ЛАБОРАТОРНЫЙ ПРАКТИКУМ
ПО ДИСЦИПЛИНЕ
“ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ СИСТЕМ УПРАВЛЕНИЯ”

Содержание

Лабораторная работа № 1
Компилятор gcc и интегрированная среда Qt Creator

Цель работы: изучить принципы работы с компилятором gcc, сборку однофайловых и многофайловых проектов. А также изучить основы работы с интегрированной средой разработки Qt Creator.

Теоретические сведения

Компиля́ция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на объектно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Компоновщик (также редактор связей или ли́нкер, отангл. *link editor, linker*) — это инструментальная программа, которая производит компоновку (*«линковку»*): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

GNU Compiler Collection (обычно используется сокращение GCC) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU. GCC, является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF) на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain. Он используется как стандартный компилятор для свободных UNIX-подобных операционных систем.

Изначально названный GNU C Compiler поддерживал только язык Си. Позднее GCC был расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java,Фортран, Ada и Go.

Программа gcc, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, gcc запускает необходимые препроцессоры, компиляторы, линкеры.

Файлы с расширением **.**cc или **.**C рассматриваются, как файлы на языке C++, файлы с расширением **.c** как программы на языке C, а файлы с расширением. o считаются объектными.

Чтобы откомпилировать исходный код C++, находящийся в файле F.cc, и создать объектный файл F**.**o, необходимо выполнить команду:

gcc -c F.cc

Опция **–c** означает «только компиляция».

Чтобы скомпоновать один или несколько объектных файлов, полученных из исходного кода - F1.o**,** F2.o, ... - в единый исполняемый файл F, необходимо ввести команду:

gcc -o F F1.o F2.o

Опция **-o** задает имя исполняемого файла.

Опции компиляции

Среди множества опций компиляции и компоновки наиболее часто употребляются те, которые перечислены в талице 1.1.

Таблица 1.1. Опции компилятора

Опция	**Назначение**
-c	Эта опция означает, что необходима только компиляция. Из исходных файлов программы создаются объектные файлы в виде name.o. Компоновка не производится.
-Dname=value	Определить имя name в компилируемой программе, как значение value. Эффект такой же, как наличие строки #define name value в начале программы. Часть =value может быть опущена, в этом случае значение по умолчанию равно 1.
-o file-name	Использовать file-name в качестве имени для создаваемого файла.
-lname	Использовать при компоновке библиотеку libname.so
-Llib-path -Iinclude-path	Добавить к стандартным каталогам поиска библиотек и заголовочных файлов пути lib-path и include-path соответственно.
-g	Поместить в объектный или исполняемый файл отладочную информацию для отладчика gdb. Опция должна быть указана и для компиляции, и для

	компоновки. В сочетании –g рекомендуется использовать опцию отключения оптимизации –O0
-MM	Вывести зависимости от заголовочных файлов, используемых в Си или C++ программе, в формате, подходящем для утилиты make. Объектные или исполняемые файлы не создаются.
-pg	Поместить в объектный или исполняемый файл инструкции профилирования для генерации информации, используемой утилитой gprof. Опция должна быть указана и для компиляции, и для компоновки. Собранная с опцией -pg программа при запуске генерирует файл статистики. Программа gprof на основе этого файла создает расшифровку, указывающую время, потраченное на выполнение каждой функции.
-Wall	Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы.
-O1 -O2 -O3	Различные уровни оптимизации.
-O0	Не оптимизировать. Если вы используете многочисленные -O опции с номерами или без номеров уровня, действительной является последняя такая опция.
-I	Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки.
-L	Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки.
-l	Передается компоновщику. Используется для добавления ваших собственных библиотек для поиска в процессе сборки.

Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это

компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Использование

make [-f make-файл] [цель] …]

Файл ищется в текущем каталоге. Если ключ -f не указан, используется имя по умолчанию для make-файла — Makefile (однако, в разных реализациях make, кроме этого могут проверяться и другие файлы, например, GNUmakefile).

Утилита make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной *цели*.

Стандартные цели для сборки дистрибутивов GNU:

- all — выполнить сборку пакета;
- install — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные каталоги);
- uninstall — удалить пакет (производит удаление исполняемых файлов и библиотек из системных каталогов);
- clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы, созданные в процессе компиляции);
- distclean — очистить все созданные при компиляции файлы и все вспомогательные файлы, созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива.

По умолчанию make использует самую первую цель в make-файле.

Make-файл

Программа make выполняет команды согласно правилам, указанным в специальном файле. Этот файл называется make-файл (makefile, мейкфайл). Как правило, make-файл описывает, каким образом нужно компилировать и компоновать программу. make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:

```
цель1 цель2 ...: реквизит1 реквизит2 ...
        команда1
        команда2
```

...

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизитов.

Правило сообщает make, что файлы, получаемые в результате работы команд (*цели*) являются зависимыми от соответствующих файлов-реквизитов. make никак не проверяет и не использует содержимое файлов-реквизитов, однако, указание списка файлов-реквизитов требуется только для того, чтобы make убедилась в наличии этих файлов перед началом выполнения команд и для отслеживания зависимостей между файлами.

Обычно цель представляет собой имя файла, который генерируется в результате работы указанных команд. Целью также может служить название некоторого действия, которое будет выполнено в результате выполнения команд (например, цель clean в make-файлах для компиляции программ обычно удаляет все файлы, созданные в процессе компиляции).

Строки, в которых записаны *команды*, должны начинаться с символа табуляции.

Рассмотрим несложную программу на Си. Пусть программа program состоит из пары файлов кода — main.c и lib.c, а также из одного заголовочного файла — defines.h, который подключён в обоих файлах кода. Поэтому, для program необходимо из пар (main.c defines.h) и (lib.c defines.h) создать объектные файлы main.o и lib.o, а затем скомпоновать их в program. При сборке вручную требуется исполнить следующие команды:

```
cc -c main.c
cc -c lib.c
cc -o program main.o lib.o
```

Если в процессе разработки программы в файл defines.h будут внесены изменения, потребуется перекомпиляция обоих файлов и линковка, а если изменим lib.c, то повторную компиляцию main.o можно не выполнять.

Таким образом, для каждого файла, который мы должны получить в процессе компиляции, нужно указать, на основе каких файлов и с помощью какой команды он создаётся. Программа make на основе этих данных выполняет следующее:

- собирает из этой информации правильную последовательность команд для получения требуемых результирующих файлов;
- инициирует создание требуемого файла только в случае, если такого файла не существует, или он старше, чем файлы, от которых он зависит.

Если при запуске make явно не указать цель, то будет обрабатываться первая цель в make-файле, имя которой не начинается с символа «.».

Для программы program достаточно написать следующий make-файл:

```
program: main.o lib.o
cc -o program main.o lib.o
main.o lib.o: defines.h
```

Стоит отметить ряд особенностей. В имени второй цели указаны два файла и для этой же цели не указана команда компиляции. Кроме того, нигде явно неуказана зависимость объектных файлов от «*.c»-файлов. Дело в том, что программа make имеет предопределённые правила для получения файлов с определёнными расширениями. Так, для цели-объектного файла (расширение «.o») при обнаружении соответствующего файла с расширением «.c» будет «cc -c» с указанием в параметрах этого «.c»-файла и всех файлов-зависимостей.

Синтаксис для определения переменных:

переменная = значение

Значением может являться произвольная последовательность символов, включая пробелы и обращения к значениям других переменных. С учётом сказанного, можно модифицировать наш make-файл следующим образом:

```
OBJ = main.o lib.o
program: $(OBJ)
    cc -o program $(OBJ)
$(OBJ): defines.h
```

Ход работы

Создание папки для работы в mikroC PRO for PIC

При выполнении лабораторной работы № 1 мы будем использовать папку с именем Lab1, которую необходимо предварительно создать.

С этой целью откройте вашу рабочую папку и создайте в ней (с помощью клавиши F7) новую папку с именем Lab1.

В дальнейшем Вы будете записывать и хранить в этой папке все файлы при выполнении лабораторной работы № 1. Полный путь к этой папке будет такой:

e:\Users\TSD\IS-21\Ivanov\Lab1

Часть I

Сборка простейших однофайловых проектов

В созданной папке создадим файл hello.c с помощью простейшего текстового редактора (например, блокнот) со следующим содержанием:

```
#include <stdio.h>

int main (void) {
  puts("Hello world!");
  getchar();
  return (0);
}
```

Сохраним файл . Затем необходимо запустить Qt 5.5 for Desktop (MinGW 4.9.2 32 bit). В появившемся окне командной строки необходимо перейти на диск E:

```
C:\Qt\Qt5.5.0\5.5\mingw492_32>E:
```

Затем зайти в папку с исходным файлом (hello.c):

```
E:\>cd e:\Users\TSD\IS-21\Ivanov\Lab1
```

И проверить наличие файла:

```
E: :\Users\TSD\IS-21\Ivanov\Lab1> dir
```

Если все выполнено правильно, то в списке вы увидите имя вашего файла (hello.c). Теперь использую компилятор gcc выполним сборку нашего проекта:

```
gcc -o hello.exe hello.c
```

Если в тексте программы присутствуют ошибки, то в командной строке вы увидите сообщения компилятора. Если все прошло удачно, то в списке файлов появится hello.exe. Запустите его, и проверьте правильность выполнения.

Часть II

Сборка многофайловых прокетов.

И так, в этом разделе разберемся каким образом через командную строку собирать проекты, которые состоят из двух и более файлов исходного кода. Для этого нам понадобиться все то, что и в предыдущим разделе, а также утилита mingw32-make. Которая аналогична утилите make.

Теперь в папке с проектом создадим следующе файлы:

- main.c (исходный текст основной программы);
- libTest.h (заголовочный файл);
- libTest.c (исходный текст библиотеки);
- Makefile (инструкции сборки проекта для утилиты make).

Содержание фалов:

main.c

```
#include <stdio.h>
#include "libTest.h"

int a;
int b;

int main() {
  puts ("Start programm");
  scanf("%d",&a);
  scanf("%d",&b);
  int Summa = Sum (a,b);
  printf("Summa=%d\r\n",Summa);
  getchar();
  return(0); }
```

libTest.h

```
#ifndef LIBTEST_H
#define LIBTEST_H

int Sum (int _a,int _b);

#endif
```

libTest.c

```
#include "libTest.h"
```

```
int Sum (int _a,int _b) {
  return (_a+_b); }
```

Makefile

```
CC=gcc
CFLAGS=-c  -static
LDFLAGS=
SOURCES= main.c libTest.c

OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE= Test.exe

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
  $(CC) $(OBJECTS) -o $@ $(LDFLAGS)

clean:
  del $(OBJECTS) $(EXECUTABLE)
```

Опции:

- CC задает компилятор, который будет использоваться для сборки проекта;
- CFLAGS задает опции компилятора;
- SOURCES перечень исходных файлов, входящих в проект;
- OBJECTS задает имя объектных файлов;
- EXECUTABLE имя исполняемого файла, который будет получен после сборки проекта.

Блок «clean:» предназначен для очистки папки нашего проекта от сборочных файлов. Для выполнения данного действия необходимо ввести команду mingw32-make clean.

И так, после того как все файлы созданы, можно приступить к процессу сборки. Для этого необходимо ввести команду:

```
mingw32-make
```

Т.к. MakeFile не указан в параметрах запуска утилиты, то она проведет поиск данного файла в текущей директории. И если данный файл присутствует, то по нему произведет сборку проекта.

Если в проекте нет ошибок, то в папке с проектом появятся следующие файлы:

- main.o;
- libTest.o;
- Test.exe.

Запустим программу и проверим ее работу.

Часть III

Интегрированная среда разработки Qt Creator

Интегрированная среда Qt Creator позволяет разрабатывать и отлаживать программы на языках C и C++, как с использованием графических библиотек Qt так и без.

Запустим данную среду через соответствующий ярлык на рабочем столе и попробуем реализовать и отладить программу, описанную в предыдущем пункте (рисунок 1.1).

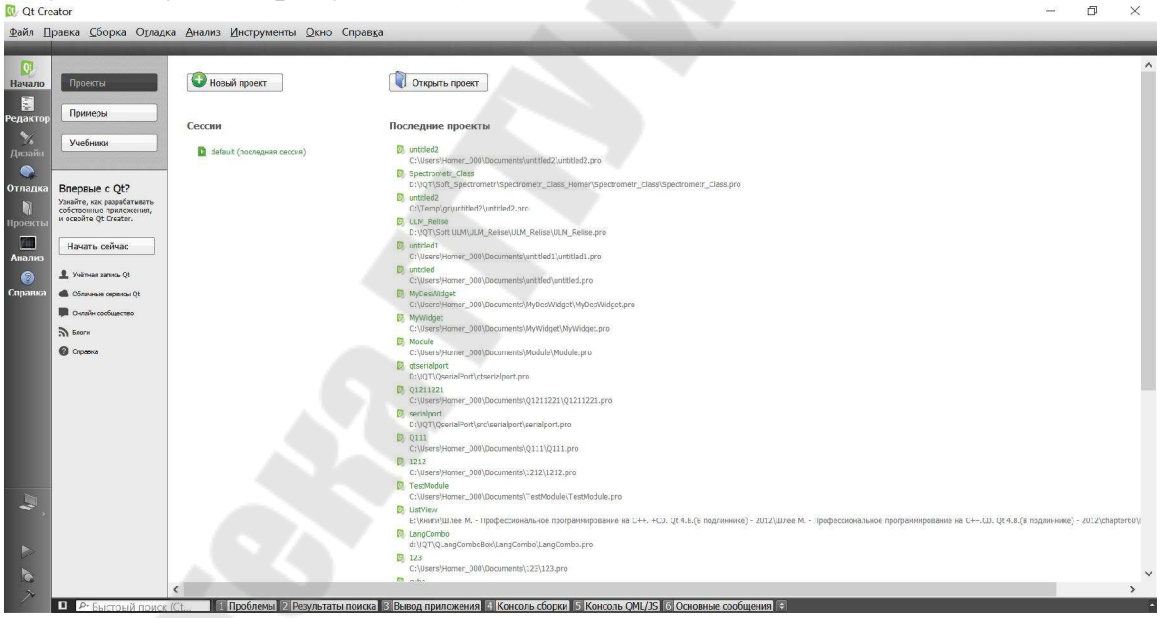


Рис. 1.1 Вид стартового окна Qt Creator

Для создания нового проекта нажмем кнопку «Новый проект» и в появившемся диалоге выберем Проект без использования Qt -> Простой проект на C (рисунок 1.2).

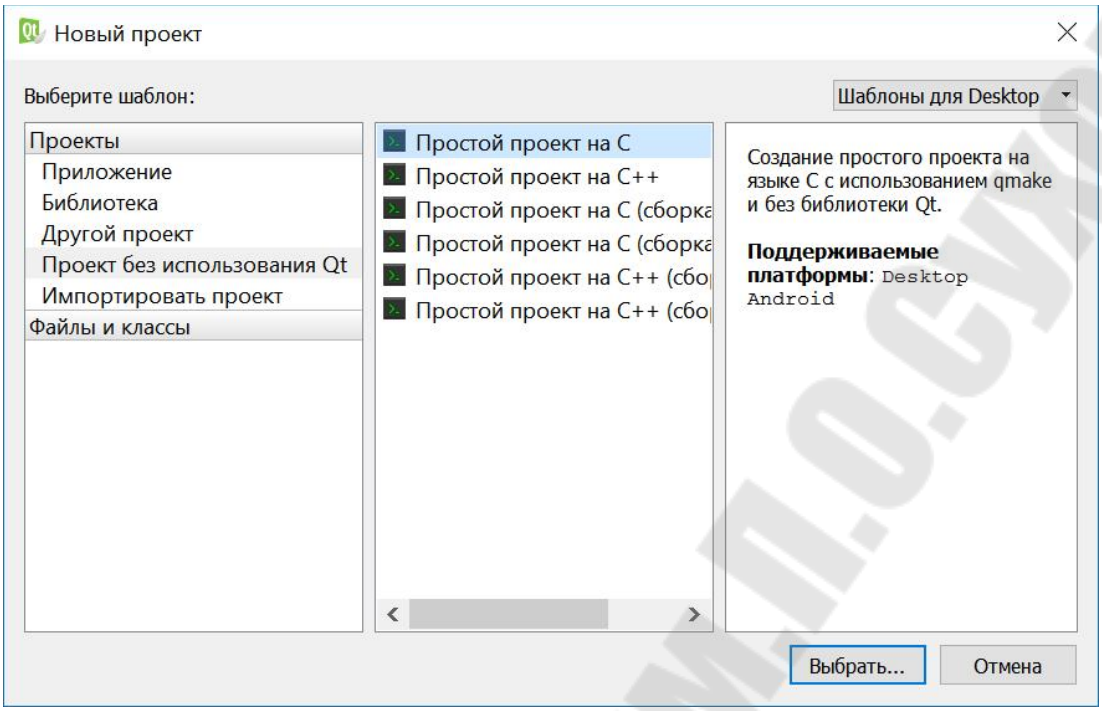


Рис. 1.2. Диалог выбора типа проекта

После появится запрос с именем проекта и его расположением. Укажем имя проекта TestQtCreator и папку с нашими проектами.

После этого появится диалог с выбором комплекта сборки. Т.к. в списке только один вариант то нажмем кнопку далее.

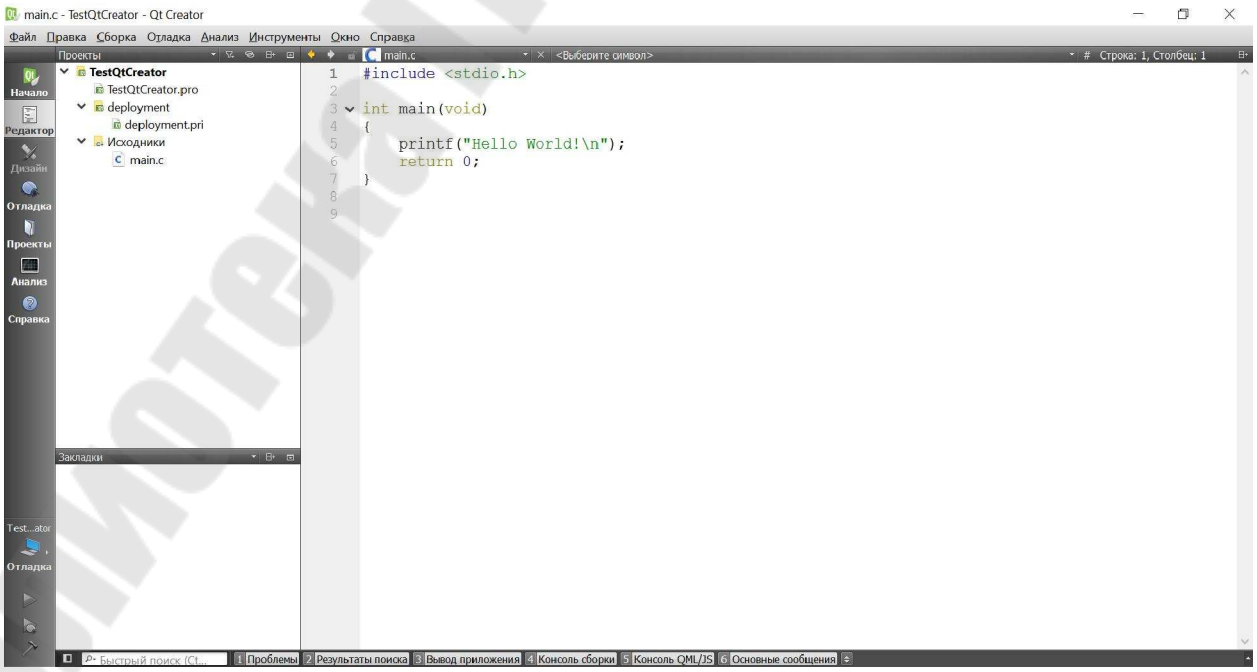


Рис. 1.3 Внещний вид основного окна после создания проекта

В левом верхнем углу располагается область менеджера проектов. В нем отображена структура нашего проекта. По умолчанию он состоит из следующих файлов:

- TestQtCreator.pro (Основной файл проекта);
- deployment.pri (Дополнительный файл проекта);
- main.c (Основной файл исходного кода проекта).

Затем нам необходимо создать библиотеку, состоящую из файлов:

- libTest.h (заголовочный файл);
- libTest.c (исходный текст библиотеки).

Для этого необходимо нажать правой кнопкой мыши по названию нашего проекта, затем в появившемся меню выбрать пункт «Добавить новый…». В появившемся диалоге создания нового файла необходимо последовательно выбрать C++ Source File (для libTest.c) и C++ Header File (для libTest.h).

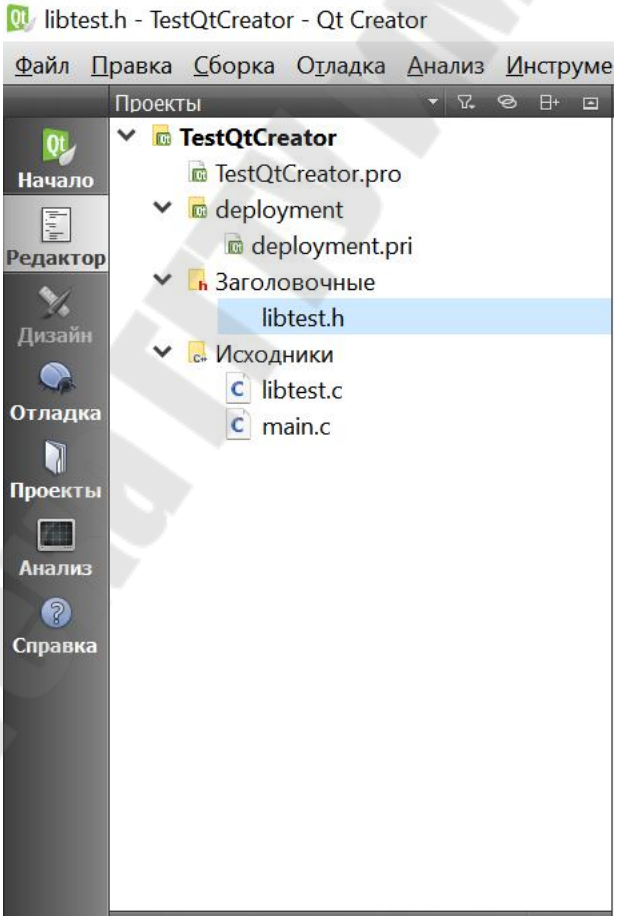


Рис. 1.4. Структура проекта после создани файлов библиотеки

Теперь из предыдущего пункта перенесем текст программы в соответствующие файлы. И запустим сборку проекта. Для этого необходимо нажать сочетание кнопок CTRL + R или слева внизу выбрать соответствующую кнопку. Если проект не содержит ошибок, то появится основное окно программы, такое же, как и в предыдущем пункте.

Часть IV

Задание для самостоятельной работы

Необходимо разработать программу, для вычисления значений следующий функций:

- f1(x) = 2x-5;
- f2(x) = 2x^2 + 6x – 10;
- f3(x) = x^2/(2x^3-6x).

Вычисления функций должны располагаться в библиотеки. Прототипы всех функций описаны в заголовочном файле. Так же программа должна иметь текстовое меню, с возможностью выбора вычисления одной из трёх функций, а также вводом значения X с клавиатуры.

Содержание отчета:

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

Контрольные вопросы:

1. Что такое компиляция?
2. Что такое компоновка?
3. В чем отличия файла с исходными кодами от заголовочного?
4. Из каких основных файлов состоит проект на Qt Creator?
5. Как создать файлы пользовательской библиотеки в Qt Creator?
6. Что такое MakeFile и для чего он предназначен?

Лабораторная работа № 2
Работа с динамической памятью, указателями на массивы данных, передача указателя на массив в качестве аргумента в функцию

Цель работы: изучить принципы работы с динамической памятью. Изучить указатели и их практическое применение при разработке программного обеспечения.

Теоретические сведения

Указатель - это переменная, содержащая адрес переменной. Указатели широко применяются в Си - отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом: в данной главе мы рассмотрим эту зависимость и покажем, как ею пользоваться. Наряду с goto указатели когда-то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, указывающий на что-нибудь совсем нежелательное.

Указатели и адреса

Начнем с того, что рассмотрим упрощенную схему организации памяти. Память типичной машины подставляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа char, двухбайтовые ячейки могут рассматриваться как целое типа short, а четырехбайтовые - как целые типа long. Указатель - это группа ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если с имеет тип char, а p - указатель на с, то ситуация выглядит следующим образом:

p = &c;

Унарный оператор & выдает адрес объекта, так что инструкция присваивает переменной p адрес ячейки с. Оператор & применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Унарный оператор * есть оператор косвенного доступа. Примененный к указателю он выдает объект, на который данный указатель указывает. Предположим, что x и y имеют тип int, а ip – укаэатель на int.

```
int x = 1, y = 2, z[10];
```

```
int *ip;    /* ip - указатель на int */
ip = &x;    /* теперь ip указывает на x */
y = *ip;    /* y теперь равен 1 */
 *ip = 0;    /* x теперь равен 0 */
 ip = &z[0]; /* ip теперь указывает на z[0] */
```

Объявления x, y и z нам уже знакомы. Объявление указателя ip int *ip; мы стремились сделать мнемоничным - оно гласит: "выражение *ip имеет тип int". Синтаксис объявления переменной "подстраивается" под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в объявлениях функций. Например, запись

double *dp, atof (char *);

означает, что выражения *dp и atof(s) имеют тип double, а аргумент функции atof есть указатель на char.

