GeekBrains

C++: разработка программ с графическим интерфейсом на Qt

Дополнительные инструменты разработки и средства сборки Qt

Отладка программного продукта. Контроль версий

Тестирование программ. Типы ошибок. QDebug

Тестирование программ

Даже опытные программисты ошибаются при реализации алгоритма. Ошибки встречаются и во фреймворках. Поэтому перед релизом программного продукта нужно проверить его на работоспособность, в том числе и на различных платформах. Тестирование в общем случае — это сравнение реального результата работы программы с ожидаемым, в том числе и в случаях, когда пользователь вводит не предусмотренные разработчиками данные. Основная цель тестирования — повысить вероятность того, что приложение будет работать правильно при любых обстоятельствах.

Типы ошибок

Логические ошибки

Это наиболее серьезные ошибки. Программа компилируется, запускается и работает без сбоев, но выдает неверный результат. Ошибки могут скрываться в алгоритме или в самом коде. Их поиск и устранение требует анализа алгоритма.

Например, рассмотрим простейший пример логической ошибки: имеется функция, которая принимает два аргумента и должна вернуть их среднее арифметическое.

```
int getMean(int var1, int var2)
{
    return var1 + var2 / 2; // вместо (var1 + var2)/2;
}
```

Результат работы этой функции будет неверным, но никакими дебаггерами такого рода ошибки обнаружить нельзя. Вот еще один пример — заполнение массива 400×400 элементов:

```
for (int i = 0; i < 400; i++)
{
    for(j = 0; i < 400; i++) // i не была заменена на j в условии и в шаге
                             // цикла
    {
        Massiv[i][j] = func(i,j);
    }
}
```

Массив заполнится не полностью.

Синтаксические ошибки

Синтаксические ошибки, в отличие от логических, обнаруживаются на стадии компиляции. Возникают они при нарушении пунктуации, например при отсутствии точки с запятой в конце строки, вводе строки или символа без кавычек.

```
int a = b + c
printf("Value a = %d", a);
QString s = Hello mistake;
QChar separate = \n;
```

Самые часто встречающиеся синтаксические ошибки таковы:

- опечатка в написании названий методов или переменных;
- пропуск знаков препинания (например, точки запятой в конце строки в С-подобных языках);
- написание операторов с ошибкой (если команда написана неверно, компилятор сообщит, что это неизвестное ему наименование оператора, типа данных и т. д.);
- неправильное использование команд;
- определение/объявление переменных (если переменная не определена, компилятор выведет сообщение об ошибке);
- потерянные скобки (количество открывающих и закрывающих скобок не совпадает).

QDebug

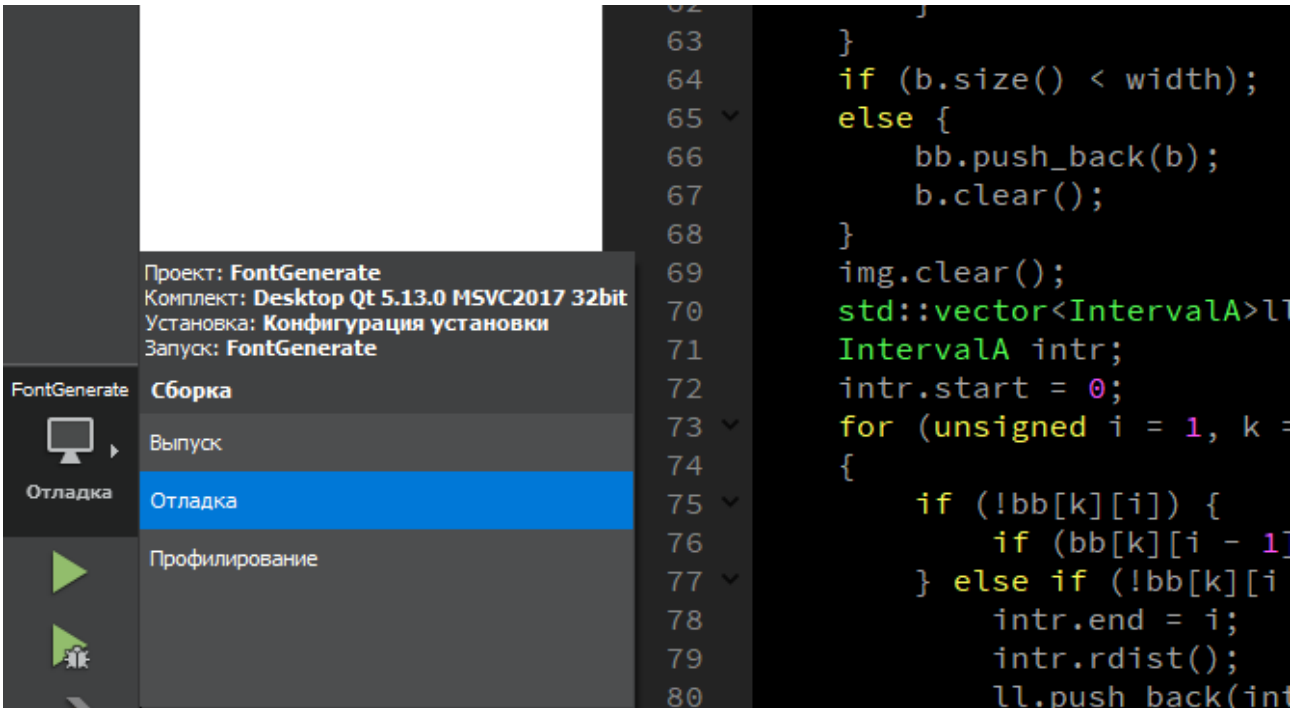
При разработке программного обеспечения возможно допустить ошибки. Для исправление ошибок возникает необходимость просматривать статус приложения, которое можно реализовать, выводя текст в терминал. Для нахождения ошибки возможна использовать вывод отладочной и трассировочной информации в виде текста. Также можно определить какие и последовательность выполнения разных кусков кодов или процедур для выполнения логических операций.

QDebug используется всякий раз, когда разработчику необходимо записать отладочную или трассировочную информацию на устройство, файл, строку или консоль.

```
for (int i = 0; i < N; i++)
{
   qDebug() << QString::number(i); // похоже на стандартный C++ std-метод cout
   ...
   qDebug() << "<Отладочная информация, которая может содержать вычисленное
значение>";
}
```

Пошаговая отладка. Контрольные точки

Бывает, что найти ошибку, просто просматривая код, проблематично — например, если в программе много ветвлений (if, switch). На этот случай были разработаны средства для отладки на прикладном уровне и на уровне ядра ОС. В Qt Creator интегрированы отладочные средства для различных платформ, под которые разрабатывается ПО. Для работы отладчика приложение должно быть собрано с отладочной информацией: консольная утилита **qmake** создает файл с правилами сборки **MakeFile.Debug**. В Qt Creator такое поведение настраивается через боковое меню.



