About Qt

Qt is a framework to develop cross-platform applications. Currently, the supported platforms are Windows, Linux, macOS, Android, iOS, Embedded Linux and some others. Generally, it means that the programmer may develop the code on one platform but compile, link and run it on another platform. But it also means that Qt needs help: to develop software it must cooperate with development tools on specific platforms, for example in case of Windows with Visual Studio.

Qt is well-known for its tools for developing graphical user interfaces (GUI) , but it also provides powerful tools for threads, networking and communication (Bluetooth, serial port, web), 3D graphics, multimedia, SQL databases, etc.

Qt has its own Integrated Development Environment (IDE): QtCreator.

History: 1991: version 1.0 as cross-platform GUI programming toolkit was developed and implemented by TrollTech (Norway). 2005: version 4.0 was a big step ahead but the compatibility with older versions was lost. 2008: TrollTech was sold to Nokia. 2012: Nokia Qt division was sold to Digia (Finland). 2014: Digia transferred the Qt business to Qt Company. The latest version is 6.9.3

Official website: https://www.qt.io/

Installation (1)

To use Qt, you need to create Qt account. For academic purposes the Qt is free of charge, but first you need to get the Educational License, see https://www.qt.io/qt-educational-license. Downloading is from page https://www.qt.io/download-dev.

Tip: the Qt installer proposes to store the stuff in folder *C:\Qt*. To avoid later complications, agree.

Tip: *QtCreator.exe* is in folder *C:\Qt\Tools\QtCreator\bin.*

Tip: QtCreator manual is on http://doc.qt.io/qtcreator/.

After download in folder *C:\Qt* open *Qt Maintenance Tool.exe* and log in.

Move: *Maintenance actions → Add or remove components → Qt → Qt 6.9.3.* Check *MSVC 2022 64-bit* and *Qt Debug Information Files,* uncheck *MinGW 13.1.0. 64-bit.*

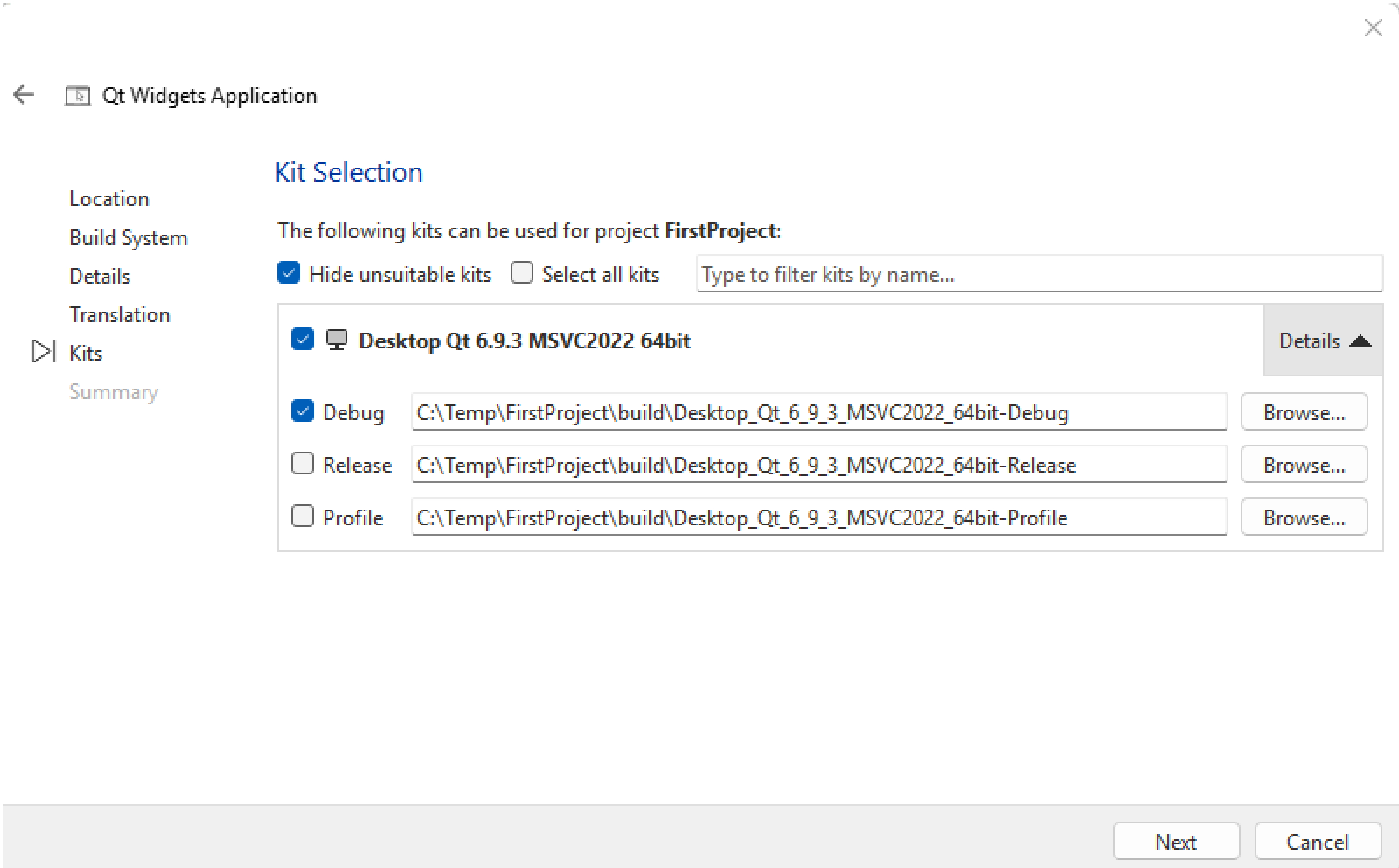
Move: *Maintenance actions → Add or remove components → Qt Creator.* Check *CDB Debugger Support.*

Update Qt. Then start QtCreator and click *Create Project:*

1. Choose *Widget Application.*
2. Set the project name to *FirstProject* and the folder to *C:\Temp*
3. Set the build system to *qmake*.
4. Accept wizard proposed *MainWindow* class.
5. Ignore the translation files.

Installation (2)

6. Check the kits. Specifying the kit you set the compiler (Qt does not have its own). In this course we must use the Visual Studio tools.



7. Accept the files proposed by wizard. File *.pro corresponds to *.sln used in Visual Studio.

Installation (3)

8. Build the project. In case of failure read the following slides.
9. Check have you now the following files and folders:

C:\Temp

- Temp
  - FirstProject
    - build
      - Desktop_Qt_6_9_3_MSVC2022_64bit-Debug
        - debug
        - release

C:\Temp\FirstProject

- build
- FirstProject.pro
- FirstProject.pro.user
- main.cpp
- mainwindow.cpp
- mainwindow.h
- mainwindow.ui

C:\Temp\FirstProject\build\Desktop_Qt_6_9_3_MSVC2022_64bit-Debug

- debug
- release
- .qmake.stash
- Makefile
- Makefile.Debug
- Makefile.Release
- ui_mainwindow.h

C:\Temp\FirstProject\build\Desktop_Qt_6_9_3_MSVC2022_64bit-Debug\debug

- FirstProject.exe
- FirstProject.ilk
- FirstProject.pdb
- FirstProject.vc.pdb
- main.obj
- mainwindow.obj
- moc_mainwindow.cpp
- moc_mainwindow.obj
- moc_predefs.h

Installation (4)

Wizard-created file *FirstProject.pro* should contain the following text:

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
CONFIG += c++17
QMAKE_PROJECT_DEPTH = 0  # if the building failed, add this row yourself
# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0
SOURCES += \
    main.cpp \
    mainwindow.cpp
HEADERS += \
    mainwindow.h
FORMS += \
    mainwindow.ui
# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Installation (5)

