Работа №2

«Объектно-ориентированное программирование»

Цель работы

Получение навыков объектно-ориентированного программирования

После выполнения работы студенты могут

- Создавать современное программное обеспечение с использованием языка C++
- Упорядочивать внутреннюю структуру своего приложения, используя классы
- Применять современные программные библиотеки, использующие объектно-ориентированную структуру

Теоретическая часть

Язык C, рассмотренный ранее, относится к так называемой **парадигме процедурного программирования**, в которой программы, составленные из последовательностей операторов, можно разделить на подпрограммы средствами самого языка. Такой подход является отражением низкоуровневой архитектуры ЭВМ. Однако использование языка C без каких либо ограничений может сравнительно быстро привести исходный код в нечитаемое состояние (так называемый «спагетти-код») за счет использования оператора безусловного перехода — `goto`.

Структурная парадигма программирования в своей основе предполагает представление программы в виде иерархической структуры блоков. Каждый блок представляет собой отдельную подпрограмму, а разработка ведется пошагово, сверху-вниз. С точки зрения языка, иерархия блоков — это порядок вызова функций. Использование оператора безусловного перехода запрещается, так как он может нарушить данную иерархию. Структурное программирование послужило мощным толчком для развития индустрии программного обеспечения, так как позволило существенно упорядочить процесс разработки и упростить сопровождение программных продуктов. Язык C, при запрете использования оператора безусловного перехода (`goto`), полностью соответствует принципам

структурного программирования.

Тем не менее, структурное и процедурное программирование имеют один существенный недостаток: отсутствие прямой связи между данными и подпрограммами (процедурами и функциями). Данная проблема усложняет процесс сопровождения проекта, так как программист вынужден выстраивать такую связь самостоятельно.

Логичным развитием структурного программирования стало **объектно-ориентированное программирование**. Как ясно из названия, в его основе лежит объект. В объекте одновременно связаны данные, которые к нему относятся, и операции, которые над этими данными могут выполняться. Объект всегда относится к определенному **классу**.

С точки зрения языка программирования, класс представляет собой сложный, составной тип данных, включающий в себя как переменные (в том числе и другие классы), так и функции. При этом, для отделения классов от обычных типов данных, переменные классов носят название **поля**, а функции, объявленные внутри классов — **методы**. Данная терминология применима для любого объектно-ориентированного языка программирования.

В процессе работы программист оперирует конкретными объектами, принадлежащими к доступным ему классам, как написанным самостоятельно, так и предоставленным программными библиотеками. Состояние каждого объекта определяется набором значений, записанных в его полях, и задача программиста сводится к заданию нужных значений полям объектов и к вызову необходимых для решения задачи методов.

Важным моментом при разработке программного обеспечения является вопрос целостности данных. Например, для представления некоей непрерывной области памяти необходимы две переменные: адрес начала этой области и ее размер в байтах. Можно записать такой тип данных на языке C:

```
typedef struct {
    void* ptr;
    size_t size;
} blob;
```

Очевидно, что переменные `ptr` и `size` не могут принимать произвольные значения, а в момент создания структуры должны быть равны 0. Изменение указателя `ptr` должно сопровождаться актуальным изменением `size`, а их изменение по отдельности невозможно в принципе. Механизмы объектно-ориентированного программирования, рассмотренные далее, позволяют решать такие проблемы автоматически.

Главной особенностью классов является обязательное наличие двух функций: конструктор и деструктор.

Конструктор — это метод, вызываемый автоматически при создании объекта, задача которого — привести класс в начальное состояние. То есть, присвоить всем его полям значения, соответствующие логике работы данного класса. В рассмотренном выше примере конструктор должен обнулить значения `ptr` и `size`.

Деструктор — это метод класса, освобождающий ресурсы, занятые классом в процессе работы. В рассмотренном примере деструктор может не делать ничего, однако если, к примеру, класс выделяет для своих нужд некий объем оперативной памяти, при вызове деструктора эту память необходимо освободить.

Для обеспечения целостности данных применяют механизмы **контроля доступа**, ограничивающие доступ к отдельным полям и методам класса для того кода, который не принадлежит классу. Методы класса всегда имеют полный доступ к полям класса, однако класс может запретить доступ к части своих полей и методов из функций, не являющихся членами класса.

Еще одной особенностью классов является их иерархическая взаимосвязь. Класс может существовать сам по себе, а может являться производным от другого класса. При этом производный класс обладает всеми теми же свойствами и методами, что и исходный. Такой механизм получил название **наследование**. Исходный класс называется также базовым или родительским, а производный — потомком или наследником.

В механизме наследования выделяют так называемые **абстрактные классы**. Такие классы предназначены не для непосредственного использования, а для описания классов-потомков. Они предоставляют один общий набор функций,

который присутствует в каждом из классов-потомков, и программисту при обращении к такому классу не обязательно знать конкретный тип, а достаточно знать тип исходного класса.

Наиболее близкая аналогия к классовому дереву — это классификация животных в биологии. Каждое животное — это объект. При этом одно и то же животное-объект может относиться к нескольким, включающим друг друга классам. К примеру, воробей относится к классам «позвоночные», «птицы» и «воробьиные». При этом аист точно так же относится к классам «позвоночные» и «птицы», но не относится к классу «воробьиные», а волк из перечисленного относится только к классу «позвоночные».

Развитие объектно-ориентированного программирования (ООП) потребовало серьезной переработки языка, поэтому в середине 80-х годов был выпущен новый язык программирования — C++. Изначально он разрабатывался как обобщение языка C и первое время существовал в виде «C с классами», поэтому программа, написанная на чистом C, будет работоспособна при сборке компилятором C++. Вследствие этого, C++ поддерживает не только объектно-ориентированное программирование, но и структурное, и поэтому пригоден для решения широкого круга задач. На сегодняшний день C++ является одним из самых популярных языков программирования, несмотря на свой возраст.

Синтаксис классов в C++

Классы в C++ объявляются аналогично структурам. Простой класс, содержащий одно поле (переменную) и один метод (функцию), можно объявить следующим образом:

```
class MyClass
{
public:
    int MyVar;
    void MyFunc(double b)
    {
        printf("b=%g, MyVar=%d\n", b, MyVar);
    }
```

```
};
```

Внутри класса можно объявлять как переменные, именуемые полями, так и функции, именуемые методами. Доступ к полям и методам класса осуществляется так же, как и к переменным в структуре, используя оператор «.». Например:

```
MyClass mc;
mc.myVar = 10;
mc.MyFunc();
```

В случае, если переменная является указателем на объект, оператор «`.`» заменяется на оператор «`->`»:

```
MyClass mc;
MyClass* mcp = &mc;
mcp->myVar = 115;
mcp->MyFunc();
```

Поля и методы класса разделяются на уровни доступа. В рамках ООП существуют три уровня:

- `public` — поля и методы, находящиеся на уровне `public`, могут быть вызваны из любой функции в любом классе или вне его.
- `private` — поля и методы, находящиеся на уровне `private`, являются частными для данного класса и могут быть вызваны только из функций данного класса.
- `protected` — поля и методы, расположенные на уровне `protected`, доступны как самому классу, так и его потомкам, но не доступны другим классам. Данный уровень является промежуточным между `private` и `public`.

Для объявления уровня доступа следует написать соответствующее ключевое слово и поставить двоеточие. Все поля и методы, объявленные после данного ключевого и до следующего уровня доступа, или до конца класса, относятся к данному уровню. Как правило, на уровне `public` оставляют методы для доступа к данным, а в `private` объявляют внутренние переменные класса. Это сделано для того, чтобы спрятать от других классов внутренние данные и обеспечить их целостность.