Для установки точки останова (контрольной точки) в IDE Qt Creator достаточно поставить курсор на строку, на которой программа должна остановиться, и нажать F9. Ее также можно установить, щелкнув мышью слева от номера строки или выбрав в меню **Отладка** пункт **Поставить/снять точку останова**. Убрать контрольную точку (точку останова) можно аналогично. На рисунке точка останова установлена на строке 49.

```
39      QString fileName = QFileDialog::getOpenFileName(
40              this,
41              "PNG Font",
42                  "C:\\Users\\Vaciv\\Pictures",
43              tr("PNG(*.png)" ),
44              &selfilter
45      );
46      if (fileName.length() == 0) return;
47      unsigned width, height;
48      std::vector<unsigned char>img;
49      std::string ch = fileName.toStdString().c_str();
50      lodepng::decode(img, width, height, ch);
51      std::vector<char>b;
52      std::vector<std::vector<char>>bb;
53      int count = img.size() >> 2;
54      int *sa = (int*)&img[0];          use of old-style cast   cast from 'unsigned char *' to 'int *' increa
55      for (int i = 0; i < count; i++) {
56          if (sa[i] == -1) b.push_back(0);
57          else b.push_back(1);
58          if (b.size() < width);
59          else {
60              bb.push_back(b);
61              b.clear();
62          }
63      }
64      if (b.size() < width);
65      else {
66          bb.push_back(b);
67          b.clear();
68      }
69      img.clear();
70      std::vector<IntervalA>ll;
71      IntervalA intr;
72      intr.start = 0;
```

Запустим программу в режиме отладки, нажатием клавиши F5 или кнопки в боковом меню (кнопка с изображением жука).

```
53  #include <QtCore/QCoreApplication>
54
55  #include <QtGui/QOpenGLContext>
56  #include <QtGui/QOpenGLPaintDevice>
57  #include <QtGui/QPainter>
58
59  //! [1]
60  OpenGLWindow::OpenGLWindow(QWindow *parent)
61      : QWindow(parent)
62      , m_animating(false)
63      , m_context(0)                                                    zero as n
64      , m_device(0)                                                     zero as n
65  {
66      setSurfaceType(QWindow::OpenGLSurface);
67  }
68  //! [1]
69
70  OpenGLWindow::~OpenGLWindow()
71  {
72      delete m_device;
73  }
74  //! [2]
75  void OpenGLWindow::render(QPainter *painter)
76  {
77      Q_UNUSED(painter);
78  }
79
80  void OpenGLWindow::initialize()
81  {
82  }
83
84  void OpenGLWindow::render()
85  {
86      if (!m_device)
```

Когда программа начинает выполнять участок кода, строка которого отмечена в качестве точки останова, программа перестает выполняться (входит в состояние паузы). Отладчик считывает значения всех переменных, существующих в программе на момент останова. Для продолжения необходимо нажать либо F5, либо кнопку в боковом меню. Все действия с отладкой можно запустить через пункт меню **Отладка**, в том числе и отвязать программу от отладчика. Отладку можно запустить и без точек останова, нажав сочетание клавиш **Shift-F10** (**Пройти через**) и **Shift-F11** (**Войти в**).

Пункт **Войти в** позволяет при отладке кода пройти по каждой строке кода внутри функции. Пункт **Пройти через** пропускает вложенный код в функцию, и переходит к следующей после функции инструкции.

Окна переменных и цепочки вызова

В данном примере продемонстрированы все переменные, используемые программой в данный момент. Переменные, объявленные в конструкторе класса, расположены в структуре **this**.

Модульное тестирование. Тестирование графического интерфейса

Модульное тестирование

Модульное тестирование (unit testing) — это тестирование атомарной функциональности приложения по отдельности в искусственно созданной среде.

Шаблон модульного теста имеет следующий вид:

```
#include <QtTest/QtTest>

class TestQString: public QObject
{
    Q_OBJECT
```

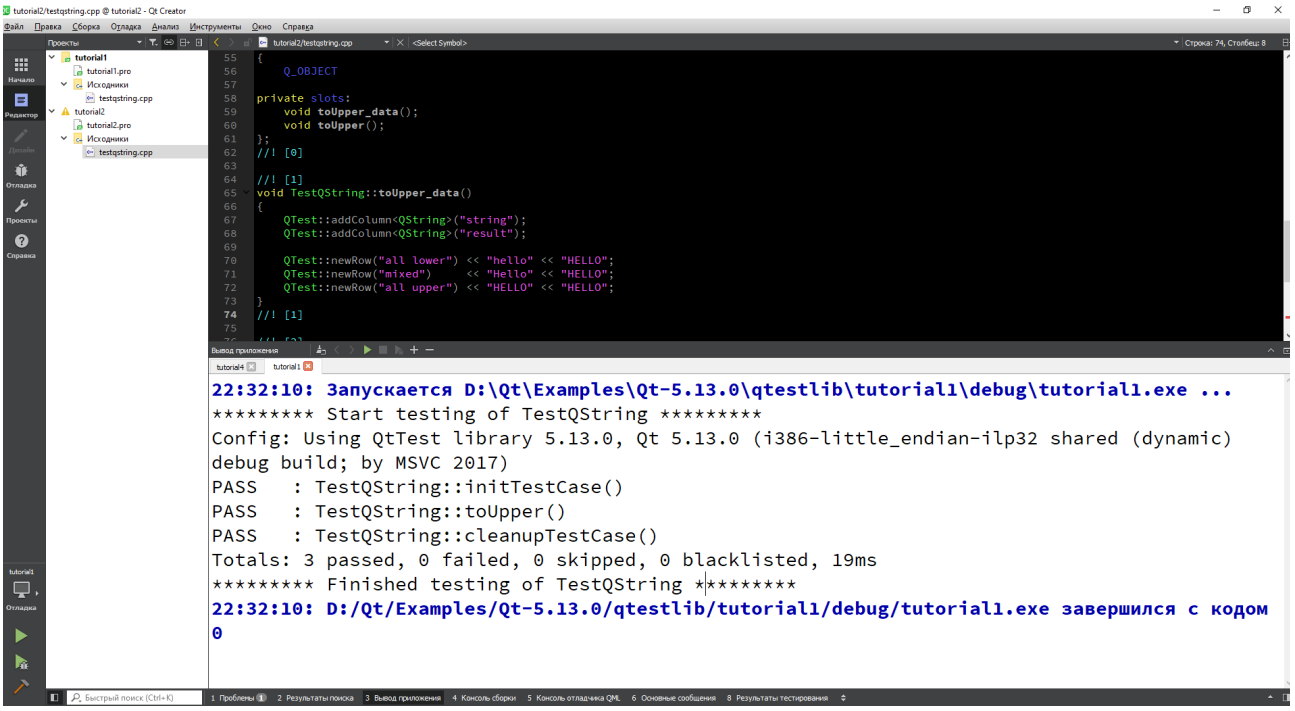
```
private slots:
   void toUpper();
};

void TestQString::toUpper()
{
   QString str = "Hello";
   QCOMPARE(str.toUpper(), QString("HELLO"));
}
QTEST_MAIN(TestQString)
#include "testqstring.moc"
```

Также необходимо подключить модуль, который содержит необходимые заголовки и библиотеки для выполнения тестирования — **QT+=testlib**.