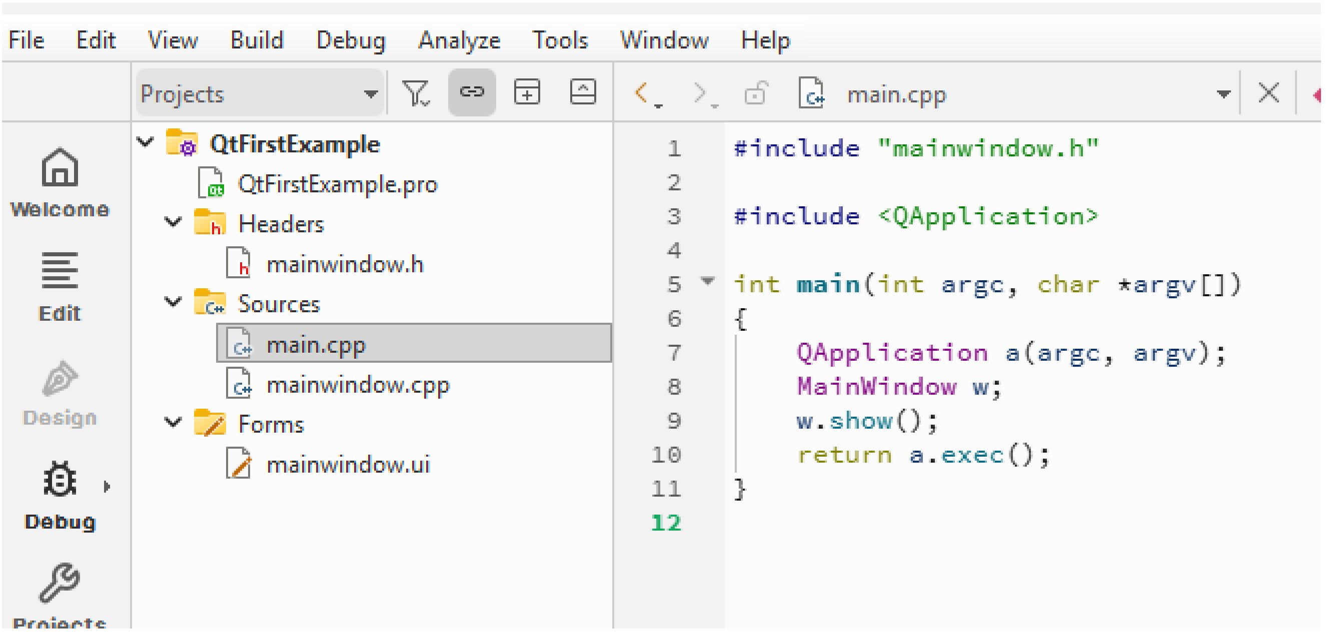
The executable from the …\build\Desktop_Qt_6_9_3_MSVC2022_64bit-Debug\debug folder cannot run outside of the QtCreator environment. Creation of stand-alone software product is out of scope of our course-.

Tips:

- To avoid later complications do not change names and other code created by wizard.
- Sometimes you may get strange and seemingly senseless error messages. If you are sure that your code is correct, try to rebuild. If it does not help try command *Run qmake* (*Build* menu). At last close QtCreator, delete the complete build folder and restart QtCreator.
- Searching help from net turn attention about which version of Qt they are discussing. Versions 4.x and partly 5.x are out of date.

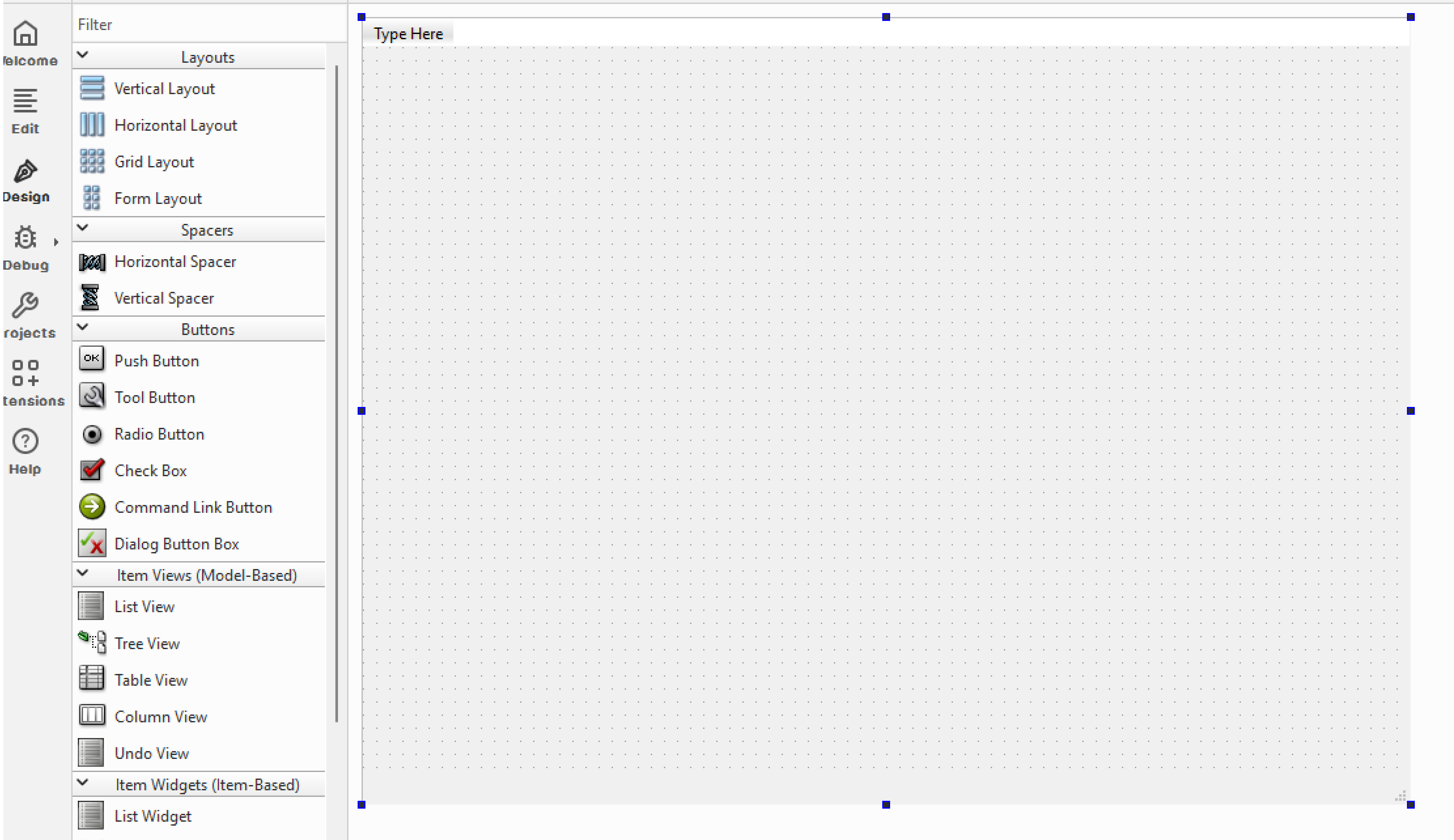
QtCreator Example #1 (1)

Create and build project QtFirstExample:



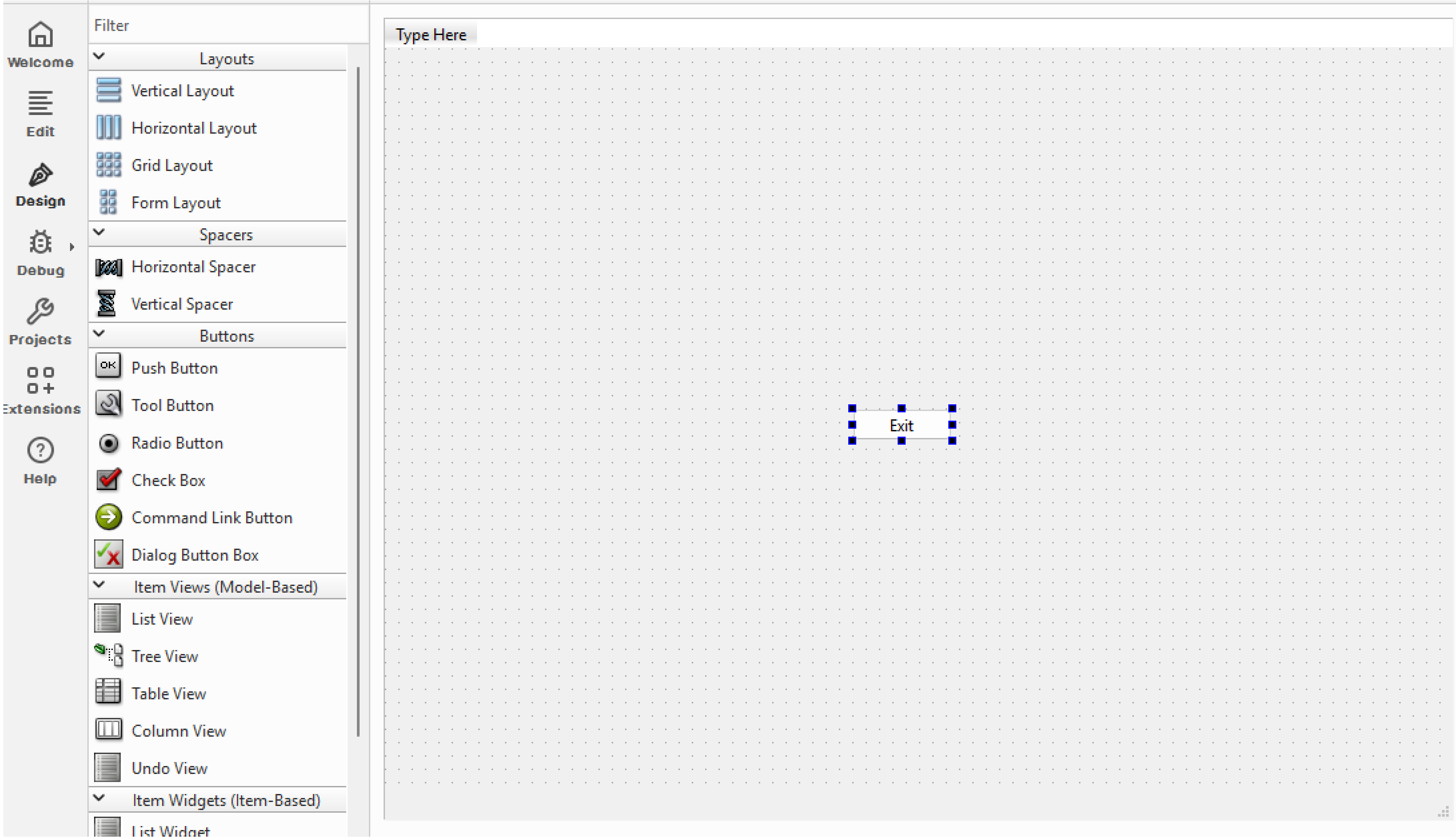
QtCreator Example #1 (2)

