**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Физико–технический факультет**

**Кафедра теоретической физики и компьютерных технологий**

**КУРСОВОЙ ПРОЕКТ**

**НАСТРОЙКА СРЕДЫ ДИНАМИЧЕСКОЙ КОМПИЛЯЦИИ ИСПОЛНЯЕМЫХ ФАЙЛОВ**

**Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Хрулев Артем Алексеевич**

**Курс 3**

**Направление 09.03.02 Информационные системы и технологии**

**Научный руководитель**

**старший преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. Н. Значко**

Нормоконтролер инженер **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Г. Д. Цой**

**Краснодар 2018**

СОДЕРЖАНИЕ

[Введение 3](#_Toc512330615)

[1 Знакомство с Qt. Обзор истории 4](#_Toc512330616)

[1.1 Основные составляющие Qt 6](#_Toc512330617)

[1.2 Обзор настроек среды Qt Creator 8](#_Toc512330618)

[2 Структура проекта. Основные типы 9](#_Toc512330619)

[2.1 Файлы проекта 9](#_Toc512330620)

[2.2 Компиляция проекта 11](#_Toc512330621)

[3 Динамические библиотеки и система расширений 14](#_Toc512330622)

[3.1 Динамические библиотеки 14](#_Toc512330623)

[3.2 Динамическая загрузка и выгрузка библиотеки 16](#_Toc512330624)

[3.3 Расширения 18](#_Toc512330625)

[3.3.1 Расширения для Qt 18](#_Toc512330626)

[3.3.2 Поддержка собственных расширений в приложениях 18](#_Toc512330627)

[4 Процессы 20](#_Toc512330628)

[5 Настройка среды динамической компиляции исполняемых файлов 22](#_Toc512330629)

[Заключение 25](#_Toc512330630)

[Список использованных источников 26](#_Toc512330631)

# ВВЕДЕНИЕ

Данная работа посвящена расширению возможностей и изменению функциональности приложения без его перекомпиляции, посредством динамической компиляции и динамической загрузки расширений приложения, то есть динамически загружаемых библиотек. Динамическая компиляция увеличивает производительность программных систем, путём компиляции исходного кода в машинный код непосредственно во время работы программы. Таким образом, достигается высокая скорость выполнения. Для решения этой задачи может помочь интегрированная среда разработки Qt Creator.

Актуальность темы заключается в том, что данная технология применима в любых средах обучения, чтобы учить элементам программирования, предоставляя обучающемуся расширить или улучшить функционал программы, описать свой алгоритм, подправить или добавить часть кода, тут же перекомпилировать и увидеть, как это отобразится в исполняемой программе.

Целью данной работы является настройка среды динамической компиляции исполняемых файлов в среде Qt Creator.

Для достижения данной цели служат следующие задачи:

– познакомиться со средой разработки Qt Creator;

– ознакомиться с инструментами и средствами, необходимыми для рабо-ты с Qt (рассмотреть структуру проекта, этапы компиляции);

– ознакомиться с принципом подключения динамических библиотек, с помощью класса QLibrary;

– ознакомиться с классом создания новых процессов – QProcess;

– применить данную теорию на конкретном примере.

# 1 Знакомство с Qt. Обзор истории

Кроссплатформенный инструментарий разработки Qt появился впервые в 1995 году благодаря своим разработчикам Хаарварду Норду и Айрику Чеймб-Ингу. С самого начала создавался как программный каркас, позволяющий создавать кроссплатформенные программы с графическим интерфейсом. Первая версия Qt вышла 24 сентября 1995. Программы, разработанные с Qt, работали как под управлением операционных систем семейства Microsoft Windows так и под управлением Unix–подобных систем [1].

За годы разработки возможности Qt значительно выросли. Работа с сетью, базами данных, графикой, мультимедиа, Интернет и другие расширения превратили его в универсальный инструментарий для создания программ. Qt превратился в полноценный и мощный инструмент разработки, который значительно превзошёл свои первоначальные возможности.

В июне 1999 года вышла вторая версия – Qt 2.0. А в 2000 году состоялся выпуск версии для встраиваемых систем, который назывался Qt Embedded. Версия Qt 3.0 – 2001 год – работала в ОС семейства Windows и многих Unix–подобных ОС, таких как MaсOS, xBSD, в различных вариантах Linux для персональных компьютеров и встраиваемых систем. Он имел 42 дополнительных класса, объём вырос до более чем 500 000 строк кода. Летом 2005 года состоялся выпуск Qt 4.0, который включал в совокупности около 500 классов и имел огромное количество существенных улучшений. Вместе с выпуском Qt 4.5 вышло и специализированная интегрированная среда разработки QtCreator.