Простейший пример сравнения ожидаемого и полученного результата:

```
#include <QtTest/QtTest>

class TestQString: public QObject
{
   Q_OBJECT

private slots:
   void toUpper_data();
   void toUpper();
};
void TestQString::toUpper_data()
{
   QTest::addColumn<QString>("string"); // Добавляем переменную в стек отладки
   QTest::addColumn<QString>("result");

   QTest::newRow("all lower") << "hello" << "HELLO";
   QTest::newRow("mixed")     << "Hello" << "HELLO";
   QTest::newRow("all upper") << "HELLO" << "HELLO";
}
void TestQString::toUpper()
{
   QFETCH(QString, string);
   QFETCH(QString, result);

   QCOMPARE(string.toUpper(), result);
}
QTEST_MAIN(TestQString)
#include "testqstring.moc"
```

Компилируем и запускаем приложение:



Тестирование графического интерфейса

Простейший пример тестирования графического интерфейса выглядит так:

```
#include <QtWidgets>
#include <QtTest/QtTest>

//! [0]
class TestGui: public QObject
{
    Q_OBJECT

private slots:
    void testGui_data();
    void testGui();
};
//! [0]

//! [1]
class TestGui: public QObject
{
    Q_OBJECT

private slots:
    void testGui();

};
//! [0]
```

```
//! [1]
void TestGui::testGui()
{
   QLineEdit lineEdit;
   QTest::keyClicks(&lineEdit, "hello world");
   QCOMPARE(lineEdit.text(), QString("hello world"));
}
//! [1]

//! [2]
QTEST_MAIN(TestGui)
#include "testgui.moc"
//! [2]
```

Через статические методы класса **QTest** можно эмулировать различные действия клавиатуры и мыши. Список доступных функций можно найти в официальной документации.

```
QTest::mouseMove(<виджет>, QPoint(40, 50));  // Перемещаем курсор
QTest::mousePress(<виджет>, Qt::LeftButton); // Имитируем нажатие кнопки
```

Практика

Подключим модульное тестирование и тестирование графического интерфейса к проекту из урока 2.

Unit-тестирование обычно проводится в созданном отдельном проекте, а не в основном коде. Основная цель тестирования — проверить насколько правильно работает участок кода класса в приложении, модуле или библиотеке.

Добавим в файл проекта модуль **testlib**, чтобы иметь возможность использовать класс **QTest**. Создадим новый класс **UnitTest** на базе класса **QObject**. Для модульного тестирования, как и тестирования графического, необходимо использовать слоты. Файл с расширением **.pro** примет следующий вид:

```
QT       += core gui testlib

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = sample
TEMPLATE = app

DEFINES += QT_DEPRECATED_WARNINGS

CONFIG += c++14

SOURCES += \
       main.cpp \
```

```
       mainwindow.cpp \
       parsetext.cpp \
       unittest.cpp

HEADERS += \
       mainwindow.h \
       parsetext.h \
       unittest.h

FORMS += \
       mainwindow.ui

DESTDIR =../
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Тестирование происходит по всем приватным слотам класса. Добавим два слота:

```
#ifndef UNITTEST_H
#define UNITTEST_H

#include <QObject>
#include <QtTest/QTest>

class UnitTest : public QObject
{
   Q_OBJECT
public:
   explicit UnitTest(QObject *parent = nullptr);

signals:

private slots: // Слоты должны быть приватными, иначе работать не будет
   void testCalc();
   void testGUI();
};
#endif // UNITTEST_H
```

Теперь, используя макросы, выполним проверку работы метода **calc()**. Для тщательной проверки необходимо написать столько кейсов, сколько необходимо для полной проверки алгоритма в различных ситуациях: например, для метода вычисления математических выражений нужно проверить работу алгоритма с различными арифметическими действиями и их комбинациями:

```
#include "unittest.h"
#include "parsetext.h"
#include "mainwindow.h"
```

```
UnitTest::UnitTest(QObject *parent) : QObject(parent)
{

}

void UnitTest::testCalc()
{
   ParseText ptxt;
   QCOMPARE(ptxt.calc("2+2="), "2+2=4");                    // Вспоминаем,
                                                            // что
                                                            // отправлялось
                                                            // в классе
   QCOMPARE(ptxt.calc("8/2="), "8/2=4");
   ...
}

void UnitTest::testGUI()
{
   MainWindow w;
   w.show();
   QTest::keyClicks(w.getPlainText(), "Test #@12+2=");      // Ввводим текст
   QCOMPARE(w.getPlainText()->toPlainText(), "Test 12+2=14"); // Проверяем
}
```

При запуске приложения никаких изменений видно не будет, так как для тестирования нужно запустить класс тестирования.

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

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

   QTest::qExec(new UnitTest(), argc, argv); // Запускаем выполнения теста
   return  0;
}
```

Виды сборок проекта (отладка, вывод, профилирование)

Основные сборки проекта:

1. Отладка. Программа создается с дополнительными спецсимволами, служащими для отладки разрабатываемого приложения.
2. Выпуск. Программа собирается с оптимизацией кода по скорости выполнения либо по размеру исполняемого файла. Эта настройка указывается в pro-файле, добавление флага **QMAKE_CXXFLAGS**.