Double-click on *mainwindow.ui*:



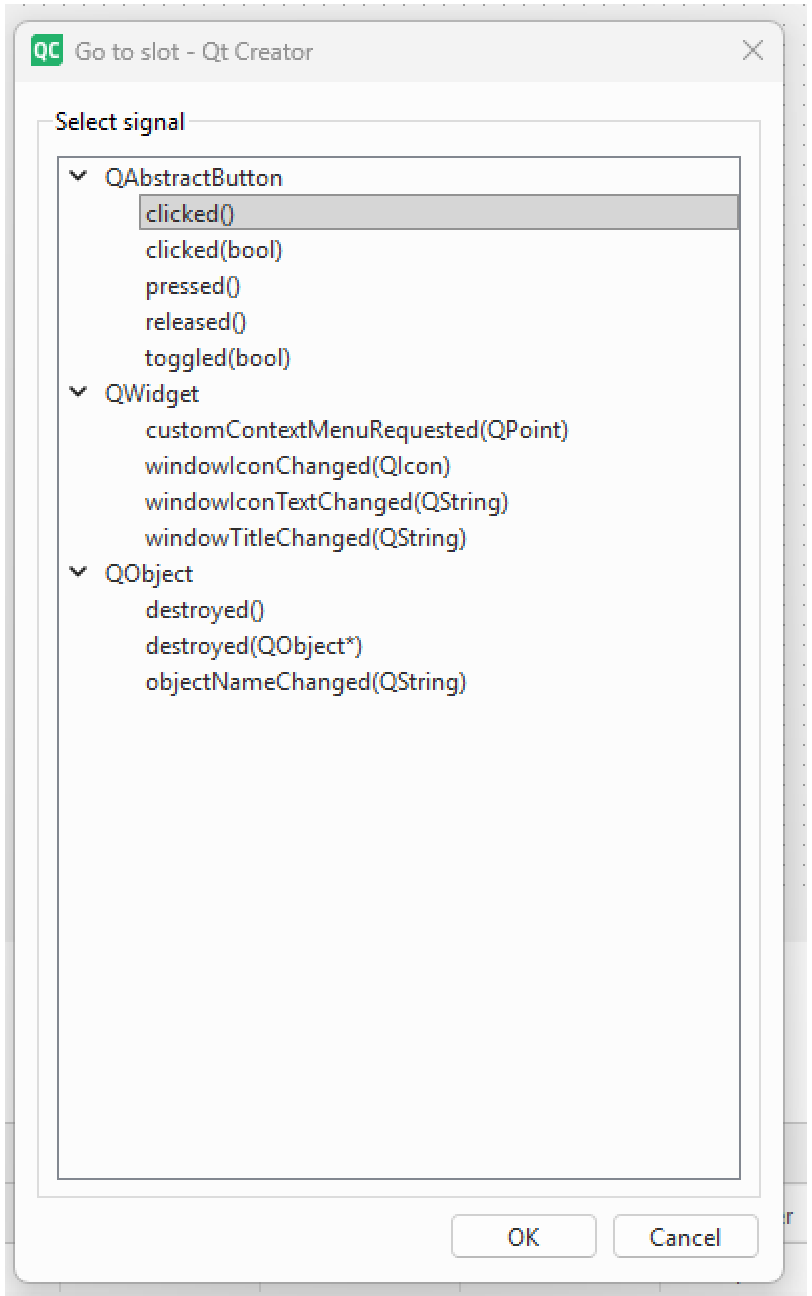
QtCreator Example #1 (3)

Drag *Push Button* from the widgets palette into the main window and right-click it. Using the pop-up menu change the title to *Exit*. Then select menu command *Goto slot*.



QtCreator Example #1 (4)

Select *QAbstractButton clicked()*. In *MainWindow* you will see wizard-created method *on_pushButton_clicked*:



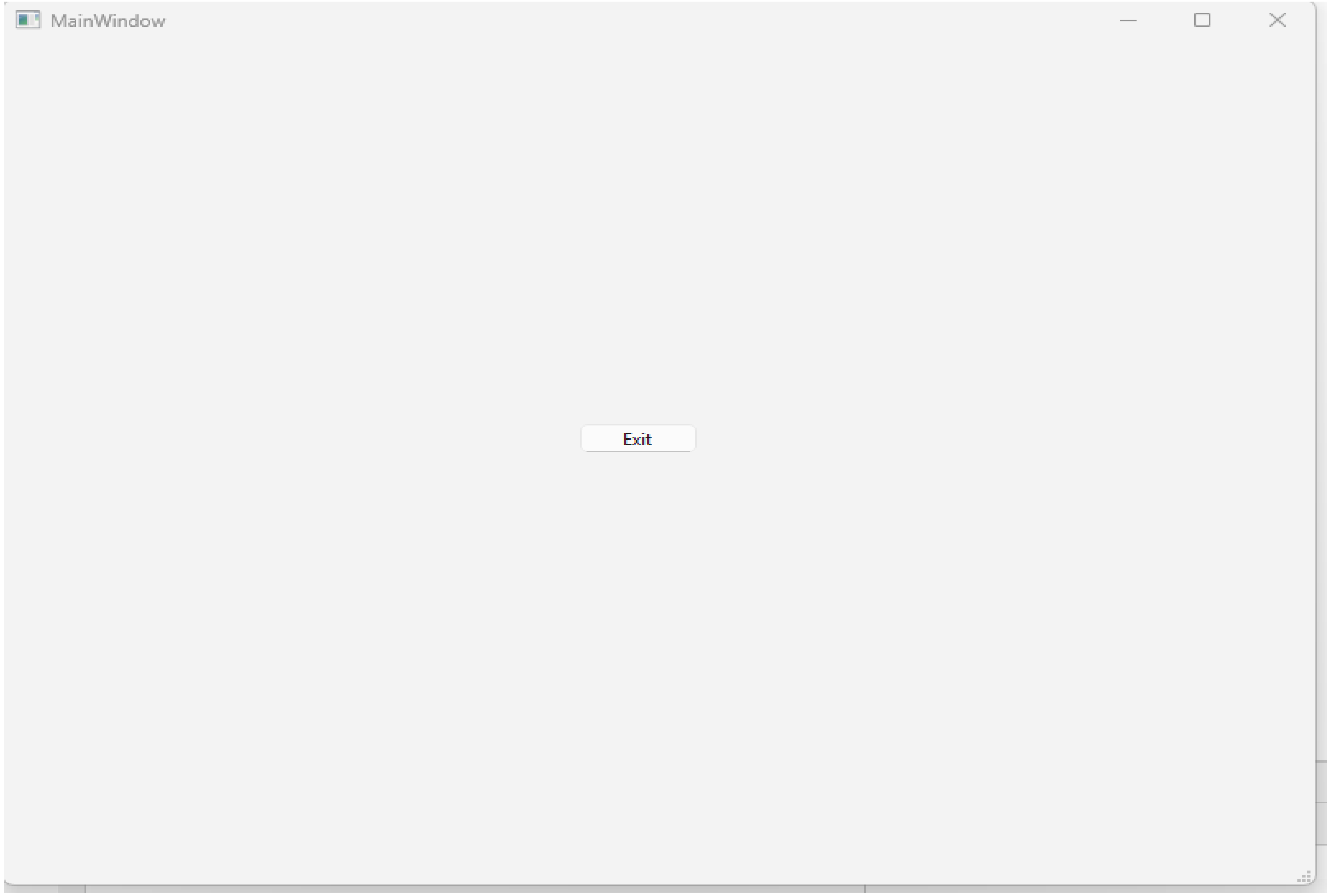

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{

}
```

QtCreator Example #1 (5)

Write into *on_pushButton_clicked* your code:

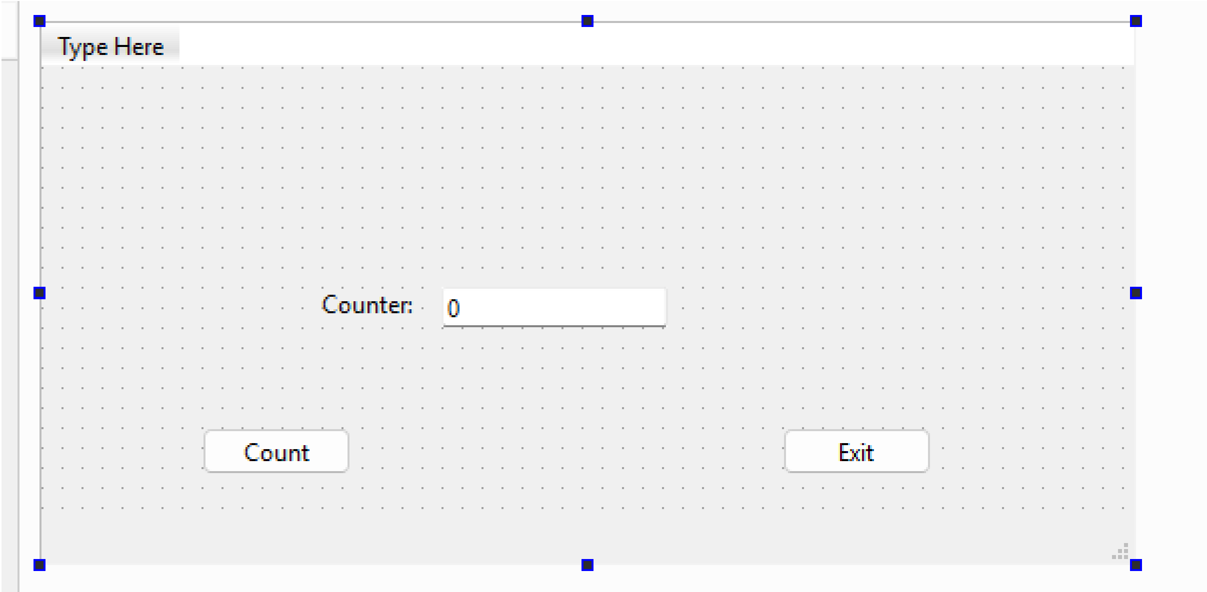
```
void MainWindow::on_pushButton_clicked()
{
    exit(0);
}
```

From menu select *Debug* → *Start debugging* → *Start debugging of startup project* or just press *F5*. The application will run. Click *Exit* to quit.



QtCreator Example #2 (1)

In this example we create the following window:



We have now 4 widgets: 2 Push Buttons, 1 Line Edit and 1 Label. They have names but it is better to introduce our own names, for example : *pushButtonExit*, *pushButtonCounter*, *lineEditCounter*, *labelCounter*. For Line Edit set text to "0". Right-click the widgets to perform the needed changes. On left upper corner right-click *Type Here* and remove the menu bar.

After each click on *pushButtonCounter* the number shown in *lineEditCounter* must be incremented.

```
void MainWindow::on_pushButtonExit_clicked()
{
    exit(0);
}


void MainWindow::on_pushButtonCounter_clicked()
{
    QString currentText = ui->lineEditCounter->text();
    int currentCounter = currentText.toInt();
    currentCounter++;
    ui->lineEditCounter->setText(currentText.setNum(currentCounter));
}
```

QtCreator Example #2 (2)

To write Qt code study class references from links:
https://doc.qt.io/qt-6/classes.html the full list of Qt standard classes
https://doc.qt.io/qt-6/qtwidgets-index.html the full list of Qt widget classes

Writing Qt software try to avoid C++ standard classes. Instead of them use the corresponding Qt classes like:

- *QString* (uses Unicode UTF-16)
- *QByteArray*
- *QFile*
- *QThread*
- Qt containers like *QVector, QList*, *QMap*, *QSet*, etc.

The C++ classes and Qt classes are similar but some differences do exist.

To get the pointer to a widget write: ui->widget_name.

For simple debugging use class QDebug (see http://doc.qt.io/qt-5/qdebug.html), for example:

```
#include <QDebug>
qDebug() << "x = " << x; // like in cout
```

Method *qDebug()* returns the *qDebug* object.

Signals and slots (1)

It is clear that a GUI must run in its own thread(s). When the user clicks a widget, types a word or just presses a key, an event has occurred. The GUI must have a listener that catches the events. An event may be ignored, but it is also possible that the event triggeres some action. This is the event-driven programming – the base of Windows and the other systems with graphical user interface.

In Qt this mechanism is implemented by signals and slots. The Qt standard classes have a common base class: *QObject*. The widgets have common base class *QWidget* inherited from *QObject*. *QObjects* are able to emit signals (i.e. send notifications informing that an event has occurred, a state parameter has changed, etc.). This signal must be processed and if necessary, some action performed. Consequently, we need a specific function called when the specific signal is emited. This function is called as slot.

In the first example, when the user clicks the button, signal *clicked()* is emitted. The wizard creates for us the corresponding slot *MainWindow::on_pushButton_clicked()*. In *mainwindow.h* you may see:

```
private slots:
    void on_pushButton_clicked();
```

Signals can have arguments. In that case the connected to it slot must be a method with input parameters. Slots may be functions or functors.

A slot may have several signals and a signal may have several slots. Read more from https://doc.qt.io/qt-6/signalsandslots.html.

Signals and slots (2)

In the previous two examples the signals from widgets were connected to slots by wizard. It is also possible and rather often necessary to do it manually.

To connect a signal and slot use method *connect* from class *QObject*:

```
connect(pointer_to_object_emitting_signal,
        &emitter_class_name::signal_name,
        pointer_to_object_receiving_signal,
        &slot_class_name::slot_name);
```

Classes created by us may also emit and receive signals. But in that case:

- They must be derived from class *QObject*
- Their declaration must start with macro *Q_OBJECT* .

Signals are declared in class declaration as ordinary methods but in their own section:

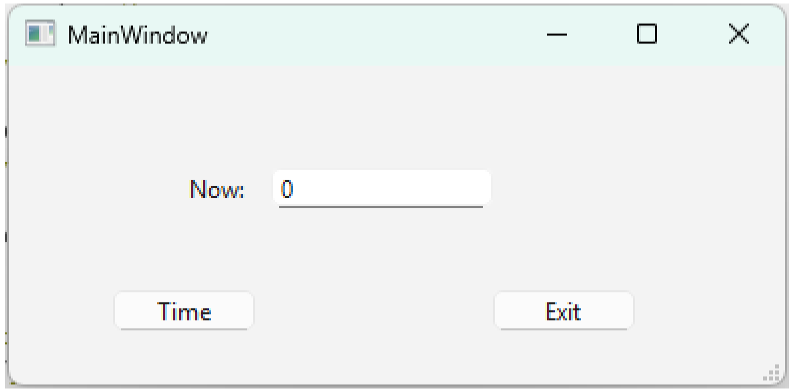
```
signals:
    void signal_name_1(signal_parameter_list);
    void signal_name_2(signal_parameter_list);
    ……………………………………………..
```

To emit a signal write:

```
emit signal_name(actual_parameters_list);
```

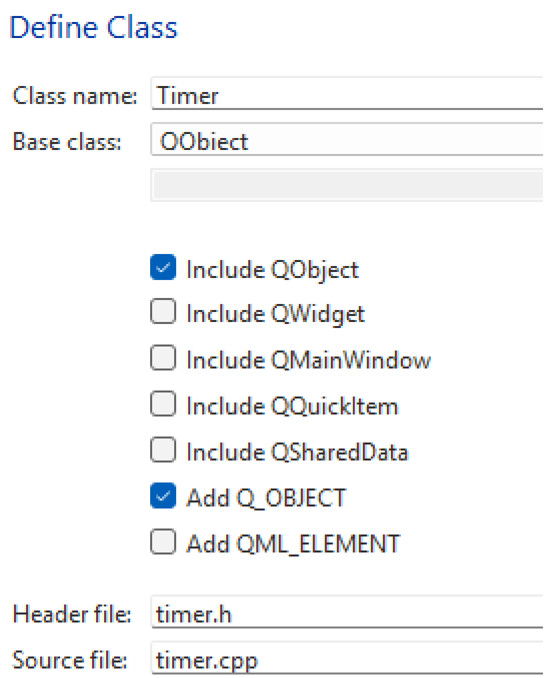
QtCreator Example #3 (1)

In this example we create the following window:



Buttons are *pushButtonTime* and *pushButtonExit, Line Edit* is *lineEditNow*.

Let the slot for signal from *pushButtonExit* to create by wizard.



Add to project new class *Timer* derived from class *QObject (*right-click *Sources* and select *Add new*… and *C++ class).* From wizard-created *Timer.h* you can see that *Timer* can emit and receive signals.

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

signals:
};
```

QtCreator Example #3 (2)

The application will work in the following way:

1. *MainWindow* constructor allocates memory for an object from class *Timer*.
2. When the user clicks *pushButtonTime*, signal *clicked()* is emitted and sent to slot *readTime()* located in *Timer* object.
3. Method *Timer::readTime()* reads the current time, transforms into text and strores in *stringstream*.
4. Before return method *Timer::readTime()* emits signal *newValue()*. Its argument is the pointer to *stringstream*.
5. Slot *showTime()* in MainWindow recieves the signal and shows the current time in *lineEditNow.*