В декабре 2012 состоялся официальный выпуск Qt5. Эта версия кроссплат-форменного средства разработки совместима с Qt4. Qt5 отличается рядом особенностей, улучшений и большим количеством новых возможностей.

Современное программное обеспечение достаточно сложное и должно соответствовать многим требованиям. Кроме пользовательских требований, налагаемых на удобство и возможности программного продукта, есть и другие требования, касающиеся разработки программного обеспечения. Большую роль здесь играют средства, которыми программист пользуется в процессе своей работы. Во многих случаях бывает удобно владеть инструментарием, который имеет достаточно широкую область применения и может служить для решения большого количества задач разного масштаба: от построения небольших программ для создания мощных программных пакетов. Также часто возникает вопрос о поддержке нескольких программных платформ, ведь, ориентируясь только на одну платформу, можно потерять большое количество потенциальных пользователей.

Инструментарий разработки Qt используют для создания кроссплатформенных программ. Здесь под этим утверждением подразумеваются программы, исходный текст которых можно скомпилировать на разных программных платформах (различные разновидности Linux, Windows, MacOS и т.д.) практически без изменений или с незначительными изменениями. Кроме того Qt используют и для разработки программ, имеющих характерный («родной», native) для программного окружения или даже собственный стилизованный интерфейс. Всё это благодаря открытому свободному программному коду, удобному и логическому API и широким возможностям применения.

Qt расширяет возможности программиста с помощью набора макросов, метаинформации и сигнально-слотовых соединений, но использует при этом лишь средства языка C++ и является совместимым со всеми распространёнными современными его компиляторами.

Наряду с традиционным для предыдущих версий Qt способом создания пользовательских интерфейсов, основанный на виджетах – визуальных элементах интерфейса (кнопки, флажки, выпадающие списки, поля ввода, слайдеры и т.д.), Qt5 ставляет большой акцент на использовании технологии QtQuick. В Qt5 некоторые нововведения коснулись и синтаксиса для создания сигнально-слотовых соединений.

Программный код, зависящий от оконной системы в Qt5, был отделён и реорганизован в отдельные библиотеки расширения, что позволило упростить перенос Qt на новые платформы и адаптации для поддержки других оконных систем. Благодаря QPA (Qt Platform Abstraction) в Qt5 реализована поддержка многих платформ для мобильных устройств.

## 1.1 Основные составляющие Qt

На рисунке 1 изображены основные составляющие Qt. Модули и инструменты доступны для разработки под целевые (Reference) и другие (Other) платформы. Средства Qt разделены по назначению на отдельные части – модули. Каждый из модулей выполнен в виде отдельной библиотеки. Разработчик имеет возможность выбрать модули, которые он использует в программе. Модули имеют взаимозависимости: одни модули используют возможности, которые предоставляют другие [1]. Основу составляют основные (Essentials) модули:

– Qt Core – основной модуль, который содержит все базовые средства Qt. На его основе построены все другие модули. Каждая программа созданная с использованием Qt, использует этот модуль;

– Qt Network – модуль для работы с сетевыми средствами;

– Qt Gui – модуль поддержки графического вывода на экран. В Qt4 он также содержит набор виджетов для создания графического интерфейса пользователя. В Qt5 виджеты вынесены в отдельный модуль;

– Qt Widgets – модуль, который содержит набор виджетов для создания графического интерфейса пользователя (Qt5)

– Qt WebKit – средства работы с Веб;

– Qt WebKit Widgets – виджеты для работы с Веб (Qt5);

– Qt Multimedia – средства работы с мультимедийными устройствами и файлами;

– Qt Multimedia Widgets – виджеты для работы с мультимедийными устройствами и файлами (Qt5);

– Qt Sql – средства работы с базами данных;

– Qt Qml – поддержка декларативной языка QML для разработки динамических визуальных интерфейсов (Qt5);

– Qt Quick – поддержка создания динамических визуальных интерфейсов (Qt5);

– Qt Quick Controls – использование технологии QtQuick для создания традиционного для рабочих столов графического интерфейса (Qt5);

– Qt Quick Layouts – компоновка для элементов QtQuick (Qt5).