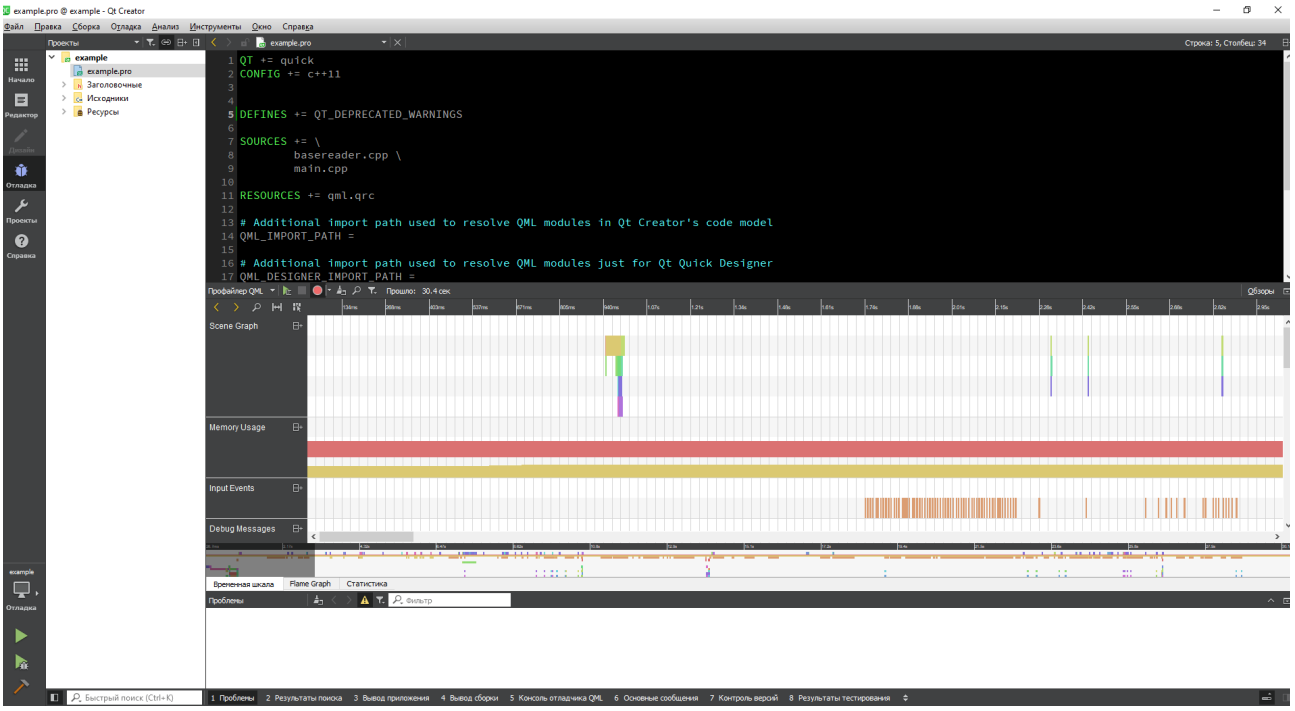
```
QT += quick
CONFIG += c++11
DEFINES += QT_DEPRECATED_WARNINGS
QMAKE_CXXFLAGS += O2 // Оптимизировать все, что можно, но только проверенные и
                     // надежные оптимизации
SOURCES += \
       basereader.cpp \
       main.cpp
RESOURCES += qml.qrc
# Additional import path used to resolve QML modules in Qt Creator's code model
QML_IMPORT_PATH =
# Additional import path used to resolve QML modules just for Qt Quick Designer
QML_DESIGNER_IMPORT_PATH =
# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
HEADERS += \
   basereader.h
```

3. Профилирование. Сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш-промахов и т. д. Для анализа работы используется инструмент, который называют профилировщиком (профайлером). Обычно профилирование выполняется совместно с оптимизацией программы. Оно позволяет отслеживать, сколько ресурсов ПК (память, события от устройств ввода, вывод отладочной информации) задействует программа, по временной шкале.

Для выполнения профилирования собираем проект с настройками профилирования. В Qt Creator можно выполнить профилирование для стандартной Qt Gui (Valgrind) и QML, эти варианты можно выбрать в меню **Анализ**. Рассмотрим профилирование на примере QML-программы из предыдущего урока.

1. Запускаем проект QML.
2. В настройках сборки устанавливаем профилирование.
3. Выполняем сборку.
4. В меню **Анализ** выбираем пункт **Профилирование QML**.
5. Выполняем какие-нибудь действия в программе (например, добавляем задачу).

6. Завершаем работу программы.

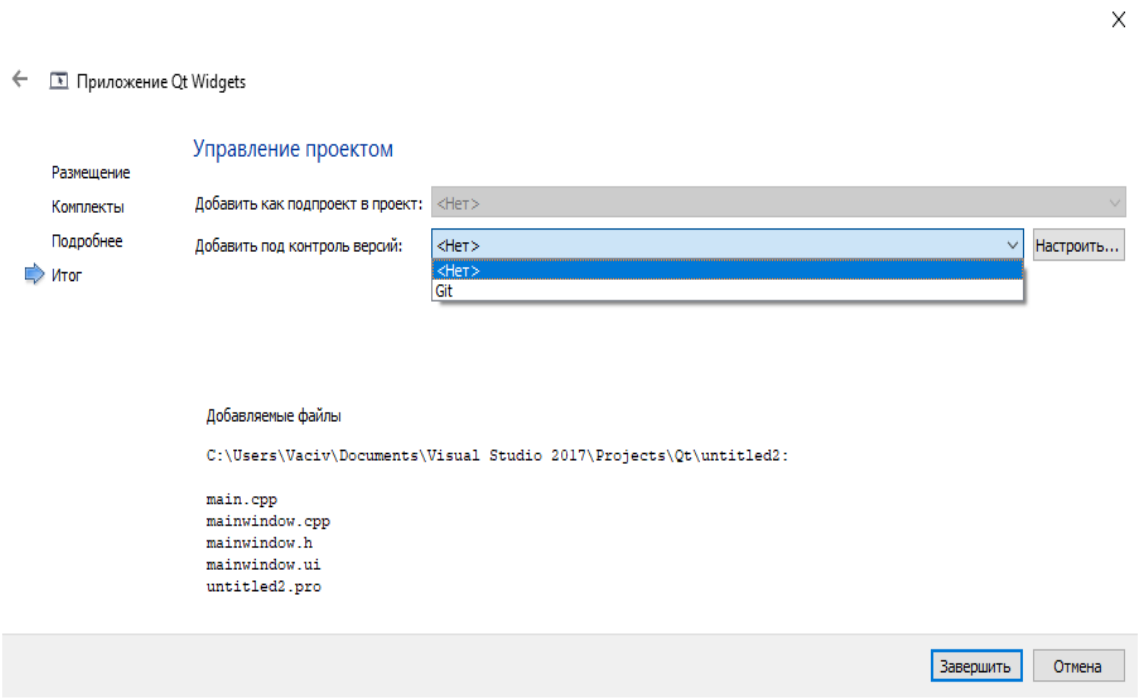


Qt Creator и системы контроля версий. Git

Крупные проекты разрабатываются группой людей, и возникает вопрос, как координировать разработку. Нередко возникает и другая ситуация: после написания основного кода возникает необходимость добавить функционал или перейти на использование других технологий. Во втором случае после правок может перестать работать и тот функционал, который работал до них. Можно, конечно, создавать копии проектов и архивировать, но это громоздкое решение: в копиях легко запутаться.

Систему контроля Git можно подключить к проекту на последнем шаге создания проекта в Qt Creator.

Все возможные действия по работе с контролем версий осуществляются через меню **Инструменты**. Если проект был создан без подключения контроля версии, то его можно добавить через меню **Инструменты — Git — Создать локальное хранилище**.