По умолчанию все поля и методы класса объявлены в секции `private`. Структура в языке C++ является классом, все поля которой объявлены как `public`. При этом структура в C++ также может содержать в себе методы, хотя для обратной совместимости с языком C не следует так делать.

Следующий листинг показывает объявление простого класса с несколькими уровнями доступа. Данный класс обслуживает внутреннюю переменную `val`, доступ к которой осуществляется функциями `setValue` и `getValue`. При этом функция `setValue` проверяет, чтобы значение `val` не выходило за пределы от 0 до 100:

```
class DoubleKeeper
{
public:
    void setValue(double d)
    {
        if(d>100)
            val = 100;
        else if (d<0)
            val = 0;
        else
            val = d;
    }
    double getValue()
    {
        return val;
    }
protected:
    double val;
};
```

Пример показывает простейшее решение по обеспечению целостности данных. Программист не может напрямую поменять значение `val`. Обращение напрямую к переменной, как показано в примере ниже, приведет к ошибке

компиляции:

```
DoubleKeeper dk;
dk.val = -4; // ! Ошибка компиляции
```

Однако задание значения через метод setValue, осуществляющий проверку вводимого значения, всегда приведет к правильному результату с точки зрения логики класса:

```
DoubleKeeper dk;
double v;
dk.setValue(-4);
v = dk.getValue();      // 0
dk.setValue(34);
v = dk.getValue();      // 34
dk.setValue(9725);
v = dk.getValue();      // 100
```

Наследование

Наследование позволяет создать производный класс, содержащий все те же поля и методы, что и исходный. При этом можно расширить функционал производного класса, добавив новые поля или методы, либо поменять его логику работы, переопределив методы родительского класса.

Для того, чтобы создать производный класс, необходимо после указания имени класса вставить двоеточие и после ключевого слова `public` — название базового класса. Пример класса, производного от `DoubleKeeper`, представлен в листинге:

```
class EvenKeeper: public DoubleKeeper
{
};
```

Новый класс `EvenKeeper` будет обладать всеми теми же полями и методами, что и класс `DoubleKeeper`. Следующий программный код будет работать без ошибок:

```
EvenKeeper k;
```

```
k.setValue(12);
double v = k.getValue();      // 12
```

При этом можно переопределить в классе `EvenKeeper` метод `setValue` таким образом, чтобы он уменьшал переданное значение до ближайшего четного:

```
class EvenKeeper: public DoubleKeeper
{
public:
    void setValue(double d)
    {
        val = d - fmod(d, 2);
    }
};
```

```
В приведенном примере мы заменили реализацию метода от родительского
класса на новую, что позволяет нам, в том числе, записывать в
EvenKeeper значения за пределами диапазона 0-100:
```

```
EvenKeeper k;
k.setValue(2345);
double v = k.getValue() // 2344
```

Разделение объявления класса и определения его методов

Очевидно, что определение методов вместе с описанием класса годится только для очень простых классов, где метод занимает несколько строчек. Гораздо разумнее оставить в классе только объявление его методов, а определение методов вынести отдельно. Для того, чтобы можно было легко реализовать это, язык C++ вводит такое понятие как пространство имен.

Пространство имен — это специальная языковая конструкция, ограничивающая набор идентификаторов внутри другого идентификатора. Простыми словами, пространство имен для идентификаторов — то же самое, что папка для файлов. Пространство имен позволяет в том числе избежать коллизии при возникновении одинаковых имен переменных. Для объявления пространства имен в языке C++ существует ключевое слово `namespace`:

```
namespace CustomNames
{
    double CustomDouble;
}
```

Для доступа к объекту, переменной или функции из пространства имен необходимо использовать специальный оператор «::». К примеру, обратиться к `CustomDouble` можно следующим образом:

```
CustomNames::CustomDouble = 0;
```

При объявлении класса автоматически создается пространство имен, которое называется так же, как и этот класс. Используя это пространство имен, можно определить функцию, объявленную в классе, за пределами объявления класса. Покажем это на примере класса `DoubleKeeper`:

```
class DoubleKeeper
{
public:
    void setValue(double d);
    double getValue();
private:
    double val;
};
//-----------------------------------------------
void DoubleKeeper::setValue(double d)
{
    if(d>100)
        val = 100;
    else if (d<0)
        val = 0;
    else
        val = d;
}
```

```
double DoubleKeeper::getValue();
{
    return val;
}
```

В данном примере специальный комментарий-сепаратор отделяет объявление класса от определения его методов. На практике эти две части помещают в разные файлы: объявление класса записывается в заголовочный файл с расширением «.h» или «.hpp», а определение его функций — в файл исходных кодов с расширением «.cpp». Обычно эти файлы называют так же, как и сам класс, но без соблюдения регистра. Для вышеприведенного класса это будут файлы `doublekeeper.h` и `doublekeeper.cpp`. Такой подход позволяет программисту, прочитав заголовочный файл, быстро понять логику работы класса, не вдаваясь в детали его реализации.

Полиморфные функции

Еще одно важное нововведение в языке C++, часто используемое при работе с классами — это **полиморфизм**. Он заключается в том, что язык допускает объявление нескольких функций с одинаковым названием, но с разными наборами аргументов. При вызове такой функции компилятор выбирает конкретную функцию в зависимости от переданных типов переменных. В приведенном примере объявляются две функции, выводящие данные на экран: одна для одиночного числа, а вторая — для массива чисел.

```
void printData(int x)
{
    printf("%d\n", x);
}

void printData(int x[], int num)
{
    for(int a=0; a<num; a++)
        printf("%d ", x[a]);
    printf("\n");
```

```
}

int main()
{
    printData(4);                 // Вывод: "4"
    int ar[4] = {1, 2, 4, 8};
    printData(ar, 4);             // Вывод: "1 2 4 8"
}
```

Методы одного класса с одинаковыми именами, но разными аргументами называются **полиморфными**. К полиморфным методам относятся как разные методы одного класса, так и один и тот же метод, определенный в родительском классе и переопределеннй в производном классе.

Конструктор и деструктор

Среди методов класса всегда выделяются два специальных метода, играющих важную роль. Они всегда объявляются неявно и присутствуют в любом классе:

Конструктор — это метод, неявно вызываемый при создании экземпляра класса. Обычно он содержит инициализацию внутренних переменных. Конструктор — это всегда метод, возвращаемый тип которого не указывается, называющийся так же, как и сам класс.

Деструктор — это метод, неявно вызываемый при удалении экземпляра класса. Деструктор — это всегда метод, возвращаемый тип которого не указывается, а название совпадает с названием класса, но начинается с символа «~».

