()	Принудительно (аварийное) завершение работы потока

В любой современной ОС потоки могут иметь различные приоритеты. Приоритеты, поддерживаемые в классе QThread, приведены в таблице 15.3.

Таблица 15.3. Приоритеты

Приоритет	**Описание**
QThread::IdlePriority	Самый низкий приоритет. Выполняется когда другие потоки не занимают время процессора
QThread::LowestPriority	Низкий приоритет
QThread::LowPriority	Выполняется реже чем обычный приоритет
QThread::NormalPriority	Приоритет по умолчанию в операционной системе
QThread::HighPriority	Высокий приоритет. Выполняется чуть выше чем обычный
QThread::HighestPriority	Выполняется чаще чем высокий приоритет
QThread::TimeCriticalPriority	Выполняется как можно чаще
QThread::InheritPriority	Использует приоритет создающего потока

2.2. Обмен сообщениями

Один из важнейших вопросов при многопоточном программировании — это обмен сообщениями.

Каждый поток может иметь свой собственный цикл событий (рисунок 15.1). Благодаря этому можно осуществлять связь между объектами. Такая связь может выполняться двумя способами: при помощи соединения сигналов и слотов или за счет обмена событиями.

Использование в классах управления потоком собственных циклов обработки событий позволяет снять ограничение, связанное с применением классов, способных работать только в одном цикле событий, и параллельно использовать необходимое количество объектов этих классов.

Для того чтобы запустить собственный цикл обработки событий в потоке, нужно вызвать метод exec () в методе run (). Цикл обработки событий потока можно завершать посредством слота quit () или метода exit (). Это очень похоже на то, как мы обычно поступаем с объектом приложения в функции main ().

Рис. 15.1. Потоки с объектами и
собственными циклами обработки событий

Класс QObject реализован так, что он обладает близостью к потокам. Каждый объект, созданный от унаследованного класса QObject, располагает ссылкой на поток, в котором был создан. Эту ссылку можно получить вызовом метода QObject: :thread(). Потоки осведомляют свои объекты. Благодаря этому каждый объект знает, к какому потоку он принадлежит. Для того чтобы определить в каком потоке исполняется код, можно вызвать статический метод QThread:: currentThreadid () и получить идентификационный номер потока. Можно так же получить и указатель на объект потока — для этого нужно вызвать статический метод QThread: : currentThreadid ().

Обработка событий осуществляется из контекста принадлежности объекта к потоку, то есть обработка его событий будет выполняться в том потоке, которому объект принадлежит. Объекты можно перемещать из одного потока в другой с помощью метода QObject: :moveToThread ().

2.3. Сигнально-слотовые соединения

Соединение при помощи метода connect () предоставляет дополнительный параметр режима обработки. Этот параметр по умолчанию имеет значение Qt: :AutoConnection, что соответствует автоматическому режиму. Как только происходит отправка сигнала, Qt проверяет, осуществляется ли связь в одном и том же потоке или в разных. Если это один и тот же поток, то отправка сигнала приведет к прямому вызову метода. В том случае, если это разные потоки, сигнал будет преобразован в событие и доставлен нужному объекту.

Реализация сигналов и слотов в Qt содержит механизм, обеспечивающий надежность при работе в потоках, а это означает, что вы можете отправлять сигналы и получать их, не заботясь о блокировке ресурсов. Вы также можете перемещать объект, созданный в одном потоке, в другой. А если вдруг обнаружится, что отправляющий объект находится в одном потоке с принимающим, то отправка сигнала будет сведена к прямой обработке соединения.

Во всем этом есть нюанс, связанный с сигнально-слотовыми соединениями и классом QThread. Очень важно понять и осознать тот факт, что класс QThread не является классом потока, — этот класс представляет собой только механизм для управления потоком, но не сам поток.

Принадлежности объектов к тому или иному потоку будет строиться согласно с тем, в контексте какого из потоков эти объекты были созданы (или помещены в потоки) вызовом метода QObject: :moveToThread(). Класс QThread унаследован от QObject и, следовательно, ничем по своему принципу не отличается от других объектов Qt. А значит, если мы определим слоты в унаследованном от QThread классе и создадим от него объект в основном потоке — например, из функции main (), то сам объект управления потоком будет принадлежать основному потоку, и его слоты станут вызываться не в потоке, которым он управляет, а в основном потоке приложения.

Следовательно, если мы хотим, чтобы слоты обрабатывались в отдельном потоке, то нам пужно создавать объекты непосредственно из класса управления потоком, либо помещать их туда методом QObject: :moveToThread().

2.4. Синхронизация (Мьютексы)

Основные сложности возникают тогда, когда потокам нужно совместно использовать одни и те же данные. С одной стороны, это просто, так как несколько потоков могут одновременно обращаться и записывать данные в одну область. Но, с другой стороны, это может привести к нежелательным последствиям. Представьте себе такую ситуацию: один поток занимается вычислениями, задействуя значения какой-нибудь глобальной переменной, а в это время другой поток вдруг изменяет значение этой переменной. Поток, занимающийся вычислениями, ничего не подозревая, продолжает свою работу и по-прежнему использует исходное, еще не измененное значение, поэтому результат вычислений может оказаться совершенно бессмысленным. Для предотвращения подобных ситуаций требуется механизм, позволяющий блокировать данные, когда один из потоков намеревается их изменить. Этот механизм получил название синхронизация.

Синхронизация позволяет задавать критические секции (critical sections), к которым в определенный момент времени имеет доступ только один из потоков.

Мьютексы (mutex) обеспечивают взаимоисключающий доступ к ресурсам, гарантирующий, что критическая секция будет обрабатываться только одним потоком. Поток, владеющий мьютексом, обладает эксклюзивным правом на использование ресурса, защищенного мьютексом, и другой поток не может завладеть уже занятым мьютексом.

Метод lock () класса QMutex выполняет блокировку ресурса. Для обратной операции существует метод unlock (), который открывает закрытый ресурс для других потоков.

Класс QMutex также содержит метод tryLock(). Его можно использовать для того, чтобы проверить, заблокирован ресурс или нет. Этот метод не приостанавливает исполнение потока и возвращается немедленно — со значением false, если ресурс уже захвачен другим потоком, не ожидая его освобождения. В случае успешного захвата ресурса этот метод вернет значение true, и это значит, что ресурс принадлежит потоку, и он вправе распоряжаться им по своему усмотрению. Основные методы данного класса приведены в таблице 15.4.

Таблица 15.4. Открытые методы класса QMutex

Метод	**Описание**
QMutex(RecursionMode *mode* = NonRecursive)	Конструктор класса. В качестве входного параметра может передаваться режим работы. По умолчанию он равен NonRecursive
bool isRecursive() const	Возвращает true в случае если установлен рекурсивный режим работы
void lock()	Блокирует ресурс
bool tryLock(int timeout = 0)	Проверяет заблокирован ли ресурс. В случае блокировки возвращает значение false. В качестве входного параметра можно передавать время ожиданя освобождения ресурса.
void unlock()	Освобождение ресурса

Рекурсивный режим работы позволяет осуществлять многократную блокировку ресурса одним и тем же потоком. Для разблокировки ресурса необходимо вызвать метод unlock столько же раз, сколько до этого вызывался метод lock. Данный режим предназначен для тех случаев, когда один и тот же поток осуществляет доступ к заблокированной секции многократно, не разблокировав ее перед этим.