Consequently, in *mainwindow.h* we need to have:

```
private slots:
    void on_pushButtonExit_clicked();
    void showTime(std::stringstream *);
signals:
    void getTime();
```

and in *Timer.h* we need to have:

```
signals:
    void newValue(std::stringstream *);
public slots:
    void readTime();
```

QtCreator Example #3 (3)

Connections between signals and slots are established in *MainWindow* constructor:

```
pTimer = new Timer;
connect(ui->pushButtonTime, &QPushButton::clicked, pTimer, &Timer::readTime);
connect(pTimer, &Timer::newValue, this, &MainWindow::showTime);
```

Timer::readTime() uses simple C time-processing tools:

```
void Timer::readTime()
{
    struct tm now_tm;
    time_t now = time(nullptr);
    localtime_s(&now_tm, &now);
    *pSout << put_time(&now_tm, "%d-%m-%Y %H:%M:%S");
    emit newValue(pSout);
}
```

MainWindow::showTime() transforms *std::string* into *QString*. The text in Qt widgets is always in *QString*.

```
void MainWindow::showTime(std::stringstream *pSout)
{
    std::string s = pSout ->str();
    QString qs = QString::fromStdString(s);
    ui->lineEditNow->setText(qs);
    pSout->str("");
}
```

Objects from user-defined classes have no right to access widgets directly. So, calls to *MainWindow::showTime()* directly from *Timer* is not allowed.

Parents and children

Objects from classes derived from *QObject* may or may not have parent object, for example the wizard-created main window has constructor:

```
explicit MainWindow(QWidget *parent = nullptr);
```

If an object has its parent object:

- If the parent is destroyed, this object as child will be also automatically destroyed.
- If the parent widget (for example a frame window) appears on the screen, all its children widgets (for example a set of buttons) will automatically appear inside it.

To find a specific child, the parent object must call method *findChild()*. To get the list of a group of children or the list of all children use method *findChildren()*. They both belong to base class *QObject*: https://doc.qt.io/qt-6/qobject.html

Qt events (1)

Slots are ordinary class functions but must be declared in their own section *private slots* or *public slots*. In **.cpp files* they are defined as ordinary class functions. Signals are declared as ordinary class functions but in their own section *signals*. As signals are notifications, they are not defined. An object may emit signal, i.e. send the signal to the connected to it slot and thus force the slot function to start running.

In addition to signals and slots Qt has a parallel mechanism: the events. An event is an object derived from abstract class QEvent. The events are distinguished by their types: for example *QEvent::KeyPress*, *QEvent::KeyRelease*, *QEvent::MouseButtonDblClick*, *QEvent::Wheel*, etc.

(see https://doc.qt.io/qt-5/qevent.html). Mostly, an event is the result of an activity outside the application (for example, a mouse click) but there may be also events that happen inside the application (for example, *QEvent::Timer*).

When an event is detected (mostly by Windows or another operating system), it is inserted into the event queue. The queue is handled by Qt event dispatcher that loops through the queue. The main event loop is started by method *exec()* from class *QApplication* (see the wizard-created *main.cpp*):

```
QApplication a(argc, argv);

………………………………

return a.exec(); // also blocks the main() until the end of application
```

When a Qt application is running, the control flow is either in the event loop or in the code implemented by us.

Qt events (2)

The dispatcher pops the event from queue and creates the corresponding event object. Each event has a receiver. For example, if we insert into our GUI a button, we also create an object of class *QPushButton*. Parameter *objectName* in the Qt designer properties window (for example *m_exitButton*) is actually the pointer to this object. If our application is running and the user clicks the *Exit* button, the receiver is the object with pointer *m_exitButton*. The dispatcher creates a *QEvent* object of type *QEvent::MouseButtonPress* and calls function *QPushButton::event()* with the event object as the actual parameter.

The event handling functions are virtual. Therefore the programmers may override them and thus change the standard features of widgets or even add some new ones. For typical operations the signals / slots mechanism is good enough and we may forget the events. But if we, for example, want that when the mouse is moving over the button then the button turns to red, we need to override event handling functions. More exactly, we have to create our own class derived from *QPushButton*.

When the user clicks icon *X* on the main window right upper corner, a *QCloseEvent* targeted to *MainWindow* is generated. If our application must before closure perform some operations (for example, close connections or store settings), we have to impelement the *QCloseEvent* handling method.

Qt events (3)

Example:

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    QMessageBox question(QMessageBox::Question, "SineGenerator", "Are you sure?\n",
                         QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes);
    if (question.exec() != QMessageBox::Yes)
    {
        event->ignore(); // reject, closing cancelled
    }
    else
    {
        event->accept(); // accept, go on with standard cancel procedures
    }
}
```

The example demonstrates also how to use a simple message box. Read more from https://doc.qt.io/qt-6/qmessagebox.html

Qt threads (1)

Each Qt application has its main thread or the GUI thread. The other threads launched in a Qt application are often referred as worker threads.

All the widgets are handled only in the GUI thread and cannot be directly accessed from the other threads.

The Qt thread has its own stack of local variables. It may also have its own event queue for events that do not belong to GUI event queue.

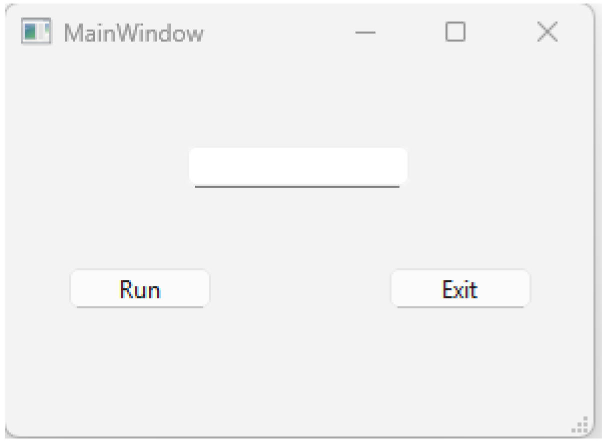
The communication between threads is organized by signals / slots and events.

The simplest way to create and execute a Qt thread is as follows:

1. Create a class derived from *QThread*.
2. In this class implement the thread entry point function:
   protected: void run() override { …… }
3. Define an object of this class.
4. Call method *start()* associated with this new object. It calls the *run()* and emits the *started* signal.
5. When *run()* exits, thread emits the *finished* signal.

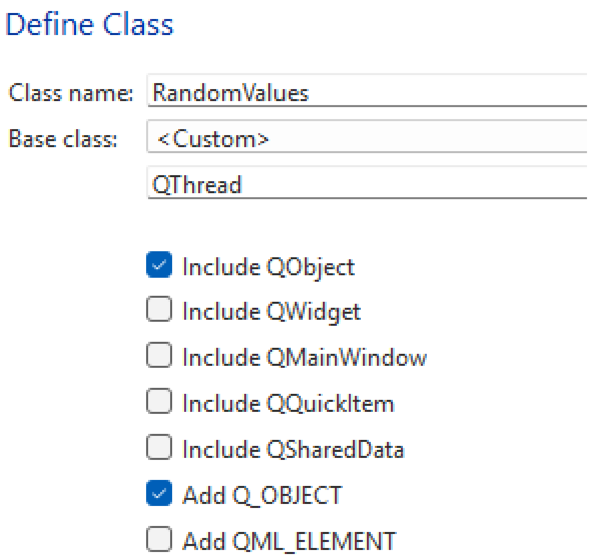
Qt threads (2)

In QtCreator Example #4 we first create a simple GUI:



The widgets are *pushButtonRun*, *pushButtonExit* and *lineEditValue*. Click on *pushButtonRun* launvhes a thread. The thread generates random numbers that are shown in *lineEditValue*. The random values are generated on occasional timepoints. The user can click on pushButtonRun just once, after click the button will be disabled (greyed out).

Add to project new class *RandomValues* derived from *QThread*:




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

class RandomValues : public QThread
{
    Q_OBJECT
public:
    RandomValues();
};
```

Qt threads (3)

Complete the code of class *RandomValues*. For simplicity let us use C++ tools. *QAtomicInt* is similar to *atomic<int>* in C++.

```
#include <QAtomicInt>

class RandomValues : public QThread
{
    Q_OBJECT
public:
    RandomValues();
    RandomValues(QAtomicInt *);
protected:
    void run() override;
signals:
    void result(int);
private:
    QAtomicInt *pEnded;
};
```

```
RandomValues::RandomValues(QAtomicInt *p)
{
    pEnded = p;
}

void RandomValues::run()
{
    default_random_engine generator;
    uniform_int_distribution<int> delay_distribution(0, 5000);
    uniform_int_distribution<int> value_distribution(0, 100);
    while(!*pEnded)
    {
        this_thread::sleep_for(chrono::milliseconds(delay_distribution(generator)));
        int x = value_distribution(generator);
        emit result(x);
    }
}
```