Класс может иметь несколько полиморфных конструкторов с разными аргументами, но только один деструктор. При этом класс обязательно содержит конструктор без аргументов, вызываемый по умолчанию. В примере показан класс `DoubleKeeper`, снабженный конструктором и деструктором, а также приведены определения конструкторов и деструктора:

```
class DoubleKeeper
{
public:
    DoubleKeeper();
    DoubleKeeper(double v);
    ~DoubleKeeper();
    void setValue(double d);
    double getValue();
private:
    double val;
};

DoubleKeeper::DoubleKeeper()
{
    val = 0;
}

DoubleKeeper::DoubleKeeper(double v)
{
    val = v;
}

DoubleKeeper::~DoubleKeeper()
{
}
```

При создании экземпляра класса конструктор вызывается немедленно, в том месте, где происходит объявление переменной:

```
int main()
{
    DoubleKeeper dk1; // Вызывается первый пустой конструктор
    DoubleKeeper dk2(); // Аналогично, вызывается первый конструктор
```

```
    DoubleKeeper dk3(5); // Вызывается второй конструктор
   // Место вызова деструкторов
}
```

В приведенном примере присутствует комментарий, обозначающий место вызова деструкторов. Для объектов, объявленных как локальная переменная, деструктор будет вызван вместе с окончанием области видимости переменной, то есть перед соответствующей закрывающей фигурной скобкой. Если объект является глобальной переменной, объявленной за пределами всех функций, то его конструктор будет вызван при старте программы, а деструктор — при завершении.

При создании класса-потомка сначала будет вызван конструктор класса-предка, а затем уже непосредственно конструктор самого класса. При удалении такого класса деструкторы будут вызываться в обратном порядке: сначала деструктор потомка, затем деструктор предка.

Виртуальные методы

Механизм наследования предполагает, что классы-потомки содержат в себе реализацию всех методов класса-предка. Рассмотрим пример: два класса, один из которых является базовым, а второй — производным. В базовом классе определена некая функция, а в производном она переопределена:

```
class Base
{
public:
    void printClassName() { puts("Base"); }
};

class Derived: public Base
{
public:
    void printClassName() { puts("Derived"); }
}
```

При создании двух экземпляров класса и вызове функции `printClassName` будет выведено имя созданного класса:

```
Base b;
Derived d;
b.printClassName(); // Вывод: Base
d.printClassName(); // Вывод: Derived
```

Однако если обращаться с классом, переданным по указателю, то поведение функции `printClassName` зависит от того, к какому типу приведён указатель на класс:

```
Derived* derivedptr = &d;
Base* baseptr = &d;
derivedptr->printClassName(); // Вывод: Dervied
baseptr->printClassName(); // Вывод: Base
```

Несмотря на то, что оба указателя `derivedptr` и `baseptr` указывают на один и тот же объект, их поведение различно. Поведение объекта определяется тем, к какому типу приведен указатель. На практике необходима ситуация, когда поведение объекта зависит от его фактического типа, а не от типа указателя, через который производится работа. Реализовать это позволяет механизм виртуальных методов.

Виртуальный метод — это метод, конкретная реализация которого определяется только типом объекта. Другими словами, виртуальный метод — это метод, определенный в родительском классе, который может быть переопределен в производном классе таким образом, что его реализация будет определяться во время исполнения программы.

Для создания виртуального метода необходимо добавить ключевое слово `virtual` при объявлении метода в базовом классе.

При использовании виртуальных методов становится возможным создание некоторого базового класса, описывающего общий принцип поведения на уровне объявления функций. Производные классы при этом определяют логику работы базового класса, однако сторонним функциям, которые пользуются этим классом, не обязательно знать конкретный тип производного класса, а достаточно знать

лишь базовый класс. Рассмотрим тот же пример, но сделаем функцию printClassName виртуальной:

```
class Base
{
    virtual void printClassName() { puts("Base"); }
};

class Derived: public Base
{
    virtual void printClassName() { puts("Derived"); }
}
```

Создадим два объекта, а также один указатель на класс Base, через который будем работать с двумя объектами:

```
Base b;
Derived d;
Base* bp;
```

При вызове функции printClassName будет вызвана реализация, зависящая от типа конкретного объекта и не зависящая от типа указателя:

```
bp = &b;
bp->printClassName(); // Вывод: Base
bp = &d;
bp->printClassName(); // Вывод: Derived
```

При этом точно так же можно использовать указатель на производный класс:

```
Derived* dp;
dp = &b;
dp->printClassName(); // Вывод: Base
dp = &d;
dp->printClassName(); // Вывод: Derived
```

Пример классов, которые используют механизм виртуальных функций — это классы ввода-вывода. Базовый класс содержит виртуальные функции для чтения и

записи, а производные классы конкретизируют процессы чтения и записи специфично для конкретного устройства: файла, сетевого соединения и т. д. При этом указателя на базовый класс оказывается достаточно для любой сторонней функции, которая просто обрабатывает данные, полученные от класса, и при этом функции не требуется знать, как именно эти данные были получены. В свою очередь, программист всегда может создать новый класс для нового устройства ввода-вывода и передать его в такую функцию, сохранив неизменной общую логику работы.

Знакомство с Qt

Qt представляет собой кросс-платформенную библиотеку, ориентированную на разработку графических приложений. Она разработана на языке C++ и состоит из набора классов, реализующих определенные элементы интерфейса или вспомогательные функции. Qt разрабатывается с 1996 года и на сегодняшний день является широко распространенным и надежным продуктом. Поскольку Qt — это программное обеспечение с открытым исходным кодом, Qt можно свободно загрузить на сайте http://qt.io. Версия для разработчиков идет в комплекте со средой разработки Qt Creator, которая сама написана на Qt. При этом возможности этой среды довольно обширны и не ограничиваются только разработкой графических приложений. Исходный код, написанный с использованием этой библиотеки, является кросс-платформенным и может быть собран на любой платформе, где присутствует библиотека Qt. На сегодняшний день это платформы MS Windows, GNU/Linux, MacOS, а также, с некоторыми особенностями — iOS и Android.

Каждый класс Qt в своем названии имеет букву Q в начале, чтобы его можно было легко отличить от остальных классов. Базовым классом, от которого так или иначе наследуются все остальные, является класс `QObject`. Все классы Qt снабжены документацией на английском языке. Бóльшая часть документации переведена на русский язык сторонними переводчиками.

Построение интерфейса Qt основано на особых графических объектах — виджетах. **Виджет** представляет собой прямоугольный элемент интерфейса, так или иначе осуществляющий взаимодействие с пользователем. Каждый виджет

является производным от базового класса `QWidget` и поэтому поддерживает стандартный, довольно широкий набор функций.

Порядок расположения виджетов в окне определяется при помощи свойства «`parent`», присущего всем виджетам: объект-«сын» размещается внутри объекта-«родителя». При этом если у виджета нет родителя (то есть в качестве `parent` указано значение `NULL`), то он становится окном верхнего уровня, и его внешний вид оформляется операционной системой: виджету добавляется рамка, название и кнопки управления приложением. На рисунке 1 показан пример простого окна, которое можно создать при помощи Qt буквально за несколько минут. В примере основное окно (MainWindow) является «родителем» для главного меню и виджета с закладками (Tab Widget), последний в свою очередь является родителем для надписи (Label) и выпадающего меню (ComboBox). В свою очередь, MainWindow является окном верхнего уровня и оформляется в соответствии с настройками операционной системы.