3. Порядок выполнения работы

Для более глубокого понимания потоков, а также области их применения рассмотрим одну и ту же программу, в первом варианте реализованную без использования потоков, а во втором с их применением.

На основной форме программы располагается QLCDNumber, в котором отображается значение переменной целого типа. С помощью бегунка можно изменять текущее значение данной переменной. Класс Increment через промежуток времени 1 с вызывает сигнал doIt() по которому производится инкрементирование значения переменной.

3.1. Реализация программы без использования потоков

Programm15_1.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm15_1
TEMPLATE = app


SOURCES += main.cpp\
        widget.cpp \
    increment.cpp

HEADERS  += widget.h \
    increment.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

increment.h

```
#ifndef INCREMENT_H
#define INCREMENT_H
```

```
#include <QObject>
#include <QThread>

class Increment : public QObject
{
   Q_OBJECT
public:
   explicit Increment(QObject *parent = 0);
   void start();
   void stop();

private:
   bool enableWork;
   void work();
signals:
   void doIt();

public slots:
};

#endif // INCREMENT_H
```

increment.cpp

```
#include "increment.h"

Increment::Increment(QObject *parent) : QObject(parent)
{
   enableWork = false;
}

void Increment::start()
{
   enableWork = true;
   work();
}
```

```
void Increment::stop()
{
   enableWork = false;
}

void Increment::work()
{
   while (enableWork)
   {
      QThread::msleep(1000);
      emit doIt();
   }
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QLCDNumber>
#include <QSlider>
#include <QPushButton>
#include "increment.h"

class Widget : public QWidget
{
   Q_OBJECT
private:
   QVBoxLayout *mainLayout;
   QLCDNumber *lcd;
   QSlider *slider;
   QHBoxLayout *layoutButton;
   QPushButton *buttonStart;
   QPushButton *buttonStop;
   Increment *increment;
```

```
public:
    Widget(QWidget *parent = 0);
    ~Widget();

private slots:
    void slotPushStart();
    void slotPushStop();
    void slotDoIt();
    void valueChanged(int a);
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    increment = new Increment;
    connect(increment,SIGNAL(doIt()),this,SLOT(slotDoIt()));
    mainLayout = new QVBoxLayout(this);
    lcd = new QLCDNumber(this);
    mainLayout->addWidget(lcd);
    slider = new QSlider(Qt::Horizontal,this);
    slider->setMinimum(0);
    slider->setMaximum(100);

connect(slider,SIGNAL(valueChanged(int)),this,SLOT(valueChanged(int)
));
    mainLayout->addWidget(slider);

    layoutButton = new QHBoxLayout;
    layoutButton->setMargin(0);
    mainLayout->addLayout(layoutButton);
```

```
    buttonStart = new QPushButton(tr("Start"),this);
    connect(buttonStart,SIGNAL(pressed()),this,SLOT(slotPushStart()));
    layoutButton->addWidget(buttonStart);

    buttonStop = new QPushButton(tr("Stop"),this);
    connect(buttonStop,SIGNAL(pressed()),this,SLOT(slotPushStop()));
    layoutButton->addWidget(buttonStop);
}

Widget::~Widget()
{
    delete increment;
}

void Widget::slotPushStart()
{
    increment->start();
}

void Widget::slotPushStop()
{
    increment->stop();
}

void Widget::slotDoIt()
{
    int a = lcd->intValue();
    a++;
    slider->setValue(a);
    lcd->display(a);
}

void Widget::valueChanged(int a) {
    lcd->display(a); }
```

3.2. Реализация программы без использования потоков

Programm15_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm15_2
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp \
    increment.cpp
HEADERS  += widget.h \
    increment.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

increment.h

```
#ifndef INCREMENT_H
#define INCREMENT_H

#include <QObject>
#include <QThread>

class Increment : public QThread
{
    Q_OBJECT
```

```
public:
    explicit Increment(QObject *parent = 0);
    void startWork();
    void stop();

private:
    bool enableWork;
    void work();
    void run();

signals:
    void doIt();

public slots:
};

#endif // INCREMENT_H
```

increment.cpp

```
#include "increment.h"

Increment::Increment(QObject *parent) : QThread(parent)
{
    enableWork = false;
}

void Increment::startWork()
{
    enableWork = true;
    start();
}

 void Increment::run()
 {
     enableWork = true;
     work();
 }
```

```
void Increment::stop()
{
   enableWork = false;
   terminate();
}

void Increment::work()
{
   while (enableWork)
   {
      QThread::msleep(1000);
      emit doIt();
   }
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QLCDNumber>
#include <QSlider>
#include <QPushButton>
#include "increment.h"

class Widget : public QWidget
{
   Q_OBJECT
private:
   QVBoxLayout *mainLayout;
   QLCDNumber *lcd;
   QSlider *slider;
   QHBoxLayout *layoutButton;
   QPushButton *buttonStart;
```

```
    QPushButton *buttonStop;
    Increment *increment;

public:
    Widget(QWidget *parent = 0);
    ~Widget();
private slots:
    void slotPushStart();
    void slotPushStop();
    void slotDoIt();
    void valueChanged(int a);};
#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    increment = new Increment;
    connect(increment,SIGNAL(doIt()),this,SLOT(slotDoIt()));
    mainLayout = new QVBoxLayout(this);
    lcd = new QLCDNumber(this);
    mainLayout->addWidget(lcd);
    slider = new QSlider(Qt::Horizontal,this);
    slider->setMinimum(0);
    slider->setMaximum(100);

connect(slider,SIGNAL(valueChanged(int)),this,SLOT(valueChanged(int)
));
    mainLayout->addWidget(slider);

    layoutButton = new QHBoxLayout;
    layoutButton->setMargin(0);
    mainLayout->addLayout(layoutButton);

    buttonStart = new QPushButton(tr("Start"),this);
    connect(buttonStart,SIGNAL(pressed()),this,SLOT(slotPushStart()));
```

```
    layoutButton->addWidget(buttonStart);

    buttonStop = new QPushButton(tr("Stop"),this);
    connect(buttonStop,SIGNAL(pressed()),this,SLOT(slotPushStop()));
    layoutButton->addWidget(buttonStop);
}

Widget::~Widget()
{
    delete increment;
}

void Widget::slotPushStart()
{
    increment->startWork();
}

void Widget::slotPushStop()
{
    increment->stop();
}

void Widget::slotDoIt()
{
    int a = lcd->intValue();
    a++;
    slider->setValue(a);
    lcd->display(a);
}

void Widget::valueChanged(int a)
{
    lcd->display(a);
}
```

3.3. Задание для самостоятельного выполнения

Реализовать текстовый редактор с набором стандартных функций (открыть файл, сохранить файл, сохранить как). При открытии файла должен использоваться режим доступа Read Only. Отдельный поток должен отслеживать изменения в редактируемом файле. При обнаружении изменений, не отображенных в текстовом поле, он должен их отобразить таким образом, чтобы те изменения, которые были внесены вручную через текстовое поле, остались и оказались в конце текстового документа.