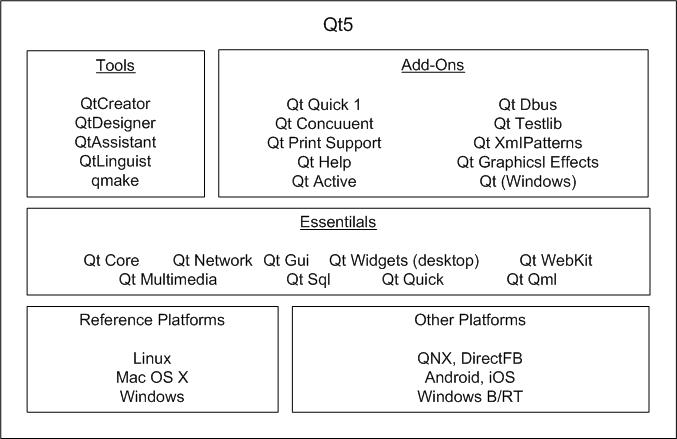


Рисунок 1 – Состав Qt5

Существует также много дополнительных (Add–On) модулей. Стоит заметить, что разделение на основные и дополнительные модули характерно Qt5 в отличие от предыдущих версий. Названия некоторых модулей в Qt5 по сравнению с Qt4 были изменены, а некоторые средства были вынесены в отдельные или перенесены в другие модули. Эти изменения необходимо учитывать при переносе программ, которые были разработаны с использованием Qt4.

## 1.2 Обзор настроек среды Qt Creator

Для разработки программ с использованием библиотеки Qt была создана интегрированная среда разработки Qt Creator. Её первая версия была представлена одновременно с официальным выпуском Qt 4.5.0. Это полноценная кроссплатформенная среда для создания новых проектов и работы с ними.

Среда Qt Creator версии 3.3.0 позволяет управлять целым рядом этапов разработки программы такими как: управление сеансами и проектами, редактирование и создание программного кода, конструирование пользовательского интерфейса программы, анализ быстродействия, анализ использования ресурсов, отладка, построение проекта, запуск программы.

Одно из первых действий, которое необходимо выполнить разработчику перед началом работы с Qt Creator – это настроить среду таким образом, чтобы с ней было удобно работать. Конечно, Qt Creator имеет стандартные настройки, которые уже достаточно удобны для работы. Для доступа к диалогу настройки используется главное меню пункт Tools–> Options...

Настроек компиляции, для управления настройками, относящимися к построению проекта, Qt Creator использует понятие комплекта (Kit). Комплект (Kit) – это конфигурация, которую составляют версия Qt, компилятор и ещё некоторые дополнительные настройки. Таким образом, QtCreator позволяет работать с несколькими различными версиями Qt, несколькими компиляторами в системе, выбирать и настраивать их комбинацию для построения проекта.

Стандартным для Linux и Mac OS X является компилятор GCC. Для Windows можно воспользоваться его свободным аналогом – MinGW, или компилятором MSVC, который входит в состав Microsoft Windows SDK или Visual Studio.

# 2 Структура проекта. Основные типы

## 2.1 Файлы проекта

Проект Qt имеет такую структуру:

– файл проекта – описывает файлы, которые входят в проект, и содержит необходимые настройки;

– файлы, входящие в проект (или другие подпроекты, если проект разбит на несколько частей).

Ключевую роль имеет файл проекта с расширением .pro. Он содержит списки файлов: исходных кодов, файлов ресурсов, файлов локализации, форм, других файлов, которые входят в проект, а также файлов подпроектов, если проект состоит с нескольких частей. Этот файл также содержит некоторые настройки программы.

Создание своего проектного файла. Для начала необходимо создать новую папку, где будет размещаться проект (например: new\_project). Далее создается файл (это будет файл проекта) вводится его имя с расширением .pro (например: new\_project .pro). Этот файл пустой, но его уже можно открыть в Qt Creator (воспользовавшись главным меню: File → Open File or Project. . .).

Создать пустой проект можно с помощью мастера построения проектов. Для этого надо воспользоваться главным меню File → New File or Project... либо комбинацией клавиш Ctrl+Shif t+N. В окне мастера нужно выбрать раздел Other Project (Другой проект) и тип проекта – Empty Qt Project.

После того, как откроется проект, Qt Creator предлагает выбрать комплект для его компиляции. В разделе Projects (Проекты) выбирается комплект по умолчанию и производится конфигурация проекта (Configure Project). В дереве проекта выбирается и открывается файл проекта.