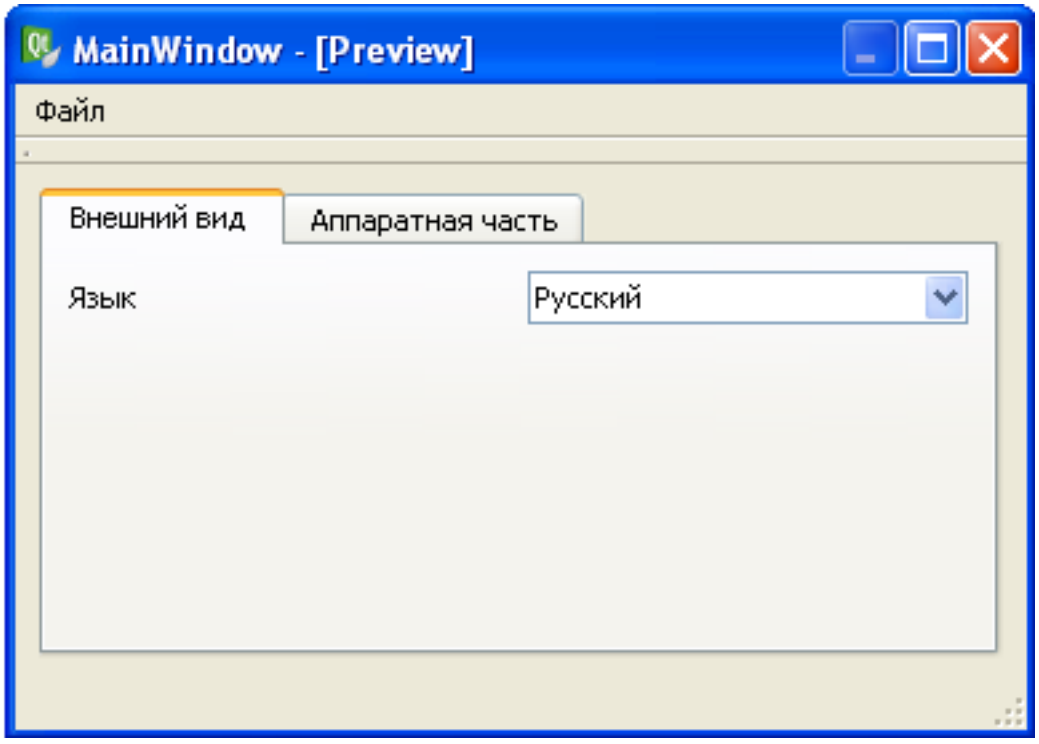

Рисунок 1 - Пример окна Qt.

Виджеты могут быть распределены произвольно внутри родительского виджета. Однако для упорядочивания виджетов существуют особые объекты —

раскладки (**Layout**). Если виджет-родитель содержит в себе раскладку, то все его виджеты-дети будут менять свои размеры в соответствии с ее правилами, причем это будет происходить автоматически при изменении размеров виджета. В приведенном примере виджет Tab Widget имеет горизонтальную раскладку, и его дочерние виджеты располагаются по горизонтали слева направо, занимая равные пропорции (виджет Label занимает половину ширины, несмотря на то, что текст занимает лишь малую его часть). Раскладка не является виджетом. Все раскладки унаследованы от общего класса `QLayout`, который в свою очередь унаследован от `QObject`. Каждый класс-раскладка привязывается к заданному виджету и реализует один тип расположения дочерних виджетов в его пределах.

Взаимодействие виджетов

Все объекты Qt могут обрабатывать события, такие как действия пользователя или оповещения от операционной системы. Механизм обработки событий в Qt копирует механизм событий в существующих операционных системах. Возникающее событие (например, нажатие кнопки мыши в пределах виджета) пересылается специальной функции-обработчику этого виджета. В качестве аргументов этой функции передаются параметры произошедшего события (координаты курсора, какая кнопка была нажата и т.д.) Событие обрабатывается в виджете, после чего оно может быть уничтожено, либо передано дальше родительскому виджету.

При необходимости сделать нестандартную обработку событий, программисту следует создать класс, производный от необходимого элемента интерфейса, и переопределить в нем обработчик событий, после чего добавить его на форму. Для мелких рутинных операций это очень неудобно. Более простой способ — поставить перехватчик событий в каком-то другом объекте. Это позволит переопределить только один класс. Однако и этот способ обладает своими недостатками, так как функция-перехватчик очень быстро разрастается по мере роста числа элементов интерфейса.

В Qt предложен свой способ обработки событий: механизм сигналов и слотов. **Сигнал** представляет собой особым образом оформленное событие и может иметь произвольные параметры. В отличие от классических событий,

сигнал не требует специального обработчика, и может быть обработан в любом классе в **функции-слоте**. При этом установка соединения между сигналом и слотом может осуществляться в любой момент в процессе работы приложения. При возникновении сигнала будут последовательно вызваны все слоты, которые с ним соединены на данный момент. При необходимости можно отключить слот от сигнала.

Сигнал описывается в теле класса в специальной секции, которая называется `signals`, как функция с произвольными аргументами и без возвращаемого значения. При этом сигнал имеет только объявление, так как он не является функцией в полном смысле этого слова. Для порождения сигнала существует ключевое слово `emit`.

Слот является полноценной функцией, и может быть при необходимости вызван напрямую. Слот объявляется в секции `slots`, которая снабжается модификатором `public`, `private` или `protected`. Тело слота следует определить так же, как и тело любой другой функции.

В приведенном примере объявляется класс `Keeper`, который позволяет задать хранимое значение `value` при помощи слота `setNumber`. При работе слота порождается сигнал `newNumber`, аргументом которого является переданное в объект число.

```
class Keeper: public QObject
{
    Q_OBJECT
signals:
    void newNumber(int n);
public slots:
    void setNumber(int a)
    {
        value = a;
        emit newNumber(value);
    }
private:
```

```
    int value;
}
```

Для соединения сигнала и слота существует функция **connect**. В качестве аргументов функция принимает следующее:

1. Указатель на объект-источник сигнала.
2. Название сигнала, порождаемого объектом, заключенное в макрос SIGNAL.
3. Указатель на объект-приемник сигнала.
4. Название слота, с которым следует соединить сигнал, заключенное в макрос SLOT.

```
class Watcher: public QObject
{
    Q_OBJECT
public:
    Watcher(QObject* parent=0)
    {
        connect(&k,     SIGNAL(newNumber(int)),
                this,   SLOT(onNewNumber(int)));
    }

public slots:
    void onNewNumber(int n);

private:
    Keeper k;
}
```

В приведенном примере в конструкторе класса Watcher производится соединение сигнала newNumber от объекта k класса Keeper со слотом onNewNumber текущего объекта класса Watcher.

Разработка графических приложений с помощью Qt Creator

Библиотека Qt доступна для скачивания в различных вариантах: как в виде исходного кода, так и в виде бинарных файлов, собранных под определенную платформу. Для среды ОС Windows выделяют следующие сборки Qt:

- Сборка для Microsoft Visual C++ (MSVC). Сборка рассчитана на работу с компилятором C++, входящим в состав Microsoft Visual Studio.
- Сборка для компилятора MinGW. Данная сборка поставляется вместе с компилятором и, по сути, представляет собой полностью независимую среду разработки.

Независимо от используемой сборки, всегда доступна среда Qt Creator. В дальнейшем будет подразумеваться использование этой среды.

Для создания приложения необходимо выбрать пункт меню «Файл» → «Создать файл или проект» и в появившемся окне выбрать пункт «Приложение Qt Widgets» (рисунок 2). Следуя указаниям мастера, необходимо указать путь к проекту, используемый компилятор (как правило, он выбран по умолчанию), а также название класса, который будет исполнять роль основного окна. По умолчанию новый класс носит название MainWindow и, в подавляющем большинстве случаев, нет причин его менять.

Внимание! Некоторые версии Qt Creator не могут работать с проектом, содержащим в пути или названии русские буквы!