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Что такое поток?
2. Для чего применяются потоки?
3. Какой класс в Qt реализует интерфейс для работы с потоками?
4. Что такое синхронизация потоков?
5. Для чего применяется синхронизация потоков?

Лабораторная работа № 16

Работа с диалогами. Класс QDialog

1. Цель работы

Изучить принципы использования стандартных диалогов, а также принципы построения собственных диалогов.

2. Основные теоретические сведения

Диалоговое окно — это центральный элемент, обеспечивающий взаимодействие между пользователем и приложением. Этот виджет может содержать ряд опций, изменение которых в ходе работы влечет за собой изменения в работе самой программы. Диалоговые окна всегда являются виджетами верхнего уровня и имеют свой заголовок.

Их можно разбить на три основные категории:

- собственные;
- стандартные;
- окна сообщений.

2.1. Стандартные диалоговые окна

Использование стандартных окон значительно ускоряет разработку тех приложений, в которых необходимо использовать диалоговые окна выбора файлов, шрифта, цвета и т. д. Вместо того чтобы тратить время на разработку своих собственных классов, можно воспользоваться готовыми классами библиотеки Qt. К достоинствам стандартных диалоговых окон можно отнести и целостность пользовательского интерфейса, так как вид окон во всех приложениях, их использующих, будет один и тот же.

2.1.1. Диалоговое окно выбора файлов

Диалоговое окно выбора файлов предназначено для выбора одного или нескольких файлов, а также файлов, находящихся на удаленном компьютере, и поддерживает возможность переименования файлов и создания каталогов. Класс QFileDialog предоставляет

реализацию диалогового окна выбора файлов (рис. 32.2) и отвечает за создание и работоспособность сразу трех диалоговых окон. Одно из них позволяет осуществлять выбор файла для открытия, второе предназначено для выбора пути и имени файла для его сохранения, а третье — для выбора каталога. В таблице 16.1 приведены основные статические методы класса QFileDialog.

Таблица 16.1. Статические методы класса QFileDialog

Метод	**Описание**
getopenFileName ()	Создает диалоговое окно выбора одного файла. Этот метод возвращает значение типа QString, содержащее имя и путь выбранного файла
getopenFileNames ()	Создает диалоговое окно выбора нескольких файлов. Возвращает список строк типа QStringList, содержащих пути и имена файлов
getSaveFileName ()	Создает диалоговое окно сохранения файла. Возвращает имя и путь файла в строковой переменной типа QString
getExistingDirectory ()	Создает окно выбора каталога. Возвращает значение типа QString, содержащее имя и путь выбранного каталога

Пример использования:

```
QString str = QFileDialog::getOpenFileName(this,
                    "Open Dialog", "", "*.cpp *.h");
```

Первый параметр this – это указатель на родителя, второй параметр "Open Dialog" – текст заголовка окна диалога, третий параметр "" – родительский каталог (каталог который будет открыт по умолчанию), четвёртый параметр "*.cpp *.h" – список фильтра файлов.

2.1.2. Диалоговое окно выбора цвета

Класс QColorDialog реализует диалоговое окно выбора цвета. Для того чтобы показать это окно, вызывается статический метод getColor (). Первым параметром в метод можно передать цветовое значение для инициализации. Вторым параметром является указатель на виджет предка. После закрытия диалогового окна метод getColor ()

возвращает объект класса QColor. Чтобы узнать, какой кнопкой было закрыто окно: OK или Cancel (Отмена), необходимо вызвать метод isValid() из возвращенного объекта QColor. Значение true означает, что была нажата кнопка OK, в противном случае — Cancel (Отмена). Например:

```
QColor color = QColorDialog::getColor(blue);
if (!color.isValid()) { // Cancel       }
```

2.1.3. Диалоговое окно выбора шрифта

Окно выбора шрифта предназначено для выбора одного из зарегистрированных в системе шрифтов, а также для задания его стиля и размера. Реализация этого диалогового окна содержится в классе QFontDialog, определенном в заголовочном файле QFontDialog.

Для того чтобы показать диалоговое окно, в большинстве случаев можно обойтись методом QFontDialog: :getFont(). Первый параметр этого метода является указателем на перемен¬ную булевого типа. Метод записывает в эту переменную значение true в том случае, если диалоговое окно было закрыто нажатием на кнопку OK, в противном случае — значение false. Во втором параметре можно передать объект класса QFont, который будет использо-ваться для инициализации диалогового окна. После завершения выбора шрифта и закрытия окна статический метод getFont() возвращает шрифт, выбранный пользователем.

Например:

```
bool bOk;
QFont fnt = QFontDialog::getFont(&bOk);
if (!bOk) {
// Была нажата кнопка Cancel
   }
```

2.2. Диалоговые окна мастера

Диалоговые окна мастера были придуманы для сопровождения пользователя при выполнении им операций, которые требуют непосредственного его участия. Для навигации по страницам

диалогового окна мастера служат две кнопки: Next (Вперед) и Back (Назад). Пользователь не имеет возможности сразу отобразить интересующую его страницу окна, не пройдя все предшествующие страницы, что гарантирует выполнение всех пунктов, содержащихся в этих диалоговых окнах.

Для создания класса мастера нужно унаследовать класс QWizard и добавить каждую новую страницу диалогового окна вызовом метода addPage (). В этот метод надо передать указатель на виджет QWizardPage, в котором вызовом метода setTitle() можно установить строку заголовка, а при помощи класса компоновки — разместить все необходимые виджеты.

2.3. Создание собственного диалогового окна

Для создания своего собственного диалогового окна нужно унаследовать класс QDialog, по умолчанию область заголовка диалогового окна содержит кнопку “?”, предназначенную для получения подробной информации о назначении виджетов. Для того чтобы ее удалить необходимо установить флаги Qt::WindowTitleHint И Qt: :WindowSystemMenuHint. Модальное диалоговое окно всегда должно содержать кнопку Cancel (Отмена). Обычно в диалоговых окнах используются две кнопки: OK и Cancel. Сигналы этих кнопок подключаются к слотам accept() и rejected() соответственно. Это делается для того, чтобы метод exec мог возвращать значения QDialog::Accepted в случает нажатия клавиши Ok и QDialog::Rejected в случае нажатия кнопки Cancel. Если вы хотите чтобы функцию exec возвращала ваш собственный код (отличный от стандартных значений), используйте функция done().

Во всем остальном построение собственного диалога схоже с построением обычного окна на основе QWidget.

3. Порядок выполнения работы

Для понимания принципов работы с диалогами рассмотрим несколько примеров, которые отображают работу как со стандартными диалогами, так и создание собственных диалогов.