Доступные операции:

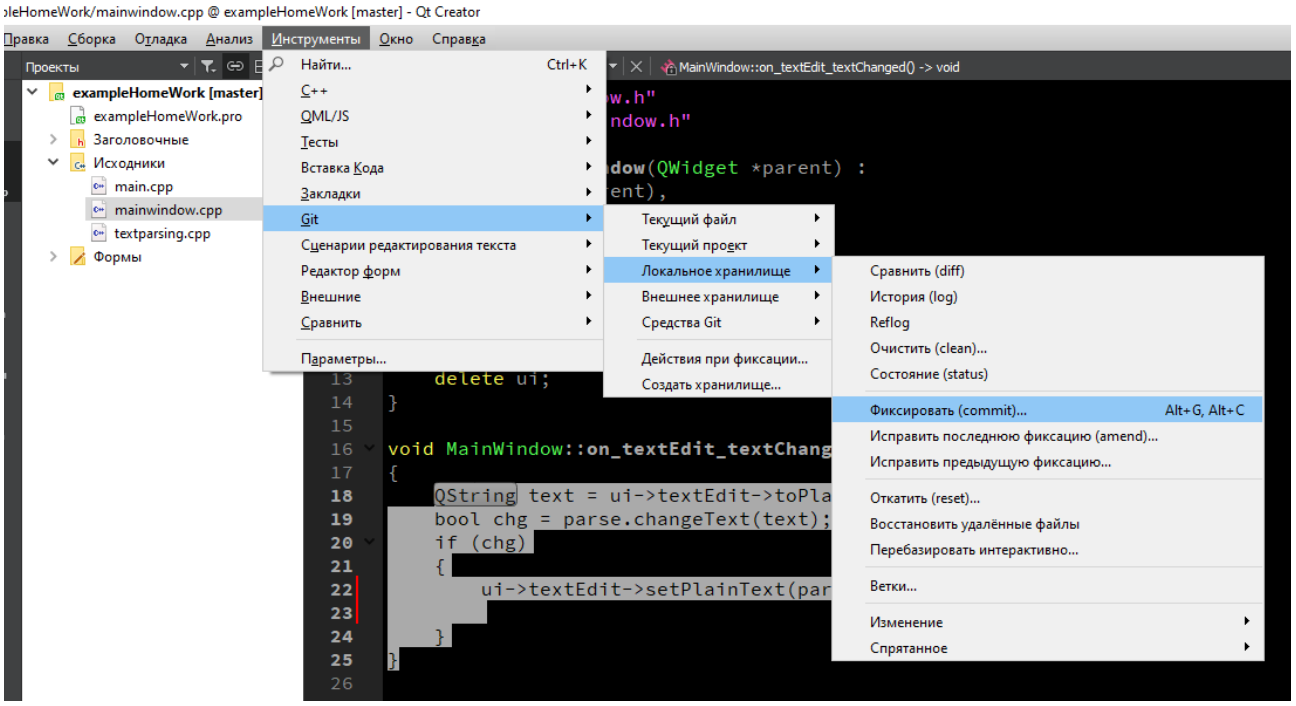
- инициализация (init) — создание локального репозитория;
- фиксации (commit) — сохранение изменений;
- сравнение (diff) — сравнение текущих изменений с последней фиксированной версией;
- ветвление (branch) — создание отдельных ветвей проекта;
- загрузка/выгрузка проекта (pull/push) — можно как отправить репозиторий на существующий сервер, хранящий репозитории (Github, BitBucket), так и создать собственный онлайн-менеджер репозиториев в локальной сети или в Интернете.

При использовании систем контроля версий можно создать несколько отдельных ветвей проекта (главная ветвь называется **master**). Обычно создаются отдельные ветви для доработки, добавлений нового функционала, а доступом к главной ветви располагает руководитель проекта, который добавляет изменения в главную ветвь (яркий пример такого подхода — работа над ядром Linux).

Например, есть такой код:

```
QString text = ui->textEdit->toPlainText();
    bool chg = parse.changeText(text);
    if (chg)
    {
        ui->textEdit->setPlainText(parse.getText());
    }
}
```

Фиксируем текущий код в репозитории:

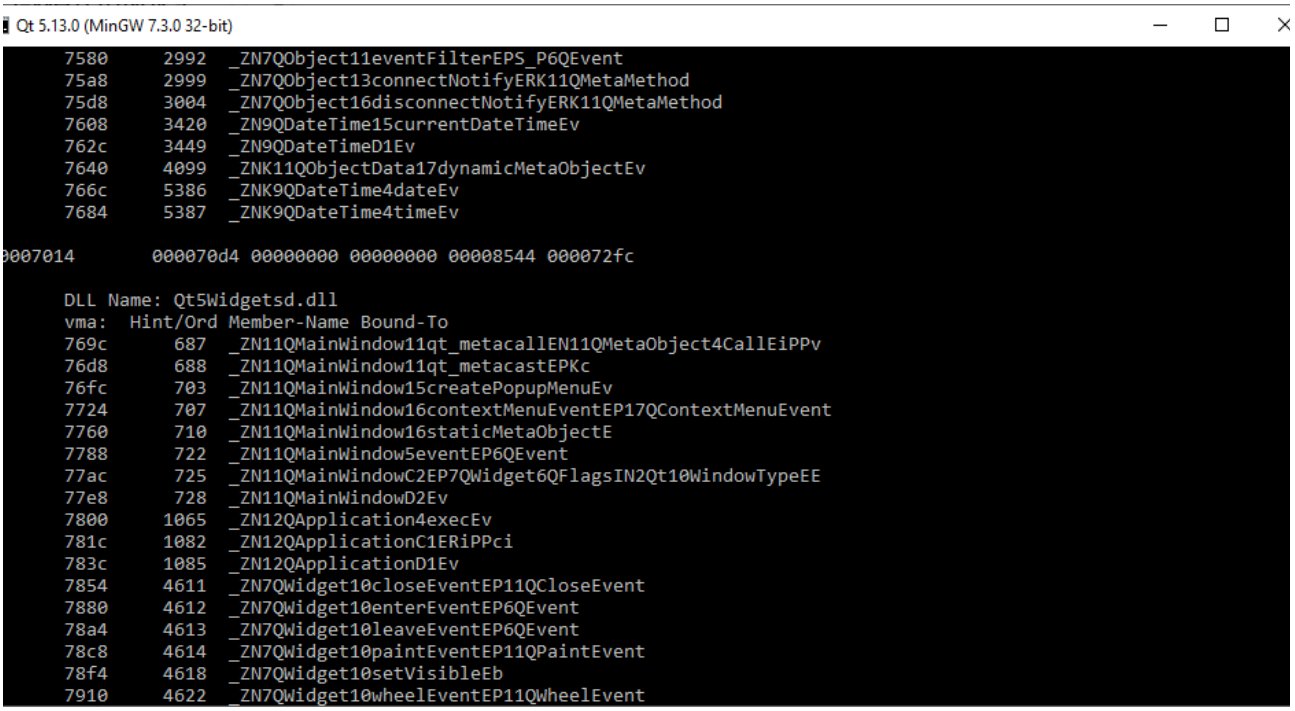


Поиск взаимосвязанных библиотек

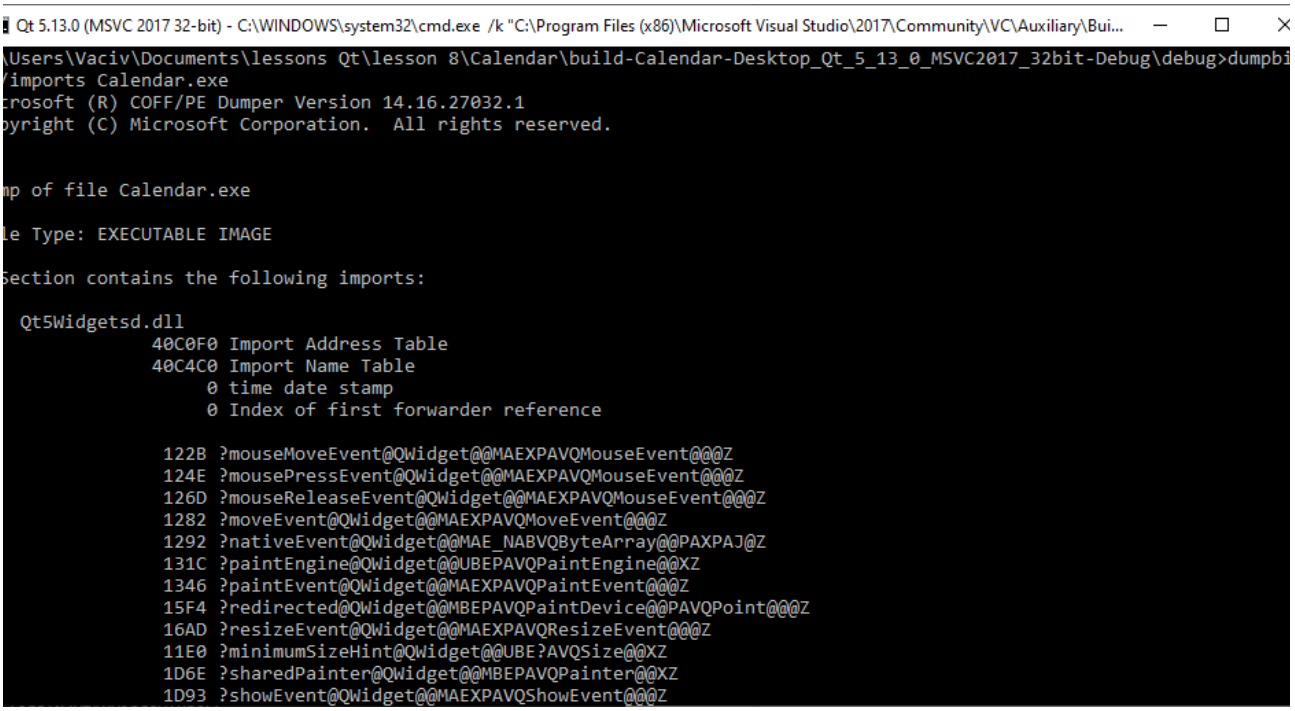
Путь до каталога поиска библиотек указывается командой **-L"<PATH>"**. У каждого исполняемого файла есть таблица импорта, где указываются используемые функции и динамические библиотеки, в которых находятся необходимые функции. Динамические библиотеки содержат таблицы импорта и экспорта. Таким образом, считывая таблицы, можно узнать, какие библиотеки необходимы проекту для корректной работы.

Для OS X и ОС семейства Linux для получения таблицы импорта можно использовать приложение **nm**. Для Windows — **dumpbin** (Visual Studio) или **objdump** (MinGW).

Определим какие динамические библиотеки использует наше приложение, считав таблицу импорта. Для Linux, OS X и Windows (при использовании MinGW) работает один и тот же способ: переходим через терминал к приложению и выполняем в терминале команду **objdump -p <имя исполняемого файла>**.



А при использовании инструментов, входящих в состав Visual Studio, вводим в терминале команду **dumpbin /imports <имя исполняемого файла>**.



Таким образом можно увидеть импортируемые функции приложением и имена динамических библиотек, которые будут подключены при запуске приложения.

Как отправить приложение заказчику (без исходников), чтобы оно точно у него запустилось

Названия библиотек, которые нужно подключить при линковке создаваемого приложения, и используемые при этом команды могут различаться в зависимости от платформы. Например, кроссплатформенное подключение осуществляется при помощи команды **LIBS+=-l<namelib>**. Если эту библиотеку нужно подключить только для ОС Windows, то строка в файле **.pro** должна быть следующей: **win32{LIBS+=-l<namelib>}**. Вариант для OS X — **osx{LIBS+=-l<namelib>}**, для ОС на базе ядра Linux — **linux{LIBS+=-l<namelib>}**.

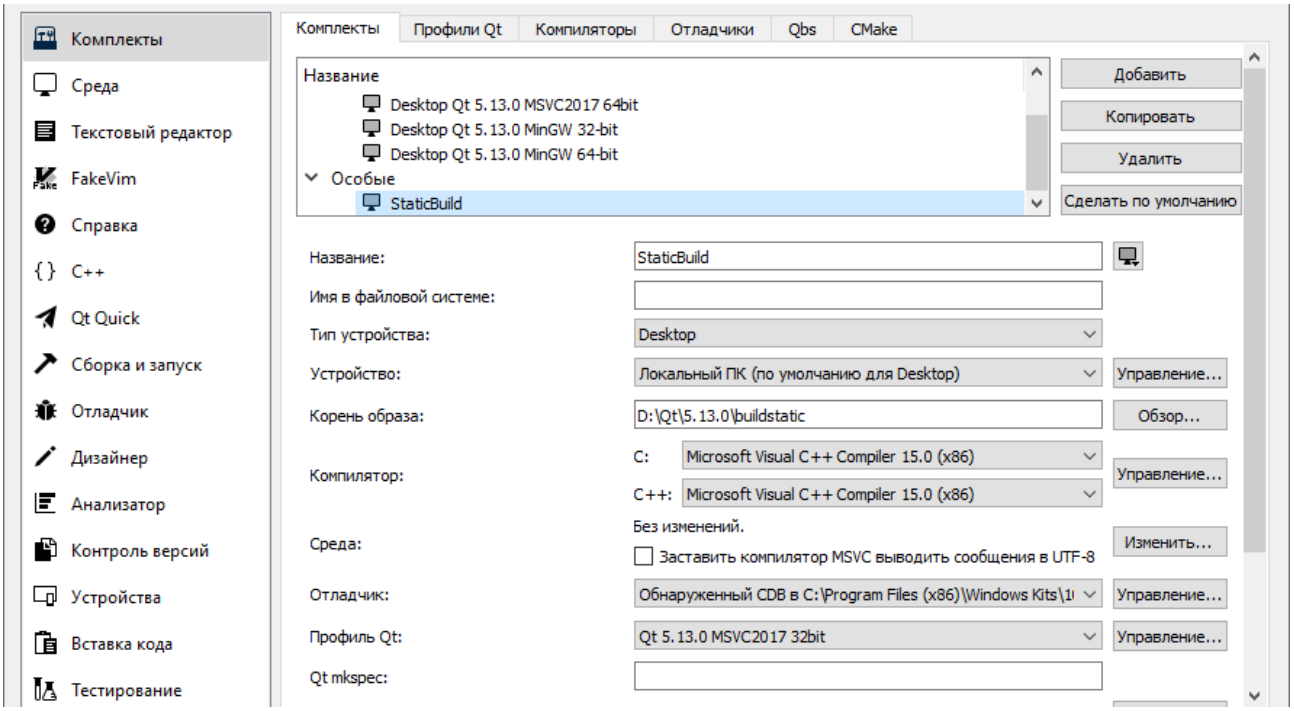
Для запуска приложения без исходного кода достаточно скопировать в папку с исполняемым файлом необходимые динамические библиотеки (Windows) или прописать зависимость библиотек для ОС Linux. Для разных сборок ОС Linux оформление зависимостей библиотек может отличаться при использовании разных утилит установки — **aptitude** (Debian, Ubuntu, Mint) или **pacman** (Arch).