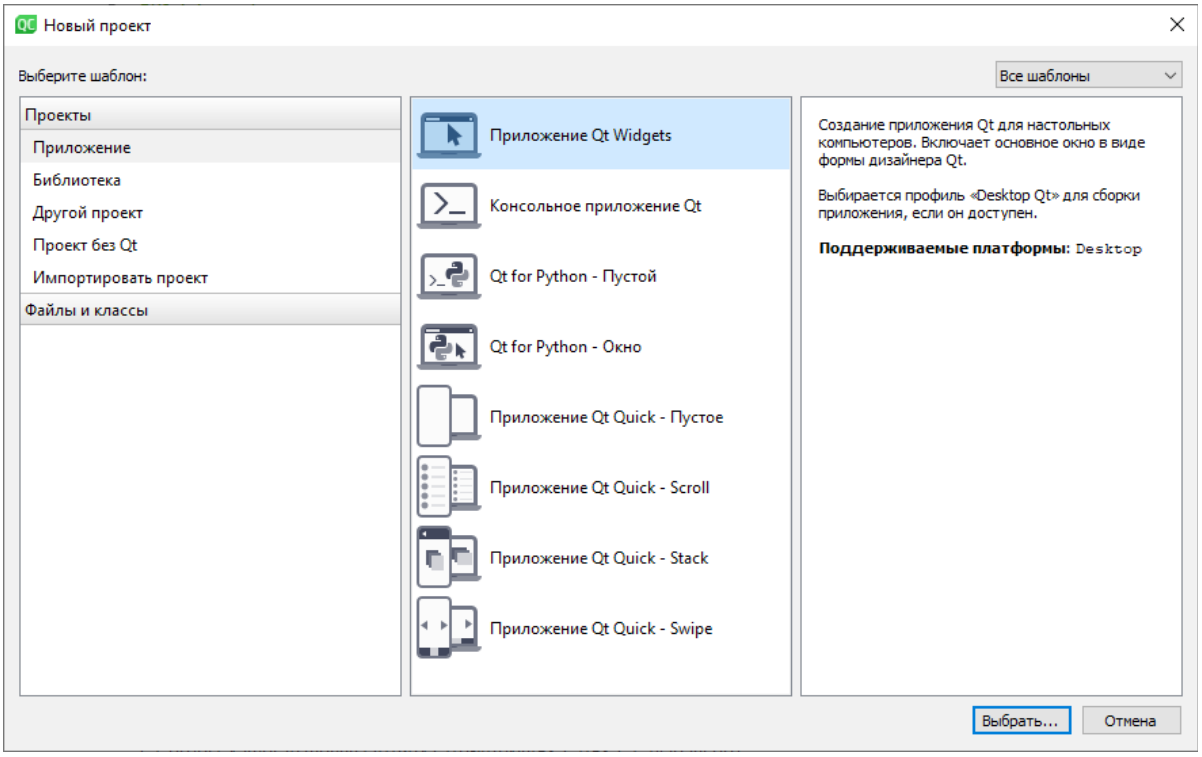

Рисунок 2 - Новый проект в Qt Creator

После завершения работы мастера в папке с проектом будут созданы следующие файлы:

- Файл с расширением .pro — это файл, хранящий описание проекта на языке QMake. Именно его следует указывать при открытии проекта.
- Файл с расширением .pro.user — это файл, хранящий настройки сборки. Содержимое этого файла привязано к конкретному компьютеру, и при переносе проекта этот файл следует удалить.
- Файл main.cpp — это основной файл, в котором описана функция `main` вашего проекта. Он создан автоматически и не требует вмешательства.
- Файл mainwindow.h — это заголовочный файл, содержащий объявление основного класса.
- Файл mainwindow.cpp — это файл, содержащий определения объявленных в заголовочном файле методов.
- Файл mainwindow.ui — это файл, содержащий описание внешнего вида приложения.

Все эти файлы представлены в виде дерева в левом верхнем углу Qt Creator (рисунок 3).

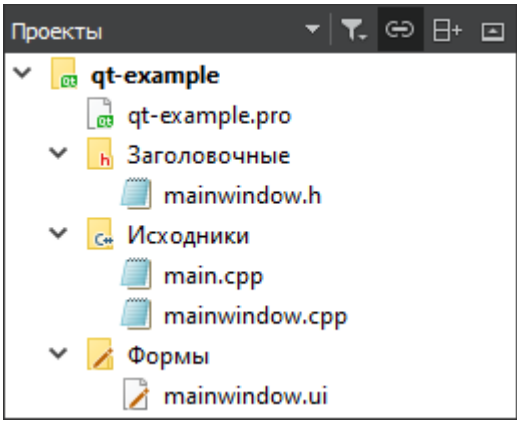

Рисунок 3 - Дерево проекта в Qt

Разработка приложения обычно начинается с описания его интерфейса. Для этого необходимо открыть файл mainwindow.ui. Описание интерфейса также носит название **форма**. Редактирование файла осуществляется в визуальном редакторе форм, который откроется автоматически.

В центре редактора форм расположен интерфейс окна программы без оформления окна (то есть, без рамок и заголовков, добавляемых операционной системой).

В левой области редактора форм располагается список виджетов, доступных для добавления. Виджеты объединены в группы в зависимости от их назначения. Наиболее часто используются следующие группы и виджеты:

- Виджеты-кнопки (Buttons)
  - Push Button — обычная кнопка, которую можно нажать мышкой.
  - Tool Button – квадратная кнопка с иконкой.
  - Radio Button – кнопка, позволяющая выбрать один из предложенных вариантов.
  - Check Box – «галочка», позволяющая выбрать предложенный вариант.
- Виджеты отображения (Display Widgets)
  - Label – обычная надпись на форме.

    - Progress Bar – индикатор прогресса с процентами.
- Виджеты ввода данных (Input Widgets)
    - Line Edit – однострочное поле ввода.
    - Text Edit – многострочное поле ввода с поддержкой форматирования текста.
    - Plain Text Edit – многострочное поле ввода для простого текста без форматирования
    - Spin Box – поле ввода целого числа (тип `int`).
    - Double Spin Box – поле ввода вещественного числа (тип `double`).
    - Horizontal Slider и Vertical Slider – виджет для быстрого ввода целого значения из заданного диапазона.
- Контейнеры (Containers) – виджеты для группировки внутри себя других виджетов.
    - Frame – панель с отображаемыми рамками.
    - Group Box – панель с рамками и подписью.
    - Tab Widget – виджет с несколькими страницами, переключаемыми с помощью закладок. Каждая страница содержит свой независимый набор виджетов.
    - Stacked Widget – аналогично Tab Widget, но страницы не могут быть переключены пользователем и меняются программно.

Для добавления виджета на форму необходимо перетащить его из левой панели на свободное место и отпустить. При необходимости можно поменять расположение виджета, его размеры или переместить его внутрь другого.

В правом верхнем углу экрана расположено дерево виджетов, в котором показаны все виджеты, находящиеся на форме. Каждый виджет в пределах одной формы должен обладать уникальным именем, при помощи которого можно обратиться к нему в исходном коде. Рекомендуется давать виджетам нестандартные имена, чтобы не потеряться в нумерации. Дерево виджетов проекта, содержащего три кнопки, показано на рисунке 4.