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private slots:
    void on_pushButtonExit_clicked();
    void on_pushButtonRun_clicked();
public slots:
    void showResult(int);
private:
    Ui::MainWindow *ui;
    RandomValues *pRandomValues = nullptr;
    QAtomicInt *pEnded = nullptr;
    void waitThreadEnd();
};
```

Signal-slot connection between thread and *MainWindow is* as it was discussed in the previous example:

```
void MainWindow::showResult(int x)
{
    QString qs = QString::number(x);
    ui->lineEditValue->setText(qs);
}
```

Qt threads (4)

```
MainWindow::~MainWindow()
{
    if (pRandomValues)
    {
        delete pRandomValues;
        delete pEnded;
    }
    delete ui;
}

void MainWindow::on_pushButtonExit_clicked()
{
    if (pEnded)
    {
        *pEnded = true;
    }
    else
    {
        exit(0);
    }
}


void MainWindow::on_pushButtonRun_clicked()
{
    pEnded = new QAtomicInt;
    *pEnded = false;
    pRandomValues = new RandomValues(pEnded);
    connect(pRandomValues, &RandomValues::finished, this, &MainWindow::waitThreadEnd);
    connect(pRandomValues, &RandomValues::finished, pRandomValues, &QObject::deleteLater);
    connect(pRandomValues, &RandomValues::result, this, &MainWindow::showResult);
    pRandomValues->start();
    ui->pushButtonRun->setEnabled(false);
}

void MainWindow::waitThreadEnd()
{
    pRandomValues->wait();
    exit(0);
}
```

If the thread is not running, click on *pushButtonExit* quits the application. If it is running, we must at first stop it, setting the flag **pEnded* to *true*.

To enable or disable a widget use method *setEnabled()*.

Method *Qthread::wait()* is similar to *join()* in C++ threads: here it allows the GUI thread to continue and exit when the *RandomValues* thread has ended.

Qt threads (5)

```
connect(pRandomValues, &RandomValues::finished, this, &MainWindow::waitThreadEnd);
connect(pRandomValues, &RandomValues::finished, pRandomValues, &QObject::deleteLater);
```

If the thread has stopped, it automatically sends signal *QThread::finished()* to the *MainWindow,* here into slot *waitThreadEnd()*. Signal *QThread::finished()* must be sent also into slot *QObject::deleteLater()*. Those two connections are compulsory.

Qt Example #4 presents a simple thread. In complicated cases the thread must have its own event loop.

All the objects derived from QObject are said to live on the thread in which they were created. Consequently, if we write in MainWindow:

WorkerThread *pWorkerThread = new WorkerThread;

then object with pointer *pWorkerThread* lives on the GUI thread. Even more, all its attributes derived from QObject and created by *WorkerThread* constructor are also living on GUI thread.

If the object that emitted a signal and the object owing the slot that receives this signal are living on the same thread, event loop is not needed. Otherwise the overridden *run()* entry point function must start the event loop calling method *exec()* (see slide *Qt events (1)*). To stop the event looping call *QThread* function *quit()*. Remember that *exec()* blocks the thread and therefore must be called on the last row of *run()*. If the event loop is present, the slot call arguments are packed up in a data structure and sent as an event to the receiving thread's event queue. In the receiving thread, the *QObject::event* method unpacks the arguments and calls the slot method.

Qt threads (6)

Consequently:

- Object of class derived from *QThread* (for example *pWorkerThread*) is just a worker thread object and not the thread itself. The thread is a sequence of machine instructions.
- The constructor of worker thread object is called in GUI thread. Consequently the thread object itself and all its attributes are living on the GUI thread.
- When the *run()* method has started, the worker thread is running. Objects created in *run()* or in methods called by *run()* are living on the worker thread.
- Methods defined in worker thread class may run as a part of worker thread (sequence of instructions) as well as a part of GUI thread.

There are several types of signal / slot connections:

1. In case of direct connection the signal is emitted from the same thread on which the receiver object is living. The slot method is called directly.
2. In case of queued connection the signal is emitted from one thread but the receiver object is living on another thread. An event is created and posted. The thread on which the receiver object is living must have event loop.
3. In case of blocked queued connection the thread from which the signal is emitted blocks until the slot method returns.

Method *connect* has an additional parameter with default value *Qt::AutoConnection* (select between direct and queued connection automatically). The other values are *Qt::QueuedConnection*, *Qt::DirectConnection*, *Qt::BlockingQueuedConnection*.

Qt thread synchronization

QMutex and *QMutexLocker* are almost identical with C++ standard *mutex* and *unique_lock*. See https://doc.qt.io/qt-6/qmutex.html and https://doc.qt.io/qt-6/qmutexlocker.html .

The Qt alternative to C++ *conditional_variable* is *QWaitCondition* (see https://doc.qt.io/qt-6/qwaitcondition.html). The differences are minor.

It is possible to use C++ mutexes and other thread control classes in QThreads. But Qt does not support latches, barriers, etc. introduced in C++ v. 20.

QSemaphore (not discussed in this course) is similar to semaphores implemented in Windows, see https://doc.qt.io/qt-6/qsemaphore.html .

QReadWriteLock is similar to *QMutex*. Its strength is that it is able to distinguish between reading and writing operations and thus allow multiple readers (but not writers) to access the data simultaneously. *QReadLocker* and *QWriteLocker* are convenience classes that automatically lock and unlock a *QReadWriteLock*. See https://doc.qt.io/qt-6/qreadwritelock.html

Qt input / output

The base class for I/O is QIODevice. All the other I/O classes are derived from it.

There are two types of I/O devices:

1. Random access devices like hard disk file (*QFile* class) have current position: we may set it (usually with method named as *seek*) to where we want and then directly read or write. We may also request the number of bytes in data set (i.e. the size). Reading and writing is fast and in most cases the multithreading is not needed. In extreme situations (for example, there is no data to read) we are informed immediately.
2. Sequental devices like sockets handle streams of data. There is no way to rewind the stream. The amount of data in stream is previously unknown. As the device is remote, the reading an writing may take time and those operations cannot be in the GUI main thread. Qt has standard classes for working with serial ports (*QSerialPort*), Bluetooth connection (*QBluetoothSocket* and several associated classes), named pipes (*QLocalSocket*) and TCP connection (*QTcpSocket* and several associated classes).

Qt file operations (1)

QFile constructor (see https://doc.qt.io/qt-6/qfile.html) creates a new object but does not create the disk file itself. It just specifies the file name:

QFile file_ name(QString_specifying_the_filename);

To operate with file we have to open it. The simplest way for that is:

file_name.open(open_mode);

The open modes are *QIODevice::ReadOnly*, *QIODevice::WriteOnly* and *QIODevice::ReadWrite*. If the file does not exist and the mode is not *QIODevice::ReadOnly*, it will created. The fundamental mode may be combined using the *bitwise or* with flags *QIODevice::Append*, *QIODevice::Truncate*, *QIODevice::Text*, *QIODevice::Unbuffered*. In case of failure the *open* method returns *false.* You may use the *errorString* method from *QIODevice* to get *QString* explaining the reason.

Example:

```
QFile file("Test.txt");
if (!file.open(QIODevice::ReadWrite | QIODevice::Text | QIODevice::Truncate))
{
  qDebug() << file.errorString();
}
```

When you do not need the file any more, close it:

file_name.close();

Qt file operations (2)

To select a file in GUI use the *QFileDialog* standard dialog box (see https://doc.qt.io/qt-6/qfiledialog.html). The simplest way to get the name of an existing file is to write:

```
QString file_name = QFileDialog::getOpenFileName(this, window_title, default_folder, filter);
```

Similar function is QFileDialog::getSaveFilename (file may not exist). Example:

```
QString fileName = QFileDialog::getOpenFileName(this, "Coursework",
    "c:\\TTU studies", "*.cpp, *.h");
if (!filename.isEmpty()) { // i.e. not cancelled
    QFile file(fileName);
}
```

To aqcuire the current position in file use method *pos()*:

```
qint64 current_pos = file_name.pos();
```

Here *qint64* corresponds to *long long int* type in Visual Studio. To shift to new position use method *seek()*:

```
file_name.seek(new_position);
```

Here the new position means the number of bytes from the beginning (and not from the current position) of file.

To get the file size use method *size()*:

```
qint64 file_size = file_name.size();
```

Example:

```
qint64 file_size = file.size();
file.seek(file_size); // to the end of file
```

Qt file operations (3)

To write into file:

```
file_name.write(pointer_to_array, number_of_bytes_to_write);
```

or

```
file_name.write(QByteArray_to_write);
```

In both cases the function returns the number of bytes that was actually written. Return value -1 means that the writing failed.

Example:

```
QFile file("Test.bin");
file.open(QIODevice::ReadWrite | QIODevice::Truncate);
char arr[] = { 'a', 'b', 'c', 'd' };
file.write(arr, 4);
QByteArray qba("efgh");
file.write(qba);
```

Remark that in Qt string constants consist of one-byte characters. But in