Вы, наверное, заметили, что указателю разрешено указывать только на объекты определенного типа.

Если ip указывает на x целочисленного типа, то *ip можно использовать в любом месте, где допустимо применение x;, например, *ip = *ip + 10; увеличивает *ip на 10.

Унарные операторы * и & имеют более высокий приоритет, чем арифметические операторы, так что присваивание y = *ip + 1; берет то, на что указывает ip, и добавляет к нему 1, а результат присваивает переменной y. Аналогично *ip += 1; увеличивает на единицу то, на что указывает ip; те же действия выполняют ++*ip; и (*ip)++;

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операторы * и ++ имеют одинаковый приоритет и порядок выполнения справа налево.

И наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора косвенного доступа. Например, если iq есть другой указатель на int, то iq = ip; копирует содержимое ip в iq, чтобы ip и iq указывали на один и тот же объект.

Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции. В программе сортировки нам понадобилась функция swap, меняющая местами два неупорядоченных элемента. Однако недостаточно написать swap(a, b); где функция swap определена следующим образом:

```
void swap(int x, int y) /* НЕВЕРНО */
```

```
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Поскольку swap получает лишь копии переменных a и b, она не может повлиять на переменные a и b той программы, которая к ней обратилась. Чтобы получить желаемый эффект, вызывающей программе надо передать указатели на те значения, которые должны быть изменены: swap(&a, &b);

Так как оператор & получает адрес переменной, &a есть указатель на a. В самой же функции swap параметры должны быть объявлены как указатели, при этом доступ к значениям параметров будет осуществляться косвенно.

Графически это выглядит следующим образом в вызывающей программе:

```
void swap(int *px, int *py) /* перестановка *px и *py */
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Аргументы-указатели позволяют функции осуществлять доступ к объектам вызвавшей ее программы и дают возможность изменить эти объекты.

Рассмотрим, например, функцию getint, которая осуществляет ввод в свободном формате одного целого числа и его перевод из текстового представления в значение типа int. Функция getint должна возвращать значение полученного числа или сигнализировать значением EOF о конце файла, если входной поток исчерпан. Эти значения должны возвращаться по разным каналам, так как нельзя рассчитывать на то, что полученное в результате перевода число никогда не совпадет с EOF.

Одно из решений состоит в том, чтобы getint выдавала характеристику состояния файла (исчерпан или не исчерпан) в качестве результата, а значение самого числа помещала согласно указателю, переданному ей в виде аргумента. Показанный ниже цикл заполняет некоторый массив целыми числами, полученными с помощью getint.

```
int n, array[SIZE], getint (int *);
```

```
for (n = 0; n < SIZE && getint (&array[n]) != EOF; n++) ;
```

Результат каждого очередного обращения к getint посылается в array[n], и n увеличивается на единицу. Заметим, и это существенно, что функции getint передается адрес элемента array[n]. Если этого не сделать, у getint не будет способа вернуть в вызывающую программу переведенное целое число.

В предлагаемом нами варианте функция getint возвращает EOF по концу файла; нуль, если следующие вводимые символы не представляют собою числа; и положительное значение, если введенные символы представляют собой число.

```
#include <ctype.h>
int getch (void); void ungetch (int);
/* getint: читает следующее целое из ввода в *pn */
int getint(int *pn)
{
   int c, sign;
   while (isspace(c = getch()))
      ; /* пропуск символов-разделителей */
   if(!isdigit(c) && c != EOF && c != '+' && c != '-') {
      ungetch (c); /* не число */
      return 0;
   }
   sign =(c =='-') ? -1 : 1;
   if (c == '+' || c == '-')
      c = getch();
   for (*pn = 0; isdigit(c); c = getch())
      *pn = 10 * *pn + (c -'0');     *pn *= sign;     if (c!= EOF)
      ungetch(c);
   return c;
}
```

Везде в getint под *pn подразумевается обычная переменная типа int. Функция ungetch вместе с getch (параграф 4.3) включена в программу, чтобы обеспечить возможность отослать назад лишний прочитанный символ.

Указатели и массивы

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя. Вариант с указателями в

общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно. Объявление

int a[10];

Определяет массив a размера 10, т. е. блок из 10 последовательных объектов с именами a[0], a[1], ..., a[9].

Запись a[i] отсылает нас к i-му элементу массива. Если pa есть указатель на int, т. е. объявлен как int *pa; то в результате присваивания pa = &a[0]; pa будет указывать на нулевой элемент a, иначе говоря, pa будет содержать адрес элемента a[0]. Теперь присваивание x = *pa; будет копировать содержимое a[0] в x.

Если pa указывает на некоторый элемент массива, то pa+1 по определению указывает на следующий элемент, pa+i - на i-й элемент после pa, а pa-i - на i-й элемент перед pa. Таким образом, если pa указывает на a[0], то

*(pa+1) есть содержимое a[1], a+i - адрес a[i], а *(pa+i) - содержимое a[i].

Сделанные замечания верны безотносительно к типу и размеру элементов массива a. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы pa+1 указывал на следующий объект, а pa+i - на i-й после pa.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания pa = &a[0]; pa и a имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание pa=&a[0] можно также записать в следующем виде: pa = a;

Еще более удивительно (по крайней мере на первый взгляд) то, что a[i] можно записать как *(a+i). Вычисляя a[i], Си сразу преобразует его в *(a+i); указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора & записи &a[i] и a+i также будут эквивалентными, т. е. и в том и в другом случае это адрес i-го элемента после a. С другой стороны, если pa - указатель, то его можно использовать с индексом, т. е. запись pa[i] эквивалентна записи *(pa+i). Короче говоря, элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель - это переменная, поэтому можно написать pa=a или pa++. Но имя массива не является переменной, и записи вроде a=pa или a++ не допускаются.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать еще одну версию функции strlen, вычисляющей длину строки.

```
/* strlen: возвращает длину строки */ int strlen(char *s)
{
   int n;
   for (n = 0; *s != '\0'; s++)
      n++;
   return n;
}
```

Так как переменная s - указатель, к ней применима операция ++; s++ не оказывает никакого влияния на строку символов функции, которая обратилась к strlen. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции strlen. Это значит, что все вызовы, такие как:

```
strlen("3дравствуй, мир"); /* строковая константа */
strlen(array);            /* char array[100]; */
strlen(ptr);             /* char *ptr; */ правомерны.
```

Формальные параметры char s[]; и char *s; в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что s есть указатель. Если функции в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно - либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если a массив, то в записях

f(&a[2]) или

f(a+2)

функции f передается адрес подмассива, начинающегося с элемента a[2]. Внутри функции f описание параметров может выглядеть как f(int arr[]) {...} или f(int *arr) {...} .

Следовательно, для f тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к

нулевому элементу; выражения p[-1], p[-2] и т.д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед p[0]. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

Ход работы

Часть I

Указатели на простые типы данных

Рассмотрим простейшую программу с указателем на переменную типа int:

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Start programm" << endl;

   int a =0;
   int *b=&a;


   cout << "sizeof(a)=" << sizeof(a) << endl;
   cout << "sizeof(b)=" << sizeof(b) << endl;

   cout << "a=" << a << endl;
   cout << "b=" << b << endl;
   cout << "*b=" << *b << endl;

   cout << "Increment a" << endl;
   a++;

   cout << "a=" << a << endl;
   cout << "b=" << b << endl;
   cout << "*b=" << *b << endl;

   cout << "Increment *b" << endl;
   (*b)++;
```

```
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;

cout << "Increment b" << endl;
b++;

cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;

return 0;
   }
```

Сначала подключаем библиотеку iostream для использования потокового ввода/вывода. Опция using namespace std указывает компилятору, что по умолчанию используются функции из пространства имен std. Если данная опция не будет указана, то команда вывода информации через cout будет выглядеть следующим образом:

```
std::cout << "a=" << a << std::endl;
```

Таким образом, когда в программе используются функции только из пространства std , наиболее удобно применять данную опцию для уменьшения исходного кода.

Объявляем переменную a типа int и указатель b на тип int. Переменной a присвоим значение 0 а указателю b адрес переменно a:

```
int a =0;
int *b=&a;
```

С помощью функции sizeof() определим объем, который занимают в памяти наша переменная и указатель. Т.к. компилятор предназначен для 32-х битной платформы, тип int a так же указатель на него будут занимать в оперативной памяти объем равный 4 байта.

```
cout << "sizeof(a)=" << sizeof(a) << endl;
cout << "sizeof(b)=" << sizeof(b) << endl;
```

Теперь проведем несколько опытов с переменной и указателем. Для начала выведем на экран значение нашей переменной, значение указателя, а также значение объекта на который указывает указатель:

```
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

В первом и третьем случае на экране будет отображено значение нашей переменной, которое равно 0. А вот во втором случае будет выведен адрес, который в памяти занимает наша переменная.

Теперь проведем инкремент переменной a и снова выведем значения на экран:

```
a++;

cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

Значении переменной a после операции инкремента увеличиться на 1. А адрес, находящийся в указателе, останется неизменный. Теперь проведем инкремент объекта, на который указывает b:

```
(*b)++;
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

На экране мы увидим те же изменения, как и в первом случае – значение переменной увеличиться на 1, а адрес останется неизменным. А вот теперь выполним инкремент самого указателя:

```
b++;
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

После выполнения данных действий переменная a останется неизменной, а вот адрес сместиться вправо на 4 байта и значение объекта, на который указывает указатель b, не будет совпадать со значением переменной a.

Часть II

Указатель как параметр функции

Указатели можно применять в качестве входного параметра функции. Рассмотрим небольшую программу:

```
#include <iostream>

using namespace std;
int a;

int pointers(int *b,int c);

int main()
{
   cout << "Start programm" << endl;
   a=5;
   int e=8;
   cout << "a=" << a << endl;
   cout << "e=" << e << endl;

   int d = pointers(&a,e);

   cout << "a=" << a << endl;
   cout << "e=" << e << endl;
   cout << "d=" << d << endl;

   return 0;
}

int pointers(int *b,int c)
{
   a++;
   (*b)++;

   int output;

   output = c + 10 ;

   return output;
```

```
}
```

В начале объявим глобальную переменную a, которая будет доступна всем функциям в текущем файле исходных кодов. Затем присвоим ей значение 5, а также объявим локальную переменную e для функции main.

Пользовательская функция pointers в качестве входных параметров принимает указатель на тип int, а так же переменную того же типа.

```
int pointers(int *b,int c);
```

В данной функции сначал производится инкремент глобальной переменной a затем инкремент объекта, на который указывает указатель b а так же возвращает c + 10.

После выполнения программы, значение a увеличиться на 2, т.к. дважды производится операция инкремента. Первый раз как инкремент глобальной переменной в функции, а второй раз как инкремент объекта, на который ссылается указатель.

Часть III

Указатель на массив

Работа с массивом через указатель заключается в том, что объявляется указатель на тип элемента массива. Затем ему присваивается адрес первого элемента, а затем, за счет инкремента адреса, выполняется перебор элементов.

```
#include <iostream>

using namespace std;

void MasIncrement(int *input,int count);

int main()
{
   cout << "Start program" << endl;

   int Mas[10] = {0,1,2,3,4,5,6,7,8,9};

   cout<<"Massive:"<< endl;
   for (int i=0;i<10;i++) cout<<"Mas["<<i<<"]="<<Mas[i]<<" ";
```

```
    cout<< endl;

    MasIncrement(&Mas[0],10);

    cout<<"Massive New:"<< endl;
    for (int i=0;i<10;i++) cout<<"Mas["<<i<<"]="<<Mas[i]<<" ";
    cout<< endl;

    return 0;
}

void MasIncrement(int *input,int count)
{
    for (int i=0;i<count;i++) (*(input+i))++;
}
```

В функции MasIncrement() выполняется последовательный перебор элементов массива и инкремент каждого элемента.

Часть IV

Приведение типа указателя

Приведение указателей к типам char * или void *- гарантированное преобразование. При приведении обратно будет получен тот же указатель. При приведении к другим типам указателей и обратно обратный указатель может быть отличным от исходного.

Рассмотрим пример программы, в которой мы выполним сериализацию и десериализацию пользовательской структуры.

```
#include <iostream>
#include <stdio.h>

using namespace std;

#pragma pack(push, 1)
struct TestUserStruct
{
    int a;
    int b;
    float c;
};
```

```
#pragma pack(pop)

int main()
{
   cout << "Start programm" << endl;
   TestUserStruct A;

   A.a=5;
   A.b=21;
   A.c=7.5;

   cout<<"A.a="<<A.a<<" A.b="<<A.b<<" A.c="<<A.c<<endl;
   cout<<"sizeof(A)="<<sizeof(A)<<endl;

   char Mas[sizeof(A)];

   //Сериализация сруктуры A
   char *ptr = (char*)&A;
   for (int i=0;i<sizeof(A);i++) Mas[i]=*(ptr+i);

   cout<<"Massiv:"<<endl;
   for (int i=0;i<sizeof(A);i++) printf ("Mas[%d]=%d ",i,Mas[i]);

   //Десериализация структуры B
   TestUserStruct B;

   char *e = (char*)&B;
   for (int i=0;i<sizeof(B);i++)
   {
      *(e+i)=Mas[i];
   }

   cout<<endl;
   cout<<"B.a="<<B.a<<" B.b="<<B.b<<" B.c="<<B.c<<endl;

   return 0;
}
```

В данной программе создается пользовательская структура, состоящая из переменных простых типов данных. Далее производится ее сериализация в массив Mas. Затем производиться десериализация данного

массива во вторую копию структуры B. Если все выполнено правильно, то структуры A и B будут содержать идентичные данные.

Часть V

Задание для самостоятельно работы

Задание 1:

Реализовать в пользовательской библиотеке набор функций, которые осуществляют основные действия над массивами. А именно ввод, вывод, сортировка по возрастанию, сортировка по убыванию. Массив должен передаваться через указатели. В программе должно присутствовать меню, с возможностью выбора необходимого действия.

Задание 2:

Реализовать программу, которая производит сериализацию заданной структуры, сохранение ее в файл. А также загрузку массива данных из файла и десериализацию. В программе должно присутствовать меню, в котором выбирается необходимое действие: Загрузка информации из файла либо сохранение.

Содержание отчета:

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

Контрольные вопросы для самопроверки:

1. Что такое указатель?
2. Какие унарные операции над указателями?
3. Как применить указатель как входной параметр функции?
4. Какие операции над указателями возможны?

Лабораторная работа № 3
Использование классов, встроенных методов, доступ к классу через объект, указатель на объект и через динамическое выделение памяти под класс

Цель работы: изучить принципы использования классов, работу с методами классов. А также технологии доступа к объекту через указатель и динамическое выделение памяти.

Теоретические сведения

Центральным понятием ООП является класс. Класс используется для описания типа, на основе которого создаются объекты (переменные типа класс).

Класс, как и любой тип данных, характеризуется множеством значений, которые могут принимать объекты класса, и множеством функций, задающих операции над объектами.

class Имя_класса { определение_членов_класса };

Члены класса можно разделить на информационные члены и функции-члены (методы) класса. Информационные члены описывают внутреннюю структуру информации, хранящейся в объекте, который создается на основе класса. Методы класса описывают алгоритмы обработки этой информации.

Данные, хранящиеся в информационных членах, описывают состояние объекта, созданного на основе класса. Состояние объекта изменяется на основе изменения хранящихся данных с помощью методов класса. Алгоритмы, заложенные в реализации методов класса, определяют поведение объекта, то есть реагирование объекта на поступающие внешние воздействия в виде входных данных.

Принцип инкапсуляции обеспечивается вводом в класс областей доступа:

- private (закрытый, доступный только собственным методам);
- public (открытый, доступный любым функциям);
- protected (защищенный, доступный только собственным методам и методам производных классов).

Члены класса, находящиеся в закрытой области (private), недоступны для использования со стороны внешнего кода. Напротив, члены класса, находящиеся в открытой секции (public), доступны для использования со

стороны внешнего кода. При описании класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class Имя_класса {
  private:
        определение_закрытых_членов_класса
  public:
          определение_открытых_членов_класса
   protected:
          определение_защищенных_членов_класса
 ...
};
```

Порядок следования областей доступа и их количество в классе – произвольны.

Служебное слово, определяющее первую область доступа, может отсутствовать. По умолчанию эта область считается private.

В закрытую (private) область обычно помещаются информационные члены, а в открытую (public) область – методы класса, реализующие интерфейс объектов класса с внешней средой. Если какой-либо метод имеет вспомогательное значение для других методов класса, являясь подпрограммой для них, то его также следует поместить в закрытую область. Это обеспечивает логическую целостность информации.

После описания класса его имя можно использовать для описания объектов этого типа.

Доступ к информационным членам и методам объекта, описанным в открытой секции, осуществляется через объект или ссылку на объект с помощью операции выбора члена класса ‘.’.

Если работа с объектом выполняется с помощью указателя на объект, то доступ к соответствующим членам класса осуществляется на основе указателя на член класса **‘->’**:

```
class X {  public:
          char c;
        int f(){…}
};

int main () {
        X x1;
```

```
        X & x2 = x1; X
        * p = & x1; int i,
        j, k; x1.c = '*'; i
        = x1.f(); x1.c =
        '+'; j = x2.f();
        x1.c = '#'; k = p
        -> f();
        …
}
```

Объекты класса можно определять совместно с описанием класса:

```
class Y {…} y1, y2;
```

Но правило «хорошего тона» требует описывать структуру класса в .h файле, а непосредственно тело методов в .cpp.

Каждый метод класса, должен быть определен в программе. Определить метод класса можно либо непосредственно в классе (если тело метода не слишком сложно и громоздко), либо вынести определение вне класса, а в классе только объявить соответствующий метод, указав его прототип.

При определении метода класса вне класса для указания области видимости соответствующего имени метода используется операции **'::'**:

Пример:

```
class x {      int
ia1;   public:
    x(){ia1 = 0;}      int
func1();
};

int x::func1(){ … return ia1; }
```

Это позволяет повысить наглядность текста, особенно, в случае значительного объема кода в методах. При определении метода вне класса с использованием операции '::' прототипы объявления и определения функции должны совпадать.

В классах C++ неявно введен специальный указатель this – указатель на текущий объект. Каждый метод класса при обращении к нему получает

данный указатель в качестве неявного параметра. Через него методы класса могут получить доступ к другим членам класса.

Указатель this можно рассматривать как локальную константу, имеющую тип X*, если X – имя описываемого класса. Нет необходимости использовать его явно. Он используется явно, например, в том случае, когда выходным значением для метода является текущий объект.

Данный указатель, как и другие указатели, может быть разыменован.

При передаче возвращаемого значения метода класса в виде ссылки на текущий объект используется разыменованный указатель this, так как ссылка, как уже было указано, инициализируется непосредственным значением.

Пример:

```
class x {
  . . .   public:
    x& f(. . .){
                . . .
                return *this;
          }
};
```

Ход работы

Часть I

Статическое объявление объекта

Рассмотрим простейший пример класса. Который имеет две переменные целого типа в качестве информационного члена класса и 4 метода для работы с ними.

Файл main.cpp

```
#include <iostream>
#include "libclass01.h"

using namespace std;

int main()
{
```

```
    cout << "Start programm" << endl;

    Numbers ObjectNumbers;

    int A;
    cout <<"A=";
    cin >> A;
    ObjectNumbers.setA(A);

    int B;
    cout << "B=";
    cin >> B;
    ObjectNumbers.setB(B);

    cout << "Sum=" << ObjectNumbers.getSum() << endl;
    cout << "Difference=" << ObjectNumbers.getDifference() << endl;
    return 0;
}
```

Файл libclass01.h

```
#ifndef LIBCLASS01
#define LIBCLASS01

class Numbers
{
private:
    int A;
    int B;

public:
    void setA (int input);
    void setB (int input);
    int getSum(void);
    int getDifference(void);


};

#endif // LIBCLASS01
```

Файл libclass01.cpp

```
#include "libclass01.h"

void Numbers::setA (int input)
{
   A=input;
}

void Numbers::setB (int input)
{
   B=input;
}

int Numbers::getSum(void)
{
   int output = A+B;
   return output;
}

int Numbers::getDifference(void)
{
   int output = A-B;
   return output;
}
```

Переменные int A и int B являются информационными членами класса. Методы setA и setB предназначены для задания значения соответствующим информационным членам класса.

Методы getSum и getDifference производят обработку информационных членов класса. Они возвращают сумму (A+B) и разность (A-B) соответственно.

Часть II

Динамическое объявление объекта

Рассмотрим небольшую программу, которая имеет класс для обработки массива целых чисел:

Файл main.cpp

```
#include <iostream>
#include "libclass02.h"

using namespace std;

int main()
{
   cout << "Start programm" << endl;



   int Count=0;
   cout << "Count=";
   cin >> Count;

   Array *Massive = new Array[Count];

   int A[Count];

   for (int i=0;i<Count;i++)
   {
      cout <<"A["<< i <<"]=";
      cin >>A[i];
   }

   Massive->setArray(A,Count);
   cout << "Source array:" <<endl;
   Massive->printArray();
   Massive->workArray();
   cout << "New array:" <<endl;
   Massive->printArray();

   delete Massive;
   return 0;
}
```

Файл libclass02.h

```
#ifndef LIBCLASS02
```

```
#define LIBCLASS02

#include <iostream>

class Array
{
private:
   int *Mas=0;
   int Count_=0;
   int indexOf (int input);

public:
   void setArray(int *input,int Count);
   void printArray(void);
   void workArray(void);

};

#endif // LIBCLASS02
```

Файл libclass02.cpp

```
#include "libclass02.h"

int Array::indexOf (int input)
{
   if ((Count_!=0)&&(Mas!=0))
   {
      for (int i=0;i<Count_;i++)
         if (Mas[i]==input)
         {
            return i;
         }
   } else return -1;

   return -1;
}

void Array::setArray(int *input, int Count)
{
   Mas = new int[Count];
   Count_=Count;
```

```
   for (int i=0;i<Count;i++) Mas[i] = input[i];
}

void Array::printArray(void)
{
   if (Count_!=0)
      for (int i=0;i<Count_;i++)
      {
         std::cout << "Mas[" << i <<"]=" << Mas[i] << std::endl;
      }
}

void Array::workArray(void)
{
   int Pos = 1;
   while (Pos>0)
   {
      Pos = indexOf(0);
      Mas[Pos]=500;
   }
}
```

В данном случае информационным членом класса у нас является массив. Который динамически формируется в оперативной памяти. Метод setArray в качестве входных параметров принимает указатель на созданный в основной программе массив, а также количество элементов в данном массиве. При его вызове в памяти формируется динамический массив. Количество элементов данного массива совпадает с количеством элементов массива, на который ссылается указатель. Затем данные из внешнего массива переносятся во внутренний.

Метод workArray производит обработку внутреннего массива. Каждый элемент массив, который равен 0, принимает значение 500. Внутренний метод indexOf осуществляет поиск элементов с заданным значение, и возвращает номер элемента. Если таковой элемент не найден, то возвращает -1.

Метод printArray осуществляет вывод текущих значений внутреннего массива на экран.

После выполнения необходимых действий, объект удаляется из оперативной памяти.

Часть III

Задание для самостоятельно работы

Разработать класс, который производит обработку двумерного массива. Дана квадратная матрица n*n (размеры матрицы вводятся с клавиатуры). Она заполняется значениями с клавиатуры. Затем с помощью метода setMatrix передается в класс. С помощью метода printMatrix, данная матрица выводится на экран в виде таблицы. Метод workMatrix производит обработку двумерного массива по заданному алгоритму (алгоритм зависит от варианта).

Объект должен быть создан динамически и после выполнения его необходимо удалить из оперативной памяти.

Пояснение:

Квадратная матрица делиться на 4 части (рисунок3.1). С помощью двух диагоналей: главной и побочной.

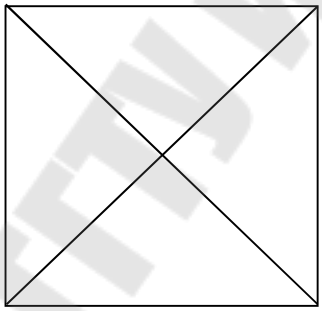


Рис. 3.1. Деление матрицы на 4 части с помощью диагоналей

В результате такого деления образуется 4 треугольника. Выбор треугольника зависит от порядкового номера студента по журналу. Необходимо взять ваш текущий номер и получить остаток от деления на 4. Например номер по журналу 13, разделим его 4 и возьмем остаток от деления = 1. По таблице 3.1 выбрать необходимый треугольник.

Таблица 3.1. Выбор треугольника

Остаток от деления	**Треугольник**
0	Левый
1	Правый
2	Верхний
3	Нижний

Затем в выбранном треугольнике необходимо обработать все элементы, лежащие в нем и на его границах. Программу оптимизировать по выполнению. Т.е. необходимо перебрать в цикле только те элементы, которые принадлежат вашему треугольнику и его границам.

Алгоритм обработки выбирается в зависимости от порядкового номера по журналу из таблицы 3.2.