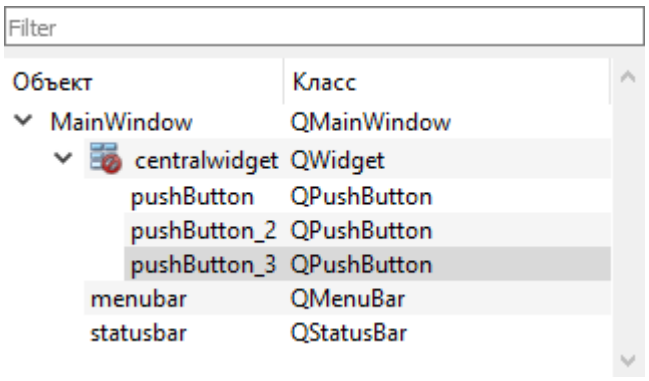

Рисунок 4 - Дерево виджетов

В правом нижнем углу экрана расположено окно свойств выбранного виджета. Свойства сгруппированы в зависимости от того, в каком классе они объявлены. У всех виджетов есть группа свойств под названием QWidget, унаследованная от одноименного класса. Свойства виджета типа QPushButton приведены на рисунке 5.

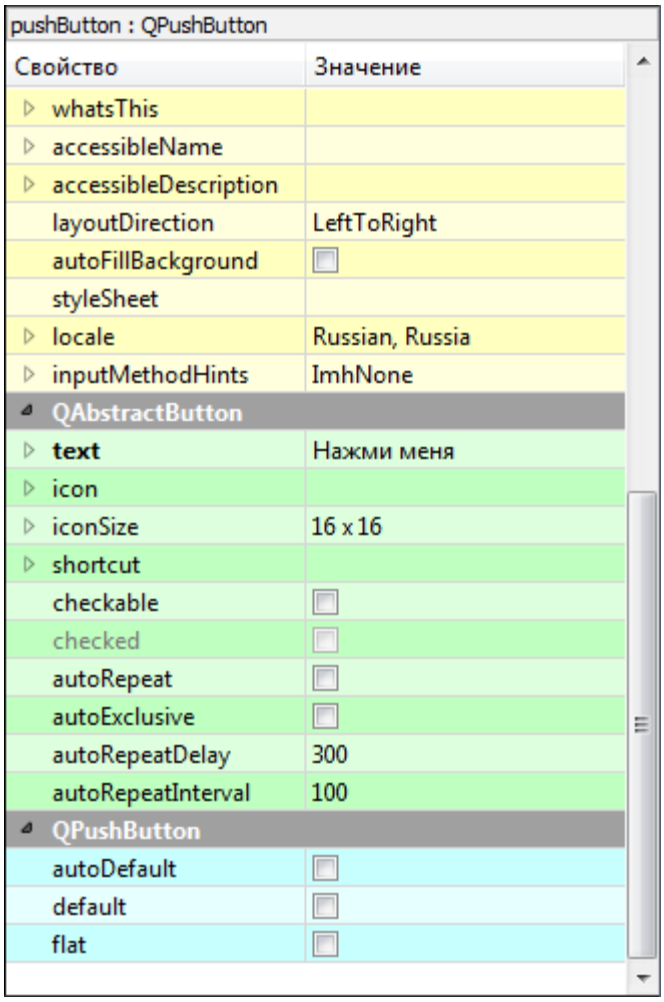

Рисунок 5 - Свойства виджета.

После добавления на форму виджеты расположены в тех местах, в которые их перетащили, и привязаны к левому верхнему углу родительского виджета. Для того, чтобы упорядочить расположение виджетов, следует применить раскладку к родительскому виджету, либо использовать объекты-раскладки (группа Layouts). Чтобы применить раскладку к виджету, необходимо выбрать его и нажать кнопку, соответствующую раскладке, на верхней панели. На рисунке 6 показано, как можно применить раскладку «по вертикали» к основному окну.

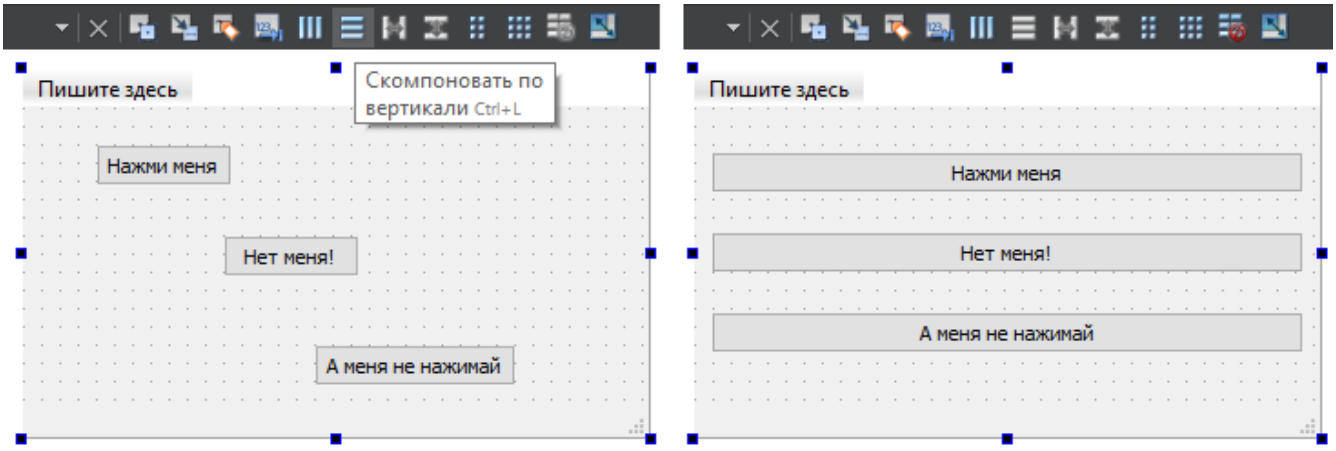

Рисунок 6 - Применение раскладки к виджету.

После создания интерфейса можно приступать к разработке логики его работы, то есть к наполнению класса программным кодом, обеспечивающим функционирование интерфейса. Классы Qt имеют большой набор встроенных функций, обеспечивающих выполнение рутинных операций, таких как отрисовка виджетов, изменение размеров окна, базовая реакция на нажатие кнопок мыши и так далее. Программисту лишь требуется написать слоты, обеспечивающие обработку событий, которые могут возникнуть в процессе работы приложения, и подключить эти слоты к соответствующим сигналам виджетов.

В Qt Creator существует удобный способ создания слота для обработки сигнала от элемента интерфейса. Для этого необходимо найти в дизайнере необходимый элемент интерфейса, правой кнопкой мыши вызвать контекстное меню и в нем выбрать пункт «Go to slot ...» / «Перейти к слоту» (рисунок 7).