Синтаксис проектных файлов Qt. Проектный файл обычно содержит несколько настроек в виде специальных переменных, каждая из которых играет свою особую роль. Среди большого количества настроек, которые задаются в .рго–файле:

– тип проекта (приложение, динамическая или статическая библиотека, проект, который состоит из подпроектов);

– общие настройки проекта;

– настройки компиляции;

– путь, где будет размещён исполняемый файл, библиотека или бинарный файл во время процесса компиляции;

– пути к файлам, библиотекам и другим частям проекта необходимым для компиляции;

– файлы, входящие в проект;

– дополнительные действия, которые будут выполняться в процессе компиляции проекта.

В таблице 1 предоставлен список переменных, которые часто участвуют в описании проекта:

Таблица 1: Некоторые важные переменные для описания настроек проекта

|  |  |
| --- | --- |
| Переменная | Описание |
| CONFIG | Разнообразные настройки конфигу–рации проекта (например: режим отладки, вывод предупреждений, компиляция динамической библиотеки и т.п.). |
| DEFINES | Макроопределения в проекте. Работает так же, как директива препро–цессора #deﬁne. |
| DESTDIR | Путь к папке, где будет создан исполняемый файл. |
| INCLUDEPATH | Путь к папкам с заголовочными файлами. |
| FORMS | Файлы форм Qt Designer. |
| HEADERS | Заголовочные файлы программы \*.h. |
| QT | Модули Qt, которые используются в программе. |

Продолжение таблицы 1

|  |  |
| --- | --- |
| Переменная | Описание |
| LIBS | Пути к динамическим библиотекам и библиотеки, которые используются в программе. |
| RESOURCES | Файл ресурсов. |
| SOURCES | Исходные тексты программы \*.cpp. |
| TARGET | Название исполняемого файла или динамической библиотеки. |
| TEMPLATE | Тип проекта (приложение, библиотека, составленный из подпроектов ...) |

## 2.2 Компиляция проекта

Компиляция проекта проходит в два этапа. Сначала выполняется предварительная обработка проекта с помощью программы qmake. Этот инструмент Qt несёт ответственность за весь процесс компиляции проекта. Он читает содержание проектного файла и генерирует необходимые промежуточные файлы (дополнительные файлы с исходным кодом и make-файлы для компиляции). Это необходимо для того, чтобы превратить все особые расширения Qt, которые были использованы в программе, в код на языке C++ и использовать дополнительные настройки для проекта, описанные в pro-файле. После этого проект готов к обработке компилятором. Вторым этапом является непосредственно процесс компиляции. Все эти действия выполняются автоматически в среде Qt Creator [2]. Процесс компиляции проекта изображен на рисунке 2.

После успешной компиляции создается исполняемый файл программы. В папке проекта будет содержаться исполняемый файл и все промежуточные файлы, сгенерированные в процессе.

Таким образом, процессом построения проекта руководит .pro-файл. При наличии исходных текстов программы и при отсутствии, pro-файла, его можно сгенерировать. Для этого из командной строки необходимо перейти в папку, которая содержит исходные тексты программы и вызвать qmake с параметром “– –project”. Этим приёмом удобно воспользоваться, чтобы сгенерировать файл проекта и использовать оболочку QtCreator для работы над программой (даже для обычных программ на C++ без Qt).

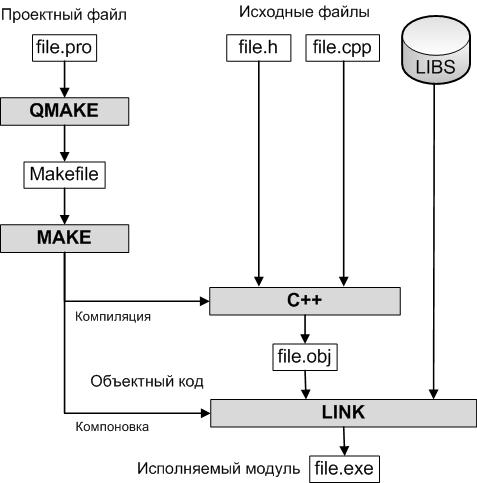


Рисунок 2 – Процесс компиляции проекта

Раздел «Projects» (Проекты) содержит набор необходимых настроек для процесса компиляции и для настройки среды запуска проекта. Одной из таких настроек есть опция Shadow Build, которая позволяет включить режим при котором для промежуточных файлов, make-файлов и продуктов компиляции создаётся отдельная папка вне папки с исходным кодом проекта (настройки размещения для неё – в поле Build directory). Это позволяет построить и хранить одновременно несколько вариантов построенного проекта для различных инструментариев. Также это сохраняет папку с исходным кодом от засорения файлами, созданными в процессе построения проекта. При выключенном Shadow build промежуточные файлы и папка с построенной программой сохраняются в папке, которая содержит файл проекта.