3.1. Работа с QFileDialog

В данном примере реализованы основные методы работы с QFileDialog. На основном окне программы располагаются три области с текстовым полем и кнопкой для вызова диалога. В первой области реализована работа с выбором имени открываемого файла, во второй области – с именем сохраняемого файла, а в третьей – с выбором каталога. После закрытия диалогового окна, возвращаемая информация помещается в соответствующее текстовое поле.

Programm16_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm16_1
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QFileDialog>
#include <QGridLayout>
#include <QLabel>
#include <QPushButton>
#include <QLineEdit>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   QGridLayout *mainLayout;
   QLabel *labelOpenFile;
   QLineEdit *editOpenFile;
   QPushButton *buttonOpenFile;
   QLabel *labelSaveFile;
   QLineEdit *editSaveFile;
   QPushButton *buttonSaveFile;
   QLabel *labelDir;
   QLineEdit *editDir;
   QPushButton *buttonDir;


private slots:
   void slotPushFile();
   void slotSaveFile();
   void slotPushDir();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"
```

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QGridLayout(this);

    labelOpenFile = new QLabel(tr("Open file"),this);
    mainLayout->addWidget(labelOpenFile,0,0,1,1);
    editOpenFile = new QLineEdit(this);
    editOpenFile->setReadOnly(true);
    mainLayout->addWidget(editOpenFile,0,1,1,1);
    buttonOpenFile = new QPushButton(tr("Open file"),this);

connect(buttonOpenFile,SIGNAL(pressed()),this,SLOT(slotPushFile()));
    mainLayout->addWidget(buttonOpenFile,0,2,1,1);

    labelSaveFile = new QLabel(tr("Save file"),this);
    mainLayout->addWidget(labelSaveFile,1,0,1,1);
    editSaveFile = new QLineEdit(this);
    editSaveFile->setReadOnly(true);
    mainLayout->addWidget(editSaveFile,1,1,1,1);
    buttonSaveFile = new QPushButton(tr("Save file"),this);
    connect(buttonSaveFile,SIGNAL(pressed()),this,SLOT(slotSaveFile()));
    mainLayout->addWidget(buttonSaveFile,1,2,1,1);

    labelDir = new QLabel(tr("Dir"),this);
    mainLayout->addWidget(labelDir,2,0,1,1);
    editDir = new QLineEdit(this);
    editDir->setReadOnly(true);
    mainLayout->addWidget(editDir,2,1,1,1);
    buttonDir = new QPushButton(tr("Dir"),this);
    connect(buttonDir,SIGNAL(clicked(bool)),this,SLOT(slotPushDir()));
    mainLayout->addWidget(buttonDir,2,2,1,1);
}

Widget::~Widget()
{

}
```

```
void Widget::slotPushFile()
{
   editOpenFile->setText(QFileDialog::getOpenFileName(this,tr("Open
File"),""));
}

void Widget::slotSaveFile()
{
   editSaveFile->setText(QFileDialog::getSaveFileName(this,tr("Save
File"),""));
}

void Widget::slotPushDir()
{
   editDir->
setText(QFileDialog::getExistingDirectory(this,tr("Directory"),""));
}
```

3.2. Работа с QColorDialog

В данном примере реализована работа с QColorDialog. На основном окне программы расположено текстовое поле, в которое будет выводится имя цвета, а также кнопка вызова диалога. После выбора цвета, его имя будет выведено в текстовую метку, а фон самой формы окрасится в сам цвет.

Programm16_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm16_2
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QColorDialog>
#include <QColor>
#include <QVBoxLayout>
#include <QLabel>
#include <QPushButton>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    QVBoxLayout *mainLayout;
    QLabel *labelColor;
    QPushButton *buttonColor;

private slots:
    void slotPushButtonColor();
```

```
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout =  new QVBoxLayout(this);
   labelColor = new QLabel(tr("Color"),this);
   mainLayout->addWidget(labelColor);

   buttonColor = new QPushButton(tr("Color"),this);

connect(buttonColor,SIGNAL(pressed()),this,SLOT(slotPushButtonColor(
)));
   mainLayout->addWidget(buttonColor);
}

Widget::~Widget()
{

}

void Widget::slotPushButtonColor()
{
   QColor col = QColorDialog::getColor(Qt::white,this,tr("Color"));
   if (col.isValid())
   {
      QPalette Pal(palette());
      Pal.setColor(QPalette::Background,col);
      this->setPalette(Pal);
      labelColor->setText(col.name());
   }
}
```

3.3. Работа с QFontDialog

В данной программе приведен пример работы с QFontDialog. На основном окне программы располагается текстовая метка, в которую выводится имя установленного шрифта, а также кнопка для вызова диалога. После выбора шрифта, он устанавливается как основной для окна программы.

Programm16_3.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm16_3
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QVBoxLayout>
#include <QLabel>
#include <QPushButton>
#include <QFontDialog>
#include <QFont>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   QVBoxLayout *mainLayout;
   QLabel *labelFont;
   QPushButton *buttonFont;

private slots:
   void slotPushFontButton();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QVBoxLayout(this);
   labelFont = new QLabel(tr("Font"),this);
   mainLayout->addWidget(labelFont);
   buttonFont = new QPushButton(tr("Font"),this);

connect(buttonFont,SIGNAL(pressed()),this,SLOT(slotPushFontButton()))
;
   mainLayout->addWidget(buttonFont);

}

Widget::~Widget()
{

}

void Widget::slotPushFontButton()
{
  bool ok = false;
  QFont currentFont = QFontDialog::getFont(&ok,this->
font(),this,tr("Font"));
  this->setFont(currentFont);
  labelFont->setText(currentFont.family());
}
```

3.4. Работа с QWizard

В данном примере показаны основные приемы работы с классом мастера (QWizard). Основной класс программы является потомком класса Qwizard.. Класс QCurrentPage является потомком класса QWizardPage и используется как основа для страниц мастера.

Programm16_4.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm16_4
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp \
    qcurrentpage.cpp
HEADERS  += widget.h \
    qcurrentpage.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

qcurrentpage.h

```
#ifndef QCURRENTPAGE_H
#define QCURRENTPAGE_H

#include <QObject>
#include <QWizardPage>
#include <QVBoxLayout>
#include <QLabel>

class QCurrentPage : public QWizardPage
{
```

```
public:
   QCurrentPage(QString text, QWidget *parent=NULL);

private:
   QVBoxLayout *mainLayout;
   QLabel *labelText; };

#endif // QCURRENTPAGE_H
```

qcurrentpage.cpp

```
#include "qcurrentpage.h"

QCurrentPage::QCurrentPage(QString text,QWidget *parent) :
QWizardPage(parent)
{
   mainLayout = new QVBoxLayout(this);
   labelText = new QLabel(this);
   mainLayout->addWidget(labelText);
   labelText->setText(text);
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QWizard>
#include <QLabel>
#include "qcurrentpage.h"

class Widget : public QWizard
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();
```

```
private:
   QCurrentPage *page_;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWizard(parent)
{


   for (quint8 i=0;i<5;i++)
   {
      page_ = new QCurrentPage(tr("Page")+" "+QString::number(i),this);
      addPage(page_);
   }
}

Widget::~Widget()
{

}
```

3.5. Создание собственных диалогов.