Можно собрать приложения со статическими библиотеками, однако это возможно при условии, что исходный код программы будет в открытом доступе, или при использовании коммерческой версии фреймворка Qt.

Для статической сборки нужно:

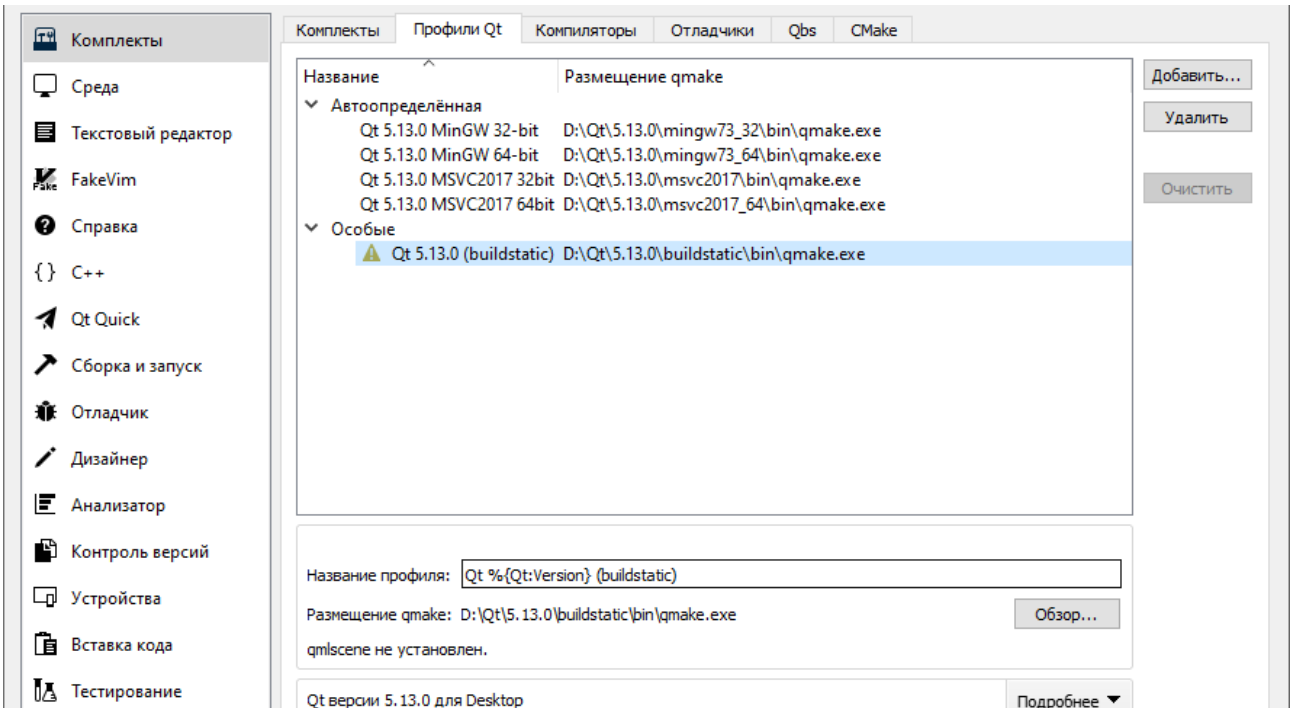
1. Скачать исходный код Qt. Если он не был установлен вместе с фреймворком, можно загрузить его через утилиту **maintenanceTool**, либо скачать архив исходного кода нужной версии с официального сайта архива Qt.
2. Установить Python (часть пакетов Qt собирается с использованием скриптов, написанных на Python).
3. Запустить конфигуратор: **configure -prefix <каталог установки платформы> -platform <платформа под которую будет произведена компиляция> -static -nomake example**.
4. Выбрать открытую лицензию или коммерческую, если приобретена коммерческая версия.
5. Принять лицензионное соглашение.
6. Создастся новый **MakeFile**. Для MS Visual Studio нужно запустить команду сборки: **nmake** (**make, mingw32-make**).
7. После завершения компиляции выполняется установка статической библиотеки (Visual Studio: **nmake(make, mingw32-make) install**).

8. Добавить новую платформу в Qt Creator (в меню **Инструменты — Параметры**):