Таблица 3.2. Выбор алгоритма обработки треугольника

№	Алгоритм обработки
1	Отсортировать элементы по возрастанию
2	Отсортировать элементы по убыванию
3	Определить минимальный элемент
4	Определить максимальный элемент
5	Определить количество отрицательных элементов
6	Определить количество положительных элементов
7	Определить количество нулевых элементов
8	Посчитать сумму минимального и максимального элемента
9	Посчитать произведение максимального и минимального элемента
10	Найти остаток от деления максимального на минимальный элемент
11	Посчитать среднее арифметическое положительных элементов
12	Посчитать среднее арифметическое отрицательных элементов
13	Посчитать произведение положительных элементов
14	Определить максимальный элемент, принадлежащий промежутку [-10,5)
15	Определить минимальный элемент, принадлежащий промежутку [-100,50]
16	Определить количество элементов, кратных 5 и принадлежащих промежутку (4,13]
17	Поменять местами максимальный и минимальный элементы
18	Определить упорядочены ли элементы, если да ,то каким образом
19	Заменить максимальный элемент средним арифметическим всех элементов

Содержание отчета:

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

Контрольные вопросы:

1. Что такое класс?
2. Что такое объект?
3. Как задаются права доступа к членам и методам класса?
4. В чем отличие статического и динамического создания объекта?

Лабораторная работа № 4
Использование методов классов: конструктора, деструктора и дружественной функции

Цель работы: изучить принципы использования конструктора и деструктора класса. А также технологию применения дружественной (friend) функции.

Теоретические сведения

Конструкторы и деструкторы являются специальными методами класса. Конструкторы вызываются при создании объектов класса и отведении памяти под них. Деструкторы вызываются при уничтожении объектов и освобождении отведенной для них памяти.

В большинстве случаев конструкторы и деструкторы вызываются автоматически (неявно), соответственно при описании объекта (в момент отведения памяти под него) и при уничтожении объекта. Конструктор (как и деструктор) может вызываться и явно, например, при создании объекта в динамической области памяти с помощью операции *new*.

Так как конструкторы и деструкторы неявно входят в интерфейс объекта, их следует располагать в открытой области класса.

Отличия и особенности описания конструктора от обычной функции:

- Имя конструктора совпадает с именем класса;
- При описании конструктора не указывается тип возвращаемого значения.

1. Конструкторы можно классифицировать разными способами:

   - по наличию параметров;
   - без параметров;
   - с параметрам.

2. По количеству и типу параметров:

   - конструктор умолчания;
   - конструктор преобразования;
   - конструктор копирования;
   - конструктор с двумя и более параметрами.

В классе может быть несколько конструкторов. В соответствии с правилами языка C++, все они имеют одно имя, совпадающее с именем

класса, что является одним из проявлений статического полиморфизма. Компилятор выбирает тот конструктор, который в зависимости от ситуации, в которой происходит создание объекта, удовлетворяет ей по количеству и типам параметров. Естественным ограничением является то, что в классе не может быть двух конструкторов с одинаковым набором параметров.

Деструкторы применяются для корректного уничтожения объектов. Часто процесс уничтожения объектов включает в себя действия по освобождению выделенной для них по операциям new памяти.

Имя деструктора: ~имя_класса

- У деструкторов нет параметров и возвращаемого значения;
- В отличие от конструкторов деструктор в классе может быть только один.

Конструктор умолчания.

Конструктор без параметров называется конструктором умолчания. Если для создания объекта не требуется каких-либо параметров, то используется конструктор умолчания. При описании таких объектов после имени класса указывается только идентификатор переменной. Роль конструктора умолчания может играть конструктор, у которого все параметры имеют априорные значения.

Конструктор преобразования и конструкторы с двумя и более параметрами.

Если для создания объекта необходимы параметры, то они указываются в круглых скобках после идентификатора переменной.

Указываемые параметры являются параметрами конструктора класса. Если у конструктора имеется ровно один входной параметр, который не представляет собой ссылку на свой собственный класс, то соответствующий конструктор называется конструктором преобразования. Этот конструктор называется так в связи с тем, что в результате его работы на основе объекта одного типа создается объект другого типа (типа описываемого класса).

Конструктор копирования.

При создании объекта его информационные члены могут быть проинициализированы значениями полей другого объекта этого же типа, то есть объект создается как копия другого объекта.

Для такого создания объекта используется конструктор копирования.

Инициализация может быть выполнена аналогично инициализации переменных встроенных типов, с использованием операции присваивания совместно с объявлением объекта.

Если класс не предусматривает создания внутренних динамических структур, например, массивов, создаваемых с использованием операции

new, то в конструкторе копирования достаточно предусмотреть поверхностное копирование, то есть почленное копирование информационных членов класса.

Конструктор копирования, осуществляющий поверхностное копирование, можно явно не описывать, он сгенерируется автоматически.

Если же в классе предусмотрено создание внутренних динамических структур, использование только поверхностного копирования будет ошибочным, так как информационные члены-указатели, находящиеся в разных объектах, будут иметь одинаковые значения и указывать на одну и ту же, размещенную в динамической памяти, структуру. Автоматически сгенерированный конструктор копирования, в данном классе, не позволит корректно создавать объекты такого типа на основе других объектов этого же типа.

В подобных случаях необходимо глубокое копирование, осуществляющее не только копирование информационных членов объекта, но и самих динамических структур. При этом, естественно, информационные члены-указатели в создаваемом объекте должны не механически копироваться из объекта-инициализатора, а указывать на вновь созданные динамические структуры нового объекта.

Автоматическая генерация конструкторов и деструкторов

Автоматически могут генерироваться только конструкторы умолчания, конструкторы копирования и деструкторы.

Если в классе явно неописано ни одного конструктора, то автоматически генерируется конструктор умолчания с пустым телом.

Если в классе явно описан хотя бы один конструктор, например, конструктор копирования, то конструктор умолчания не будет автоматически генерироваться, даже, если он необходим в соответствии с постановкой задачи.

В случае отсутствия в классе явно описанного конструктора копирования, он всегда генерируется автоматически и обеспечивает поверхностное копирование.

Если в классе не описан деструктор, то всегда автоматически генерируется деструктор, который не производит никаких действий.

Таким образом, даже если в классе не описаны конструкторы и деструктор, они все равно неявно присутствуют в нем.

Дружественная функция

Иногда требуется, чтобы функция – не член класса, имела доступ к скрытым членам класса. Основная причина использования таких функций состоит в том, что некоторые функции нуждаются в привилегированном

доступе более, чем к одному классу. Такие функции получили название дружественных.

Для того, чтобы функция – не член класса имела доступ к private-членам класса, необходимо в определение класса поместить объявление этой дружественной функции, используя ключевое слово friend. Объявление дружественной функции начинается с ключевого слова friend и должно находиться только в определении класса.

```
void func(){...}
class A{
...
friend void func();
};
```

Дружественная функция, хотя и объявляется внутри класса, функцией-членом не является. Поэтому не имеет значения, в какой части тела класса (private, public) она объявлена.

Функция-член одного класса может быть дружественной для другого класса.

```
class A
{...
  int func();
};
class B
{...
  friend int A :: func();
};
```

Функция-член func() класса A является дружественной для класса B. Если все функции-члены одного класса являются дружественными функциями второго класса, то можно объявить дружественный класс: friend class ИмяКласса;

```
class A {
...
};
class B
{...
  friend class A;
};
```

Ход работы

Часть I

Применение конструктора и деструктора

Рассмотрим применение конструктора и деструктора, а также технологию полиморфизма.

Пример:

libconstructor.h

```
#ifndef LIBCONSTRUCTOR
#define LIBCONSTRUCTOR
#include <string>
#include <iostream>


class Constructor
{
public:
   Constructor();
   Constructor(std::string input);
   ~Constructor();
private:

};

        #endif // LIBCONSTRUCTOR
```

libconstructor.cpp

```
#include "libconstructor.h"

Constructor::Constructor()
{
   std::cout << "The object is created(1)" << std::endl;
}

Constructor::Constructor(std::string input)
{
   std::cout << "The object is created(2)" << std::endl;
}
```

```
Constructor::~Constructor()
{
   std::cout << "Object deleted" << std::endl;
}
```

main.cpp

```
#include <iostream>
#include <clocale>
#include "libconstructor.h"

using namespace std;

int main()
{
   Constructor *Constr;

   cout << "1: Start programm" << endl;
   cout << "2: Create an object..." << endl;
   Constr = new Constructor();
   cout << "3: Remove the object..." << endl;
   delete Constr;

   cout << "4: Create an object..." << endl;
   Constr = new Constructor("Test");
   cout << "5: Remove the object..." << endl;
   delete Constr;

   return 0;
}
```

Основным классом программы является Constructor. В своем составе он имеет два конструктора и один деструктор:

1. Constructor();
2. Constructor(std::string input);
3. ~Constructor();

Первый конструктор является конструктором умолчания. Он не имеет входных параметров. Он вызывается, когда при создании класса не задается ни одного входного параметра. Второй конструктор является

конструктором с параметром, и вызывается когда при создании класса указывается входной параметр типа std::string.

При использовании первого конструктора на экране должна появится надпись "The object is created(1)" , а при вызове второго появится надпись "The object is created(2)" .

Часть II

Использование дружественной (friend) функции

Как уже было сказано дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

Рассмотрим пример:

libvect.h

```
#ifndef LIBVECT_H
#define LIBVECT_H

class vect
{
public:
   vect(unsigned int n);
   void add(unsigned int input);
   unsigned int gatValue (unsigned char number);

   friend unsigned int getMaxValue(vect *input);

private:
   unsigned int *vactor_;
   unsigned char count=0;
};

unsigned int getMaxValue(vect *input);

      #endif // LIBVECT_H
```

libvect.cpp

```
#include "libvect.h"

using namespace std;
```

```
vect::vect(unsigned int n)
{
   vactor_ = new unsigned int [n];
}

void vect::add(unsigned int input)
{
   vactor_[count] = input;
   count++;
}

unsigned int vect::gatValue (unsigned char number)
{
   return vactor_[number];
}

unsigned int getMaxValue(vect *input)
{
   unsigned int Max=input->vactor_[0];
   for (unsigned int i=1;i<input->count;i++)
      if (input->vactor_[i]>Max) Max = input->vactor_[i];

   return Max;
}
```

main.cpp

```
#include <iostream>
#include "libvect.h"
using namespace std;


int main()
{
   unsigned int n;



   cout<<"Start programm"<<endl;
   cout<<"Enter the number of elements "<<endl;
   cin>>n;
```

```
    vect *vector = new vect(n);

    cout<<"n="<<n<<endl;

    for (unsigned int i=0;i<n;i++)
    {
       cout<<"vector["<<i<<"]=";
       unsigned int buf;
       cin>>buf;
       vector->add(buf);
    }

    cout<<"vector =";

    for (unsigned int i=0;i<n;i++)
    {
       cout<<" "<<vector->gatValue(i);
    }
    cout<<endl;

    cout<<"Maximum element="<<getMaxValue(vector)<<endl;
    return 0;
}
```

В данной программе дружественной функцией является getMaxValue. Она получает доступ к закрытым членам класса через указатель и производит их обработку.

Часть III

Задание для самостоятельно работы

Разработать класс, который в качестве основного члена данных содержит строку (массив типа char) в области private. А также имеет методы для заполнения строки, вывода ее на экран, вставки и удаления символа. А также с помощью дружественной функции реализовать обработку данного массива в соответствии с индивидуальным заданием, которое выбирается из таблицы 4.1.

Таблица 4.1. Выбор алгоритма обработки строки

№	Алгоритм обработки
1	Определить количество заданных символов в строке
2	Определить количество букв в строке
3	Определить количество запятых в строке
4	Определить количество точек в строке
5	Определить количество слов в первом предложении
6	Определить количество слов во всей строке
7	Определить количество слов в последнем предложении
8	Определить количество слов в предложении, длина которых меньше заданной
9	Определить количество слов в предложении, длина которых больше заданной
10	Определить количество предложений в строке
11	Заменить все запятые в строке точками
12	Определить количество двоеточий в строке
13	Заменить первое наибольшее слово заданным
14	Заменить первое наименьшее слово заданным
15	Заменить последнее наибольшее слово заданным
16	Заменить последнее наименьшее слово заданным
17	Удалить все слова, начинающиеся с заданной буквы
18	Удалить все слова короче заданной длины
19	Заменить первое наибольшее слово последним наименьшим

Содержание отчета:

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

Контрольные вопросы:

1. Что такое конструктор?
2. Что такое деструктор?
3. Какие виды конструкторов существую и в чем их отличия?
4. В каких случаях отрабатывает конструктор?
5. В каких случаях отрабатывает деструктор?

Лабораторная работа № 5
Наследование классов, механизм виртуальных функций, переопределение операций

Цель работы: изучить принципы наследования классов, переопределения операций и механизма виртуальных функций.

Теоретические сведения

Правила наследования

Наследование является одним из трех основных механизмов ООЯП. В результате использования механизма наследования, осуществляется формирование иерархических связей между описываемыми типами. Тип-наследник уточняет базовый тип.

Пример:

```
struct A {    int
x,y;
};

struct B: A {    int z;
};

A a1; B b1;
b1.x = 1;
b1.y = 2;
b1.z = 3; a1
= b1;
```

Объект типа *B* наследует свойства объекта типа *A*. Таким образом, объект типа-наследника содержит внутри себя члены базового типа (рисунок 5.1).

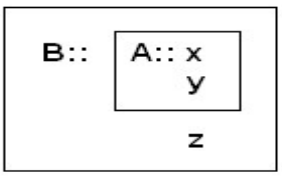


Рис. 5.1. Наследование структур

При наследовании наследуются не только информационные члены, но и методы.

Не наследуются:

- Конструкторы;
- Деструктор;
- Операция присваивания.

Правила видимости при наследовании.

Наследование свойств и поведения могут контролироваться с помощью квалификаторов доступа, задаваемых при наследовании: public, protected, private. Названия квалификаторов доступа совпадают с названиями ранее описанных областей доступа в классах и структурах. Квалификаторы доступа ограничивают видимость полностью или частично, для полностью или частично открытых членов. Закрытые члены всегда остаются закрытыми. При наследовании можно уменьшить видимость членов, но не расширить их видимость.

Если не указан тип наследования, то тип наследования по умолчанию определяется описанием типа наследника. Если тип-наследник описывается классом, то тип наследования – закрытый (private), если же это структура, то наследование по умолчанию будет открытым (public).

Пример:

```
struct A {    int x;
};

class C: A {}; int
main(){
  C c;
  // c.x = 1; // ошибка: в классе C из-за закрытого по
                    // умолчанию наследования поле x
                       // становится закрытым.
  return 0;
}
```

Если тип-наследник описывается структурой, то наследование по умолчанию становится открытым.

Пример:

```
class A {    public:
int x;    private:
int y; };

struct C: A {};

int main(){
   C c;
   c.x = 1;     // ошибки нет, т.к. наследование – открытое.
   return 0;
}
```

При необходимости открытого наследования членов базового типа, если тип-наследник описывается с использованием класса, следует явно указывать квалификатор public:

```
class C: public A {    int z;
};
```

Защищенный вид доступа (protected) означает, что члены базового типа в типе-наследнике доступны только для методов своего (базового) типа, а также для методов производного типа. Во всех остальных случаях они ведут себя так же, как члены с закрытым видом доступа (private).

Ограничение видимости при наследовании ограничивает манипуляции с членами базового типа только в объектах типа-наследника и его потомках. Поэтому при преобразовании указателя типа-наследника к указателю на объекты базового типа, работа с объектом осуществляется в соответствии с правилами видимости для базового класса.

Закрытое (private) наследование

Закрытые члены базового класса недоступны напрямую с использованием дополнительных методов класса-наследника (при любом способе наследования). Работа внутри класса-наследника, с такими получаемыми закрытыми членами базового класса, возможна только с использованием открытых и защищенных методов базового класса.

Закрытые и защищенные получаемые методы недоступны для манипулирования с объектом вне класса. Они могут использоваться как подпрограммы другими методами класса.

При закрытом наследовании открытые и защищенные члены базового класса (любые) доступны только внутри производного класса и недоступны извне (через объекты производного класса), как и его собственные закрытые члены.

Таким образом, приведенная таблица показывает вид доступа для членов в типе наследнике для типа наследника следующего уровня. Но, для текущего типа-наследника доступность зависит от вида доступа в базовом типе.

Перекрытие имен

В производном классе могут использоваться имена членов класса, перекрывающие видимость таких же имен в базовом классе (overriding). При перекрытии имен при работе с объектом через указатель будет исполняться тот метод, который содержится в классе, используемом в объявлении указателя, независимо от типа объекта, на который указывает указатель.

Члены базового класса с именами, совпадающими с именами членов производного класса, доступны в производном классе. Для доступа к ним необходимо указывать квалификатор (имя базового класса) с использованием операции **‘::’**, так как данные члены находятся в доступной области видимости, которая не совпадает с текущей областью видимости. Также метод базового класса доступен через указатель класса-наследника при условии использования квалификатора.

Виртуальную функцию можно использовать, даже если у её класса нет производных классов. Производный класс, который не нуждается в собственной версии виртуальной функции, не обязан её реализовывать.

Интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызывается, в то время как интерпретация вызова не виртуальной функции-члена класса зависит от типа указателя или ссылки, указывающей на этот объект.

Этот механизм делает производные классы и виртуальные функции ключевыми понятиями при разработке программ на C++. Базовый класс определяет интерфейс, для которого производные классы обеспечивают набор реализаций. Указатель на объект класса может передаваться в контексте, где известен интерфейс, определённый одним из его базовых классов, но этот производный класс неизвестен. Механизм виртуальных функций гарантирует, что этот объект всё равно будет обрабатываться функциями, определёнными для него, а не для базового класса.

Спецификатор virtual предполагает принадлежность функции классу, поэтому виртуальная функция не может быть ни глобальной функцией, ни статическим членом класса, поскольку вызов виртуальной функции нуждается в конкретном объекте для выяснения того, какую именно функцию следует вызывать.

Виртуальные функции

Метод класса может содержать спецификатор virtual. Такая функция называется виртуальной. Спецификатор virtual может быть использован только в объявлениях нестатических функций-членов класса.

Если некоторый класс содержит виртуальную функцию, а производный от него класс содержит функцию с тем же именем и типами формальных параметров, то обращение к этой функции для объекта производного класса вызывает функцию, определённую именно в производном классе. Функция, определённая в производном классе, вызывается даже при доступе через указатель или ссылку на базовый класс. В таком случае говорят, что функция производного класса подменяет функцию базового класса. Если типы функций различны, то функции считаются разными, и механизм виртуальности не включается.

Ход работы

Часть I

Простое наследование. Расширение базового класса

Рассмотри программу. Она содержит в себе три класса:

1. parentClass;
2. childClass;
3. childClass2.

Основной класс parentClass содержит в себе конструктор с двумя входными параметрами целого типа. А так же метод summ() , который возвращает сумму входных параметров конструктора. Класс childClass является наследником от parentClass. И в нем добавлен метод proizved (), который возвращает произведение входных параметров конструктора базового класса. Класс childClass2 является наследником обоих предыдущих классов. В нем добавлен метод Raznost (), который возвращает разность входных параметров конструктора первого класса. Рассмотри листинг программы:

libparentclass.h

```
#ifndef PARENTCLASS_H
#define PARENTCLASS_H
#include <iostream>
```

```
using namespace std;

class parentClass
{
public:
   parentClass();
   parentClass(int a_, int b);
   int summ(void);

protected:
   int a;
   int b;
};

    #endif // PARENTCLASS_H
```

libparentclass.cpp

```
#include "libparentclass.h"


parentClass::parentClass()
{

}

parentClass::parentClass(int a_, int b_)
{
   a=a_;
   b=b_;
}

int parentClass::summ(void)
{
   return a+b;
}
```

libchildclass.h

```
#ifndef CHILDCLASS_H
#define CHILDCLASS_H
```

```
#include "libparentclass.h"

class childClass : public parentClass
{
public:
   childClass();
   childClass(int a_, int b_);
   int proizved (void);
};

#endif // CHILDCLASS_H
```

libchildclass.cpp

```
#include "libchildclass.h"


childClass::childClass()
{
   cout<<"Constructor child 1"<<endl;
}
childClass::childClass(int a_, int b_) : parentClass(a_,b_)
{
   cout<<"Constructor child 2"<<endl;
}

int childClass::proizved()
{
   return a*b;
}
```

libchildclass2.h

```
#ifndef CHILDCLASS2_H
#define CHILDCLASS2_H
#include "libchildclass.h"

class childClass2 : public childClass
{
public:
   childClass2(int a_, int b_);
   int Raznost (void);
```

```
};

#endif // CHILDCLASS2_H
```

libchildclass2.cpp

```
#include "libchildclass2.h"

childClass2::childClass2(int a_, int b_) : childClass(a_,b_)
{

}

int childClass2::Raznost (void)
{
   return a-b;
}
```

main.cpp

```
#include <iostream>
#include "libparentclass.h"
#include "libchildclass.h"
#include "libchildclass2.h"

using namespace std;

int main(int argc, char *argv[])
{
   int a,b;
   cout<<"Enter a=";
   cin>>a;

   cout<<"Enter b=";
   cin>>b;

   parentClass *par = new parentClass(a,b);
   cout<<"Sum = "<<par->summ()<<endl;

   childClass *child = new childClass(a,b);
   cout<<"Sum = "<<child->summ()<<endl;
   cout<<"Proizved = "<<child->proizved()<<endl;
```

```
    childClass2 *child2 = new childClass2(a,b);
    cout<<"Sum = "<<child2->summ()<<endl;
    cout<<"Proizved = "<<child2->proizved()<<endl;
    cout<<"Raznost = "<<child2->Raznost()<<endl;
}
```

Примечание:

Как уже было сказано, конструкторы не наследуются. Поэтому при необходимости вызова конструктора базового класса при создании наследника необходимо применять конструкцию childClass::childClass(int a_, int b_) : parentClass(a_,b_). Где parentClass(a_,b_) – конструктор базового класса который необходимо вызвать, при вызове конструктора класса наследника childClass(int a_, int b_).

Часть II

Виртуальные функции

Как уже было сказано, виртуальные функции предназначены для замены методов базового класса методами класса наследника, при вызове наследника через указатель на базовый. Рассмотрим пример использования виртуальной функции при наследовании классов:

libclassvirtual.h

```
#ifndef CLASSVIRTUAL_H
#define CLASSVIRTUAL_H
#include <iostream>

using namespace std;

class classVirtual
{
public:
    classVirtual();
    void f1(void);
    virtual void f2(void);
};

#endif // CLASSVIRTUAL_H
```

libclassvirtual.cpp

```
#include "libclassvirtual.h"

classVirtual::classVirtual()
{

}

void classVirtual::f1()
{
   cout<<"parent f1"<<endl;
}

void classVirtual::f2()
{
   cout<<"parent f2"<<endl;
}
```

libclassvirtualchild.h

```
#ifndef CLASSVIRTUALCHILD_H
#define CLASSVIRTUALCHILD_H
#include "libclassvirtual.h"

class classVirtualChild : public classVirtual
{
public:
   classVirtualChild();
   void f1(void);
   void f2(void);
};

#endif // CLASSVIRTUALCHILD_H
```

libclassvirtualchild.cpp

```
#include "libclassvirtualchild.h"

classVirtualChild::classVirtualChild()
{
```

```
}

void classVirtualChild::f1()
{
   cout<<"child f1"<<endl;
}

void classVirtualChild::f2()
{
   cout<<"child f2"<<endl;
}
```

main.cpp

```
#include <iostream>
#include "libclassvirtual.h"
#include "libclassvirtualchild.h"

using namespace std;

int main(int argc, char *argv[])
{
   cout << "Start programm" << endl;

   classVirtual *virt = new classVirtual();
   virt->f1();
   virt->f2();

   cout<<"---------------------------------------"<<endl;

   classVirtualChild *virt2 = new classVirtualChild();
   virt2->f1();
   virt2->f2();

   cout<<"---------------------------------------"<<endl;

   delete virt;
   virt = new classVirtualChild();
   virt->f1();
   virt->f2();
}
```

Часть III

Задание для самостоятельно работы

Необходимо выбрать стандартный базовый класс (например, класс стринг) и реализовать класс-наследник, с двумя оригинальными методами.

Содержание отчета:

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

Контрольные вопросы:

1. Что такое наследование?
2. Какие составляющие класса не наследуются?
3. Как влияют квалификаторы доступа на наследование?
4. Что такое виртуальная функция и какова область ее применения?

Лабораторная работа № 6
Программирование шаблонов функций и классов

Цель работы: изучить принципы создания и использования шаблонов функций и классов.

Теоретические сведения

Параметрический полиморфизм

Параметрический полиморфизм позволяет применить один и тот же алгоритм к разным типам данных. При этом тип является параметром тела алгоритма. Механизм шаблонов, реализующий параметрический полиморфизм, позволяет легче разрабатывать стандартные библиотеки.

Шаблон представляет собой предварительное описание функции или типа, конкретное представление которых зависит от параметров шаблона. Так, если необходимо написать функции нахождения суммы элементов числовых массивов разных типов (например, int, float или double), то вместо трех различных функций можно написать один шаблон.

Параметры шаблона

Для описания шаблонов используется ключевое слово template, вслед за которым указываются аргументы (формальные параметры шаблона), заключенные в угловые скобки. Формальные параметры шаблона перечисляются через запятую, и могут быть как именами объектов, так и параметрическими именами типов (встроенных или пользовательских). Параметр-тип описывается с помощью служебного слова class или служебного слова typename.

В соответствии со Стандартом ISO 1998 C++ параметром шаблона может быть:

- параметрическое имя типа;
- параметр-шаблон;
- параметр одного из следующих типов:
  - ✓ интегральный;
  - ✓ перечислимый;
  - ✓ указатель на объект любого типа (встроенного или пользовательского) или на функцию;
  - ✓ ссылка на объект любого типа (встроенного или пользовательского) или на функцию;
  - ✓ указатель на член класса, в частности, указатель на метод класса.