Для создания собственного класса необходимо произвести наследование от класса QDialog. В приведенном примере реализован диалог, который позволяет вводить текстовую строку. После его выполнения данная строка отображается в текстовом поле на основном окне программы.

Programm16_5.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm16_5
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp \
    qvaluedialog.cpp
HEADERS  += widget.h \
    qvaluedialog.h
```

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

qvaluedialog.h

```
#ifndef QVALUEDIALOG_H
#define QVALUEDIALOG_H

#include <QObject>
#include <QWidget>
#include <QDialog>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
```

```
#include <QDebug>

class QValueDialog : public QDialog
{
   Q_OBJECT
public:
   QValueDialog();
   ~QValueDialog();
   QString static getStringValue(bool *ok=NULL);

private:
   void createForm();
   QVBoxLayout *mainLayout;
   QHBoxLayout *layoutValue;
   QLabel *labeValue;
   QLineEdit *editValue;
   QHBoxLayout *layoutButtons;
   QPushButton *buttonOk;
   QPushButton *buttonCancel;

};

#endif // QVALUEDIALOG_H
```

qvaluedialog.cpp

```
#include "qvaluedialog.h"

QValueDialog::QValueDialog()
{
   createForm();
}

QValueDialog::~QValueDialog()
{

}
```

```
void QValueDialog::createForm()
{
   this->setModal(true);

   mainLayout = new QVBoxLayout(this);
   layoutValue = new QHBoxLayout;
   layoutValue->setMargin(0);
   mainLayout->addLayout(layoutValue);

   labeValue = new QLabel(tr("Value"),this);
   layoutValue->addWidget(labeValue);
   editValue = new QLineEdit(this);
   layoutValue->addWidget(editValue);

   layoutButtons = new QHBoxLayout;
   layoutButtons->setMargin(0);
   mainLayout->addLayout(layoutButtons);
   layoutButtons->addStretch();
   buttonOk = new QPushButton(tr("Ok"),this);
   connect(buttonOk,SIGNAL(pressed()),this,SLOT(accept()));
   layoutButtons->addWidget(buttonOk);
   buttonCancel = new QPushButton(tr("Cancel"),this);
   connect(buttonCancel,SIGNAL(pressed()),this,SLOT(reject()));
   layoutButtons->addWidget(buttonCancel);

   show();
}

QString QValueDialog::getStringValue(bool *ok)
{
   QString Output;
   QValueDialog dlg;
   int B = dlg.exec();
   if (ok != NULL) *ok = B;
   Output = dlg.editValue->text();
   return Output;
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QHBoxLayout>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include "qvaluedialog.h"

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = 0);
   ~Widget();

private:
   QHBoxLayout *mainLayout;
   QLabel *labelVal;
   QLineEdit *editVal;
   QPushButton *buttonVal;


private slots:
   void slotPushButtonValue();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"
```

```
Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QHBoxLayout(this);
   this->setLayout(mainLayout);
   labelVal = new QLabel(tr("Value"),this);
   mainLayout->addWidget(labelVal);
   editVal = new QLineEdit(this);
   editVal->setReadOnly(true);
   mainLayout->addWidget(editVal);
   buttonVal = new QPushButton(tr("Get value"),this);

connect(buttonVal,SIGNAL(pressed()),this,SLOT(slotPushButtonValue())
);
   mainLayout->addWidget(buttonVal);
}