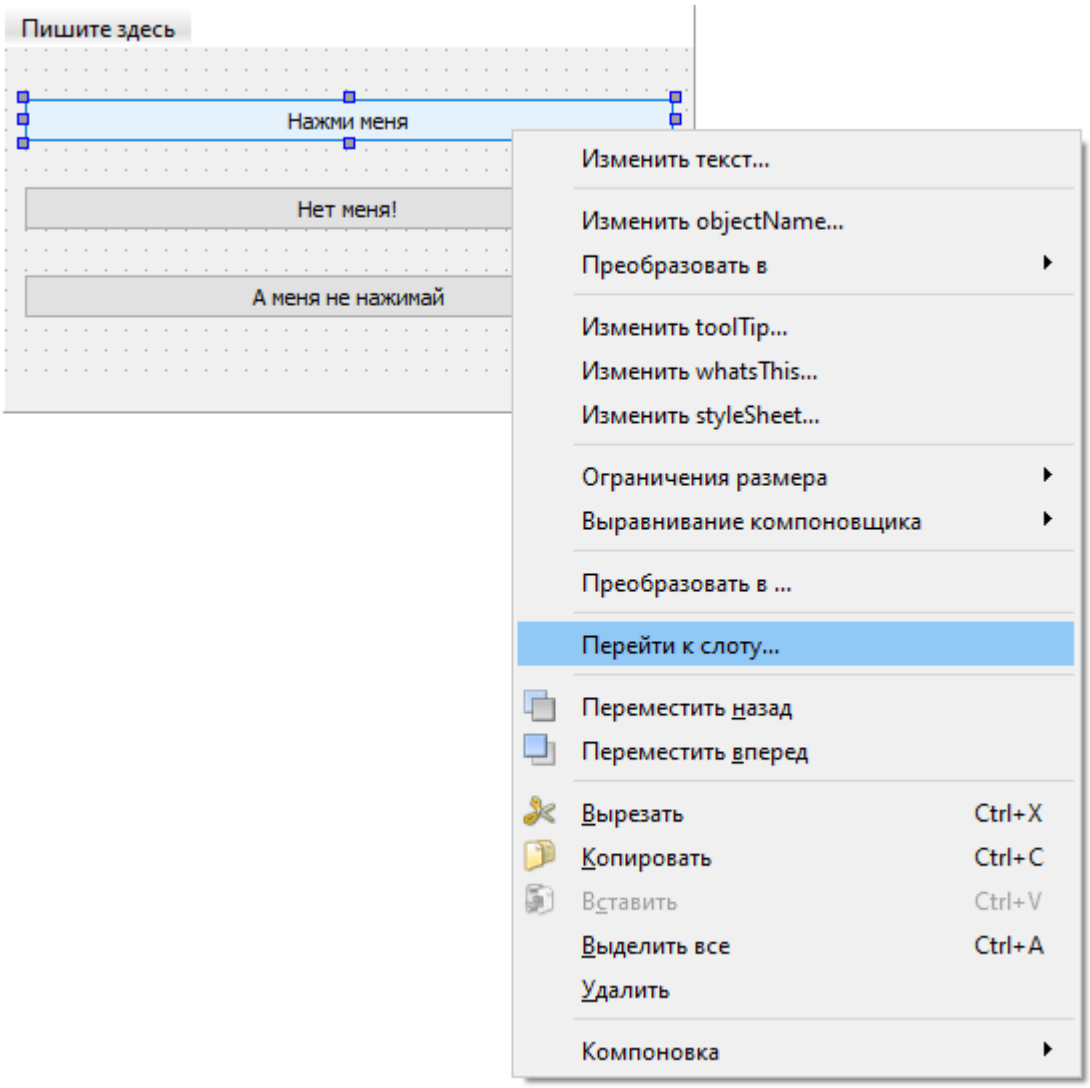

Рисунок 7 - Создание автослота.

В появившемся меню следует выбрать тот сигнал, к которому следует сделать слот, и нажать ОК (рисунок 8).

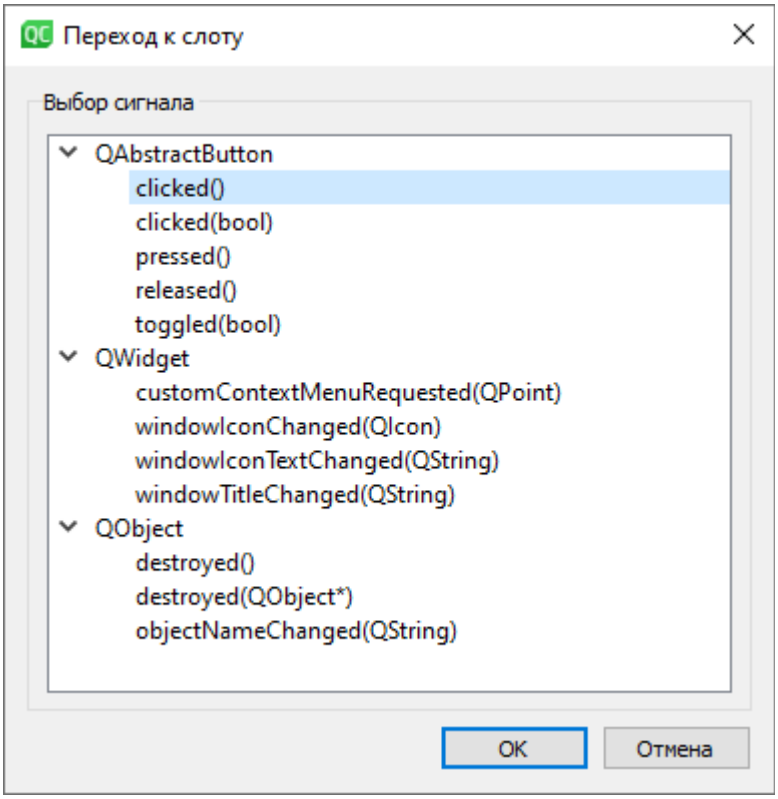

Рисунок 8 - Меню выбора сигнала.

После этого будет создан слот, обрабатывающий сигнал `clicked` от объекта `pushButton`, и Qt Creator перейдет в режим редактирования кода данного слота. Данный слот является так называемым **автослотом** и не требует принудительного вызова `connect()`. Он именуется по шаблону `on_<OBJECT>_<SIGNAL>()`, где вместо `<OBJECT>` подставляется имя объекта, а вместо `<SIGNAL>` — сигнал, к которому слот будет подключен.

В программном коде все элементы интерфейса, добавляемые на форму, доступны при помощи специальной переменной `ui`. Для того, чтобы обратиться к кнопке `pushButton` для изменения написанного на ней текста, необходимо записать следующий программный код:

```
QString s = ui->pushButton->text();
s = QString("_")+ s +QString("_");
ui->pushButton->setText(s);
```

После того, как изменения закончены, необходимо сохранить все файлы исходных кодов, собрать и запустить приложение. Для этого необходимо нажать на

зеленую стрелку в левом нижнем углу экрана. В приведенном примере нажатие на кнопку `pushButton` будет приводить к тому, что текст на кнопке будет увеличиваться на один символ подчеркивания слева и справа.

Краткое справочное руководство по классам Qt

Библиотека Qt содержит большое количество различных классов, решающих разные рутинные задачи. Каждый класс содержит набор методов, при помощи которых можно с ним взаимодействовать. При этом часть методов объявлена непосредственно в классе, а часть унаследована от родительского класса. Все методы класса документированы на английском языке. Справка доступна непосредственно в среде Qt Creator, либо через приложение Qt Assistant.