Интегральные типы:

- − знаковые и беззнаковые целые типы;
- − bool, char, wchar_t.

При использовании типа в качестве параметра перед параметром, являющимся параметрическим именем типа, необходимо использовать одно из ключевых слов: либо class, либо typename.

Таким образом, параметром шаблона не может быть формальный параметр нецелочисленного типа, например, float или double. Это можно объяснить тем, что основным назначением формальных параметров в качестве параметров шаблона является определение числовых характеристик параметрических типов (например, размер вектора, стека и так далее).

Вместе с тем параметром шаблона может быть указатель, в том числе и на объект нецелочисленного типа. При этом, естественно, указатель – фактический параметр может содержать как адрес одного объекта, так и массива однородных объектов.

Использование шаблонов позволяет создавать универсальные алгоритмы, без привязки к конкретным типам.

Шаблоны функций.

Пример шаблона функции:

```
template <class Type>
  Type min2( Type a, Type b ) {
    return a < b ? a : b;
}
```

Обращение:

```
min2( 10, 20 )
```

При обращении к функции-шаблону после имени функции в угловых скобках указываются фактические параметры шаблона - имена реальных типов или значения объектов.

Ход работы

Часть I

Простое наследование. Расширение базового класса

Рассмотри программу. Она содержит в себе три класса:

1. parentClass;
2. childClass;
3. childClass2.

Основной класс parentClass содержит в себе конструктор с двумя входными параметрами целого типа. А так же метод summ() , который возвращает сумму входных параметров конструктора. Класс childClass является наследником от parentClass. И в нем добавлен метод proizved (), который возвращает произведение входных параметров конструктора базового класса. Класс childClass2 является наследником обоих предыдущих классов. В нем добавлен метод Raznost (), который возвращает разность входных параметров конструктора первого класса. Рассмотри листинг программы:

libparentclass.h

```
#ifndef PARENTCLASS_H
#define PARENTCLASS_H
#include <iostream>

using namespace std;

class parentClass
{
public:
   parentClass();
   parentClass(int a_, int b);
   int summ(void);

protected:
   int a;
   int b;
};

    #endif // PARENTCLASS_H
```

libparentclass.cpp

```
#include "libparentclass.h"


parentClass::parentClass()
{

}

parentClass::parentClass(int a_, int b_)
{
   a=a_;
   b=b_;
}

int parentClass::summ(void)
{
   return a+b;
}
```

libchildclass.h

```
#ifndef CHILDCLASS_H
#define CHILDCLASS_H
#include "libparentclass.h"

class childClass : public parentClass
{
public:
   childClass();
   childClass(int a_, int b_);
   int proizved (void);
};

#endif // CHILDCLASS_H
```

libchildclass.cpp

```
#include "libchildclass.h"
```

```
childClass::childClass()
{
   cout<<"Constructor child 1"<<endl;
}
childClass::childClass(int a_, int b_) : parentClass(a_,b_)
{
   cout<<"Constructor child 2"<<endl;
}

int childClass::proizved()
{
   return a*b;
}
```

libchildclass2.h

```
#ifndef CHILDCLASS2_H
#define CHILDCLASS2_H
#include "libchildclass.h"

class childClass2 : public childClass
{
public:
   childClass2(int a_, int b_);
   int Raznost (void);
};

#endif // CHILDCLASS2_H
```

libchildclass2.cpp

```
#include "libchildclass2.h"

childClass2::childClass2(int a_, int b_) : childClass(a_,b_)
{

}

int childClass2::Raznost (void)
{
   return a-b;
}
```

main.cpp

```
#include <iostream>
#include "libparentclass.h"
#include "libchildclass.h"
#include "libchildclass2.h"

using namespace std;

int main(int argc, char *argv[])
{
   int a,b;
   cout<<"Enter a=";
   cin>>a;

   cout<<"Enter b=";
   cin>>b;

   parentClass *par = new parentClass(a,b);
   cout<<"Sum = "<<par->summ()<<endl;

   childClass *child = new childClass(a,b);
   cout<<"Sum = "<<child->summ()<<endl;
   cout<<"Proizved = "<<child->proizved()<<endl;

   childClass2 *child2 = new childClass2(a,b);
   cout<<"Sum = "<<child2->summ()<<endl;
   cout<<"Proizved = "<<child2->proizved()<<endl;
   cout<<"Raznost = "<<child2->Raznost()<<endl;
}
```

Примечание:

Как уже было сказано, конструкторы не наследуются. Поэтому при необходимости вызова конструктора базового класса при создании наследника необходимо применять конструкцию childClass::childClass(int a_, int b_) : parentClass(a_,b_). Где parentClass(a_,b_) – конструктор базового класса который необходимо вызвать, при вызове конструктора класса наследника childClass(int a_, int b_).

Часть II

Виртуальные функции

Как уже было сказано виртуальные функции предназначены для замены методов базового класса методами класса наследника, при вызове наследника через указатель на базовый. Рассмотрим пример использования виртуальной функции при наследовании классов:

libclassvirtual.h

```
#ifndef CLASSVIRTUAL_H
#define CLASSVIRTUAL_H
#include <iostream>

using namespace std;

class classVirtual
{
public:
   classVirtual();
   void f1(void);
   virtual void f2(void);
};

#endif // CLASSVIRTUAL_H
```

libclassvirtual.cpp

```
#include "libclassvirtual.h"

classVirtual::classVirtual()
{

}

void classVirtual::f1()
{
   cout<<"parent f1"<<endl;
}

void classVirtual::f2()
{
   cout<<"parent f2"<<endl;
```

```
}
```

libclassvirtualchild.h

```
#ifndef CLASSVIRTUALCHILD_H
#define CLASSVIRTUALCHILD_H
#include "libclassvirtual.h"

class classVirtualChild : public classVirtual
{
public:
   classVirtualChild();
   void f1(void);
   void f2(void);
};

#endif // CLASSVIRTUALCHILD_H
```

libclassvirtualchild.cpp

```
#include "libclassvirtualchild.h"

classVirtualChild::classVirtualChild()
{

}

void classVirtualChild::f1()
{
   cout<<"child f1"<<endl;
}

void classVirtualChild::f2()
{
   cout<<"child f2"<<endl;
}
```

main.cpp

```
#include <iostream>
#include "libclassvirtual.h"
#include "libclassvirtualchild.h"
```

```
using namespace std;

int main(int argc, char *argv[])
{
   cout << "Start programm" << endl;

   classVirtual *virt = new classVirtual();
   virt->f1();
   virt->f2();
   cout<<"----------------------------------------"<<endl;
   classVirtualChild *virt2 = new classVirtualChild();
   virt2->f1();
   virt2->f2();
   cout<<"----------------------------------------"<<endl;
   delete virt;
   virt = new classVirtualChild();
   virt->f1();
   virt->f2(); }
```

Часть III

Задание для самостоятельно работы

Необходимо выбрать стандартный базовый класс (например, класс стринг) и реализовать класс-наследник, с двумя оригинальными методами.

Содержание отчета:

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

Контрольные вопросы:

1. Что такое наследование?
2. Какие составляющие класса не наследуются?
3. Как влияют квалификаторы доступа на наследование?
4. Что такое виртуальная функция и какова область ее применения?

Лабораторная работа № 7
Файловый ввод-вывод с применением файловых потоков

Цель работы: изучить принципы работы с файлами через потоки.

Теоретические сведения

Файлом называют способ хранения информации на физическом устройстве. Файл — это понятие, которое применимо ко всему — от файла на диске до терминала.

В C++ отсутствуют операторы для работы с файлами. Все необходимые действия выполняются с помощью функций, включенных в стандартную библиотеку. Они позволяют работать с различными устройствами, такими, как диски, принтер, коммуникационные каналы и т.д. Эти устройства сильно отличаются друг от друга. Однако файловая система преобразует их в единое абстрактное логическое устройство, называемое потоком.

Текстовый поток — это последовательность символов. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток — это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Классическая работа с файловыми потоками

В C++ для работы с файловыми потоками используется библиотека fstream. Для ее подключения в заголовке исходного файла необходимо указать команду #include <fstream>.

Для записи информации в файл используется поток вывода ofstream, а для чтения информации из файла применяется поток ввода ifstream.

Для связи файла с потоком вывода необходимо выполнить команду:

ofstream имя логического файла;

Для связи файла с потоком ввода необходимо выполнить команду:

ifstream имя логического файла;

Например:

```
ofstream OutputFile;
ifstream InputFile;
```

Для открытия файла применяется метод open соответствующего потока. OutputFile.open(имя физического файла) и InputFile.open(имя физического файла).

Например:
OutputFile.open(“Output.dat”);
InputFile.open(“Input.dat”);

Чтение информации осуществляется аналогично, как и с потоком ввода.

Например, из файла InputFile нам необходимо прочитать целое число. Для этого выполним следующую команду:

```
int number;
InputFile >> number;
```

Запись информации в файл так же аналогична выводу информации в поток вывода. Например, нам необходимо записать целое число 1024 в файл OutputFile. Для этого выполним команду:

```
int number = 1024;
OutputFile << number;
```

После завершение всех операций чтения и записи, необходимо выполнить команду закрытия файла. Иначе в процессе выполнения программы могут возникать ошибки. А также файл открыты для записи будет заблокирован вашей программой и другая программа не сможет получить к нему доступ. Для этого необходимо выполнить метод close() файлового потока.

Например закроем файлы InputFile и OutputFile:

```
InputFile.close();
OutputFile.close();
```

Операции над файлами с использованием стандартных классов Qt.

Для работы с файлами в Qt присутствует класс QFile. Для работы с ним необходимо произвести подключение соответствующей библиотеки #include <QFile>.

Основные открытые методы класса QFile приведены в таблице 7.1.

Таблица 7.1. Основные метода класса QFile

Метод	Описание
QFile()	Конструктор умолчания. Не имеет входных параметров. Все необходимые данные устанавливаются с помощью отдельных методов

QFile(const QString &name)	Конструктор, входным параметром является имя открываемого файла. При использовании данного конструктора нет необходимости указывать файл с помощью других методов
copy(const QString &newName)	Метод копирования файлов. Указанный перед вызовом этого метода файл копируется в файл, указанный в входном параметре данного метода. Данные метод возвращает значение типа bool. В случае удачного выполнения операции оно принимает значение true.
copy(const QString &fileName, const QString &newName)	Данный метод тоже производит копирование файлов. Однако в отличии от предыдущего имя исходного файла так же, как и имя файла назначения, указывается в качестве входного параметра.
exists(const QString &fileName)	Метод проверяет существование файла, указанного во входном параметре. В случае если файл существует – возвращается true.
exists() const	Метод проверяет существование ранее указанного файла.
fileName() const	Возвращает значение типа Qstring, в котором указано имя файла с которым работает данный объект.
open(OpenMode mode)	Данный метод осуществляет открытие файла, указанного ранее. В качестве входного параметра передается режим доступа. В случае удачного открытия файла возвращается true.
remove()	Метод, осуществляет удаление ранее указанного файла. В случае удачи возвращает true.
remove(const QString &fileName)	Данный метод аналогичен предыдущему, однако осуществляется удаление файла, указанного в качестве входного параметра.
rename(const QString &newName)	Метод производит переименование ранее указанного файла. Новое имя файла принимается в качестве входного параметра. В случае удачи возвращает true.
rename(const QString &oldName, const QString &newName)	Метод так же производит переименование файла, однако исходное имя файла, как и новое, передается в качестве входного параметра. При успешном выполнении возвращает true.
setFileName(const QString &name)	Метода задает имя текущего файла. Данный метод применятся в случае когда текущий файл

	не задан в конструкторе, либо необходимо изменить тешущий файл.
read(qint64 maxSize)	Данный метод читает из файла заданное количество байт. Возвращает массив типа QByteArray
readAll()	Метод читает файл полностью и возвращает данные массивом типа QByteArray
write(const QByteArray &byteArray)	Метод записывает в файл массив типа QByteArray. Возвращает количество записанных байт
atEnd() const	Данный метод сигнализирует о конце файла. Если при операции чтения достигнут конец файла, данный метод возвращает true.
size() const	Метод возвращает размер текущего файла.

Для открытия файла применяется метод open(OpenMode mode). Как уже было сказано, в качестве входного параметра в него передается режим доступа. Список возможных режимов доступа к файлу приведен в талице 7.2.

Таблица 7.2. Режимы доступа к файлу

Режим	Значение
QIODevice::NotOpen	файл не открыт (это значение не имеет смысла передавать в метод open()).
QIODevice::ReadOnly	открытие файла только для чтения данных.
QIODevice::writeOnly	открытие файла только для записи данных.
QIODevice::ReadWrite	открытие файла для чтения и записи данных.
QIODevice::Append	открытие файла для добавления данных.
QIODevice::Unbuffered	Открытие файла для непосредственного доступа к данным, в обход промежуточных буферов чтения и записи.
QIODevice::Text	применяются преобразования символов переноса строки в зависимости от платформы. Для ОС Windows, например — \r\n, а для MacOS X и UNIX — /r.
QIODevice::Truncate	все данные файла, по возможности, должны быть удалены при открытии.

Пример открытия файла для чтения:

QFile file(“Input.txt”); //Создаем объект типа QFile и передаем имя фала

```
file.open(QIODevice::ReadOnly); //Открываем файл для чтения
QByteArray buf;
buf  = file.readAll(); //Считываем все содержимое файла в массив
```

Пример открытия файла для записи:

```
QFile file(“Output.txt”); //Создаем объект типа QFile и передаем имя фала
file.open(QIODevice::writeOnly); //Открываем файл для записи
QString buf = “Test string”; //Формируем строку для записи
file.write(buf.toLatin1()); //Записываем данные в файл
```

Примечение: Как видно из таблицы 1, метод write() принимает в качестве параметра массив типа QByteArray. Для того чтобы записать строку необходимо выполнить преобразование. Для этого вызываем метод toLatin1().

Применение классов QTextStream и QDataStream при работе с файлами.

Класс QTextStream предназначен для чтения текстовых данных. Числовые данные, передаваемые в поток, автоматически преобразуются в текст. Можно управлять форматом их преобразования, например, метод QTextStream::setRealNumberPrecision() задает количество знаков после запятой. Следует использовать этот класс для считывания и записи текстовых данных, находящихся в формате Unicode.

Класс QDateStream очень похож на предыдущий, но рассчитан для работы с двоичными данными.

Пример записи данных в текстовый файл через поток:

```
QFile file("file.txt");
QTextStream stream(&file);
file.open(QIODevice::writeOnly); //Открываем файл для записи
QString str = "This is a test";
stream << str.toUpper();
file.close();
```

Более подробную информацию по данным классам можно найти в программе Qt Assistant.

Ход работы

Часть I

Файловый ввод-вывод с использованием стандартной библиотеки языка С.

Рассмотрим простейший пример работы с файлом в языке С++. Программа осуществляет запись целого числа в файл и считывание его из файла.

main.cpp

```
#include <iostream>
#include <fstream> //Подключаем библиотеку для работу с файловыми потоками

using namespace std; //Задаем пространство имен std по умолчанию для всего проекта

int main(int argc, char *argv[])
{
    cout<<"Enter number"<<endl;
    int number;
    cin>>number; //Считываем число с клавиатуры

    ofstream *OutputFile = new ofstream;
    //Динамически создаем объект на основе класса
    //ofstream для записи информации в файл
    OutputFile->open("information.dat");
   //Открываем файл information.dat который будет
   //располагаться в папке с исполняемым файлом
   *(OutputFile) << number; //Записываем введенное число в файл
    OutputFile->close();  //Закрываем файл
    delete OutputFile;  //Удаляем объект

    ifstream InputFile;
    //Статически объявляем объект на основе класса ifstream для чтения
    //информации из файла
    InputFile.open("information.dat");
   //Открываем файл information.dat находящийся в папке
    //с исполняемым файлом
    int inputNumber;
```

```
  //Объявляем переменную для считывания в нее информации из файла
   InputFile >> inputNumber;
  //Считываем информацию
   InputFile.close();
  //Закрываем файл
   cout<<"inputNumber="<<inputNumber<<endl;
  //Выводим значние переменной на экран

   return 0;
}
```

В результате выполнения программы в папке с исполняемым файлом должен появится файл information.dat. В котором будет записано число, введенное с клавиатуры.

Часть II

Ввод-вывод с использованием класса QFile.

Рассмотри консольное приложение Qt, которое выполняет запись и чтение файла через класс QFIle.

Для создания консольного приложения с использованием библиотеки Qt, необходимо выбрать пункт «Консольное приложение Qt» (рисунок 7.1).

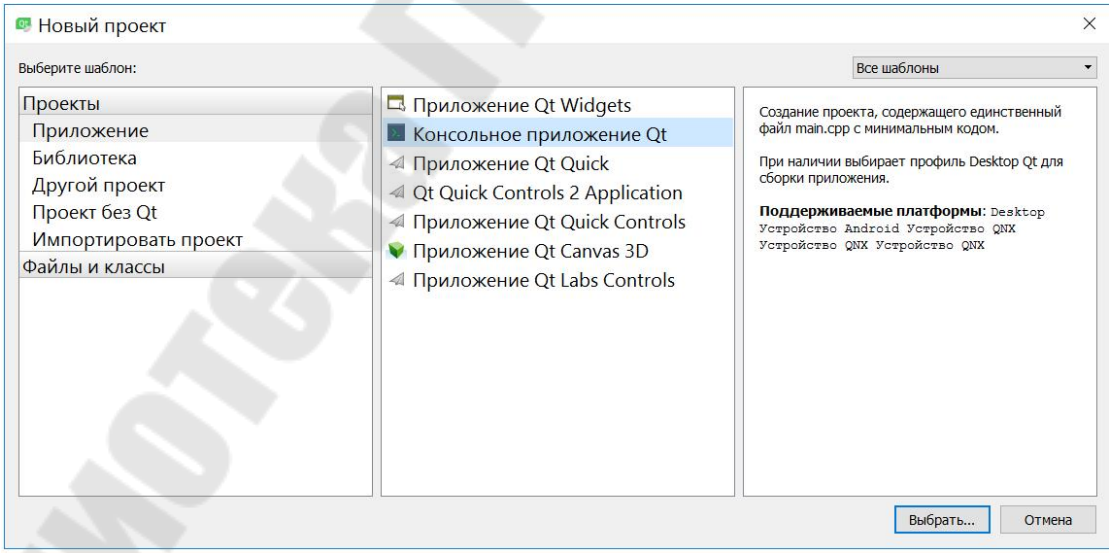


Рисунок 7.1. – Создание консольного приложения с использованием библиотеки Qt

Содержание файлов проекта:

Файл Programm7_2.pro

```
QT += core
QT -= gui

CONFIG += c++11

TARGET = Programm7_2
CONFIG += console
CONFIG -= app_bundle

TEMPLATE = app
SOURCES += main.cpp
```

Файл main.cpp

```
#include <QDebug>
#include <QFile>
#include <QByteArray>

int main(int argc, char *argv[])
{
    qDebug()<<"Start programm";
    QString testString = "Test string"; //Создаем тестовую строку
    qDebug()<<"Write to file string: "<<testString;
     //Вывовим ее значение на экран
    QFile outputFile("information.txt");
    //Создаем объект для работы с файлом и задаем имя файла
    outputFile.open(QIODevice::WriteOnly);
    //Открываем файл с опцией доступа по записи
    outputFile.write(testString.toLatin1());
    //Записываем строку, предварительно сконвертировав ее в формат
    //QByteArray
    outputFile.close(); //Закрываем файл
    qDebug()<<"File writed";
    //Выводим сообщение на экран о том то файл записан

    QFile *inputFile = new QFile;
    //Динамически создаем объект для работы с файлом
    inputFile->setFileName("information.txt");
    //Устанавливаем имя файла
```

```
    inputFile->open(QIODevice::ReadOnly);
    //Открываем файл для чтения
    QByteArray readString;
    //Создаем переменную для хранение считанных данных
    readString = inputFile->readAll();
    //Считываем весь файл
    inputFile->close(); //Закрываем файл
    delete inputFile; //Удаляем объект
    qDebug()<<"String from file: "<<QString(readString);
    //Выводи значение считанные данные, сконвертировав их в формат
    //QString

    return 0;
}
```

В результате выполнения программы будет создан файл information.txt. В котором будет содержаться тестовая строка. Файл будет находится в той же папке, что и исполняемый файл.

Часть III

Ввод-вывод с использованием потоков библиотеки Qt.

Рассмотрим проект, осуществляющий работу с файлами через потоки библиотеки Qt.

Файл Programm7_3.pro

```
QT += core
QT -= gui

CONFIG += c++11

TARGET = Programm7_3
CONFIG += console
CONFIG -= app_bundle

TEMPLATE = app

SOURCES += main.cpp
```

Файл main.cpp

```
#include <QFile>
#include <QTextStream>
#include <QDebug>

int main(int argc, char *argv[])
{
   qDebug()<<"Start programm";
   QFile *file = new QFile("file1.txt");
   file->open(QIODevice::WriteOnly);
   QTextStream *textStream = new QTextStream(file);
   qDebug()<<"Write information to the file";
   QString testString = "Information for writing to a file";
   *(textStream)<<testString;
   file->close();
   qDebug()<<"Information is writed";

   delete textStream;
   delete file;

   qDebug()<<"Read information";
   file - new QFile("file1.txt");
   file->open(QIODevice::ReadOnly);
   textStream = new QTextStream(file);
   QString readString;
   readString = textStream->readAll();
   qDebug()<<"Read the string: "<<readString;
   file->close();

   delete textStream;
   delete file;


   return 0;
}
```

Часть IV

Задание для самостоятельно работы

Файл содержит несколько строк, в каждой из которых записано единственное выражение вида a#b (без ошибок), где a, b - целочисленные величины, # - операция +, -, /, *. Написать программу, которая считывает

эти строки, выводит выражения на экран, а также, производит их вычисления. Реализовать два варианта программы:

1. Работу с файлами реализовать через потоки стандартного C++
2. Работу с файлами реализовать с использованием классов Qt

Содержание отчета:

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

Контрольные вопросы:

1. Что такое файл?
2. Что такое текстовый поток?
3. Что такое двоичный поток?
4. Как организовывается работа с файлами стандартными средствами C++?
5. Назначение класса QFile?
6. Каким образом реализуется работа с файлами через потоки Qt?

Лабораторная работа № 8
Функций обработки исключительных ситуаций

Цель работы: изучить принципы определения и обработки исключительных ситуаций.

Теоретические сведения

Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой исключением.

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: деление на 0, выход за границы массива или истощение свободной памяти. Такие исключения нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать. В C++ имеются встроенные средства для их возбуждения и обработки. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (или независимо разработанным) фрагментам программы обмениваться информацией об исключении.

Фундаментальная идея обработки исключительных ситуаций состоит в том, что функция, обнаружившая проблему, но не знающая как её решить, генерирует исключение в надежде, что вызвавшая её (непосредственно или косвенно) функция сможет решить возникшую проблему. Функция, которая может решать проблемы данного типа, указывает, что она перехватывает такие исключения.

Для реализации обработки исключений в C++ используйте выражения try, throw и catch.

Блок try {...} позволяет включить один или несколько операторов, которые могут создавать исключение. Выражение throw используется только в программных исключениях и означает, что исключительное условие произошло в блоке try. В качестве операнда выражения throw можно использовать объект любого типа. Обычно этот объект используется для передачи информации об ошибке.

Для обработки исключений, которые могут быть созданы, необходимо реализовать один или несколько блоков catch сразу после блока try. Каждый блок catch указывает тип исключения, которое он может обрабатывать.

Сразу за блоком try находится защищенный раздел кода. Выражение throw вызывает исключение, т.е. создает его. Блок кода после catch является обработчиком исключения. Он перехватывает исключение, вызываемое, если типы в выражениях throw и catch совместимы. Если оператор catch задает многоточие (...) вместо типа, блок catch обрабатывает все типы исключений.

Поскольку блоки catch обрабатываются в порядке программы для поиска подходящего типа, обработчик с многоточием должен быть последним обработчиком для соответствующего блока try. Как правило, блок catch(...) используется для ведения журнала ошибок и выполнения специальной очистки перед остановкой выполнения программы.

```
try { ... // защищенный раздел кода
  throw параметр;
}
catch (параметр) { // обработка исключения }
catch (...) { // обработка остальных исключений }
```

Ход работы

Часть I

Простейший пример генерации исключительной ситуации и ее обработки. Рассмотрим простейший пример работы с исключительными ситуация.

Файл Programm7_1.pro

```
T
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += main.cpp
A
Файл main.cpp
#include <iostream>
using namespace std;
int main() {
  try {
    cout << "Exception: "<<endl;
    throw 1;
```

```
    cout << "No exception!"<<endl;
  } catch (int a) {
    cout << "catch:" << a <<endl;

r
e
t
u
r
n
```

В данной программе часть кода заключена в try блок. Данн код сначала осуществляет вывод на экран сообщения "Exception: ", затем генерирует исключение с кодом 1, а после выводит сообщение "No exception!". После try-блока расположен обработчик исключительных ситуаций с целочисленным кодом catch (int a). При возникновении такой исключительной ситуации на экран будет выведено сообщение catch:" с числовым кодом данного исключения.

Часть II

Применение обработки исключительных ситуаций при использовании функций. В данном примере будет рассмотрено использование исключительных ситуаций при применении функций.