Cозданные промежуточные файлы не являются непосредственной частью проекта. Они были сгенерированы, и будут перезаписываться при необходимости во время компиляции. Поэтому не стоит добавлять их к рго-файлу или делать любые изменения в них. Также не стоит их добавлять в систему контроля версий, если её используют при разработке.

Иногда сгенерированные файлы вместе с объектными и make-файлами бывает необходимо удалить. Это необходимо делать перед тем как заархивировать проект для сохранения, поскольку сгенерированные файлы занимают довольно много места на диске по сравнению с объёмом исходного кода. Порой могут возникать проблемы с компиляцией, когда после значительных изменений в структуре программы промежуточные файлы не были достаточно хорошо заново сгенерированы. В таких случаях возникает необходимость очистить проект. Для этого в главном меню Build → Clean Project (Сборка → Очистить проект). Это позволит удалить сгенерированные файлы, кроме скомпилированного исполняемого файла и make-файлов.

Для того чтобы очистить проект полностью, необходимо изменить некоторые настройки. В разделе Projects (Проекты) и в разделе Clean Steps (Этапы очистки) в пункте Details (Подробнее) и изменить параметр Make arguments (Аргументы make) с clean на distclean.

Очистив проект – все сгенеровапиые файлы, включительно с исполняемым файлом и make-файлами, будут удалены.

# 3 Динамические библиотеки и система расширений

Зачастую приходится реализовывать свои дополнения, предназначенные для использования в одной или нескольких программах [3]. Это можно сделать при помощи динамических библиотек, которые реализуют специальный интерфейс.

## 3.1 Динамические библиотеки

В самом деле, на практике очень часто возникают случаи, когда требуется совместное использование какой-либо функции сразу в нескольких программах, работающих на одном компьютере [4]. Не совсем экономично, если каждая из этих программ будет содержать одинаковый код, – значит, необходим механизм для объединения общего кода таких функций в отдельных файлах (библиотеках), который позволял бы воспользоваться им в необходимых случаях. Такие файлы должны подгружаться в процессе работы самих программ динамически – по мере необходимости и в зависимости от потребностей. Совместно используемая динамически подключаемая библиотека (далее просто динамическая библиотека) по структуре представляет собой группу, содержащую объектные файлы. Использование динамических библиотек предоставляет следующие преимущества:

– программа занимает меньше места в оперативной памяти и на диске. Когда динамическая библиотека подключается к программе, то в исполняемый файл включается не код самой библиотеки, а лишь ссылка на нее;

– разные программы могут использовать одну и ту же библиотеку, только ссылаясь на нее;

– после обновления динамической библиотеки не обязательно перекомпилировать использующие ее программы, что позволяет обновлять динамические библиотеки, не затрагивая связанные с ними приложения. То есть, если в библиотеке будет обнаружена ошибка, достаточно просто заменить ее файл другим.

Очень важно также уточнить, что динамическая библиотека – это не просто группа объектных файлов, из которой компоновщиком выбираются нужные для разрешения ссылки. Все объектные файлы, которые содержатся в самой библиотеке, объединяются в единый объектный файл. Это дает преимущество для программ, которые компонуются вместе с библиотекой, потому что они всегда будут иметь доступ сразу ко всему содержимому библиотеки, а не какой-либо отдельной ее части.

Для того чтобы создать динамическую библиотеку, в проектный файл (файл с расширением pro), необходимо включить следующие строки:

TEMPLATE = lib

CONFIG += dll

Первая опция – TEMPLATE говорит о том, что проект – это библиотека, однако этого недостаточно, потому что библиотеки бывают как динамические, так и статические. Поэтому нужна вторая опция – CONFIG, в ней и указывается, что нужна именно динамическая библиотека. Теперь, после компоновки этого проекта, получится файл динамической библиотеки. В Windows такие файлы носят расширение dll, в Mac OS X – dylib, а в Linux – so.

Чтобы использовать динамическую библиотеку в любом проекте другой библиотеки или исполняемой программы, нужно компоновать проект совместно с этой библиотекой. Допустим, нужная библиотека называется Tools. Тогда в pro-файл нужно включить ее вместе с путем ее размещения, а также и путь ее заголовочных файлов:

LIBS += –L../../lib/ –lTools

INCLUDEPATH = ../../include

Таким образом, если в библиотеке Tools определены классы, то можно получать к ним доступ абсолютно так же, как если бы эти классы определялись внутри самого проекта, который компонуется с этой библиотекой.