```
QString qstr("efgh");
```

the characters are converted into two-byte *QChar* (i.e. Unicode) characters.

Example:

```
QChar qarr[] = { 'i', 'j', 'k', 'l'}; // 8 bytes: 0x00, 0x69, 0x00, 0x6A, 0x00, 0x6B, 0x00, 0x6C
file.write(reinterpret_cast<char *>(qarr), sizeof(qarr));
```

Remark: Qt does not like C-style castings.

Qt file operations (4)

Instead of method *write* you may use streams:

```
QDataStream stream_name(pointer_to_Qfile_object);
QTextStream stream_name(pointer_to_Qfile_object);
```

In both streams (see https://doc.qt.io/qt-6/qdatastream.html and https://doc.qt.io/qt-6/qtextstream.html) the data transfer is implemented with overloaded *operator*<< methods. During transfer the data is serialized. Several of the Qt standard classes support serialization. In our own classes, of course, we have to write *operator*<< methods ourselves. Examples:

```
QFile file1("Test.bin");
file1.open(QIODevice::ReadWrite | QIODevice::Truncate);
QDataStream stream1(&file1);
int i = 15;
stream1 << i; // stores 0x00, 0x00, 0x00, 0x0F
const char *p = "abcd";
stream1 << p; // stores 0x00, 0x00, 0x00, 0x05, 0x61, 0x62, 0x63, 0x64, 0x00
              // and not just 0x61, 0x62, 0x63, 0x64, 0x00 (C-string serialization rules)!
QPoint point(10, 12);
stream1 << point; // stores 0x00, 0x00, 0x0A, 0x00, 0x00, 0x0C
file2.open(QIODevice::ReadWrite | QIODevice::Text | QIODevice::Truncate);
QTextStream stream2(&file2);
stream2 << i << ' ' << p << ' ' << point.x() << ' ' << point.y(); // as cout in standard C++
```

Qt file operations (5)

Methods for reading:

qint64 nr_of_bytes_actually_read = file_name.read(pointer_to_buffer, nr_of_bytes_to_read);
QByteArray results = file_name.read(nr_bytes_to_read);
QByteArray results = file_name.readAll();

If the number of actually read data is 0, the file is empty. Return value -1 means that reading has failed. On the both extreme cases an empty *QByteArray* is returned.

From text files we may read by lines:

qint64 nr_of_bytes_actually_read = file_name.readLine(pointer_to_buffer, buffer_length);

The result is a regular *char* * C-string. *"\r\n"* at the end of line is replaced by *"\n\0"*.

Also, the reading is possible with *operator>>* from *QDataStream* (targets may be variables of Qt standard types but not arrays or other containers) and *QTextStream* (targets may be also of class *QString* and *QByteArray* as well as pointer to array).

Examples:

```
QDataStream stream1(&file);
char buf1[1024];
stream1 >> buf1; // compile error
QTextStream stream2(&file);
stream2 >> buf1; // correct, but if the buffer is too small, crashes
QByteArray buf2;
stream2 >> buf2;  // advised to use
```

Qt remote device operations (1)

As the base class is always *QIODevice*, the main ideas of Qt remote device operations are almost the same for all of them.

To start, we have to open the device and establish the connection. It may take time and therefore should be in a separate thread. During connecting procedure this thread is blocked. Therefore we need to set timeout. If the specified time has elapsed, the connecting has failed. When the connection has been established, the waiting is interrupted:

```
void run() {
   ...................................
   connecting_fun(connecting_parameters);
   if (!waiting_fun(timeout_value))    {
      emit error_message_to_GUI;
   }
   else {
       emit success_message_to_GUI;
   }
}
```

There is an additional way to know that the connection was successful: at the end of procedure the device emits signal *connected* and we have to write a slot for it (for some devices only).

Similar thread with timeout and / or slot for signal *disconnected* is also necessary for disconnecting and closing. This signal is emitted also when for some reason the connection has broken off.

Qt remote device operations (2)

For writing we may use methods *write* inherited from *QIODevice* (see slide *Qt file operations (3)*).

```
void run() {
   ………………………………
  qint64 n = write(data_to_write);
  if (!waitForBytesWritten(timeout_value))   {
     emit error_message_to_GUI;
  }
  else {
     emit number_of_written_bytes_to_GUI
  }
}
```

There is an additional way – signal inherited from *QIODevice*:

```
void bytesWritten(nr_written_bytes);
```

Any Qt remote device emits this signal when the writing operation has finished. The slot must inform the user and / or the other modules of current program that they may continue.

The user may break off the pending writing operation by closing the device.

Qt remote device operations (3)

For reading we need to write a slot for signal

```
void readyRead();
```

QIODevice emits this signal when there is some available data. The slot must perform the actual reading using functions presented on slide *Qt file operations (5)*.

To set the time we can wait for arrival of data use method

```
bool result = waitForReadyRead(timeout_in_ms);
```

Return value *false* means that the operation is timed out or an error occurred.

The user may break off the pending reading operation by closing the device.

Some devices may also emit error signals (specific to this device, not inherited from *QIODevice*).

Settings (1)

It is very cumbersome to fill after each start all the fields in GUI window. To remember and restore the settings use class *QSettings* (see https://doc.qt.io/qt-6/qsettings.html).

To work with Qt settings mechanism your *MainWindow* class should contain an object of class *QSettings*:

```
QSettings settings_name(organization_name, application_name);
```

The both arguments are *QString*s. Example:

```
QSettings *pSettings = new QSettings("TTU", "Coursework");
```

To store the settings consider to write a *MainWindow* method that is called from the *QCloseEvent* handler as well as from the *Exit* button slot. In Windows the settings are stored in the system registry.

To read and view the settings consider to write a *MainWindow* method that is called from the constructor.

To store a value, use method *setValue()*:

```
settings_name.setValue(key, value);
```

Here *QString* key specifies the settings name. The value is a *QVariant* – meaning that it may be any of the most common Qt types (*bool*, *int*, *double*, *QChar*, *QString*, *QByteArray*, *QDate*, etc., read more from https://doc.qt.io/qt-6/qvariant.html). Example:

```
pSettings->setValue("nPoints", ui->nPointsLineEdit->text()); // save as text
```

or

```
pSettings->setValue("nPoints", ui->nPointsLineEdit->text().toInt()); //save as integer
```

Settings (2)

To restore a value use method *value()*:

```
settings_name.value(key).convertor_from_QVariant();
```

Example:

```
ui->nPointsLineEdit->setText(pSettings->value("nPoints").toString());
```

If the setting with specified key was not found, the *value()* method returns *QVariant* with default zero value, that may be converted to integer zero, to empty string, etc. This zero may be replaced by any other default value:

```
settings_name.value(key, default_value).convertor_from_QVariant();
```

In Windows system registry the keys are case-insensitive.

Settings may be grouped. In that case the key consists of two parts separated by slash, for example: *"mainwindow/size"*.

You may store the settings when the application is running (for example, to avoid loosing settings in case of crash):

```
settings_name.sync();
```

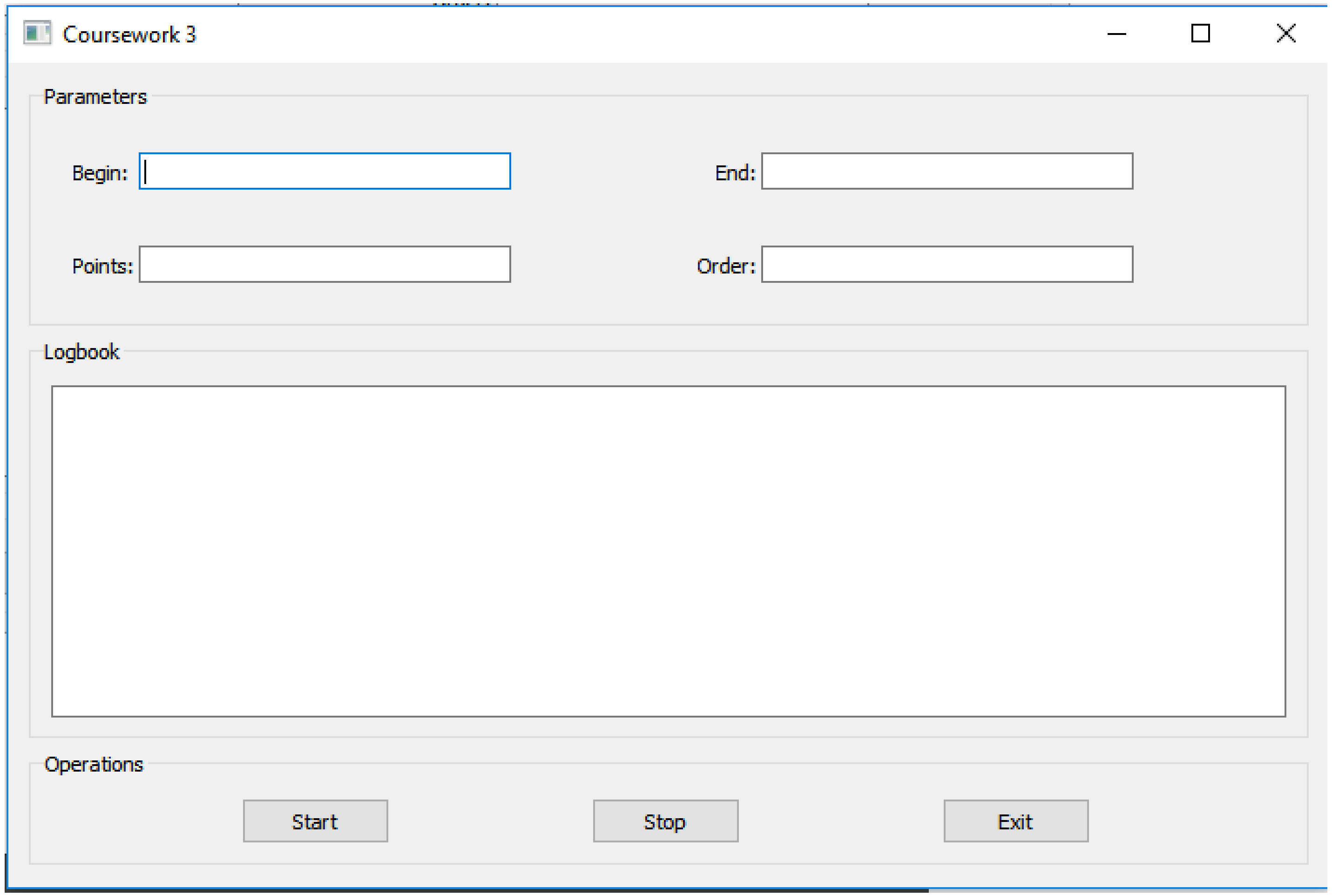
It is also possible to remove a setting from registry or to clear all the current settings:

```
settings_name.remove(key);
settings_name.clear();
```

Layouts (1)

A widget must automatically adjust itself to the changes of window size. Therefore just to drag a widget onto the main window box is not enough. We have to set the layouts.

Normally, the simple widgets like buttons or edit boxes are put into containers like group boxes or frames, for example as in QtCreator Example #5:



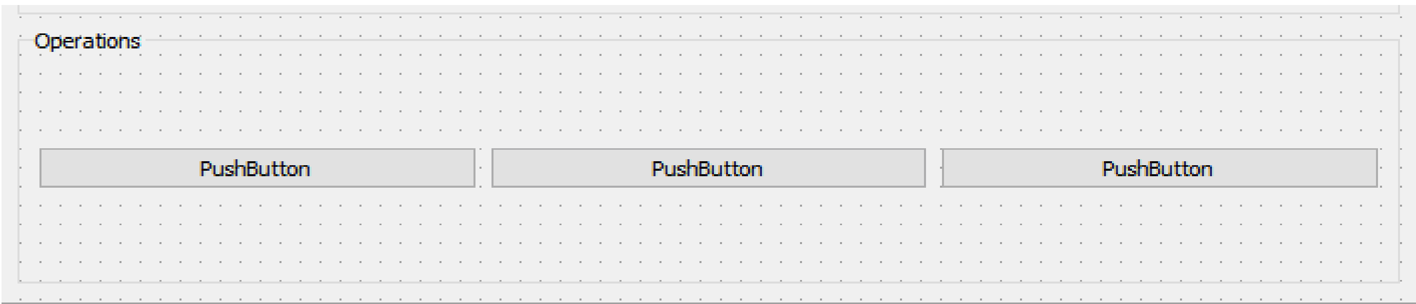
Layouts (2)

To start with GUI design, first take a piece of paper and draw a sketch. Then open the *mainwindow.ui* (double-click on the projects panel). Everything you see in the main window is actually located in the container called *centralWidget* (see the box on the right upper corner). By default the *centralWidget* contains widgets *menuBar* and *statusBar.* If your GUI does not need them, remove them (right-click and select the pop-up menu command *Remove*).

In our example the widgets are divided between three group boxes. So drag three group boxes from the palette into the central widget. Then right-click the main window and from pop-up menu select *Lay out → Lay out vertically*. You get three group boxes with equal dimensions covering the whole main window. Run the application and enlarge and shrink the main window: the group boxes will automatically adjust their sizes.

Click on the first group box and using the properties table (left lower corner) set the values for properties *objectName* and *title*. Do the same for the second and third group box.

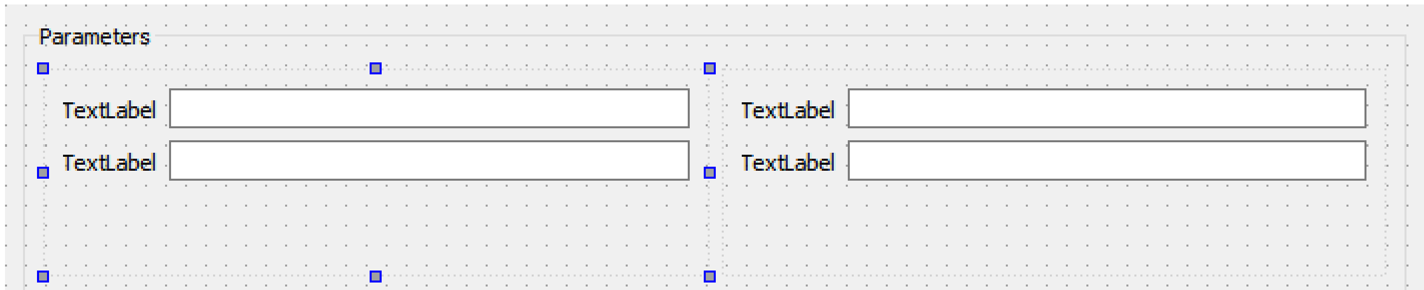
Now drag three buttons into the lower group box. Right-click the group box and from pop-up menu select *Lay out → Lay out horizontally*. You get three buttons located side by side. They cover the whole width of their container:



Layouts (3)

If we enlarge or shrink the main window, the buttons will automatically adjust their size. But we need buttons with fixed size. Click on a button and set values for properties *minimumSize* and *maximumSize* (the same values for both, for example 80 for width and 25 for height). Also set the values for properties *objectName* and *title*. Do the same for other two buttons too. Run the application: if you enlarge or shrink the main window, distances between buttons will enlarge or shrink also but the dimensions of buttons are kept.

Into the upper group box drag a frame. Then drag into the frame two line edit widgets and two label widgets. Right-click the frame and from pop-up menu select *Lay out →* *Lay out in a form layout*. In the same way create another frame. At last right-click the group box and from pop-up menu select *Lay out → Lay out horizontally*:



Run the application: distances between labels and line edit boxes does not change but the width of line edit boxes is adjusted. If you do not like it, set the maximum and / or minimum width to proper value.

Layouts (4)

For labels set their texts, for line edit boxes their names Then click the frame and adjust the layout properties: for example the *layoutHorizontalSpacing*, *layoutVerticalSpacing*, *layoutLabelAlignment*, etc.

At last drag a plain text edit widget into the middle group box and set its name. Set the group box layout to *horizontal*.

To set the main window title click on point not covered by group boxes and set the value for property *windowTitle*.

You should always first create the container and only after that put the widgets into it. If you do not follow this rule, the Qt designer may not be able to grab widgets because they are now lying under the container.

By default all the group boxes from the central widget have the same dimensions. To change it set new values for the central widget layout stretches.

More about layouts read https://doc.qt.io/qt-6/designer-layouts.html .

About application icons read https://doc.qt.io/qt-6/appicon.html

In QtCreator designer turn attention to widget called simply as "*widget*" It is unvisible but you may set its *minimumSize* and *maximumSize*. You may use it to insert padding between other widgets.

Third-party DLLs

Suppose your Qt widget application name is *TestDll* and its folder is *C:\TestDll*.The application uses *ThirdPartyDll.dll*. As we use implicit linking here, we need also *ThirdpartyDll.h* and *ThirdPartyDll.lib*. Then:

1. Create folder *C:\TestDll\ThirdPartyDll* and put *ThirdpartyDll.h* and *ThirdPartyDll.lib* (not *ThirdPartyDll.dll*) into it.
2. Open your project file *C:\TestDll\TestDll.pro* and add into it (for example after row *Forms += mainwindow.ui*):
   INCLUDEPATH += "ThirdPartyDll"
   LIBS += -L"c:\TestDll\ThirdPartyDll" –l "ThirdPartyDll"
3. Write you code. Calls to functions exported by third-party DLL are as calls to any other C++ function. Do not forget to add:
   #include "ThirdPartyDll.h"
   Use filenames as they are in Windows Explorer – do not change the case.
4. From *Build* menu select *Rebuild All*.
5. Put *ThirdPartyDll.dll* into *C:\TestDll\build\Desktop_Qt_6_9_3_64bit-Dedug*
6. Click the green arrow (left edge of QtCreater window) – the application should run.