Файл Programm7_2.pro

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += main.cpp
```

Файл main.cpp

```
#include <iostream>
using namespace std;
float divAB (int a,int b) {
    float Output;
    if (b == 0) throw -1;
        else Output = a/b;
    return Output;
}
int main() {
  cout<<"Start programm"<<endl;
  int a = 10, b =0;
```

```
 try   {
     cout << divAB(a,b) << endl;
 }
 catch (int i) {
    cout<<"Error"<<i<<endl;
 }
 cout<<"End programm"<<endl;
 return 0;
}
```

Функция divAB() осуществляет деление входного параметра a на входной параметр b. В случае если значение b равно 0 генерируется исключение с кодом -1. В основной программе вызов данной функции взять в блок try. В случае генерации исключения с целочисленным кодом вызывается обработчик, который в свою очередь выводи код данной ошибки на экран. Обратите внимание, что генерация исключительной ситуации происходит внутри тела функции, а его обработка — в основной программе.

Часть III

Применение обработки исключительных ситуаций в классах.

Рассмотри пример реализации обработки исключительных ситуаций на примере класса стека.

Файл Programm7_3.pro

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += main.cpp \
   stack.cpp
HEADERS += \
   stack.h
```

Файл stack.h

```
#ifndef STACK_H
#define STACK_H
class stack {
```

```
private:
    int* body;
    int size;
    int top;
public:
    stack(int sz = 10);
    ~stack();
    int pop();
    void push(int x);
};
#endif // STACK_H
```

Файл stack.cpp

```
#include "stack.h"
stack::stack(int sz) {
    size = sz;
    top = 0;
    body = new int[size];
}
stack::~stack() {
    delete[] body;
}
int stack::pop() {
    top--;
    if (top == 0) throw -1;
    else return body[top];
}
void stack::push(int x) {
    if (top >= size) throw -2;
    else
    {
        body[top] = x;
        top++;
    }

}
```

Файл main.cpp

```
#include <iostream>
```

```
#include "stack.h"
using namespace std;
int main() {
   cout << "Start programm" << endl;
   stack Stack(10);

   try {
      cout << "Filling the stack" << endl;
      for (int i=0;i<10;i++)
         Stack.push(i);
      cout << "Extracting the stack" << endl;
      for (int i=0;i<12;i++)
         cout << Stack.pop() << endl;
   }
   catch (int i) {
      switch (i) {
      case -1:  cout << "Stack is empty" << endl;
         break;
      case -2:  cout << "Stack is full" << endl;
         break;
      }
   }
   return 0;
}
```

В данном примере реализован класс стека, у которого производится контроль при добавлении значения и извлечения. Если стек заполнен и производится попытка добавления нового значения, то генерируется исключительная ситуация с кодом -2, а в случае если производится извлечение значения из стека, но он пуст, то генерируется исключительная ситуация с кодом -1.

В основной программе вызовы методов push и pop взяты в try-блок, для перехвата исключительных ситуаций. В случае если возникла исключительная ситуация вызывается catch-обработчик и по коду определяется тип исключительной ситуации.

Часть IV

Задание для самостоятельно работы

Реализовать класс, который позволит обрабатывать одномерный массив целого типа. Размер массива должен передаваться в качестве входного параметра конструктора. Добавления нового элемента в массив должно осуществляться с помощью метода add(). Метод at(int i) должен возвращать значение элемента с номером i. Удаление элементов должно производиться с помощью метода remove (int i), где i номер элемента который должен быть удален. Метод length () должен возвращать текущее количество элементов в шаблоне. Так же класс должен содержать методы для обработки массива:

- определение максимального элемента;
- определение минимального элемента;
- подсчет среднего арифметического всех присутствующих элементов массива;
- сортировку элементов массива по возрастанию;
- сортировку массива по убыванию.

При удалении объекта – массив должен освобождать используемую память.

Так же продумать возможные исключительные ситуации и реализовать их обработку.

Содержание отчета:

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

Контрольные вопросы:

1. Что такое исключение?
2. Как производится обработка исключения?
3. Для чего предназначена инструкция try ?
4. Для чего предназначена инструкция catch?
5. Для чего предназначена инструкция throw?

Лабораторная работа № 9
Основы построения графического интерфейса. Базовый класс Qwidget. Система сигналов и слотов

Цель работы: изучить принципы построения графического интерфейса.

Теоретические сведения

Класс QWidget

Класс QWidget является фундаментальным для всех классов виджетов. Его интерфейс содержит 254 метода, 53 свойства и массу определений, необходимых каждому из виджетов, например, для изменения размеров, местоположения, обработки событий и др.

Сам класс QWidget унаследован от класса QObject, а значит, может использовать механизм сигналов/слотов и механизм объектной иерархии. Благодаря этому виджеты могут иметь потомков, которые отображаются внутри предка. Это очень важно, так как каждый виджет может служить контейнером для других виджетов, — то есть в Qt нет разделения между элементами управления и контейнерами. Виджеты в контейнерах могут выступать в роли контейнеров для других виджетов, и так до бесконечности. Например, диалоговое окно содержит кнопки Ok и Cancel (Отмена)— следовательно, оно является контейнером. Это удобно еще и потому, что если виджет-предок станет недоступным или невидимым, то виджеты-потомки автоматически примут его состояние.

Виджеты без предка называются виджетами верхнего уровня (top-level widgets) и имеют свое собственное окно. Все виджеты без исключения могут быть виджетами верхнего уровня. Позиция виджетов-потомков внутри виджета-предка может изменяться методом setGeometry () вручную или автоматически, с помощью специальных классов компоновки (layouts). Для отображения виджета на экране вызывается метод show (), а для скрытия — метод hide ().

Класс Qwidget и большинство унаследованных от него классов имеют конструктор с двумя параметрами:

QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)

Из определения видно, что не обязательно передавать параметры в конструктор, так как они равны нулю по умолчанию. А это значит, что если конструктор вызывается без аргументов, то созданный виджет станет виджетом верхнего уровня. Второй параметр Qt: :WindowFlags служит для задания свойств окна, и с его помощью можно управлять внешним видом окна и режимом отображения (чтобы окно не перекрывалось другими

окнами и т. д.). Чтобы изменить внешний вид окна, необходимо во втором параметре конструктора передать значения модификаторов, объединенные с типом окна побитовой операцией ИЛИ, обозначенной символом |. Аналогичного результата можно добиться вызовом метода setwindowFlags ().

Например:

```
wgt.setWindowFlags(Qt::Window | Qt::WindowTitleHint |
Qt::WindowStaysOnTopHint);
```

При помощи слот-метода setwindowTitle () устанавливается надпись заголовка окна. Но это имеет смысл только для виджетов верхнего уровня.

Например:

```
wgt.setwindowTitle("My Window");
```

Слот setEnabled () устанавливает виджет в доступное (enabled) или недоступное (disabled) состояние. Параметр true соответствует доступному, а false — недоступному состоянию. Чтобы узнать, в каком состоянии находится виджет, вызовите метод isEnabled (). На рисунке 9.1 приведен внешний вид виджетов верхнего уровня.

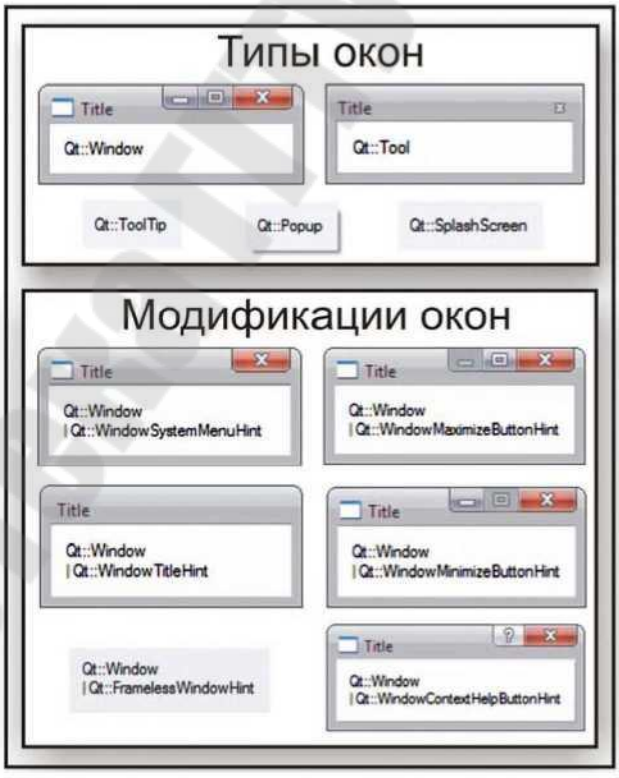


Рис. 9.1. Вид окон виджетов верхнего уровня

Механизм сигналов и слотов

Элементы графического интерфейса определенным образом реагируют на действия пользователя и посылают сообщения. Существует несколько вариантов такого решения.

Проблема расширения языка C++ решена в Qt с помощью специального препроцессора MOC (Meta Object Compiler, метаобъектный компилятор). Он анализирует классы на наличие в их определении специального макроса Q OBJECT И внедряет в отдельный файл всю необходимую дополнительную информацию. Это происходит автоматически, без непосредственного участия разработчика.

Макрос Q OBJECT должен располагаться сразу на следующей строке после ключевого слова class с определением имени класса. Очень важно помнить, что после макроса не должно стоять точки с запятой. Внедрять макрос в определение класса имеет смысл в тех случаях, когда созданный класс использует механизм сигналов и слотов или если ему необходима информация о свойствах.

Механизм сигналов и слотов полностью замещает старую модель функций обратного вызова, он очень гибок и полностью объектно-ориентирован. Сигналы и слоты — это краеугольный концепт программирования с использованием Qt, позволяющий соединить вместе несвязанные друг с другом объекты. Каждый унаследованный от QObject класс способен отправлять и получать сигналы.

В программировании с использованием Qt под понятием сигналы подразумеваются методы, которые в состоянии осуществлять пересылку сообщений.

Причиной для появления сигнала может быть сообщение об изменении состояния управляющего элемента — например, перемещение ползунка. На подобные изменения присоединенный объект, отслеживающий такие сигналы, может соответственно отреагировать, что, впрочем, и не обязательно. Это очень важный момент — он говорит о том, что соединяемые объекты могут быть абсолютно независимы и реализованы отдельно друг от друга. Такой подход позволяет объекту, отправляющему сигналы, не беспокоиться о том, что впоследствии будет происходить с этими сигналами. Объект, отправляющий сигналы, может даже и не догадываться, что их принимают и обрабатывают другие объекты. Благодаря такому разделению, можно разбить большой проект на компоненты, которые будут разрабатываться разными программистами по отдельности, а потом соединяться при помощи сигналов и слотов вместе.

Сигналы определяются в классе, как и обычные методы, только без реализации. С точки зрения программиста они являются лишь прототипами методов, содержащихся в заголовочном файле определения класса. Всю

дальнейшую заботу о реализации кода для этих методов берет на себя MOC. Методы сигналов не должны возвращать каких-либо значений, и поэтому перед именем метода всегда должен стоять возвращаемый параметр void.

Сигнал не обязательно соединять со слотом. Если соединения не произошло, то он просто не будет обрабатываться. Подобное разделение отправляющих и получающих объектов исключает возможность того, что один из подсоединенных слотов каким-то образом сможет помешать объекту, отправившему сигналы.

Библиотека предоставляет большое количество уже готовых сигналов для существующих элементов управления. В основном, для решения поставленных задач хватает этих сигналов, но иногда возникает необходимость реализации новых сигналов в своих классах.

Из сказанного становится ясно, что не имеет смысла определять сигналы как private, protected или public, поскольку они играют роль вызывающих методов.

Слоты (slots) — это методы, которые присоединяются к сигналам. По сути, они являются обычными методами. Самое большое их отличие состоит в возможности принимать сигналы. Как и обычные методы, они определяются в классе как public, private или protected. Если необходимо сделать так, чтобы слот мог соединяться только с сигналами сторонних объектов, но не вызываться как обычный метод со стороны, то тогда слот нужно объявить как protected или private. Во всех других случаях объявляйте их как public. В объявлениях перед каждой группой слотов должно стоять соответственно: private slots:, protected slots: или public slots:. Слоты могут быть и виртуальными.

Правда, есть небольшие ограничения, отличающие обычные методы от слотов. В слотах нельзя использовать параметры по умолчанию— например: slotMethod(int n = 8), или определять слоты как static. Классы библиотеки содержат целый ряд уже реализованных слотов.

Соединение объектов осуществляется при помощи статического метода connect (), который определен в классе QObject. В общем виде вызов метода connect () выглядит следующим образом:

```
QObject::connect(const QObject*      sender,
                 const char*       signal,
                 const QObject*receiver,
                 const char*       slot,
                 Qt::ConnectionType type =
                Qt:Autoconnection );
```

Ему передаются пять следующих параметров:

- sender — указатель на объект, отправляющий сигнал;
- signal — это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в

специальный макрос SIGNAL (method ( )) ;

- receiver — указатель на объект, который имеет слот для обработки сигнала;
- slot — слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальный макрос SLOT (method ( )) ;
- type — управляет режимом обработки. Имеется три возможных значения: Qt::DirectConnection — сигнал обрабатывается сразу вызовом соответствующего метода слота, Qt: :QueuedConnection — сигнал преобразуется в событие (см. главу 14) и ставится в общую очередь для обработки, Qt: Autoconnection — это автоматический режим, который действует следующим образом: если отсылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим Qt: : DirectConnection, В противном случае — режим Qt: :QueuedConnection. Этот режим (Qt: Autoconnection) определен в методе connection () по умолчанию. Вам вряд ли придется изменять режимы «вручную», но полезно знать, что такая возможность есть.

Существует альтернативный вариант метода connect (), преимущество которого заключается в том, что все ошибки соединения сигналов со слотами выявляются на этапе компиляции программы, а не при ее исполнении, как это происходит в классическом варианте метода connect (). Прототип альтернативного метода выглядит так:

```
QObject::connect(const QObject* sender,
                 const QMetaMethods signal,
                 const QObject* receiver, const
                 QMetaMethods slot,
                 Qt::ConnectionType type =
                Qt:Autoconnection );
```

Параметры этого метода полностью аналогичны предыдущему за исключением тех параметров, которые были объявлены в предыдущем методе как const char*. Вместо них используются указатели на методы сигналов и слотов классов напрямую. Благодаря именно этому, если вы вдруг ошибетесь с названием сигнала или слота, ваша ошибка будет выявлена сразу в процессе компиляции программы. К недостаткам альтернативного метода можно отнести то, что при каждом соединении нужно явно указывать имена классов для сигнала и слота и следить за совпадением их параметров.

Если есть возможность соединения объектов, то должна существовать и возможность их разъединения. В Qt при уничтожении объекта все связанные с ним соединения уничтожаются автоматически, но

в редких случаях может возникнуть необходимость в уничтожении этих соединений «вручную». Для этого существует статический метод disconnect (), параметры которого аналогичны параметрам статического метода connect (). В общем виде этот метод выглядит таким образом:

QObject::disconnect(sender, signal, receiver, slot);

Ход работы

Часть I

Создание графического интерфейса на основе Qwidget и его потомков.

Рассмотрим пример создания окна приложение на основе класса текстовой метки (QLabel). Проект состоит из двух файлов Programm9_1.pro и main.cpp.

Листинг программы:

Programm9_1.pro

```
QT += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm9_1
TEMPLATE = app

SOURCES += main.cpp
```

main.cpp

```
#include <QLabel>
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv); //Передаем классу QApplication парамтры запуска программы
    QLabel w; //Создам виджет верхнего уровня
    w.setText("Hello world"); //Устанавливаем текс
    w.show(); //Делаем виджет видимым
```

```
    return a.exec();
}
```

Рассмотри более подробно файл Programm9_1.pro. Опция QT += указывает какие модули должны быть подключены к проекту. При этом += означает то, что к уже подключенным модулям добавляются указанные в этой строке. В данном примере подключаются модули core, gui и widgets. Опция TARGET задает имя исполняемого файла, который получится в результате сборки проекта. В данном примере (TARGET = Programm9_1) мы получим исполняемый файл Programm9_1.exe. Параметр TEMPLATE указывает на то, что данный проект является приложением (в результате сборки должен получиться исполняемый файл), а не библиотека или модуль. Опция SOURCES задает список файлов с исходным кодом включенных в проект. В данном случае к проекту относится только один файл main.cpp, который располагается в той же папке, что и файл проекта.

Теперь рассмотрим исходный код программы. Библиотека QApplication содержит одноименный класс, который обеспечивает общение программы с графическим интерфейсом и операционной системой. Библиотека QLabel содержит виджет текстовой метки, который в данном случае является окном программы (виджетом верхнего уровня).

При выполнении программы будет создано графическое окно, сдержащее текст Hello world.

Часть II

Создание окна на основе собственного потомка класса QWidget

В данном примере будет объявлен класс-потомок от QWidget, и на основе его будет создано основное окно программы (виджет верхнего уровня). Рассмотри листинг программы.

Programm9_2.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm9_2
TEMPLATE = app


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

```
    HEADERS  += widget.h
```

widget.h

```
  #ifndef WIDGET_H
  #define WIDGET_H

  #include <QWidget>
  #include <QDebug>
  #include <QLabel>

  class Widget : public QWidget
  {
     Q_OBJECT

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

  private:
     QLabel *label1=NULL;
  };

      #endif // WIDGET_H
```

widget.cpp

```
  #include "widget.h"

  Widget::Widget(QWidget *parent)
     : QWidget(parent)
  {
     qDebug()<<"Create window";
     this->setWindowTitle("Main Window");

     label1 = new QLabel("Hello world",this);
  }

  Widget::~Widget()
  {
     qDebug()<<"Destroy window";
```

```
}
```

main.cpp

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

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

    return a.exec();
}
```

В файле проекта присутствуют аналогичные параметры, как и в предыдущем примере. За исключением опции HEADERS. Она задает список заголовочных файлов, относящихся к проекту.

Заголовочный файл widget.h содержит описание структуры класса, наследника от QWidget. На основе которого создается основное окно программы. Так же данный класс является контейнером, т.к. содержит в себе объект класса QLabel.

- В конструкторе класса основного окна производим следующие действия;
- Выводим текстовое сообщение в отладочную консоль программы (qDebug()<<"Create window";);
- Устанавливаем текст заголовка окна (this->setWindowTitle("Main Window"););
- Создаем виджет текстовой метки (label1 = new QLabel("Hello world",this););

В деструкторе класса содержится только вывод информации в отладочную консоль приложения (qDebug()<<"Destroy window";). В результате запуска приложения, будет создано окно, с заголовком Main Window и содержащее текстовую метку. При выполнении конструктора класса (при создании окна) в отладочную консоль будет выдано сообщение Create window, а при закрытии окна будет вызван деструктор и в отладочной консоли появится сообщение Destroy window.

Часть III

Система сигналов и слотов

Programm9_3.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm9_3
TEMPLATE = app


SOURCES += main.cpp\
        widget.cpp

HEADERS  += widget.h
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QDebug>

class Widget : public QWidget
{
    Q_OBJECT

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

private:
    QPushButton *button=NULL;
    quint16 pressCount=0;

private slots:
    void slotPressButton();
```

```
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   button = new QPushButton(tr("Press me"),this);
   connect(button,SIGNAL(pressed()),this,SLOT(slotPressButton()));
}

Widget::~Widget()
{

}

void Widget::slotPressButton()
{
   qDebug()<<"Button pressed "<<QString::number(pressCount);
   pressCount++;
}
```

main.cpp

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

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

   return a.exec();
}
```

В данном примере практически все аналогично предыдущему, кроме того, что в качестве дочернего виджета используется кнопка. Сигнал кнопки

подключается к слоту, расположенному в классе основного окна. При нажатии происходит инкремент переменной счетчика и в отладочную консоль приложения выводится надпись.

Часть IV

Задание для самостоятельно работы

Файл содержит несколько строк, в каждой из которых записано единственное выражение вида a#b (без ошибок), где a, b - целочисленные величины, # - операция +, -, /, *. Написать программу, которая при нажатии на кнопку считывает эти строки, выводит выражения в отладочную консоль приложения, а также, производит их вычисления. Работу с файлами реализовать с использованием классов Qt.

Содержание отчета:

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

Контрольные вопросы:

1. Что такое QWidget?
2. В чем особенность виджета верхнего уровня?
3. Каким образом происходит создание графического интерфейса?
4. Что такое сигнал?
5. Что такое слот?
6. Соединение сигналов и слотов?

Лабораторная работа № 10

Управление компоновкой элементов на форме

Цель работы: изучить принципы автоматической компоновки элементов на форме.

Теоретические сведения

Классы компоновки виджетов (Layouts) являются одной из сильных сторон Qt. По сути, это контейнеры, которые после изменения размеров окна автоматически приводят в соответствие размеры и координаты виджетов, находящихся в нем. Хотя они ничего не добавляют к функциональной части самой программы, тем не менее, они очень важны для внешнего вида окон приложения. Компоновка определяет расположение различных виджетов относительно друг друга.

Конечно, можно вручную размещать виджеты в окнах приложения, но это существенно усложняет разработку. Ведь тогда, чтобы заново упорядочить элементы, нужно будет отлавливать и обрабатывать изменение размеров окна приложения. Подобные ситуации хорошо известны программистам на языке Visual Basic, которые вынуждены писать для этого сложные методы.

Еще один из недостатков размещения вручную состоит в том, что если приложение поддерживает несколько языков, то, поскольку слова в разных языках имеют разную длину, необходим механизм, который мог бы в процессе работы программы динамически поправлять и изменять размеры и координаты виджетов, — иначе части текста на другом языке могут оказаться «отрезанными». Классы компоновки библиотеки Qt выполняют эту непростую работу за вас. Более того, классы компоновки могут инвертировать направление размещения элементов, что может быть полезно для пишущих справа налево, — например, в Израиле или в странах арабского Востока. Иерархия классов компановки приведена на рисунке 10.1.

От класса QLayout унаследованы классы QGridLayout и QBoxLayout. Класс QGridLayout управляет табличным размещением, а от QBoxLayout унаследованы два класса QHBoxLayout и QVBoxLayout для горизонтального и вертикального размещения.

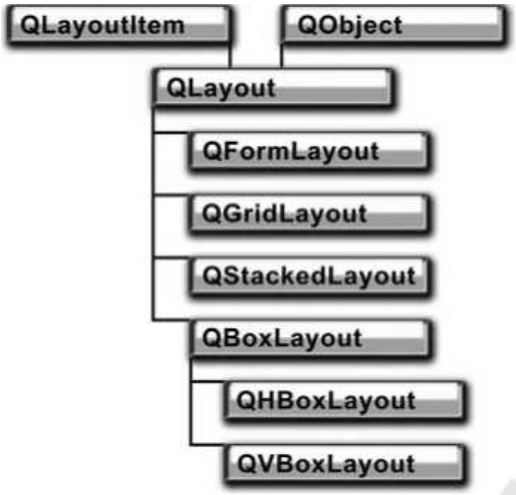


Рис. 10.1. Иерархия классов менеджеров компоновки

Класс QBoxLayout

Объект класса QBoxLayout может управлять как горизонтальным, так и вертикальным размещением. Для того чтобы задать способ размещения, первым параметром конструктора должно быть одно из следующих значений:

- LeftToRight — горизонтальное размещение, заполнение осуществляется слева направо;
- RightToLeft — горизонтальное размещение, заполнение выполняется справа налево;
- TopToBottom — вертикальное размещение, заполнение осуществляется сверху вниз;
- BottomToTop — вертикальное размещение, заполнение выполняется снизу вверх.

Этот класс расширяет класс QLayout методами вставки на заданную позицию:

- Виджета — insertWidget();
- встроенной компоновки — insertLayout();
- расстояния между виджетами — insertspacing();
- фактора растяжения — insertStretch().

К компоновке при помощи метода addSpacing() можно добавить заданное расстояние между двумя виджетами.

Класс QBoxLayout определяет свой собственный метод addwidget() для добавления виджетов в компоновку с возможностью указания, в

дополнительном параметре, фактора растяжения (по умолчанию этот параметр равен нулю).

Горизонтальное размещение QHBoxLayout. Объекты класса QHBoxLayout упорядочивают все виджеты только в горизонтальном

порядке — слева направо. Его применение аналогично использованию класса QBoxLayout**,** но передавать в конструктор дополнительный параметр, задающий горизонтальный порядок размещения, не нужно. Пример горизонтальной компановки виджетов приведен на рисунке 10.2.

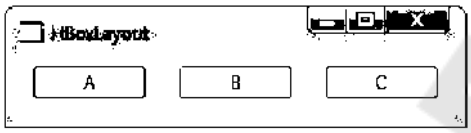


Рис.10.2. Размещение кнопок по горизонтали

Вертикальное размещение QVBoxLayout

Компоновка QVBoxLayout унаследована от QBoxLayout и упорядочивает все виджеты только по вертикали — сверху вниз. В остальном она ничем не отличается от классов QBoxLayout и QHBoxLayout. Пример вертикальной компановки приведен на рисунке 10.3.

Рис.10.3. Размещение кнопок по вертикали

Факторы растяжения можно самостоятельно добавлять в компоновки, для чего существует метод addstretcho. В этом случае фактор растяжения образно можно сравнить с пружиной, которая находится между виджетами и может иметь различную упругость в соответствии с задаваемым параметром. На Рисунке 4 показан пример добавления фактора растяжения между двумя виджетами для расположения виджетов по краям окна.

Рис.10.4. Фактор растяжения

Вложенные размещения

Размещая одну компоновку внутри другой, можно создавать размещения практически любой сложности. Для организации вложенных размещений существует метод addLayout, в который вторым параметром передается фактор растяжения для добавляемой компоновки.

На Рисуноке 10.5 показан пример вложенного размещения двух менеджеров компоновки. В компоновку QVBoxLayout помещается компоновка QHBoxLayout. Пример вложенных размещений изображен на рисунке 10.5.

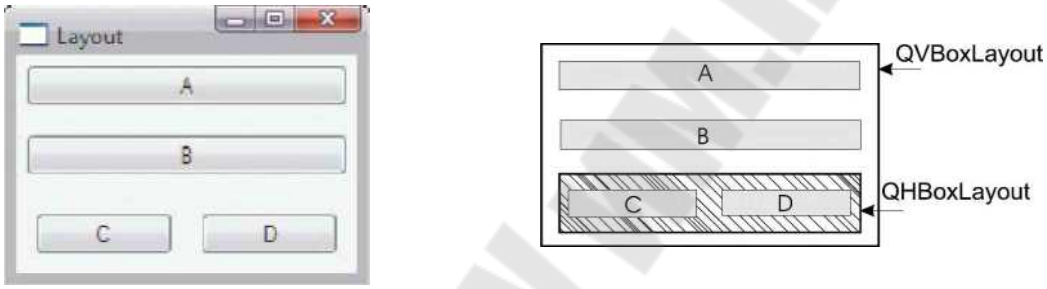


Рис.10.5. Вложенное размещение

Табличное размещение QGridLayout

Для табличного размещения используется класс QGridLayout, с помощью которого можно быстро создавать сложные по структуре размещения. Таблица состоит из ячеек, позиции которых задаются строками и столбцами.

Добавить виджет в таблицу можно с помощью метода addwidget (), передав ему позицию ячейки, в которую помещается виджет. Иногда необходимо, чтобы виджет занимал сразу несколько позиций, чего можно достичь тем же методом addwidget(), указав в дополнительных параметрах количество строк и столбцов, которые будет занимать виджет. В последнем параметре можно задать выравнивание (таблица 10.1), например, по центру:

playout->addWidget(widget, 17, 1, Qt::AlignCenter);

Таблица 10.1. Значения флагов Alignment Flag пространства имен Qt

Константа	**Значение**	**Описание**
AlignLeft	0x0001	Расположение текста слева
AlignRight	0x0002	Расположение текста справа
AlignHCenter	0x0004	Центровка текста по горизонтали

AlignJustify	0x0008	Растягивание текста по всей
AlignTop	0x0010	Расположение текста вверху
AlignBottom	0x0020	Расположение текста внизу
AlignVCenter	0x0040	Центровка текста по вертикали

Фактор растяжения устанавливается методами setRowStretch () и setColumnStretch (), но не для каждого виджета в отдельности, а для строки или столбца. Расстояние между виджетами также устанавливается для столбцов или строк методом setspacing ().

Пример, показанный на рисунке 10.6, размещает четыре кнопки: A, B, C и D в таблице размером 2 на 2 ячейки.

Рис. 10.6. Размещение кнопок в табличном пордке

Порядок следования табулятора

Пользователь может взаимодействовать с виджетами при помощи мыши и клавиатуры. В последнем случае для выбора нужного виджета используется клавиша табуляции — <TaЬ>, при нажатии которой происходит переход фокуса согласно установленному порядку от одного виджета к другому. Иногда возникает необходимость в изменении этого порядка, который по умолчанию соответствует очередности установки дочерних виджетов в виджете предка. На рис. 10.7 цифрами изображен порядок смены фокуса с помощью табулятора. При появлении диалогового окна с тремя кнопками фокус будет установлен на кнопке C и после нажатия на клавишу табуляции он перейдет на кнопку B, а затем — на кнопку A.

Рис.10.7. а. Порядок смены фокуса, б. Измененный порядок смены фокуса

Изменить порядок смены фокуса можно с помощью статического метода QWidget:: setTabOrder (), получающего в качестве параметров два

указателя на виджеты. Следующие вызовы изменят порядок следования табулятора, на более логичный порядок:

```
QWidget::setTabOrder(A, B);
QWidget::setTabOrder(B, C);
```

Ход работы

Часть I

Рассмотрим несколько примеров программ, которые демонстрируют основные возможности автоматической компоновки элементов на форме.

Компоновка с помощью QBoxLayout.

Данный пример реализует компоновку 6 кнопок на основной форме, в два ряда. Первый ряд располагает элементы слева на право, а второй с права на лево.

Programm10_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm10_1
TEMPLATE = app
SOURCES += main.cpp\
                widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QBoxLayout>
#include <QPushButton>

class Widget : public QWidget
{
   Q_OBJECT

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

private:
   QBoxLayout *mainLayout=NULL;

   QBoxLayout *layout1=NULL;
   QPushButton *buttonA=NULL;
   QPushButton *buttonB=NULL;
   QPushButton *buttonC=NULL;

   QBoxLayout *layout2=NULL;
   QPushButton *buttonD=NULL;
   QPushButton *buttonE=NULL;
   QPushButton *buttonF=NULL;

};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

```
    mainLayout = new QBoxLayout(QBoxLayout::TopToBottom,this);

    layout1 = new QBoxLayout(QBoxLayout::LeftToRight);
    mainLayout->addLayout(layout1);

    buttonA = new QPushButton(tr("Button A"),this);
    layout1->addWidget(buttonA);

    buttonB = new QPushButton(tr("Button B"),this);
    layout1->addWidget(buttonB);

    buttonC = new QPushButton(tr("Button C"),this);
    layout1->addWidget(buttonC);

    layout2 = new QBoxLayout(QBoxLayout::RightToLeft);
   mainLayout->addLayout(layout2);

    buttonD = new QPushButton(tr("Button D"),this);
    layout2->addWidget(buttonD);

    buttonE = new QPushButton(tr("Button E"),this);
    layout2->addWidget(buttonE);

    buttonF = new QPushButton(tr("Button F"),this);
    layout2->addWidget(buttonF);
}

Widget::~Widget()
{
    delete buttonA;
    delete buttonB;
    delete buttonC;
    delete buttonD;
    delete buttonE;
    delete buttonF;

    delete layout1;
    delete layout2;
}
```

Файлы Programm10_1.pro и main.cpp отдельно рассматривать не стоит, они типичны и хорошо описаны в предыдущих лабораторных.

Рассмотри класс основного окна программы. Вначале файла widget.h производим подключение необходимых библиотек, для использования следующих классов:

- QWidget – класс-родитель для класса основного окна;
- QBoxLayout - класс автоматической компановки элементов на форме;
- QPushButton – класс кнопки.

Затем в структуре класса основного окна объявляются необходимые указатели: 3 указателя на менеджеры компоновки и указатели на 6 кнопок. Указатель mainLayout будет содержать объект основного менеджера компоновки, на нем будут находится оставшиеся классы компоновки. Объект layout1 будет отвечать за компоновку первой строки виджетов, а layout2 соответственно за компоновку виджетов второй строки.

Теперь рассмотри конструктор класса основного окна программы. Сначала создается основной объект копоновки:

```
mainLayout = new QBoxLayout(QBoxLayout::TopToBottom,this);
```

Элементы будут на нем располагаться вертикально и порядок заполнения будет сверху в низ (за это отвечает опция QBoxLayout::TopToBottom, переданная в конструктор). Нам не следует явно задавать что данный объект будет основным для виджета окна, т.к. в параметрах указателя основное окно программы передано в качестве родителя.

Затем создается объект компоновки первой строки виджетов и укладывается на основной менеджер компоновки:

```
layout1 = new QBoxLayout(QBoxLayout::LeftToRight);
mainLayout->addLayout(layout1);
```

Виджеты, которые будут уложены на него, будут располагаться горизонтально, с порядком заполнения с лева на право.

Аналогичным образом поступаем и со вторым объектом компоновки:

```
Layout2 = new QBoxLayout(QBoxLayout::RightToLeft);
mainLayout->addLayout(layout2);
```

Только, в отличии от первого, порядок заполнения элементами у него установлен с право на лево. Затем создаем кнопки и укладываем их на соответствующие объекты компоновки:

```
buttonA = new QPushButton(tr("Button A"),this);
layout1->addWidget(buttonA);
.
.
.
buttonF = new QPushButton(tr("Button F"),this);
layout2->addWidget(buttonF);
```

В результате выполнения данной программы мы получим форму, на которой виджеты (кнопки) расположены в два ряда. В первом ряду порядок кнопок будет с лева на право, а во втором наоборот.

Часть II

Компоновка с помощью QVBoxLayout и QHBoxLayout.

Рассмотрев более общий класс компоновки виджетов, рассмотрим специализированные QVBoxLayout и QHBoxLayout. Как уже было сказано в теоритической части первый класс компоновки осуществляет вертикальное расположение виджетов, а второй горизонтальное. Пример применения данных классов для размещения виджетов на форме.

Programm10_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm10_2
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

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

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

class Widget : public QWidget
{
   Q_OBJECT

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

private:
   QVBoxLayout *mainLayout=NULL;
   QHBoxLayout *layout1=NULL;
   QLabel *labelA=NULL;
   QLineEdit *editA=NULL;
   QHBoxLayout *layout2=NULL;
   QPushButton *buttonSetA=NULL;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

   layout1 = new QHBoxLayout;
   mainLayout->addLayout(layout1);

   labelA = new QLabel(tr("Enter A"),this);
   layout1->addWidget(labelA);
   editA = new QLineEdit(this);
   layout1->addWidget(editA);
```

```
    layout2 = new QHBoxLayout;
    mainLayout->addLayout(layout2);

    layout2->addStretch();
    buttonSetA = new QPushButton(tr("Set A"),this);
    layout2->addWidget(buttonSetA);
}

Widget::~Widget()
{

}
```

Как обычно, в заголовочном файле класс основного окна произведем подключение необходимых библиотек для данного проекта:

- QWidget – класс-родитель класса основного окна;
- QHBoxLayout – горизонтальный компоновщик виджетов;
- QVBoxLayout – вертикальный компоновщик виджетов;
- QLabel – класс текстовой метки;
- QLineEdit – класс текстового поля;
- QPushButton – класс кнопки.

В структуре класса основного окна произведем объявление указателей на необходимые объекты.

В конструкторе класса основного окна произведем создание объектов и расположение их на форме.

Сначала создаем основной компоновщик виджетов:

```
mainLayout = new QVBoxLayout(this);
```

Затем создаем компоновщики первой и второй строки и располагаем их на основном компоновщике:

```
layout1 = new QHBoxLayout;
mainLayout->addLayout(layout1);
layout2 = new QHBoxLayout;
mainLayout->addLayout(layout2);
```

Создаем элементы (виджеты) и располагаем их на нужных компоновщиках. В первой строке расположим текстовую метку и текстовое поле. Их создание типичное, и не требует дополнительных пояснений. Во второй строке расположим кнопку. Однако необходимо сделать так, чтобы при изменении ширины формы кнопка не меняла своего размера.

```
layout2->addStretch();
buttonSetA = new QPushButton(tr("Set A"),this);
```

```
layout2->addWidget(buttonSetA);
```

Для этого перед расположение формы создадим фактор растяжения (layout2->addStretch()), а только после этого создадим объект кнопки и расположим его на компоновщике.

Часть III

Компоновка с помощью QGridLayout.

Данный метод компоновки элементов на форме позволяет выравнивать положение виджетов по сетке. Рассмотрим пример.

Programm10_3.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm10_3
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
```

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

class Widget : public QWidget
{
   Q_OBJECT

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

private:
   QGridLayout *mainLayout=NULL;
   QLabel *labelA=NULL;
   QLineEdit *editA=NULL;
   QLabel *labelB=NULL;
   QLineEdit *editB=NULL;
   QPushButton *buttonSet=NULL;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

   labelA = new QLabel(tr("Enter A"),this);
   mainLayout->addWidget(labelA,0,0,1,1);
   editA = new QLineEdit(this);
   mainLayout->addWidget(editA,0,1,1,2);

   labelB = new QLabel(tr("Enter B"),this);
   mainLayout->addWidget(labelB,1,0,1,1);
   editB = new QLineEdit(this);
   mainLayout->addWidget(editB,1,1,1,2);
```

```
    buttonSet = new QPushButton(tr("Set"),this);
    mainLayout->addWidget(buttonSet,2,2,1,1);
}

Widget::~Widget()
{

}
```

Принцип использования QgridLayout аналогичен предыдущим, однако при расположении виджета необходимо указывать дополнительные параметры.

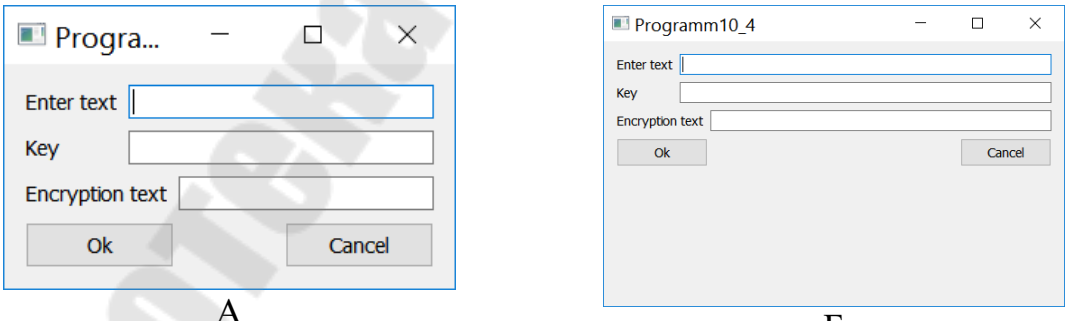
```
mainLayout->addWidget(labelA,0,0,1,1);
```

Первый параметр, как и у всех компоновщиков, является указателем на виджет. Затем обязательно указывается строка и столбец, в котором необходимо расположить элемент, а также количество ячеек, которые должен занимать виджет по горизонтали и вертикали. В данном случаем виджет labelA располагается в 0 строке, 0 столбце и занимает по одной ячейке в каждом направлении.

Часть IV

Задание для самостоятельно работы

Реализовать программу, которая будет создавать форму, аналогичную той, что изображена на рисунке 10.8.



А Б

Рис.10.8. Вид формы окна для задания на самостоятельную работу. А) Исходное состояние формы. Б) Вид формы при изменении размера

Содержание отчета

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

Контрольные вопросы для самопроверки

1. Что такое классы управления автоматического размещения элементов?
2. Особенности применения QBoxLayout?
3. Особенности применения QHBoxLayout?
4. Особенности применения QVBoxLayout?
5. Особенности применения QGridLayout?
6. Что такое фактор растяжения?
7. Как задается порядок следования табулятора?

Лабораторная работа № 11
Работа с элементами отображения

Цель работы: изучить основные элементы отображения и методы их использования при построении формы.

Теоретические сведения

Элементы отображения не принимают активного участия в действиях пользователя и используются для информирования его о происходящем. Эта информация может носить как текстовый, так и графический характер (картинки, графика).

Надписи (Qlabel)

Виджет надписи служит для показа состояния приложения или поясняющего текста и представляет собой текстовое поле, текст которого не подлежит изменению со стороны пользователя. Информация, отображаемая этим виджетом, может изменяться только самим приложением. Таким образом, приложение может сообщить пользователю о своем изменившемся состоянии, но пользователь не может изменить эту информацию в самом виджете. Класс виджета надписи QLabel определен в заголовочном файле QLabel:

#include <QLabel>

Виджет надписи унаследован от класса QFrame и может иметь рамку. Отображаемая им информация может быть текстового, графического или анимационного характера, для передачи ее используются слоты setText (), setPixmap () и setMovie ().

Расположением текста можно управлять при помощи метода setAlignment (). Метод использует большое количество флагов, некоторые из них приведены в табл. 11.1. Обратите внимание, что значения не пересекаются, и это позволяет комбинировать их друг с другом с помощью логической операции | (ИЛИ). Наглядным примером служит значение AlignCenter, составленное из значений AlignVCenter И AlignHCenter.

Таблица 11.1. Значения флагов Alignment Flag пространства имен Qt

Константа	Значение	Описание
AlignLeft	0x0001	Расположение текста слева
AlignRight	0x0002	Расположение текста справа
AlignHCenter	0x0004	Центровка текста по
AlignJustify	0x0008	Растягивание текста по всей
AlignTop	0x0010	Расположение текста вверху
AlignBottom	0x0020	Расположение текста внизу
AlignVCenter	0x0040	Центровка текста по
AlignCenter	AlignVCenter |	Центровка текста по

Например:

```
QLabel *labelText = new QLabel(tr("Count"),this);
labelText->setAlignment(Qt::AlignVCenter | Qt::AlignHCenter);
```

Индикатор процесса

Индикатор процесса (progress bar) — это виджет, демонстрирующий процесс выполнения операции и заполняющийся слева направо. Полное заполнение индикатора информирует о завершении операции. Этот виджет необходим в том случае, когда программа выполняет продолжительные действия, — виджет дает пользователю понять, что программа не зависла, а находится в работе. Он также показывает, сколько уже проделано и сколько еще предстоит сделать. Класс QProgressBar виджета индикатора процесса определен в заголовочном файле QProgressBar. Обычно индикаторы процесса располагаются в горизонтальном положении, но это можно изменить, передав в слот setOrientation () значение Qt::vertical, — после этого он будет расположен вертикально.

```
QProgressBar  *progres = new QProgressBar();
progres->setMinimum(0); //Задаем минимальное значени
progres->setMaximum(100); //Задаем максимальное значение
progres->setOrientation(Qt::Vertical); //Устанавливаем
//вертикальную ориентацию
```

Электронный индикатор

Класс QLCDNumber виджета электронного индикатора определен в заголовочном файле QLCDNumber. По внешнему виду этот виджет представляет собой набор сегментных указателей — как, например, на электронных часах. С помощью виджета электронного индикатора отображаются целые числа. Допускается использование точки, которую можно отображать между позициями сегментов или как отдельный символ, вызывая метод setSmallDecimalPoint () и передавая в него true или false соответственно. Количество отображаемых сегментов можно задать в

конструкторе или с помощью метода setNumDigits (). В том случае, когда для отображения числа не хватает сегментов индикатора, отсылается сигнал overflow ().

По умолчанию стиль электронного индикатора соответствует стилю Outline, но его можно изменить, передав методу setSegmentstyle() одно из следующих значений: QLCDNumber:: Outline, QLCDNumber: : Filled ИЛИ QLCDNumber: : Flat. В таблице 11.1 показан внешний ВИД ВИДЖета для каждого из перечисленных стилей.

Ход работы

Часть I

Вывод текстовой информации в Qlabel.

Programm11_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm11_1
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QHBoxLayout>
#include <QLabel>
#include <QPushButton>

class Widget : public QWidget
{
   Q_OBJECT

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

private:
   quint16 Count=0;
   QHBoxLayout *mainLayout=NULL;
   QLabel *labelText=NULL;
   QPushButton *buttonIncr=NULL;

private slots:
   void slotPushButton();


};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QHBoxLayout(this);
   labelText = new QLabel(tr("Count"),this);
   labelText->setAlignment(Qt::AlignVCenter | Qt::AlignHCenter);
```

```
    mainLayout->addWidget(labelText);
    buttonIncr = new QPushButton(tr("Increment"),this);
    connect(buttonIncr,SIGNAL(pressed()),this,SLOT(slotPushButton()));
    mainLayout->addWidget(buttonIncr);
}

Widget::~Widget()
{

}

void Widget::slotPushButton()
{
    Count++;
    labelText->setText(tr("Count")+"="+QString::number(Count));
}
```

Часть II

Использование HTML в QLabel.

Programm11_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm11_2
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

```
    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>

class Widget : public QWidget
{
    Q_OBJECT

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

private:
    QVBoxLayout *mainLayout=NULL;
    QLabel *labelHTML=NULL;
    QLabel *labelLink=NULL;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    labelHTML = new QLabel("<strong>Hello</strong> "
                           "<font color=red>World !",this);
    mainLayout->addWidget(labelHTML);
```

```
    labelLink = new QLabel("<CENTER><A
HREF=\"http://www.google.ru\">www.google.ru</A></CENTER>",this)
;
    labelLink->setOpenExternalLinks(true);
    mainLayout->addWidget(labelLink);
}

Widget::~Widget()
{

}
```

Использование QProgressBar.

Programm11_3.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm11_3
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QPushButton>
#include <QProgressBar>

class Widget : public QWidget
{
   Q_OBJECT

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

private:
   quint8 Position=0;
   QHBoxLayout *mainLayout=NULL;
   QVBoxLayout *layout1=NULL;
   QHBoxLayout *layout2=NULL;
   QProgressBar *progres1=NULL;
   QProgressBar *progres2=NULL;
   QPushButton *buttonIncr=NULL;
   QPushButton *buttonReset=NULL;

private slots:
   void slotPushIncr();
   void slotPushReset();


};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
```

```
    : QWidget(parent)
{
    mainLayout = new QHBoxLayout(this);

    layout1 = new QVBoxLayout;
    mainLayout->addLayout(layout1);

    progres1 = new QProgressBar(this);
    progres1->setMinimum(0);
    progres1->setMaximum(100);
    layout1->addWidget(progres1);

    layout2 = new QHBoxLayout;
    layout1->addLayout(layout2);

    buttonIncr = new QPushButton(tr("Increment"),this);
    connect(buttonIncr,SIGNAL(pressed()),this,SLOT(slotPushIncr()));
    layout2->addWidget(buttonIncr);

    buttonReset = new QPushButton(tr("Reset"),this);
    connect(buttonReset,SIGNAL(pressed()),this,SLOT(slotPushReset()));
    layout2->addWidget(buttonReset);

    progres2 = new QProgressBar(this);
    progres2->setMinimum(0);
    progres2->setMaximum(100);
    progres2->setOrientation(Qt::Vertical);
    mainLayout->addWidget(progres2);
}

Widget::~Widget()
{

}

void Widget::slotPushIncr()
{
    Position++;
    progres1->setValue(Position);
    progres2->setValue(Position);
}
```

```
void Widget::slotPushReset()
{
   Position=0;
   progres1->setValue(0);
   progres2->setValue(0);
}
```

Часть III

Применение QLCDNumber

Programm11_4.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm11_4
TEMPLATE = app
SOURCES += main.cpp\
        mainwindow.cpp
HEADERS  += mainwindow.h
```

main.cpp

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

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

   return a.exec();
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWidget>
#include <QVBoxLayout>
#include <QLCDNumber>
```

```
#include <QPushButton>

class MainWindow : public QWidget
{
   Q_OBJECT

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

private:
   quint16 Count=0;
   QVBoxLayout *mainLayout=NULL;
   QLCDNumber *lcdNumber=NULL;
   QPushButton *buttonIncr=NULL;

private slots:
   void slotPushIncr();
};

      #endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"

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

   lcdNumber = new QLCDNumber(this);
   lcdNumber->display(0);
   lcdNumber->setFixedHeight(30);
   mainLayout->addWidget(lcdNumber);

   buttonIncr = new QPushButton(tr("Increment"),this);
   connect(buttonIncr,SIGNAL(pressed()),this,SLOT(slotPushIncr()));
   mainLayout->addWidget(buttonIncr);
}

MainWindow::~MainWindow()
```

```
{

}

void MainWindow::slotPushIncr()
{
   Count++;
   lcdNumber->display(Count);
}
```

Часть IV

Задание для самостоятельного выполнения

Программа «Таблица умножения»

Реализовать программу, которая по нажатию кнопки производит вывод в Qlabel таблицы умножения.

Программа «Визитная карточка»

Необходимо реализовать программу, которая будет выводить о вас информацию. Фотографию, ФИО, группу и дату рождения вывести в QLabel с использованием html. Ваш возраст (количество полных лет) вывести в QLCDNumber. А с помощью QProgressBar отобразить в процентном отношении количество дней, оставшихся до вашего следующего дня рождения, от текущей даты.

Содержание отчета

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

Контрольные вопросы для самопроверки

1. Что такое класс QLabel и какие основные возможности он имеет?
2. Основные возможности QProgressBar и область его применения?
3. Описание, основные возможности и область применения QLCDNumber?
4. Способы вывода форматированного текста в QLabel?

Лабораторная работа № 12
Элементы ввода, управления и выбора.
Часть 1

Цель работы: изучить основные элементы ввода и методы их использования.

Теоретические сведения

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

Однострочное текстовое поле (QLineEdit)

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

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

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

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

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

Т.к. QLineEdit является наследником классов QObject и QWidget, он имеет все те же слоты, что и классы-родители. Однако есть и собственные. Собственные слоты класса представлены в таблице 12.1

Таблица 12.1. Собственные открытые слоты

Слот	**Описнаие**
void QLineEdit::clear()	Очистить текстовое поле

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

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

Таблица 12.2. Собственные сигналы

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

Собственные открытые методы представлены в таблице 12.3.

Таблица 12.3. Открытые собственные методы

Метод	**Описание**
QLineEdit::QLineEdit(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на родителя, по умолчанию равен нулю
QLineEdit::QLineEdit(const QString &contents, QWidget *parent = Q_NULLPTR)	Так же конструктор, но в отличие от предыдущего, имеет параметр contents, которым можно задать текст в

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

Счетчик QSpinBox

Виджет QSpinBox предоставляет пользователю доступ к ограниченному диапазону чисел.