Для работы с динамическими библиотеками в среде операционной системы нужно описать место их размещения. Лучше всего это сделать, включив путь динамических библиотек в переменную среды path [5]:

– в Windows: set PATH=%PATH%;с:\Projects\cpp\lib\win32

– в Mac OS X:

export DYLD\_LIBRARY\_PATH =

/Projects/cpp/lib/mac/:$DYLD\_LIBRARY\_PATH

– в Linux:

export LD\_LIBRARY\_PATH=/Projects/cpp/lib/xll/:$LD\_LIBRARY\_PATH

## 3.2 Динамическая загрузка и выгрузка библиотеки

Существуют два способа использования динамических библиотек. В первом способе связывание с динамической библиотекой осуществляется в процессе компоновки самой программы. В этом случае динамическая библиотека загружается автоматически при запуске использующего ее приложения.

Второй способ предоставляет возможность загрузки некоторого кода без явной компоновки во время работы самой программы. Это необходимо, например, в случаях, когда нужно предоставить сторонним разработчикам возможность создавать дополнительные модули для расширения функциональности вашей программы. Суть способа заключается в использовании класса QLibrary [6]. Этот класс заботится и о том, чтобы загруженная библиотека оставалась в памяти на протяжении всего времени работы приложения.

В файле проекта, для создания динамической библиотеки нужно установить в секции template значение lib. Готовая библиотека будет расположена на один уровень выше каталога с ее исходными файлами, для этого в секции destdir задается значение “..”.

Ввиду того, что создаваемая библиотека не нуждается в элементах пользовательского интерфейса, оператором –= исключается опция gui в секции QT.

В динамическую библиотеку должны быть экспортированы прототипы функций для их дальнейшего использования. Эти функции необходимо заключить в спецификатор extern "C" \_\_declspec(dllexport). Тогда компилятор C++ не будет прикреплять информацию о типе к символьной сигнатуре функции. Такой спецификатор нужно указать, если необходимо, чтобы динамическую библиотеку можно было загружать в процессе работы основной программы, например, при помощи класса QLibrary. C помощью ключевого слова \_\_declspec(dllexport), которое хранит имена функций в таблице экспорта DLL, экспортируются данные, функции, классы или функции членов класса из библиотеки DLL [7].

Чтобы использовать динамическую библиотеку в программе, нужно создать объект класса QLibrary и передать в его конструктор имя файла динамической библиотеки без расширения. Это связано с тем, что на разных платформах файлы динамических библиотек имеют различные расширения. Как уже отмечалось ранее, в ОС Windows файл библиотеки будет иметь расширение dll, в UNIX/Linux – so, а в Мае OS X – dylib. Передавая только имя, библиотека Qt автоматически подставляет нужное расширение.

Указатели на экспортируемые функции извлекаются с помощью метода resolve(). В этот метод передается символьная сигнатура, по которой будет осуществляться поиск нужной функции. Возвращает метод указатель на тип void, который представляет собой адрес найденной функции. Если метод resolve() вернет нулевой указатель, это означает, что функция не найдена. Для вызова функции этот указатель необходимо привести к нужному типу. В случае успешной проверки указателя вызывается сама функция. В завершение для отображения виджета надписи на экране вызывается метод show().

## 3.3 Расширения

Использование расширений – это неотъемлемая часть любого профессионального приложения. Расширение – это совместно используемая динамическая библиотека, предназначенная для загрузки в процессе исполнения основного приложения, которая обязательно должна реализовывать хотя бы один специальный интерфейс. Расширения делятся на две группы:

– расширения для Qt;

– расширения для собственных приложений.

## 3.3.1 Расширения для Qt

Библиотека Qt предоставляет различные типы расширений, предназначенных для дополнения ее возможностей. Они могут использоваться для поддержки новых форматов растровых файлов, новых драйверов баз данных и т. д. Всего их более 20-ти. Вот некоторые из них:

– QSqlDriverPlugin – для драйверов баз данных;

– QPictureFormatPlugin, QImageIOPlugin – для поддержки различных форматов растровых изображений;

– QTextCodecPlugin – для реализации кодировок текста;

– QStylePlugin – для стилей элементов управления.

Все эти классы наследуют класс QObject, который является базовым для всей системы расширений Qt. Свои расширения для Qt нужно строить на базе этих классов, реализуя в них нужные методы.