Widget::~Widget()
{

}

void Widget::slotPushButtonValue()
{
   bool result;
   QString Output = QValueDialog::getStringValue(&result);
   if (result)
   {
      editVal->setText(Output);
   }
}
```

3.6. Задание для самостоятельного выполнения

Разработать программу «Телефонный справочник». Данные должны сохраняться и загружать из бинарного файла с использованием сериализации. Функции добавления записи, а также поиск должны быть реализованы с использование собственного диалогового окна. В настройках программы должен задаваться

шрифт, используемый интерфейсом, а также цвет окон приложения. Все параметры должны сохраняться и загружаться с использованием класса QSettings.

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Что такое диалог?
2. Для чего применяются диалоги?
3. Для чего предназначен класс QWizard?
4. Какой стандартный диалог предназначен для выбора параметров шрифта?
5. Какой стандартный диалог предназначен для работы с файлами и папками?
6. Какой стандартный диалог предназначен для настройки цвета?

Лабораторная работа № 17

Построения основного окна приложения на основе QmainWindow

1. Цель работы

Изучить принципы использования класса QMainWindow для построения основного окна программы.

2. Основные теоретические сведения

Многие приложения имеют сходный интерфейс — окно включает меню, рабочую область, строку состояния и т. д. Неудивительно, что библиотека Qt содержит классы, помогающие создавать такие приложения.

2.1 Класс главного окна QMainWindow

Класс QMainWindow — это очень важный класс, который реализует главное окно, содержащее в себе типовые виджеты, необходимые большинству приложений, — такие как меню, секции для панелей инструментов, рабочую область, строки состояния и т. п. В этом классе внешний вид окна уже подготовлен, и его виджеты не нуждаются в дополнительном размещении, поскольку уже находятся в нужных местах. На рис. 1.17 изображен пример основного окна программы.

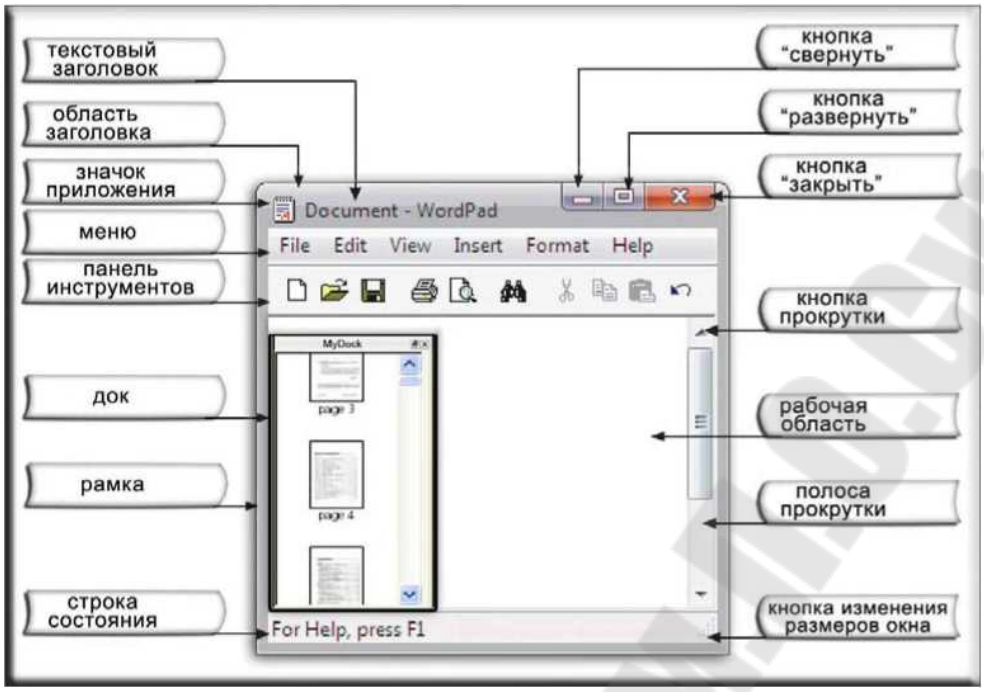

Рис. 17.1. Основное окно программы

В таблице 17.1 отображены основные открытые методы класса QMainWindow.

Таблица 17.1. Основные открытые методы класса QMainWindow

Метод	**Описание**
QMainWindow(QWidget *parent = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags())	Конструктор класса. В качестве родителя передается указатель на Qwidget
void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar)	Данный метод позволяет добавить панель инструментов. Первый параметр задает положения панели по умолчанию, а вторым параметром задается указатель на класс панели инструментов
void addDockWidget(Qt::DockWidgetArea area, QDockWidget *dockwidget)	Метод для добавления док виджета на форму основного окна. В качестве первого параметра передается положение виджета по умолчанию, а в качестве второго

	параметра передается указатель на док виджет
QWidget *centralWidget() const	Метод возвращает указатель на центральный виджет
void setIconSize(const QSize &iconSize)	Метод устанавливает размер иконок на основном окне
void setMenuBar(QMenuBar *menuBar)	Метод позволяет установить строку меню. В качестве входного параметра передается указатель на класс строки меню
void setStatusBar(QStatusBar *statusbar)	Метод задает объект строки статуса
QStatusBar *statusBar() const	Метод возвращает указатель на объект строки состояния

2.1.1. Класс действия QAction

Класс действия QAction предоставляет очень мощный механизм, ускоряющий разработку приложений. Если бы его не было, то вам пришлось бы создавать команды меню, соединять их со слотами, затем создавать панели инструментов и соединять их с теми же слотами и т. д. Это приводило бы к дублированию программного кода и вызывало бы проблемы при синхронизации элементов пользовательского интерфейса. Например, при необходимости сделать одну из команд меню недоступной, вам нужно было бы сделать ее недоступной в меню, а потом проделать то же самое и с кнопкой панели инструментов этой команды.

Объекты класса QAction предоставляют решение этой проблемы и значительно упрощают программирование. Например, если команда меню File | New (Файл | Создать) дублируется кнопкой на панели инструментов, для них можно создать один *объект действия.* Тем самым, если вдруг команду потребуется сделать недоступной, мы будем иметь дело только с одним объектом.

QAction объединяет следующие элементы интерфейса пользователя:

- текст для всплывающего меню;

- текст для всплывающей подсказки;
- текст подсказки «Что это»;
- «горячие» клавиши;
- ассоциированные значки;
- шрифт;
- текст строки состояния.

В таблице 17.2 отображены основные открытые методы класса QAction.

Таблица 17.2. Открытые методы класса QAction.

Метод	**Описание**
QAction(QObject *parent = nullptr)	Конструктор класса
QFont font() const	Возвращает текущий шрифт компонента
QIcon icon() const	Возвращает текущую иконку компонента
void setFont(const QFont &font)	Устанавливает шрифт компонента
void setIcon(const QIcon &icon)	Устанавливает иконку компонента
void setShortcut(const QKeySequence &shortcut)	Устанавливает клавиши быстрого доступа (например Ctrl+A)
void setText(const QString &text)	Устанавливает текст компонента
void setWhatsThis(const QString &what)	Устанавливает текст подсказки
QString text() const	Возвращает текст компонента

2.1.2 Панель инструментов (QToolBar)

Основная цель панели инструментов (Tool Bar) — предоставить пользователю быстрый доступ к командам программы одним нажатием кнопки мыши. Это делает панель инструментов более удобной, чем меню, в котором нужно сделать, по меньшей мере, два нажатия. Еще одно достоинство панели инструментов состоит в том, что она всегда видима, а это освобождает от необходимости тратить время на поиски в меню необходимой команды или вспоминать комбинацию клавиш ускорителя.

Панель инструментов представляет собой область, в которой расположены кнопки, дублирующие часто используемые команды

меню. Для панелей инструментов библиотека Qt предоставляет класс QToolBar.

Для того чтобы поместить кнопку на панель инструментов, необходимо вызвать метод addAction (), который неявно создаст объект действия. В этот метод можно передать растровое изображение, поясняющий текст и соединение со слотом.

Наряду с кнопками, в панели инструментов могут быть размещены и любые другие виджеты. Для этого в классе QToolBar определен метод addwidget ().

Пример создания и добавления действия на панель инструментов:

```
        actionNew = new
QAction(QIcon(":/icon/new.png"),tr("New"));
        toolBar->addAction(actionNew);
```

2.1.3. Строка состояния

Этот виджет располагается в нижней части главного окна и отображает, как правило, текстовые сообщения, содержащие информацию о состоянии приложения или короткую справку о командах меню или кнопках панелей инструментов. Строку состояния реализует класс QStatusBar, определенный в заголовочном файле QStatusBar. Различают следующие типы сообщений строки состояния:

- промежуточный — вызывается методом showMessage(). Для очистки строки состояния следует вызвать метод clearMessage (). Если во втором параметре метода showMessage () задан временной интервал, то строка состояния будет очищаться автоматически по его истечении. Примером промежуточного отображения является вывод поясняющего текста для команд меню;
- нормальный — служит для отображения часто изменяющейся информации (например, для отображения позиции указателя мыши). Для этого рекомендуется поместить в строку состояния отдельный виджет. Например, если приложение должно отображать процесс выполнения какой-либо операции, то лучше разместить в строке состояння индикатор этого процесса. Чтобы

разместить виджет в строке состояния, нужно передать его указатель в метод addwidget (). Виджеты можно также удалять из строки состояния с помощью метода removeWidget ();

- постоянный — отображает информацию, необходимую для работы с приложением. Например, для отображения состояния клавиатуры в строке состояння могут быть внесены виджеты надписей, отображающие состояние клавиш <Caps Lock>, <Num Lock>, <Insert> и т. д. Это достигается посредством вызова метода addPermanentwidget (), принимающего указатель на виджет в качестве аргумент.

Пример добавления виджета в строку состояния:

```
statusBar()->addWidget(new QLabel(tr("Status bar")));
```

2.2. MDI-приложение

MDI-приложение позволяет пользователю работать с несколькими открытыми документами. По своей сути оно очень напоминает обычный рабочий стол, только в виртуальном исполнении. Пользователь может разложить в его области несколько окон документов или свернуть их. Окна документов могут перекрывать друг друга, а могут быть развернуты на всю рабочую область.

Рабочая область, внутри которой размещаются окна документов, реализуется классом QMdiArea. Виджет этого класса выполняет «закулисное» управление динамически создаваемыми окнами документов. Упорядочивание таких окон осуществляется при помощи слотов tileSubwindows () и cascadeSubwindows(), определенных в этом классе. Метод QMdiArea:: subwindowList () возвращает список всех содержащихся в нем окон, созданных ОТ класса QmdiSubWindow. Рисунок 17.2 поясняет понятие приложения с MDI интерфейсом.

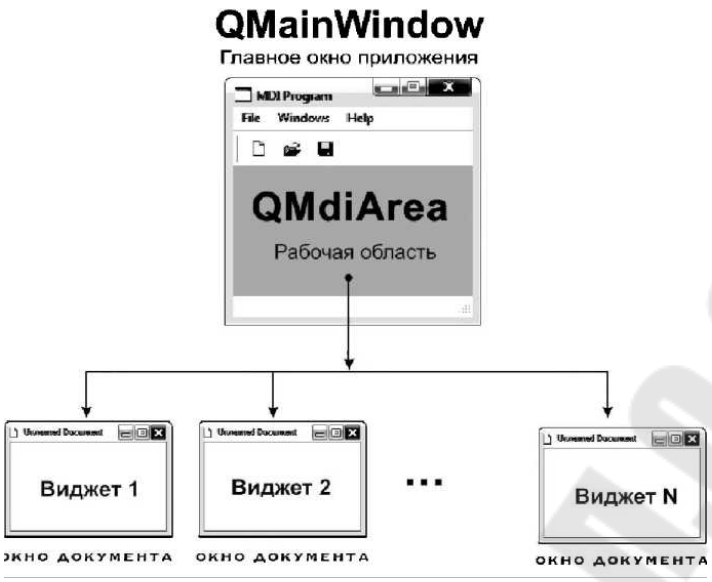

Рис. 17.2. MDI Приложение

В таблице 17.3 приведены основные открытые методы класса QmdiArea.

Таблица 17.3. Основные открытые методы QMdiArea

Метод	**Описание**
QMdiArea(QWidget *parent = Q_NULLPTR)	Конструктор класса, в качестве входного параметра передается указатель на родителя
QMdiSubWindow *activeSubWindow() const	Возвращает указатель на активное дочернее окно
QList<QMdiSubWindow *> subWindowList(WindowOrder order = CreationOrder) const	Возвращает список дочерних окон
void activateNextSubWindow()	Активировать следующее дочернее окно
void activatePreviousSubWindow()	Активировать предыдущее дочернее окно
void cascadeSubWindows()	Расположить каскадом дочерние окна
void closeActiveSubWindow()	Закрыть активное дочернее окно

void closeAllSubWindows()	Закрыть все дочерние окна
void setActiveSubWindow(QMdiSubWindow *window)	Установить активным дочернее окно, в качестве входного параметра передается указатель на дочернее окно

3. Порядок выполнения работы

Для понимания принципов построения основного окна программы рассмотри пару примеров по созданию MDI и SDI приложений.

3.1. Работа с классом QMainWindow

В данном примере реализовано основное рабочее окно программы по технологии SDI. В качестве основного рабочего виджета используется объект типа Qwidget. Он устанавливается с помощью следующей команды:

```
CentralWidget = new QWidget(this);
setCentralWidget(CentralWidget);
```

На нем располагается компоновщик с текстовой меткой типа Qlabel. Так же реализованы следующие методы:

- actionOpen;
- actionSave;
- actionHelp;
- actionExit.

Действие actionExit производит закрытие программы. Для добавления действия в основное меню программы выполняются следующие команды:

```
QMenu *menu = menuBar()->addMenu(tr("&File"));
//Создаем пункт меню File
menu->addAction(actionOpen);
//Добавляем действие actionOpen в пункт меню File
menu->addAction(actionSave);
```

```
        //Добавляем действие actionSave в пункт меню File
         menu->addAction(actionExit);
        //Добавляем действие actionExit в пункт меню File
```

Programm17_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm17_1
TEMPLATE = app
SOURCES += main.cpp\
        mainwindow.cpp
HEADERS  += mainwindow.h
RESOURCES += \
    icons.qrc
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QToolBar>
```

```
#include <QAction>
#include <QMenuBar>
#include <QStatusBar>

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();

private:
   QWidget *CentralWidget;
   QVBoxLayout *mainLayout;
   QLabel *labelText;
   QToolBar *toolBar;

   QAction *actionOpen;
   QAction *actionSave;
   QAction *actionHelp;
   QAction *actionExit;


};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
   CentralWidget = new QWidget(this);
   setCentralWidget(CentralWidget);

   mainLayout = new QVBoxLayout(CentralWidget);
```

```
    CentralWidget->setLayout(mainLayout);
    labelText = new QLabel(tr("CentralWidget"),CentralWidget);
    mainLayout->addWidget(labelText,0,Qt::AlignCenter);

    toolBar = new QToolBar(this);
    toolBar->setIconSize(QSize(48,48));
    addToolBar(toolBar);

    actionOpen = new QAction(QIcon(":/icon/open.png"),tr("Open"));
    actionOpen->setShortcut(Qt::CTRL+Qt::Key_O);
    toolBar->addAction(actionOpen);

    actionSave = new QAction(QIcon(":/icon/save.png"),tr("Save"));
    actionSave->setShortcut(Qt::CTRL+Qt::Key_S);
    toolBar->addAction(actionSave);

    actionHelp = new QAction(QIcon(":/icon/help.png"),tr("Help"));
    toolBar->addAction(actionHelp);

    actionExit = new QAction(QIcon(":/icon/exit.png"),tr("Exit"));
    actionExit->setShortcut(Qt::ALT+Qt::Key_X);
    connect(actionExit,SIGNAL(triggered(bool)),this,SLOT(close()));
    toolBar->addAction(actionExit);

    QMenu *menu = menuBar()->addMenu(tr("&File"));

    menu->addAction(actionOpen);
    menu->addAction(actionSave);
    menu->addAction(actionExit);

    menu = menuBar()->addMenu(tr("&Help"));
    menu->addAction(actionHelp);

    statusBar()->addWidget(new QLabel(tr("Status bar")));

}

MainWindow::~MainWindow()
{
```

```
}
```

icons.qrc

```
<RCC>
   <qresource prefix="/">
      <file>icon/open.png</file>
      <file>icon/save.png</file>
      <file>icon/help.png</file>
      <file>icon/exit.png</file>
   </qresource>
</RCC>
```

3.2. Создание MDI приложения

В данном примере реализована программа по MDI технологии. В качестве центрального виджета используется объект типа QMdiArea. В качестве дочернего окна используется виджет текстовой метки Qlabel. Пример создания дочернего окна:

```
QLabel *labelText = new QLabel(tr("SubWindow"),m_pma);
m_pma->addSubWindow(labelText);
labelText->show();
```

Класс QMdiArea содержит два метода tileSubWindows и cascadeSubWindows. Они позволяют автоматически выстраивать дочерние окна. Первый метод выстраивает дочерние окна по сетке, а второй – каскадом.