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

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

```
spinEdit->setRange(0,10); //Устанавливаем интервал от 0 до 10
```

spinEdit-> setValue (5); //Устанавливаем значение 5

int Value = spinEdit-> value (); //Забираем значение и заносим его в переменную

При изменении значений посылаются сразу два сигнала valueChanged (): один с параметром типа int, а другой — const QString&. Можно изменить способ отображения с помощью методов setPrefix() и setSuffix(). Например, вызов следующих методов приведет к тому, что число будет отображаться в скобках:

spinEdit ->setPrefix("(") ;

spinEdit ->setSuffix(")") ;

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

Таблица 12.4. Собственные сигналы QSpinBox.

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

Таблица 12.5. Собственные слоты QSpinBox.

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

Таблица 12.6. Собственные методы QSpinBox

Метод	**Описание**
QSpinBox::QSpinBox(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на объект-родитель. По умолчанию равен 0
QString QSpinBox::cleanText() const	Этот метод возвращает текст виджета, без суффиксов и префиксов

int QSpinBox::displayIntegerBase() const	Метод возвращает систему счисления, в которой отображаются значения(например 10-ая, 2-ая и т.д.)
int QSpinBox::maximum() const	Возвращает текущее максимальное значение
int QSpinBox::minimum() const	Возвращает текущее минимальное значение
QString QSpinBox::prefix() const	Возвращает текущий префикс
void QSpinBox::setDisplayIntegerBase(int base)	Устанавливает систему счисления для отображения чисел
void QSpinBox::setMaximum(int max)	Устанавливает максимум
void QSpinBox::setMinimum(int min)	Устанавливает минимум
void QSpinBox::setPrefix(const QString &prefix)	Устанавливает префикс (отображается перед значением). prefix - текстовый префикс для отображения в виджете
void QSpinBox::setRange(int minimum, int maximum)	Устанавливает рабочий интервал виджета (максимальное и минимальное значение). minimum – минимальное значение, maximum – максимальное значение
void QSpinBox::setSingleStep(int val)	Устанавливает размер единичного шага значений
void QSpinBox::setSuffix(const QString &suffix)	Устанавливает текстовый суффикс (отображается после значения)
int QSpinBox::singleStep() const	Возвращает единичный шаг
QString QSpinBox::suffix() const	Возвращает суффикс
int QSpinBox::value() const	Возвращает текущее значение

Элемент ввода даты и времени

Существует три виджета ввода даты и времени:

- QDateTimeEdit;
- QDateEdit;
- QTimeEdit.

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

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

И, соответственно, наследует их сигналы, слоты и методы. Более подробно рассмотрим собственные открытые сигналы, соты и методы класса QDateTimeEdit. Сигналы, слоты и методы класса QDateTimeEdit представлены в таблица 12.7, 12.8 и 12.9.

Таблица 12.7. Собственные сигналы QDateTimeEdit.

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

Таблица 12.8. Собственные слоты QDateTimeEdit.

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

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

Таблица 12.9. Собственные открытые методы QDateTimeEdit.

Метод	**Описание**
QDateTimeEdit::QDateTimeEdit(QWidget *parent = Q_NULLPTR)	Конструктор класса. parent – указатель на родителя, по умолчанию равен 0.

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

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

Таблица 12.10. Основные открытые методы QDateTime.

Метод	**Описание**
QDate QDateTime::date() const	Возвращает установленную дату в формате QDate

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

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

Таблица 12.11. Основные открытые методы QDate.

Метод	**Описание**
QDate::QDate(int y, int m, int d)	Конструктор класса. в качестве входных параметров задается дата: y – год, m – месяц, d - день
int QDate::day() const	Возвращает день установленной даты
int QDate::dayOfWeek() const	Возвращает день недели установленной даты
int QDate::dayOfYear() const	Возвращает номер установленного дня в году
int QDate::daysInMonth() const	Возвращает количество дней в установленном месяце
int QDate::daysInYear() const	Возвращает количество дней в установленном году
void QDate::getDate(int *year, int *month, int *day) const	Возвращает дату в виде чисел. В качестве параметров передаются указатели на целочисленные

	переменные, в которые будет помещена дата
bool QDate::setDate(int year, int month, int day)	Устанавливает дату из целых чисел
QString QDate::toString(const QString &format) const	Преобразовывает текущую дату в текстовую строку, в соответствии с форматом
int QDate::year() const	Возвращает установленный год

Основные открытые методы класса QTime представлены в таблице 12.12.

Таблица 12.12. Основные открытые методы QTime.

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

Редактор текста

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

Класс QTextEdit содержит следующие методы:

- setReadOnly () — устанавливает или снимает режим блокировки изменения текста;

- text () — возвращает текущий текст.

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

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

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

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

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

Для помещения обычного текста в область виджета можно воспользоваться методом setPlainText() или слотом insertPlainText(). При

помощи слота append() осуществляется добавление текста, причем добавленный текст не вносится в список операций, действие которых можно вернуть с помощью слота undo(), что делает этот слот быстрым и не требующим дополнительных затрат памяти. Метод find() может быть использован для поиска и выделения заданной строки в тексте.

Ход работы

Часть I

Использование QLineEdit.

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

Programm12_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_1
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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


class Widget : public QWidget
{
   Q_OBJECT

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

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



private slots:
   void slotTextChanged(QString text);
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
```

```
{
    mainLayout = new QGridLayout(this);

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

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

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

Widget::~Widget()
{

}


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

Часть II

Использование счетчика QSpinBox.

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

Programm12_2.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_2
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

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

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

```
    return a.exec();
}
```

widget.cpp

```
#include "widget.h"

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

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

connect(spinValue,SIGNAL(valueChanged(int)),SLOT(slotValueChange
d(int)));
    mainLayout->addWidget(spinValue);
}

Widget::~Widget()
{

}

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

Часть III

Использование QDateTimeEdit.

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

Programm12_3.pro

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

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

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

private slots:
    void slotTimeOut();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

Widget::~Widget()
{
    timerClock->stop();
    delete timerClock;
}

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

Часть IV

Использование QTextEdit.

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

Programm12_4.pro

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

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
```

```
{
    Q_OBJECT

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

Widget::~Widget() {
}
```

Часть V

Задание для самостоятельного выполнения

Программа «Пароль»

Реализовать программу, которая производит вычисление функции, указанной в таблице (функция выбирается в соотвествии с порядковым номером в журнале). При этом вход в программу осуществляется по

паролю. После запуска программы появляется окно ввода пароля. Символы в поле ввода пароля должны быть скрыты. Если пароль введен верно, появляется основное окно программы с формой для расчёта функции. Если пароль введен неверно, то выдается ошибка и программа завершается. Значения переменных вводятся в QLineEdit на основной форме. Формула для расчета выбирается по талице 12.13.

Таблица 12.13. Формулы для расчета

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

Программа «Текстовый редактор»

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

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Часть 2

Цель работы: изучить основные элементы ввода и методы их использования.

Теоретические сведения

Кнопки

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

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

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

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

Таблица 12.14. Сигналы класса QAbstractButton

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

Для опроса текущего состояния кнопок в классе QAbstractButton определены три метода. Методы опроса состояния кнопки представлены в таблице 12.15.

Таблица 12.15. Методы опроса состояния.

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

Класс QPushButton виджета нажимающейся кнопки определен в заголовочном файле QPushButton.

Создать нажимающуюся кнопку можно следующим образом:

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

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

Таблица 12.16. Варианты кнопок

Кнопка	**Описание**
Normal Button	(Обычная кнопка) соответствует той самой кнопке, которую мы привыкли видеть в большинстве случаев. После отпускания кнопка всегда возвращается в свое исходное положение;
Toggle Button	(Выключатель) может пребывать в двух состояниях: нажатом или не нажатом, которые соответствуют положениям «включено» или «выключено». Логика действия этой кнопки идентична, например, логике обычного комнатного электровыключателя;

Flat Button	Flat Button (Плоская кнопка) по своим функциональным особенностям идентична обычной кнопке. Разница лишь во внешнем виде. Например, благодаря тому, что конту- ры этой кнопки не видны, ею можно воспользоваться для размещения «секретной кнопки» диалогового окна;
Pixmap Button	(Кнопка с изображением) представляет собой кнопку, содержащую растровое изображение

Рис. 12.1. Внешний вид различных вариантов кнопок

Для управления внешним видом кнопок используются методы указанные в таблице 12.16, а сам внешний вид кнопок приведен на рисунке 12.1..

Таблица 12.17. Методы для управления внешним видом кнопок

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

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

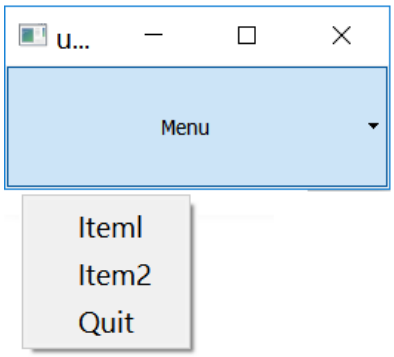


Рис.12.2. Всплывающее меню на кнопке

Виджет флажка QCheckBox

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

Флажок представляет собой маленький прямоугольник и может содержать поясняющий текст или картинку. При щелчке на виджете в прямоугольнике появится отметка. Этого же можно добиться нажатием клавиши <Пробел>, когда виджет находится в фокусе. Виджет флажка устанавливается в положение «включено» или «выключено» и является, по логике действия, кнопкой-выключателем (toggle button). Но, в отличие от последней, флажок может иметь еще и третье состояние — неопределенное. Внешнний вид данного виджета представлен на рисунке 12.3, а его основные методы в таблице 12.18.

Пример использования такого состояния можно увидеть в диалоговом окне Properties (Свойства) Проводника в ОС Windows при выборе нескольких файлов, имеющих разные атрибуты.

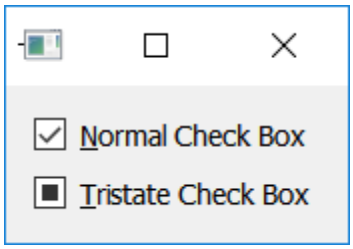


Рис.12.3. Внешний вид

Таблица 12.18. Основные методы QCheckBox

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

Переключатели QRadioButton

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

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

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

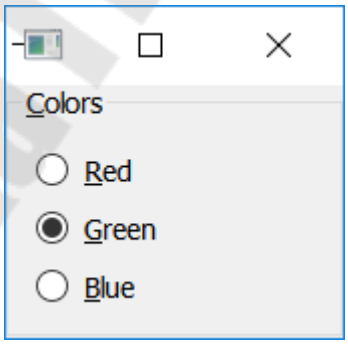


Рис.12.4. Внешний вид переключателей

Ползунок QSlider

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

Класс QSlider содержит метод, управляющий размещением рисок (шкалы) ползунка. Риски очень важны при отображении ползунка. Они

дают пользователю визуально более четкое представление о его местонахождении и показывают шаг.

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

Таблица 12.19. Значения setTickPosition

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

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

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

Ход работы

Часть I

Использование QPushButton.

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

Programm12_5.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm12_5
```

```
TEMPLATE = app
SOURCES += main.cpp\
        widget.cpp
HEADERS  += widget.h
RESOURCES += \
    icon.qrc
```

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QDebug>

class Widget : public QWidget
{
    Q_OBJECT

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

private:
    QVBoxLayout *mainLayout=NULL;
    QPushButton *buttonA=NULL;
```

```
    QPushButton *buttonB=NULL;
    QPushButton *buttonC=NULL;
    QPushButton *buttonD=NULL;

private slots:
    void slotPushButtonA();
    void slotButtonBToggled(bool input);
    void slotPushButtonC();
    void slotPushButtonD();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

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

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

    buttonD = new QPushButton(tr("Button D"),this);
    QPixmap pix(":/battery.png");
    buttonD->setIcon(pix);
    buttonD->setIconSize(QSize(64,64));
```

```
    connect(buttonD,SIGNAL(pressed()),this,SLOT(slotPushButtonD()));
    mainLayout->addWidget(buttonD);
}

Widget::~Widget()
{

}

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

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

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

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

icon.qrc

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

Стоит отметить, что данный пример включается в себя файл ресурсов. Ресурсы – дополнительные материалы, такие как картинки, текстовые документы и т.д., которые при сборке вносятся в исполняемый файл и могут использоваться в процессе выполнения программы. В данном случае файл ресурсов содержит картинку, которая отображается на последней кнопке.

Для добавлени файла ресурсов необходимо вызвать выпадающее меню, на имени проекта, выбрать пункт «Добавить новый…» и, в появившимся окне, выбрать файл ресурсов. Затем открыть его и добавить необходимые материалы.

Часть II

Использование QCheckBox.

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

Programm12_6pro

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

main.cpp

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

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

   return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
   Q_OBJECT

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

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

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

#endif // WIDGET_H
```

Widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   mainLayout = new QVBoxLayout(this);
   checkEnableSettings = new QCheckBox(tr("Enable settings"),this);
```

```
    mainLayout->addWidget(checkEnableSettings);
    groupSetings = new QGroupBox(tr("Settings"),this);
    groupSetings->setEnabled(checkEnableSettings->isChecked());

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

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

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

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

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

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

Widget::~Widget()
{

}

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

    } else if ((!checkParam2->isChecked()) && (!checkParam3-
>isChecked()))
    {
        checkParam1->setChecked(false);
```

```
    } else
    {
       checkParam1->setCheckState(Qt::PartiallyChecked);
       checkParam1->setTristate(true);
    }
}
```

Часть III

Использование QRadioButton.

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

Programm12_7.pro

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

main.cpp

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

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

   return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVBoxLayout>
#include <QRadioButton>
#include <QGroupBox>
#include <QPalette>

class Widget : public QWidget
{
   Q_OBJECT

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

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

```
    mainLayout = new QVBoxLayout(this);
    groupColors = new QGroupBox(tr("Colors"),this);
    mainLayout->addWidget(groupColors);
    layoutGroupColor = new QVBoxLayout(groupColors);

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

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

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

connect(radioGreen,SIGNAL(clicked(bool)),this,SLOT(slotClickRadiokG
reen(bool)));
    layoutGroupColor->addWidget(radioGreen);

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

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

Widget::~Widget()
{

}

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

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

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

Часть IV

Использование QSlider.

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

Programm12_8.pro

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

main.cpp

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

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

   return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
   Q_OBJECT

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

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

connect(sliderSize,SIGNAL(valueChanged(int)),this,SLOT(slotValueCha
nged(int)));
   sliderSize->setMinimum(10);
   sliderSize->setMaximum(200);
```

```
    mainLayout->addWidget(sliderSize);

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

Widget::~Widget()
{

}

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

Часть V

Задание для самостоятельного выполнения

Программа «Пароль» модернизированная

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

Таблица 12.20. Формулы для расчета

№	Задание	№	Задание
1	y = 15a +45b – 123c	11	y= (88a+5)*b/c
2	y = 10sin(a) – 12b + c^2	12	y=10a/5c + 15b

3	y= 56sin(a) – 10cos(b) + 13c	13	y= sin(a)*cos(b) + 5c
4	y= 11a – 3sin(b) +16c	14	y= 192-4a+5c/b
5	y= 2a^2 + 15b – 150/c	15	y= 4ab/c
6	y= 2(a+b)^2-16c	16	y=11a^2+b^3+22c
7	y= 21/a + 14b = c^2	17	y= 11(a+b) + c/10
8	y= 4a/b + c^6	18	y= 77(a^2+2b) – 3c/4
9	y= (11a^4 +b)*c	19	y= 11(a+b+c)
10	y= a^2+b^2+c^2	20	y= 90a – ¾(b-c^2)

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Лабораторная работа № 13
Управление событиями

Цель работы: изучить основные события и их использование при построении графического интерфейса.

Теоретические сведения

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

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

События клавиатуры QKeyEvent

Класс QKeyEvent содержит данные о событиях клавиатуры. С его помощью можно получить информацию о клавише, вызвавшей событие, а также ASCII-код отображенного символа (American Standard Code for Information Interchange, американский стандартный код для обмена информацией). Объект события передается в методы QWidget::keyPressEvent() и QWidget:: keyReleaseEvent (), определенные в классе QWidget. Событие может вызываться нажатием любой клавиши на

клавиатуре, включая <Shift>, <Ctrl>, <Alt>, <Esc> и <F1>- <F12>. Исключение составляют клавиша табулятора <Tab> и ее совместное нажатие с клавишей <Shift>, которые используются методом обработки QWidget:: event () для передачи фокуса следующему виджету.

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

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

Таблица 13.1. Некоторые значения перечислений кеупространства имен Qt

Константа	**Значение**	**Константа**	**Значение**	**Константа**	**Значение**
Key_Space	20	Key_B	42	Key_Insert	1000006
Key_NumberSi	23	Key_C	43	Key_Delete	1000007
Key_Dollar	24	Key_D	44	Key_Pause	1000008
Key_Percent	25	Key_E	45	Key_Print	1000009
Ke y_Ampe r s	26	Key_F	46	Key_Home	1000010
Key_Apostroph	27	Key_G	47	Key_End	1000011
Key_ParenLeft	28	Key_H	48	Key_Left	1000012
Key_ParenRigh	29	Key_I	49	Key_Up	1000013
Key_Asterisk	2A	Key_J	4A	Key_Right	1000014
Key_Plus	2B	Key_K	4B	Key_Down	1000015
Key_Comma	2C	Key_L	4C	Key_PageUp	1000016
Key_Minus	2D	Key_M	4D	Key_PageDo	1000017
Key_Period	2E	Key_N	4E	Key_Shift	1000020
Key_Slash	2F	Key_O	4F	Key_Control	1000021
Key_0	30	Key_P	50	Key_Meta	1000022
Key_l	31	Key_Q	51	Key_Alt	1000023
Key_2	32	Key_R	52	Key_CapsLo	1000024
Key_3	33	Key_S	53	Key_NumLo	1000025
Key_4	34	Key_T	54	Key_ScrollL	1000026
Key_5	35	Key_U	55	Key_Fl	1000030
Key_6	36	Key_V	56	Key_F2	1000031
Key_7	37	Key_W	57	Key_F3	1000032
Key_8	38	Key_X	58	Key_F4	1000033
Key_9	39	Key_Y	59	Key_F5	1000034
Key_Colon	ЗА	Key_Z	5A	Key_F6	1000035

Key_Semicolon	3B	Key_Back	5C	Key_F7	1000036
Key_Less	3C	Key_Esca	1000000	Key_F8	1000037
Key_Equal	3D	Key_Tab	1000001	Key_F9	1000038
Key_Greater	3E	Key_Back	1000003	Key_F10	1000039
Key_Question	3F	Key_Retur	1000004	Key_Fll	100003A
Key_A	41	Key_Enter	1000005	Key_F12	100003B

Событие обновления контекста рисования

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

События мыши Класс QMouseEvent

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

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

По умолчанию метод mouseMoveEvent () вызывается при перемещении указателя мыши, только если одна из ее кнопок нажата. Это

позволяет не создавать лишних событий во время простого перемещения указателя. Если же необходимо отслеживать все перемещения указателя мыши, то нужно воспользоваться методом setMouseTracking() класса QWidget, передав ему параметр true.

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

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

Вызвав метод button (), можно узнать, какая из кнопок мыши была нажата в момент наступления события (табл. 13.2). Метод buttons() возвращает битовую комбинацию из приведенных в табл. 13.2 значений. Как видно из этой таблицы, значения не пересекаются, поэтому можно применять операцию | (ИЛИ) для их объединения. Значения перечисления MouseButton представлены в таблице 13.2.

Таблица 13.2. Некоторые значения перечисления MouseButton пространства имен Qt

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

Ход работы

Часть I

Работы с событиями клавиатуры QKeyEvent.

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

Programm13_1.pro

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

main.cpp

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    editNumber = new QNUmberLineEdit(this);
    mainLayout->addWidget(editNumber);
}

Widget::~Widget()
{

}
```

qnumberlineedit.h

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

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

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

#endif // QNUMBERLINEEDIT_H
```

qnumberlineedit.cpp

```
#include "qnumberlineedit.h"

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

}

void QNUmberLineEdit::keyPressEvent(QKeyEvent *event)
{
   qDebug()<<"Key pressed";

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

Часть II

Работа с событиями обновления контекста рисования.

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

Programm31_2.pro

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

main.cpp

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

```
#include <QApplication>

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
```

```
{
}

Widget::~Widget()
{

}

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

Часть III

Работа с событиями мыши QMouseEvent.

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

Programm11_3.pro

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

main.cpp

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

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

   return a.exec();
```

```
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QLabel
{
   Q_OBJECT

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

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

```
Widget::~Widget()
{

}

void Widget::mousePressEvent(QMouseEvent* event)
{
   dumpEvent(event, "Mouse Pressed");
}

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

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


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


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

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

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


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

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

Часть IV

Задание для самостоятельного выполнения

Программа «Виджет журнала»

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

Программа «виджет консоли»

На основе текстового поля реализовать виджет консоли (командной строки). Строка приглашения должна содержать символы >> . В данный виджет должны вводиться только числа в 16-ти ричной системе. При

нажатии клавиши ввод должен вызываться сигнал, в который передается последняя команда. А также сама команла должна дублироваться в поле виджета. Так же реализовать возможность выбора последних введенных команлд с помощью клавиш «стрелка вверх» и «стрелка вниз»

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Лабораторная работа № 14
Компоненты для работы с мультимедиа.
Часть 1

Цель работы: изучить основные компоненты для работы с графикой, а также методы их применения.

Теоретические сведения

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

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

Классы геометрии

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

Точка

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

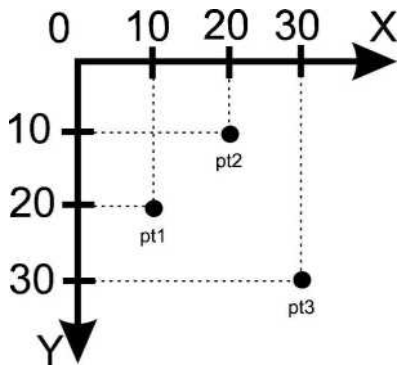


Рис.14.1. Создание и сложение точек

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

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

Например:

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

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

Например:

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

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

Например:

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

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

Например:

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

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

Двумерный размер

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

Классы QSize и QSizeF, как и классы QPoint, QPointF, предоставляют

операторы сравнения ==, ! = и метод isNull (), возвращающий значение true в том случае, если высота и ширина равны нулю.

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

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

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

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

Прямоугольник

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

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

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

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

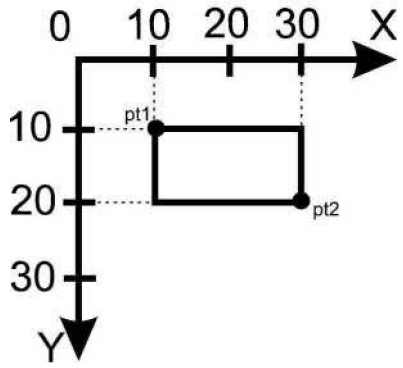


Рис.14.2. Прямоугольная область

Прямая линия

Классы QLine и QLineF описывают прямую линию или, правильней сказать, отрезок на плоскости в целочисленных и вещественных координатах соответственно. Позиции начальной точки можно получить при помощи методов xi () и yi (), а конечной — x2 () и y2 (). Аналогичного результата можно добиться вызовами pi () и p2 (), которые возвращают объекты класса QPoint/QPointF, описанные ранее. Методы dx() и dy() возвращают величины горизонтальной и вертикальной проекций прямой на оси X и Y соответственно. Прямую, показанную на рисунке 14.3, можно создать при помощи одной строки кода:

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

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

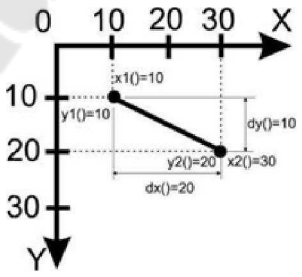


Рис.14.3. Прямая линия

Многоугольник

Многоугольник (или полигон) — это фигура, представляющая собой замкнутый контур, образованный ломаной линией. В Qt эту фигуру

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

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

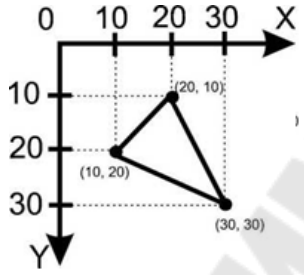


Рис.14.4. Задание многоугольника

Цвет (класс QColor)

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

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

С помощью класса QColor можно сохранять цвета моделей RGB и HSV. Его определение находится в заголовочном файле QColor. Объекты

класса QColor можно сравнивать при помощи операторов == и ! =, присваивать и создавать копии.

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

Первый параметр задает оттенок красного, второй — зеленого, а третий — оттенок синего цвета. Диагональ куба, идущая от черного цвета к белому, — это оттенки серого цвета. Диапазон каждого из трех значений может изменяться в пределах от 0 до 255 (либо от 0 до 1 в вещественном представлении), где 0 означает полное отсутствие оттенка цвета, а 255 — его максимальную насыщенность.

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

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

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

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

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

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

переменным структуры QRgb и 32-битное значение цвета. Например, синий цвет устанавливается сразу несколькими способами:

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

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

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

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

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

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

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

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

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

- оттенок (Hue)— это «цвет» в общеупотребительном смысле этого слова, например, красный, оранжевый, синий и т. д., который задается углом в цветовом круге, изменяющемся от 0 до 360 градусов;
- насыщенность (Saturation) обозначает наличие белого цвета в оттенке. Значение насыщенности может изменяться в диапазоне от 0 до 255. Значение, равное 255 в целочисленном числовом представлении либо единице в вещественном представлении,

соответствует полностью насыщенному цвету, который не содержит оттенков белого. Частично насыщенный оттенок светлее — например, оттенок красного с насыщенностью, равной 128, либо 0,5 в вещественном представлении, соответствует розовому;

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

Установку значения цвета в координатах HSV можно выполнить с помощью метода QColor::setHsv() ИЛИ QColor::setHsvF(): color.setHsv(233, 100, 50);

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

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

Класс QPainter

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

Чтобы использовать объект QPainter, необходимо передать ему адрес объекта контекста, на котором должно осуществляться рисование. Этот адрес можно передать как в конструкторе, так и с помощью метода QPainter:: begin (). Смысл метода begin () состоит в том, что он позволяет рисовать на одном контексте несколькими объектами класса QPainter. При использовании метода begin() нужно по окончании работы с контекстом вызвать метод QPainter:: end (), чтобы отсоединить установленную этим методом связь с контекстом рисования, давая возможность для рисования другому объекту.