## 3.3.2 Поддержка собственных расширений в приложениях

Связь с расширением осуществляется с помощью интерфейса, поэтому приложение должно предоставлять по меньшей мере один интерфейс для использования расширения. Расширения загружаются приложением при помощи класса QPluginLoader, который содержит несколько методов. Самый часто используемый из них – это метод instance(), создающий и возвращающий указатель на объект расширения. Класс QPluginLoader автоматически загружает расширение, чье имя файла указывается в его конструкторе. Выгрузку расширения, если в этом есть необходимость, можно осуществить с помощью метода unload().

Интерфейс – это класс, который содержит только чисто виртуальные определения методов.

# 4 Процессы

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

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

По своей сути, процессы – это независимые друг от друга программы, обладающие своими собственными данными. Коротко процесс можно охарактеризовать как совокупность кода, данных и ресурсов, необходимых для его работы. Под ресурсами подразумеваются объекты, запрашиваемые и используемые процессами в период их работы. Любая прикладная программа, запущенная на компьютере, представляет собой не что иное, как процесс.

Создание процесса может оказаться полезным для использования функциональных возможностей программ, не имеющих графического интерфейса и работающих с командной строкой. Особенно это имеет смысл при запуске команд или программ, время работы которых не продолжительно.

Процессы можно создавать с помощью класса QProcess, который определен в заголовочном файле QProcess [8]. Объекты этого класса позволяют считывать информацию, выводимую запущенными процессами, и даже реагировать на их запросы ввода данных. Кроме того, класс QProcess содержит методы для манипулирования системными переменными процесса. Работа с объектами этого класса осуществляется в асинхронном режиме, что позволяет сохранять работоспособность графического интерфейса программы в моменты, когда запущенные процессы находятся в работе. При появлении данных или других событий объекты класса QProcess отправляют сигналы. Например, при возникновении ошибок объект процесса вышлет сигнал error() с кодом этой ошибки.

Для создания процесса его нужно запустить. Запуск процесса выполняется методом start(), в который необходимо передать имя команды и список ее аргументов, либо все вместе: команду и аргументы одной строкой. Как только процесс будет запущен, отправляется сигнал started(), а после завершения его работы – сигнал finished(). Сигнал finished() сообщает код и статус завершения работы процесса. Для получения статуса выхода можно вызвать метод exitStatus(), который возвращает только два значения: NormalExit (нормальное завершение) и CrashExit (аварийное завершение).

Для чтения данных запущенного процесса класс QProcess предоставляет два разделенных канала: канал стандартного вывода (stdout) и канал ошибок (stderr). Эти каналы можно переключать с помощью метода setReadChannel(). Если процесс готов предоставить данные по установленному текущему каналу, то отправляется сигнал readyRead(). Отправляются и сигналы для каждого канала в отдельности: readyReadStandardOutput() и readyReadStandardError().

Считывать и записывать данные в процесс можно с помощью методов: write(), read(), readLine() и getChar(). Для чтения также можно воспользоваться методами, привязанными к конкретным каналам: readAllStandardOutput() и readAllStandardError().

Метод start() исполняется асинхронно, а если логике работы программы требуется блокирующий подход, то используются методы waitForStarted() и waitForFinished(). Первый метод блокирует выполнение потока программы до тех пор, пока программа не будет запущена, а второй, пока программа не будет завершена.

# 5 Настройка среды динамической компиляции исполняемых файлов

Для достижения поставленной цели, в среде Qt Creator, была разработана программа, реализующая работу с рассмотренными классами (QLibrary, QProcess), и, показывающая принцип динамической компиляции исполняемых файлов.

Настройка среды динамической компиляции исполняемого файла,заключается в следующих этапах.

Первым этапом необходимо создать проект нового приложения, который включает в себя следующие файлы:

– program.cpp – файл с исходными текстами программы;

– program.h – заголовочный файл;

– program.pro – файл проекта;

– main.cpp – файл исходников;

– program.ui – дерево виджетов формы в формате XML.

В этих файлах описывается, непосредственно, разрабатываемое приложение.

На втором этапе создается подпроект, который будет отвечать за создание/редактирование динамически загружаемой библиотеки. Этот подпроект помещается в отдельную папку исходного проекта, и содержит следующие файлы:

– library.cpp – файл с исходными текстами функций;

– library.h – заголовочный файл;

– library.pro – файл проекта;

– library\_global.h – заголовочный файл.

В динамической библиотеке описываются функции, которые будут редактироваться во время исполнения приложения. Данные функции продумываются на этапе разработки приложения. Формат описания функции имеет вид: extern "C" \_\_declspec(dllexport) тип\_возвращаемого\_значения имя\_функции(список\_параметров) { объявления и операторы } [9]. Они помещаются в файл "library.cpp".