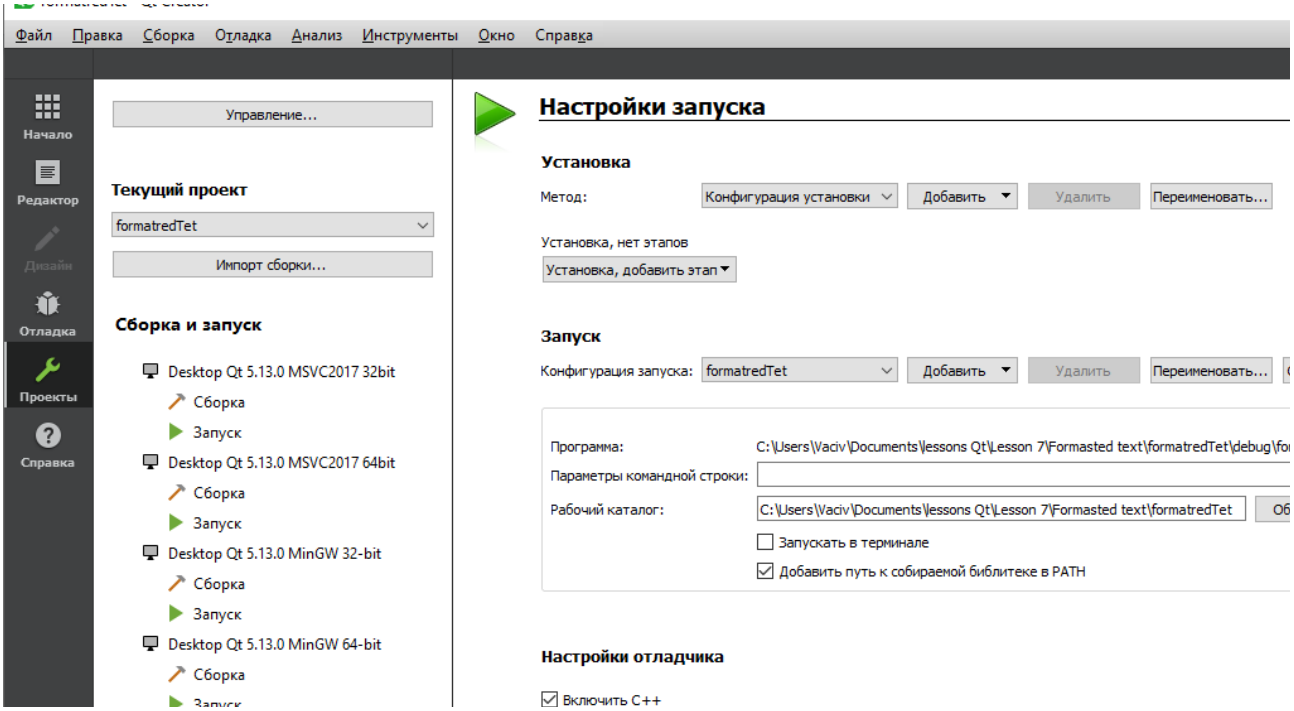
Указать имя сборки. Указать путь к каталогу (**Корень сборки**), который в конфигурации был указан в параметре **prefix.**

9. Указать профиль Qt. Нажать **Управление**. Добавить новый профиль, указав в **qmake** путь до собранного проекта: **<Путь к каталогу>/bin/qmake**.

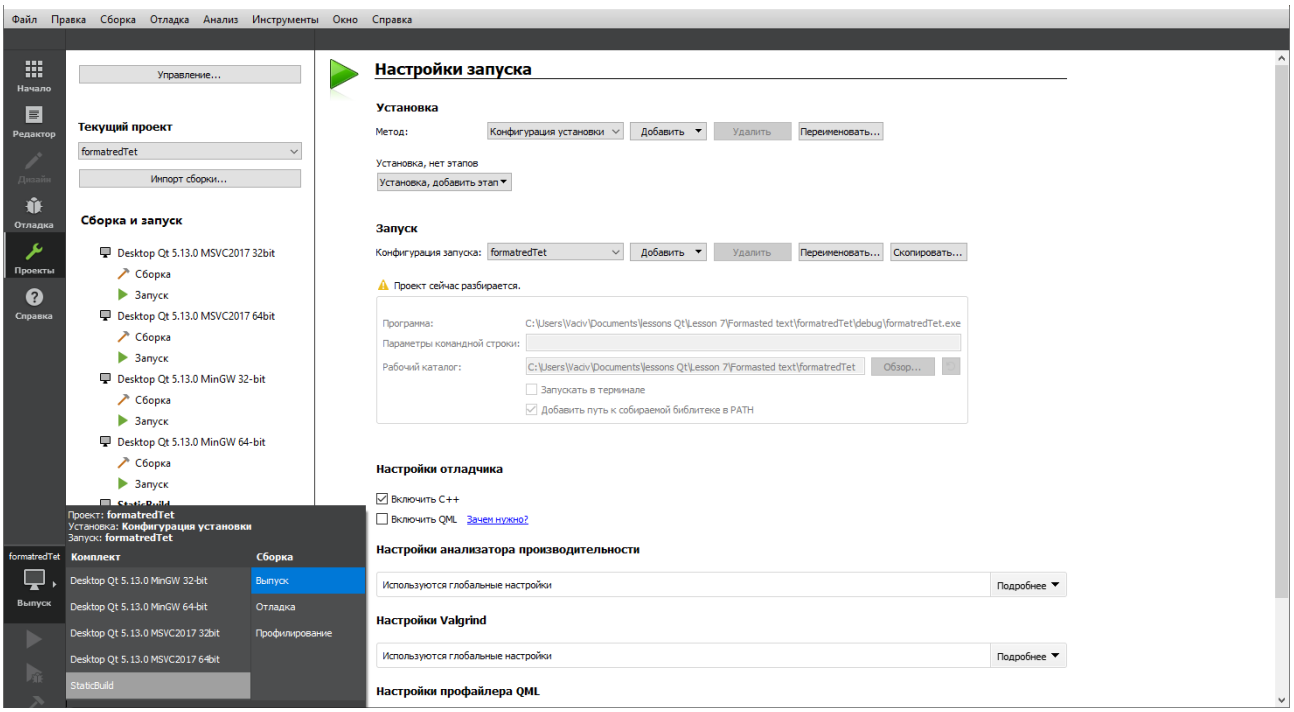


10. Принять изменения.
11. Открыть проект, для которого необходима статическая сборка.
12. В редакторе нажать кнопку **Проекты** (с изображением гаечного ключа).

13. В поле **Сборка и запуск** выбрать пункт с названием добавленного комплекта:



14. Выбрать платформу для сборки:



15. Собрать проект.

Теперь для запуска программы на другом компьютере будет достаточно одного исполняемого файла.

В настройках конфигурации сборки пакета Qt нужно внимательно указывать путь установки и параметры. Желательно исключить из сборки примеры (**-nomake examples**), так как сборка достаточно долгая (со сборкой примеров занимает более семи часов). Если при конфигурации была допущена ошибка с параметрами включения модулей или не было указано, что нужна статическая

сборка, то пакет нужно переконфигурировать. Для сброса конфигурации нужно выполнить в терминале следующие команды: **(make, nmake, mingw32-make) distclean.** Если же необходимо выполнить переконфигурацию сборки Qt (скажем, вы забыли указать флаг статической сборки), то нужно добавить параметр **-recheck-all** (шаг 3). Все параметры конфигурации можно получить, введя в терминал команду **configure --help**.

Практическое задание

1. Для проекта из предыдущего урока (планировщик задач) добавить отображение количества введенных задач. При этом должны учитываться и те задачи, которые сохранены в файле.
2. Добавить валидацию вводимых данных. Название задачи - непустая строка, дата - строка в формате dd.mm.yyyy, прогресс задачи - число от 0 до 10. Если хотя бы одно значение не пройдет валидацию, не добавлять задачу и вывести предупреждение пользователю.
3. *(По желанию) Провести модульное тестирование текстового редактора.

Дополнительные материалы

1. Документация QTest. https://doc.qt.io/qt-5/qtest.html
2. Основы Git. https://git-scm.com/book/ru/v1/%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Документация QTest.
2. Основы Git.