```
QPainter painterl;
QPainter painter2;
```

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

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

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

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

Ход работы

Часть I

Рисование геометрических объектов.

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

PainterPath.pro

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

main.cpp

```
#include <QtWidgets>

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

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

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

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

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

   PainterPathWidget wgt;

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

   return app.exec();
```

```
}
```

Часть II

Наложение объектов.

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

Programm31_2.pro

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

main.cpp

```
#include <QtWidgets>

QLabel* lbl(const QPainter::CompositionMode& mode)
{
   QLabel* plbl = new QLabel;
   plbl->setFixedSize(100, 100);

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

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

   QImage resultImage(rect.size(), QImage::Format_ARGB32_Premultiplied);
   painter.begin(&resultImage);
   painter.setRenderHint(QPainter::Antialiasing, true);
   painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
   painter.setPen(QPen(QColor(0, 255, 0), 4));
```

```
    painter.setBrush(QBrush(QColor(255, 0, 0)));
    painter.drawEllipse(rect);
    painter.setCompositionMode(mode);
    painter.drawImage(rect, sourceImage);
    painter.end();

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


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

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

```
  pgrd->addWidget(new QLabel("<CENTER>DestinationAtop</CENTER>"),
3, 4);
  pgrd->addWidget(lbl(QPainter::CompositionMode_Xor), 2, 5);
  pgrd->addWidget(new QLabel("<CENTER>Xor</CENTER>"), 3, 5);
  wgt.setLayout(pgrd);

  wgt.show();

  return app.exec();
}
```

Часть III

Задание для самостоятельного выполнения

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

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Часть 2

Цель работы: изучить основы работы со звковыми устройствами. Освоить на практике алгоритмы записи и воспроизведения звуковых сигналов

Теоретические сведения

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

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

К соновным параметрам оцифровки сигнала можно отнести:

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

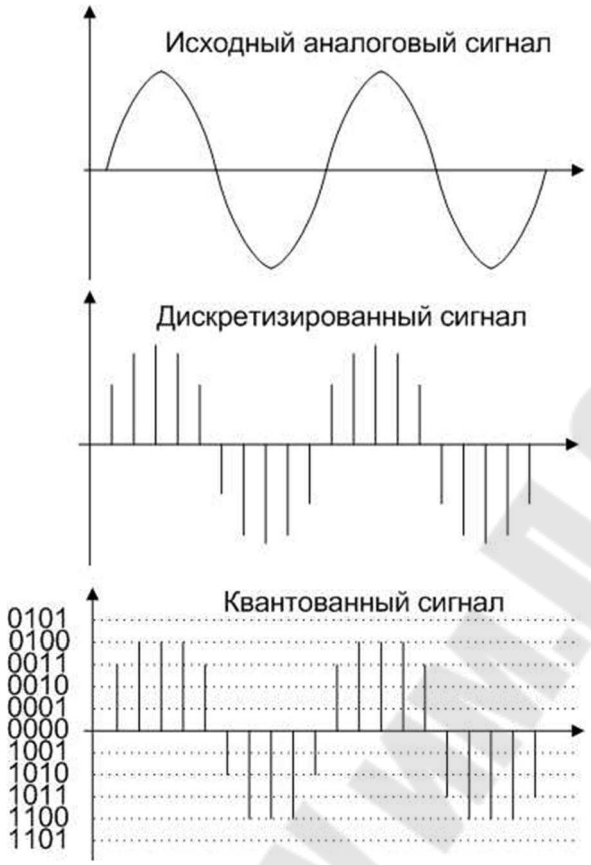


Рис.14.5. Процесс оцифровки сигнала.

Класс QIODevice

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

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

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

устройствах все данные располагаются последовательно и после чтения удаляются из буфера.

К наиболее распространенным наследникам QIODevice можно отнести такие классы как:

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

Таблица 14.1. Основные виртуальные методы QIODevice

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

	Ожидание окончания записи данных
	Ожидание чтения данных
	Возвращает true если может быть прочитана строка данных

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

Таблица 14.2. Защищенные виртуальные методы класса QIODevice

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

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

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

Класс QAudioFormat

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

Таблица 14.3. Основные методы класса QaudioFormat

Метод	**Описнние**
	Возвращает установленный порядок следования байтов
	Возращает имя установленного кодека
	Возвращает количество каналов
	Возвращает частоту дискретизации
	Возвращает разрядность
	Возвращает тип отсчета
void setByteOrder(QAudioFormat::Endian byteOrder)	Устанавливает порядок следования байтов
	Устанавливает количество каналов
void setCodec(const QString &codec)	Устанавливает кодек
	Устанавливает частоту дескритизации
	Устанавливает разрядность
void setSampleType(QAudioFormat::SampleType sampleType)	Устанавливает тип отсчета

Пример использования:

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

В данном примере частота дискретизации равно 8000 Гц, разрядность — 16 бит, количество каналов — 1, имя кодека - audio/pcm, порядок следования байтов — от младшего к старшему, тип отсчета — знаковый целочисленный тип.

Класс QAudioInput

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

Основные открытые методы класса приведены в таблице 14.4.

Таблица 14.4. Основные методы класса QAudioInput

Метод	**Описание**
	Возвращает установленные параметры звукового устройства
	Сбрасывает устройство
	Устанавливает размер внутреннего буфера
	Устанавливает громкость аудиоустройства
	Данный метод запускает работу с устройством, в качестве входного параметра необходимо задать устройство, в которое будет производиться запись данных

	Возвращает текущее состояние аудиоустройства
	Приостанавливает работу устройства
	Возращает текущее значение громкости

Пример использование:

```
QFile destinationFile;   // Class member
QAudioInput* audio; // Class member
{
   destinationFile.setFileName("test.raw");
   destinationFile.open( QIODevice::WriteOnly | QIODevice::Truncate );

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

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

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

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

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

Класс QAudioOutput

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

Пример использования:

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

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

    QAudioDeviceInfo info(QAudioDeviceInfo::defaultOutputDevice());
    if (!info.isFormatSupported(format)) {
        qWarning() << "Raw audio format not supported by backend, cannot play audio.";
        return;
    }

    audio = new QAudioOutput(format, this);
```

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

Класс QBuffer

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

Таблица 14.5. Основные методы класса QBuffer

Метод	Описание
QBuffer(QByteArray *byteArray, QObject *parent = nullptr)	Конструктор класса. В качестве входного параметра можно задать QbyteArray, в котором будут храниться данные
	Возвращает буфер с данными
	Устанавливает буфер для данных
void setData(const QByteArray &data)	Заполняет буфер данными из data
void setData(const char *data, int size)	Заполняет буфер данными из data

Ход работы

Часть I

Пример программы для работы с аудиоустройствами

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

Файл LR14_1.pro

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

```
deprecated before Qt 6.0.0

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

Файл main.cpp

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

Файл widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget {
   Q_OBJECT
public:
   Widget(QWidget *parent = 0);
private:
   QAudioInput *input;
   QAudioOutput *output;
   QAudioFormat formatAudio;
   QBuffer *buffer;
   QByteArray data;
   QHBoxLayout *mainLayout;
   QPushButton *buttonRecord;
```

```
    QPushButton *buttonPlay;

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

Файл widget.cpp

```
#include "widget.h"

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

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

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

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

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

void Widget::slotPushPlay() {
```

```
    buffer->seek(0);
    buttonRecord->setEnabled(false);
    output→start(buffer); }

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

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

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

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

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

Часть II

Задание для самостоятельно работы

В данной лабораторной работе необходимо самостоятельно реализовать две программы. В первой должен быть реализован диктофон,

позволяющий делать голосовые записи, сохранять их в файлы и затем прослышивать выбранную запись.

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

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Лабораторная работа № 15

Многопоточность в приложении.

Синхронизация потоков

Цель работы: изучить принципы построения многопоточных приложений, а также классы, позволяющие создавать и синхронизировать потоки.

Теоретические сведения

Потоки

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

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

Многопоточность требуется для выполнения действий в фоновом режиме, параллельно с действиями основной программы, и позволяет разбить выполнение задач на параллельные потоки, которые могут быть абсолютно независимы друг от друга. А если приложение выполняется на компьютере с несколькими процессорами, то разделение на потоки может значительно ускорить работу всей программы, так как каждый из процессоров получит отдельный поток для выполнения. В последнее время появляется все больше компьютеров, оснащенных многоядерными процессорами, что делает многопоточное программирование еще более популярным.

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

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

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

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

Например:

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

Расммотри более подробно интерфейс класс QThread. В таблице 15.1 приведены основные сигналы класса.

Таблица 15.1. Сигналы

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

В таблице 15.2 приведены основные слоты класса.

Таблица 15.2. Слоты

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

Таблица 15.3. Приоритеты

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

Обмен сообщениями

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

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

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

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

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

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

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

Сигнально-слотовые соединения

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

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

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

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

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

Синхронизация (Мьютексы)

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

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

Мьютексы (mutex) обеспечивают взаимоисключающий доступ к ресурсам, гарантирующий, что критическая секция будет обрабатываться только одним потоком. Поток, владеющий мьютексом, обладает эксклюзивным правом на использование ресурса, защищенного мьютексом,

и другой поток не может завладеть уже занятым мьютексом.

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

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

Таблица15. 4. Открытые методы класса QMutex

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

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

Ход работы

Часть I

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

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

Реализация программы без использования потоков.
Programm15_1.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm15_1
TEMPLATE = app


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

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

main.cpp

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

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

    return a.exec();
}
```

increment.h

```
#ifndef INCREMENT_H
#define INCREMENT_H

#include <QObject>
#include <QThread>

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

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

public slots:
};

#endif // INCREMENT_H
```

increment.cpp

```
#include "increment.h"

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

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

void Increment::stop()
```

```
{
    enableWork = false;
}

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

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

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

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

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

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

```
}

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

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

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

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

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

Часть II

Реализация программы без использования потоков.

Programm15_2.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Programm15_2
TEMPLATE = app
```

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

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

main.cpp

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

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

    return a.exec();
}
```

increment.h

```
#ifndef INCREMENT_H
#define INCREMENT_H

#include <QObject>
#include <QThread>

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

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

```
signals:
    void doIt();

public slots:
};

#endif // INCREMENT_H
```

increment.cpp

```
#include "increment.h"

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

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

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

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

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

```
            emit doIt();
        }
    }
```

widget.h

```
    #ifndef WIDGET_H
    #define WIDGET_H

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

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

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

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

    #endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

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

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

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

void Widget::slotPushStart()
```

```
{
   increment->startWork();
}

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

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

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

Часть III

Задание для самостоятельного выполнения

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

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Лабораторная работа № 16
Работа с диалогами.
Класс QDialog

Цель работы: изучить принципы использования стандартных диалогов, а также принципы построения собственных диалогов.

Теоретические сведения

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

Их можно разбить на три основные категории:

- собственные;
- стандартные;
- окна сообщений.

Стандартные диалоговые окна

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

Диалоговое окно выбора файлов

Диалоговое окно выбора файлов предназначено для выбора одного или нескольких файлов, а также файлов, находящихся на удаленном компьютере, и поддерживает возможность переименования файлов и создания каталогов. Класс QFileDialog предоставляет реализацию диалогового окна выбора файлов и отвечает за создание и работоспособность сразу трех диалоговых окон. Одно из них позволяет осуществлять выбор файла для открытия, второе предназначено для выбора пути и имени файла для его сохранения, а третье — для выбора каталога.

Таблица 16.1. Статические методы класса QFileDialog.

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

Пример:

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

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

Диалоговое окно выбора цвета

Класс QColorDialog реализует диалоговое окно выбора цвета. Для того чтобы показать это окно, вызывается статический метод getColor (). Первым параметром в метод можно передать цветовое значение для инициализации. Вторым параметром является указатель на виджет предка. После закрытия диалогового окна метод getColor () возвращает объект класса QCoior. Чтобы узнать, какой кнопкой было закрыто окно: OK или Cancel (Отмена), необходимо вызвать метод isValid() из возвращенного объекта QColor. Значение true означает, что была нажата кнопка OK, в противном случае — Cancel (Отмена). Например:

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

```
    // Cancel
}
```

Диалоговое окно выбора шрифта

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

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

Например:

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

Диалоговые окна мастера

Диалоговые окна мастера были придуманы для сопровождения пользователя при выполнении им операций, которые требуют непосредственного его участия. Для навигации по страницам диалогового окна мастера служат две кнопки: Next (Вперед) и Back (Назад). Пользователь не имеет возможности сразу отобразить интересующую его страницу окна, не пройдя все предшествующие страницы, что гарантирует выполнение всех пунктов, содержащихся в этих диалоговых окнах.

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

Создание собственного диалогового окна

Для создания своего собственного диалогового окна нужно унаследовать класс QDialog, по умолчанию область заголовка диалогового окна содержит кнопку “?”, предназначенную для получения подробной

информации о назначении виджетов. Для того чтобы ее удалить необходимо установить флаги Qt: :WindowTitleHint И Qt: :WindowSystemMenuHint. Модальное диалоговое окно всегда должно содержать кнопку Cancel (Отмена). Обычно в диалоговых окнах используются две кнопки: OK и Cancel. Сигналы этих кнопок подключаются к слотам accept() и rejected() соответственно. Это делается для того, чтобы метод exec мог возвращать значения QDialog::Accepted в случает нажатия клавиши Ok и QDialog::Rejected в случае нажатия кнопки Cancel. Если вы хотите чтобы функция exec возвращала ваш собственный код (отличный от стандартных значений), используйте функция done().

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

Ход работы

Часть I

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

Работа с QFileDialog.

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

Programm16_1.pro

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

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

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

```
    QLineEdit *editDir;
    QPushButton *buttonDir;


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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

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

connect(buttonSaveFile,SIGNAL(pressed()),this,SLOT(slotSaveFile()));
    mainLayout->addWidget(buttonSaveFile,1,2,1,1);

    labelDir = new QLabel(tr("Dir"),this);
```

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

Widget::~Widget()
{

}

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

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

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

Часть II

Работа с QColorDialog.

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

Programm16_2.pro

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

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

```
    ~Widget();

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

private slots:
    void slotPushButtonColor();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

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

Widget::~Widget()
{

}

void Widget::slotPushButtonColor()
{
    QColor col = QColorDialog::getColor(Qt::white,this,tr("Color"));
    if (col.isValid())
    {
```

```
        QPalette Pal(palette());
        Pal.setColor(QPalette::Background,col);
        this->setPalette(Pal);
        labelColor->setText(col.name());
    }
}
```

Часть III

Работа с QFontDialog.

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

Programm16_3.pro

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

main.cpp

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

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

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
   Q_OBJECT

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

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

private slots:
   void slotPushFontButton();
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

```
    labelFont = new QLabel(tr("Font"),this);
    mainLayout->addWidget(labelFont);
    buttonFont = new QPushButton(tr("Font"),this);

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

}

Widget::~Widget()
{

}

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

Часть IV

Работа с QWizard

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

Programm16_4.pro

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

main.cpp

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

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

   return a.exec();
}
```

qcurrentpage.h

```
#ifndef QCURRENTPAGE_H
#define QCURRENTPAGE_H

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

class QCurrentPage : public QWizardPage
{
public:
   QCurrentPage(QString text, QWidget *parent=NULL);

private:
   QVBoxLayout *mainLayout;
   QLabel *labelText;

};

#endif // QCURRENTPAGE_H
```

qcurrentpage.cpp

```
#include "qcurrentpage.h"

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

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWizard
{
   Q_OBJECT

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

private:
   QCurrentPage *page_;
};

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

```
{


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

Widget::~Widget()
{

}
```

Часть V

Создание собственных диалогов.

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

Programm16_5.pro

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

main.cpp

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

int main(int argc, char *argv[])
```

```
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

qvaluedialog.h

```
#ifndef QVALUEDIALOG_H
#define QVALUEDIALOG_H

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

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

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

};
```

```
#endif // QVALUEDIALOG_H
```

qvaluedialog.cpp

```
#include "qvaluedialog.h"

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

QValueDialog::~QValueDialog()
{

}

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

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

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

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

```
    show();
}

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

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

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

class Widget : public QWidget
{
    Q_OBJECT

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

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

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

#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"

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

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

Widget::~Widget()
{

}

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

Часть VI

Задание для самостоятельного выполнения

Разработать программу «Телефонный справочник». Данные должны сохраняться и загружать из бинарного файла с использованием сериализации. Функции добавления записи, а также поиск должны быть реализованы с использование собственного диалогового окна. В настройках программы должен задаваться шрифт, используемый интерфейсом, а также цвет окон приложения. Все параметры должны сохраняться и загружаться с использованием класса QSettings.

Содержание отчета

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

Контрольные вопросы для самопроверки

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

Лабораторная работа № 17
Построения основного окна приложения на основе QMainWindow

Цель работы: изучить принципы использования класса QMainWindow для построения основного окна программы.

Теоретические сведения

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

Класс главного окна QMainWindow

Класс QMainWindow — это очень важный класс, который реализует главное окно, содержащее в себе типовые виджеты, необходимые большинству приложений, — такие как меню, секции для панелей инструментов, рабочую область, строки состояния и т. п. В этом классе внешний вид окна уже подготовлен, и его виджеты не нуждаются в дополнительном размещении, поскольку уже находятся в нужных местах. На рисунке 17.1 изображен пример основного окна программы.

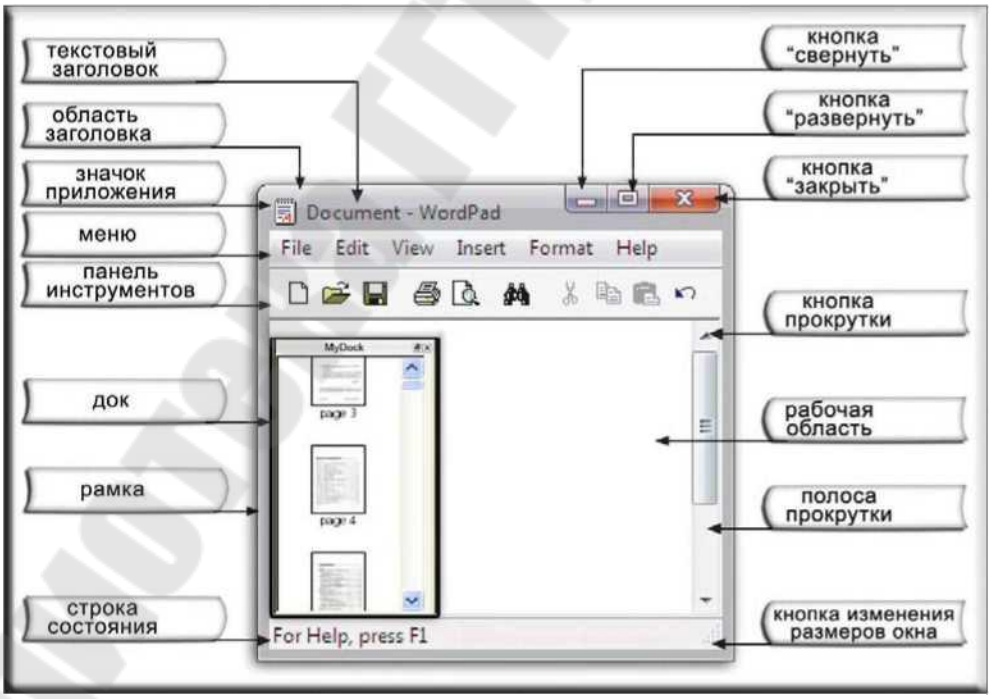


Рис.17.1. Основное окно программы

В таблице 17.1 отображены основные открытые методы класса QMainWindow.

Таблица 17.1. Основные открытые методы класса QMainWindow

Метод	**Описание**
QMainWindow(QWidget *parent = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags())	Конструктор класса. В качестве родителя передается указатель на Qwidget.
void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar)	Данный метод позволяет добавить панель инструментов. Первый параметр задает положения панели по умолчанию, а вторым параметром задается указатель на класс панели инструментов
void addDockWidget(Qt::DockWidgetArea area, QDockWidget *dockwidget)	Метод для добавления док виджета на форму основного окна. В качестве первого параметра передается положение виджета по умолчанию, а в качестве второго параметра передается указатель на док виджет
QWidget *centralWidget() const	Метод возвращает указатель на центральный виджет
void setIconSize(const QSize &iconSize)	Метод устанавливает размер иконок на основном окне
void setMenuBar(QMenuBar *menuBar)	Метод позволяет установить строку меню. В качестве входного параметра передается указатель на класс строки меню
void setStatusBar(QStatusBar *statusbar)	Метод задает объект строки статуса
QStatusBar *statusBar() const	Метод возвращает указатель на объект строки состояния

Класс действия QAction

Класс действия QAction предоставляет очень мощный механизм, ускоряющий разработку приложений. Если бы его не было, то вам пришлось бы создавать команды меню, соединять их со слотами, затем создавать панели инструментов и соединять их с теми же слотами и т. д. Это

приводило бы к дублированию программного кода и вызывало бы проблемы при синхронизации элементов пользовательского интерфейса. Например, при необходимости сделать одну из команд меню недоступной, вам нужно было бы сделать ее недоступной в меню, а потом проделать то же самое и с кнопкой панели инструментов этой команды.

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

QAction объединяет следующие элементы интерфейса пользователя:

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

В таблице 17.2 отображены основные открытые метода класса QAction.

Таблица 17.2. Открытые метода класса QAction.

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

Панель инструментов (QToolBar)

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

Панель инструментов представляет собой область, в которой расположены кнопки, дублирующие часто используемые команды меню. Для панелей инструментов библиотека Qt предоставляет класс QToolBar.

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

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

Пример создания и добавления действия на панель инструментов:

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

Строка состояния

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

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

отображать процесс выполнения какой-либо операции, то лучше разместить в строке состояння индикатор этого процесса. Чтобы разместить виджет в строке состояния, нужно передать его указатель в метод addwidget (). Виджеты можно также удалять из строки состояния с помощью метода removeWidget ();

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

Пример добавления виджета в строку состояния:

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

MDI-приложение

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

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

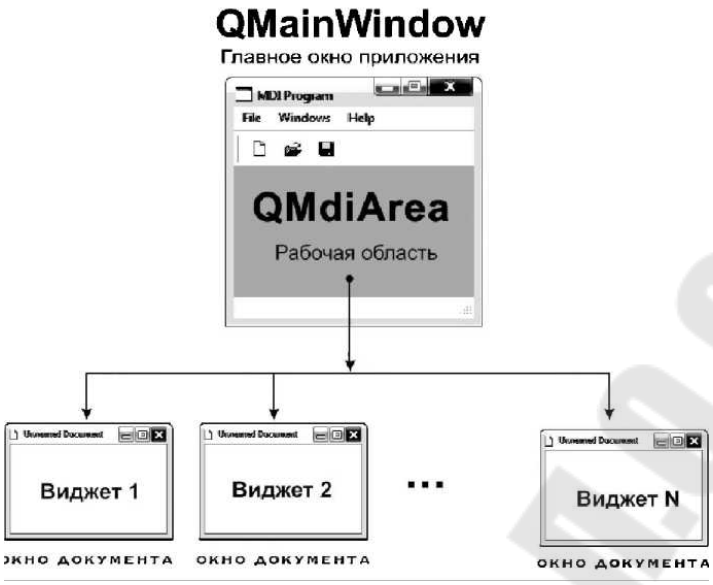


Рис.17.2. MDI Приложение

Таблица17. 3. Основные открытые методы QMdiArea

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

Ход работы

Часть I

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

Работа с классом QMainWindow.

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

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

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

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

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

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

Programm17_1.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = Programm17_1
TEMPLATE = app
```

```
SOURCES += main.cpp\
        mainwindow.cpp
HEADERS  += mainwindow.h
RESOURCES += \
    icons.qrc
```

main.cpp

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

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

    return a.exec();
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

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

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private:
```

```
    QWidget *CentralWidget;
    QVBoxLayout *mainLayout;
    QLabel *labelText;
    QToolBar *toolBar;

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


};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"

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

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

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

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

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

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

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

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

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

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

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

}

MainWindow::~MainWindow()
{

}
```

icons.qrc

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

Часть II

Создание MDI приложения.

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

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

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

Programm17_2.pro

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

main.cpp

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

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

    return a.exec();
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

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

class MainWindow : public QMainWindow
{
   Q_OBJECT

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

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

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

   QAction *actionCascade;
   QAction *actionTile;

private slots:
   void slotNew();
};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"

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



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

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

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

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

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

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

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

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

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

connect(actionTile,SIGNAL(triggered(bool)),m_pma,SLOT(tileSubWind
ows()));
    toolBar->addAction(actionTile);

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

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

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

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

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

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

}

MainWindow::~MainWindow()
{

}

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

icons.qrc

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

Часть III

Задание для самостоятельного выполнения

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

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

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

Содержание отчета

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

Контрольные вопросы для самопроверки

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