Можно выделить следующие методы, которые используются наиболее часто:

- Методы `text()` и `setText()` служат для получения и задания текста виджета соответственно. Они применимы ко всем виджетам, хотя некоторые виджеты игнорируют отображаемый текст (например, слайдеры).

```
ui->pushButton->setText("Special button");
```

- Методы `value()` и `setValue()` служат для получения и задания значения в таких элементах, как SpinBox, DoubleSpinBox и в слайдерах.

```
ui->spinBox->setValue(23);
```

- Метод `hide()` позволяет скрыть виджет, а метод `show()` - показать его обратно на экран. Скрытие виджета не удаляет его с формы, последующий вызов `show()` вернет его на прежнее место.

```
ui->label->show();
ui->pushButton->hide();
```

- Метод `setEnabled()` позволяет переключить виджет в выключенное состояние (`false`) и обратно в обычное (`true`).

```
ui->pushButton->setEnabled(false);
```

- Методы `isChecked()` и `setChecked()` позволяют проверять и устанавливать флажки в CheckBox и RadioButton, а также «залипание» кнопок.

```
if (ui->radioButton->isChecked())
```

```
ui->pushButton->setEnabled(true);
```

Среди наиболее часто используемых классов в Qt следует выделить класс `QString`. Он используется для хранения строк и позволяет совершать различные преобразования. Основные методы `QString` следующие:

- Для создания новой строки следует записать `QString("")` и в кавычках указать хранимую строку.

```
QString s1 = QString("Hello!");
QString s2("World);
```

- Для создания строки на русском языке следует использовать функцию `QString::fromUtf8("")`, если кодировка файла исходных кодов UTF-8, или `QString::fromLocal8Bit("")`, если кодировка файла — Windows-1251. Кодировку файла можно узнать при помощи пункта меню «Правка» → «Выбрать кодировку».

```
QString s = QString::fromUtf8("Привет!");
```

- Строки `QString` можно складывать, используя оператор «+», а также присваивать при помощи оператора «=»:

```
QString s1("Hello");
QString s2("World");
QString s3 = s1 + " " + s2;  // "Hello world"
```

- Для создания строки из числа следует использовать функцию `QString::number()`.

```
double c = 3.25;
QString s = QString::number(c);     // "3.25"
```

- Для преобразования из числа в строку существуют функции `toInt()`, `toLong()`,`toFloat()`,`toDouble()`:

```
QString s("1233");
int i = s.toInt();                  // 1233
double d = s.toDouble();            // 1233.0
```

Документация по классам Qt доступна онлайн https://doc.qt.io/qt-5/reference-overview.html (на английском языке), http://doc.crossplatform.ru/qt/4.8.x/html-qt/ (на русском языке для версии Qt 4.8, основной функционал не отличается от Qt5) и в программе Qt Assistant, устанавливаемой вместе со средой разработки.

Порядок проведения работы

В работе необходимо разработать графическое приложение, содержащее несколько простых переменных и позволяющее пользователю их изменять. Для этого необходимо выполнить следующие действия:

1. Разработать основу приложения

   1.1. Запустить Qt Creator и создать новый проект. В качестве типа проекта следует выбрать «Qt Widget Project» и далее — «Qt GUI Application». При создании проекта следует убедиться, что полный путь не содержит русских символов, так как это может впоследствии привести к ошибке сборки.

   1.2. Перейти к описанию интерфейса проекта (файл `mainwindow.ui`) и при помощи дизайнера нарисовать интерфейс, содержащий HorizontalSlider, Label, LineEdit, DoubleSpinBox и PushButton, а также два пункта главного меню. Виджеты в окне следует расположить в раскладке «Сетка» и при необходимости растянуть их, чтобы они занимали соседние ячейки. Также следует поджать виджеты с использованием «пружин» - Spacer. Рекомендуется расположить кнопку внутри отдельной горизонтальной раскладки и также поджать при помощи пружин.

   Для добавления пунктов главного меню необходимо нажать на надпись сверху формы «Пишите здесь» и добавить новое меню, после чего аналогичным образом добавить в него отдельные пункты.

   Внешний вид интерфейса должен соответствовать рисунку 9.

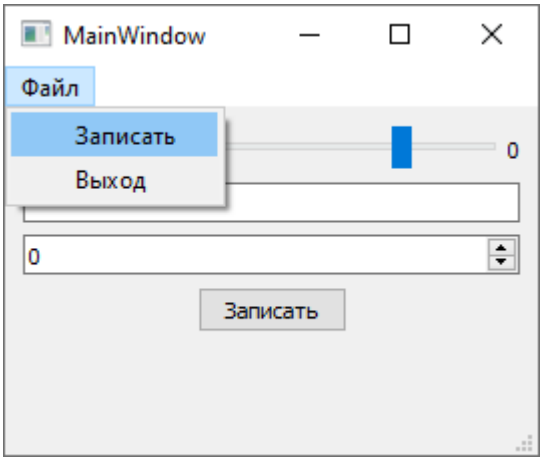

Рисунок 9 - Интерфейс приложения Qt.

1.3. Используя редактор сигналов и слотов (Signals and Slots Editor) в окне редактора форм, соединить сигнал `triggered()` от действия, соответствующего пункту меню «Выход», со слотом `close()` от `MainWindow`. Для этого в редакторе сигналов и слотов необходимо создать новое соединение, нажав на зеленый «плюс» и в выпадающих меню в появившейся строке последовательно выбрать объект-источник сигнала (`action`), порождаемый им сигнал, объект-получатель и слот, который должен быть вызван.

1.4. Также соединить сигнал `triggered()` от действия, соответствующего пункту меню «Записать», со слотом `click()` одноименной кнопки.

1.5. Запустить полученное приложение и убедиться в работоспособности интерфейса.

2. Запрограммировать логику работы приложения

2.1. Создать слот, связанный с сигналом `valueChanged(int)`, порождаемым в слайдере. Аргумент, передаваемый в слот — это новое значение, выставленное в слайдере.

2.2. В теле созданного слота необходимо вывести на Label справа от слайдера число, которое в данный момент выставлено на слайдере. Следует помнить, что для вывода на Label число нужно преобразовать в строку, используя `QString::number`.

2.3. Создать слот, связанный с сигналом `clicked()` кнопки «Записать».

2.4. В теле нового слота необходимо создать три переменные, в которые нужно записать значения из слайдера, текстового поля ввода (LineEdit) и спинбокса.

2.5. Интерфейс MainWindow содержит строку статуса — `statusBar`. Используя метод `showMessage` объекта `statusBar`, вывести в статусную строку сообщение, содержащее значение трех переменных. Строку следует записать в формате «Slider=A, LineEdit=B, SpinBox=C», где A, B и C — значения переменных, полученные на предыдущем этапе.

2.6. Запустить приложение и убедиться в его работоспособности.

3. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ. Вывести полученные значения в файл.

3.1. В теле слота, обрабатывающего нажатие кнопки, используя один из известных вам способов, вывести в произвольный текстовый файл полученные из интерфейса значения.

3.2. Рекомендуется использовать класс `QFile` для работы с текстовыми файлами. При создании экземпляра класса следует передать в конструктор имя файла, затем открыть файл, используя метод `open`, передав в качестве аргумента константу `QFile::WriteOnly`. Запись строки в файл осуществляется методом `write`, однако строку предварительно нужно преобразовать в байтовый массив при помощи метода `toLatin1()` или `toUtf8()` в зависимости от желаемой кодировки:

```
QFile f("test.txt");
f.open(QFile::WriteOnly);
QString s("Sample string");
f.write(s.toUtf8());
```

После записи следует закрыть файл, используя метод `close()`.

3.3. Запустить приложение и убедиться, что файл создается, и в него корректно записываются нужные значения.

Вопросы для самоконтроля и подготовке к защите работы №2

1. В чем заключаются основные особенности объектно-ориентированного программирования?
2. Что такое класс и объект? В чем между ними разница?
3. Для чего используется наследование?
4. Каким образом разделяются объявление класса и определение его методов?
5. Что такое конструктор и деструктор?
6. Для чего нужен механизм виртуальных функций?
7. Что такое виджет? Является ли виджетом текстовое поле ввода?
8. Как связаны между собой файлы mainwindow.h, mainwindow.cpp и mainwindow.ui?
9. Каким образом можно взаимодействовать с элементом интерфейса из программного кода?
10. Для чего нужны сигналы и слотоы?