Programm17_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm17_2
TEMPLATE = app
SOURCES += main.cpp\
        mainwindow.cpp
HEADERS  += mainwindow.h
```

```
RESOURCES += \
    icons.qrc
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QToolBar>
#include <QAction>
#include <QMenuBar>
#include <QStatusBar>
#include <QMdiArea>
#include <QDebug>

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
```

```
    ~MainWindow();

private:
    QMdiArea*      m_pma;
    QVBoxLayout *mainLayout;
    QLabel *labelText;
    QToolBar *toolBar;

    QAction *actionNew;
    QAction *actionOpen;
    QAction *actionSave;
    QAction *actionHelp;
    QAction *actionExit;

    QAction *actionCascade;
    QAction *actionTile;

private slots:
    void slotNew();
};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    m_pma = new QMdiArea;
    m_pma->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    m_pma->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    setCentralWidget(m_pma);



    toolBar = new QToolBar(this);
    toolBar->setIconSize(QSize(48,48));
```

```
    addToolBar(toolBar);

    actionNew = new QAction(QIcon(":/icon/new.png"),tr("New"));
    actionNew->setShortcut(Qt::CTRL+Qt::Key_N);
    connect(actionNew,SIGNAL(triggered(bool)),this,SLOT(slotNew()));
    toolBar->addAction(actionNew);

    actionOpen = new QAction(QIcon(":/icon/open.png"),tr("Open"));
    actionOpen->setShortcut(Qt::CTRL+Qt::Key_O);
    toolBar->addAction(actionOpen);

    actionSave = new QAction(QIcon(":/icon/save.png"),tr("Save"));
    actionSave->setShortcut(Qt::CTRL+Qt::Key_S);
    toolBar->addAction(actionSave);

    actionHelp = new QAction(QIcon(":/icon/help.png"),tr("Help"));
    toolBar->addAction(actionHelp);

    actionExit = new QAction(QIcon(":/icon/exit.png"),tr("Exit"));
    actionExit->setShortcut(Qt::ALT+Qt::Key_X);
    connect(actionExit,SIGNAL(triggered(bool)),this,SLOT(close()));
    toolBar->addAction(actionExit);

    actionCascade = new
QAction(QIcon(":/icon/cascade.png"),tr("Cascade"));

connect(actionCascade,SIGNAL(triggered(bool)),m_pma,SLOT(cascadeS
ubWindows()));
    toolBar->addAction(actionCascade);

    actionTile = new QAction(QIcon(":/icon/tile.png"),tr("Tile"));

connect(actionTile,SIGNAL(triggered(bool)),m_pma,SLOT(tileSubWindo
ws()));
    toolBar->addAction(actionTile);

    QMenu *menu = menuBar()->addMenu(tr("&File"));

    menu->addAction(actionOpen);
```

```
    menu->addAction(actionSave);
    menu->addAction(actionExit);

    menu = menuBar()->addMenu(tr("&Window"));

    menu->addAction(actionTile);
    menu->addAction(actionCascade);

    menu = menuBar()->addMenu(tr("&Help"));
    menu->addAction(actionHelp);

    statusBar()->addWidget(new QLabel(tr("Status bar")));

}

MainWindow::~MainWindow()
{

}

void MainWindow::slotNew()
{
    qDebug()<<"New";
    QLabel *labelText = new QLabel(tr("SubWindow"),m_pma);
    m_pma->addSubWindow(labelText);
    labelText->show();
}
```

icons.qrc

```
<RCC>
    <qresource prefix="/">
        <file>icon/open.png</file>
        <file>icon/save.png</file>
        <file>icon/help.png</file>
        <file>icon/exit.png</file>
        <file>icon/new.png</file>
        <file>icon/cascade.png</file>
```

```
    <file>icon/tile.png</file>
  </qresource>
</RCC>
```

3.3. Задание для самостоятельного выполнения

Разработать два варианта текстового редактора, с использованием класса QMainWindow. Первый вариант по SDI технологии, а второй по MDI. В программе должны присутствовать основные методы для работы с текстовыми файлами:

- Открытие файлов;
- Создание файлов;
- Сохранение файлов;
- Диалог поиска;
- Установка шрифтов для текста документа;
- Справка (сведения о программе и разработчике, сведения о Qt);
- Работа с буфером обмена (копирование, вставка и т.д.).

Все методы должны быть реализованы как через основное меню программы, так и через панель инструментов

4. Содержание отчета

Наименование и цель работы. Краткая основная теоретическая часть, которая применялась вами при выполнении работы. Комментарии, а также снимки экрана для каждого пункта лабораторной работы, которые отражали бы весь ход выполнения. Вывод о полученных навыках и знаниях после выполнения работы.

Контрольные вопросы для самопроверки

1. Для чего предназначен класс QMainWondow?
2. Какие основные компоненты содержит в себе данный класс?
3. В чем суть SDI технологии?
4. В чем суть MDI технологии?
5. Что такое QToolBar?
6. Что такое QstatusBar?

Литература

1. Лафоре, Р. Объектно-ориентированное программирование в С++ / Р. Лафоре. – СПб. : Питер, 2019. – 928 с.
2. Саммерфилд, М. Qt – профессиональное программирование: разработка кроссплатформенных приложений на С++ / М. Саммерфилд. – СПб. : Символ-Плюс, 2011. – 560 с.
3. Боровский, А. Н. Qt4.7 Практическое программирование на С++ / А. Н. Боровский. – СПб. : БВХ-Петербург, 2012 – 496 с.
4. Лаптев, В. В. С++ объектно ориентированное программировании / В. В. Лаптев. – СПб. : Питер, 2008. – 464 с.
5. Кьоу, Д. Объектно ориентированное программирование / Д. Кьоу. – СПб. : Питер, 2016. – 505 с.
6. Шлее, М., Qt 5.3. Профессиональное программирование на С++ / М. Шлее. – СПб. : БХВ-Петербург, 2015. – 928 с.

Содержание

	стр.
Лабораторная работа № 12. Элементы ввода, управления и выбора ……	3
Лабораторная работа № 13. Управление событиями……..	44
Лабораторная работа № 14. Компоненты для работы с мультимедиа……..	58
Лабораторная работа № 15. Многопоточность в приложении. Синхронизация потоков....	89
Лабораторная работа № 16. Работа с диалогами. Класс QDialog……...	107
Лабораторная работа № 17. Построения основного окна приложения на основе MainWindow……..…….	129
Литература……..….	146

Сахарук Андрей Владимирович

ОСНОВНЫЕ КЛАССЫ QT И ИХ ПРИМЕНЕНИЕ

Практикум
по выполнению лабораторных работ
по дисциплине «Технология разработки программного обеспечения систем управления»
для студентов специальности
1-53 01 07 «Информационные технологии и управление в технических системах»
дневной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 20.01.21.
Рег. № 74Е.
http://www.gstu.by