Третий этап заключается в описании методов, в исходном проекте, компиляции, загрузки и выгрузки динамически загружаемой библиотеки.

Метод, описывающий компиляцию библиотеки, приведен в листинге 1.

void program:: compilation\_dll() //компиляция dll

{

QProcess process(this);

process.start("cmd");

process.write("PATH=C:\\Qt\\Qt5.6.2\\5.6\\mingw49\_32\\

bin;C:\\Qt\\Qt5.6.2\\Tools\\mingw492\_32\\bin\n");

//определение директории исполняемых файлов

process.waitForBytesWritten();

process.waitForReadyRead(); //ожидание записи данных

process.write("qmake lib\_tennis.pro\n");

//запуск сборки dll

process.waitForBytesWritten();

process.waitForReadyRead(); //ожидание записи данных

process.write("mingw32-make -j4\n"); //компиляция dll

process.waitForBytesWritten();

process.waitForReadyRead(); //ожидание записи данных

process.close();

}

Листинг 1 – Компиляция библиотеки, с помощью класса QProcess

Метод, позволяющий выгрузить библиотеку, приведен в листинге 2.

void program:: unloading\_dll()

{

lib.unload();

}

Листинг 2 – Метод, выгружающий библиотеку

Метод, описывающий загрузку библиотеки, приведен в листинге 3.

void program:: loading\_dll() //загрузка dll

{

QLibrary lib("library");

if (!lib.load())

qDebug() << lib.errorString();

else

qDebug() << "lib open!!!" << lib.load();

typedef int (\*Function)(int, int);

Function fun;

fun = (Function)lib.resolve("name\_function\_in\_dll");

}

Листинг 3 – Загрузка библиотеки, с помощью класса QLibrary

После загрузки библиотеки, вызвать ту или иную функцию можно с помощью переменной fun из листинга 3. Вызов данной функции приведён в листинге 4.

if (fun)

{

int sum;

sum = fun(50,10);

}

Листинг 4 – Использование функции из библиотеки

# ЗАКЛЮЧЕНИЕ

Основные результаты курсового проекта состоят в следующем:

1 В среде разработки Qt Creator рассмотрены инструменты и средства, необходимые для работы с Qt (изучена структура проекта и этапы компиляции).

2 Выявлены принципы загрузки и выгрузки динамических библиотек, с помощью класса QLibrary.

3 Рассмотрен класс создания новых процессов – QProcess.

4 Разработана программа, реализующая работу с рассмотренными классами (QLibrary, QProcess), и, показывающая принцип динамической компиляции исполняемых файлов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Шлее М. Qt 5.3. Профессиональное программирование на C++ / М. Шлее. – СПб.: БХВ–Петербург, 2015. – 928 c.

2 Программирование на языке С++ в среде Qt Creator / Е. Р. Алексеев, Г. Г. Злобин, Д. А. Костюк,О. В. Чеснокова, А. С. Чмыхало. – М.: ALT Linux, 2015. – 448 c.

3 Леденев А. В. Динамически загружаемые библиотеки: структура, архитектура и применение / А. В. Леденев, И. А. Семенов, В. А. Сторожевых // Прикладная информатика. – 2008. – № 2. – С. 31-84.

4 Шарыгин Е. Ю. Обзор методов динамической компиляции запросов /

Е. Ю. Шарыгин, Р. А. Бучацкий // Труды института системного програм-мирования РАН. – 2017. – Т. 29. – № 3. – С. 179-224.

5 GitHub [Электронный ресурс]; Сборка под Windows через консоль // – (Engl). – URL: https://github.com/qreal/qreal/wiki/Сборка-под-Windows-через-консоль [24 марта 2018].

6 CrossPlatform.RU Все о кроссплатформенном программировании [Электронный ресурс]; QLibrary Class Reference. // – (Engl). – URL: http://doc.crossplatform.ru/qt/4.3.2/qlibrary.html [17 марта 2018].

7 Microsoft. Developer Network [Электронный ресурс]; Exporting from a DLL Using \_\_declspec(dllexport). // – (Engl). – URL: https://msdn.microsoft.com/en-us/library/a90k134d.aspx [23 апреля 2018].

8 CrossPlatform.RU Все о кроссплатформенном программировании [Электронный ресурс]; Описание класса QProcess. // – (Рус.). – URL: http://doc.crossplatform.ru/qt/4.7.x/qprocess.html [2 апреля 2018].

9 Дейтел Х., Дейтел П. Как программировать на C++ / Х. Дейтел, П. Дейтел. – М.: Бином, 2017. – 1021 с.