Introduction to Qt

Third Party Libraries

Python has third party packages: numpy, pandas, matplotlib, requests, etc.

Python has environments: venv, conda, etc.

How does C++ implement similar things?

Third Party Libraries

The core idea of a “third party library” is that it’s some source code files (.py, .h, .cpp) which you include together with your program.

numpy defines a class **ndarray** which you use for computation.

requests defines a function **get()** which provides a simple interface for GET requests.

my_python_module defines a function **solve_problem_2()** which returns the solution to problem 2 of the contest.

C++ doesn’t have an official package manager.

There are some package managers, but another way to “install” libraries is to copy their code to the project folder and manage everything **manually**.

CMake helps manage these things.
MSYS2 and Homebrew help “install” packages in a more convenient way.

Third Party Libraries

C++ has many useful third party libraries. Some examples:

- Machine Learning
  - libtorch
  - tensorflow
- Images
  - OpenCV
  - Matplot++
- GUI
  - ImGUI
  - GTK
  - **Qt**

https://en.cppreference.com/w/cpp/links/libs

Qt

- Both a UI library and a general framework
  - QVariant as a multi-type value, QString, QList, etc.
- Advanced GUI design tools
  - Qt Designer, allows you to drag-and-drop buttons instead of writing all coordinates manually
- Cross-platform
- The de-facto standard

- Qt for Python - https://www.qt.io/qt-for-python

Qt Structure

- Qt Framework (qt-base) - basic necessary libraries and modules
  - Precompiled binaries
    - Allow you to connect Qt libraries to your project quickly, without recompiling the whole framework
  - Sources
    - Allow you to analyze sources to debug them and to set up IDE highlights better
- Docs
- Examples
- Tools
  - Qt Creator (IDE)
  - Qt Designer (GUI Design tool)
- Plugins

Qt Widgets vs Qt Quick

Qt Widgets

Everything in C++

Older (stable and known)

Focused on desktop

Focus of this course

Qt Quick

Some parts in QML (markup language)

Newer (new paradigms)

Mobile-friendly

Creating a Qt Widgets Based Application with CMake

Use this template for Qt projects - https://github.com/dsba-z/qt-empty-project
Qt Creator makes these templates automatically.

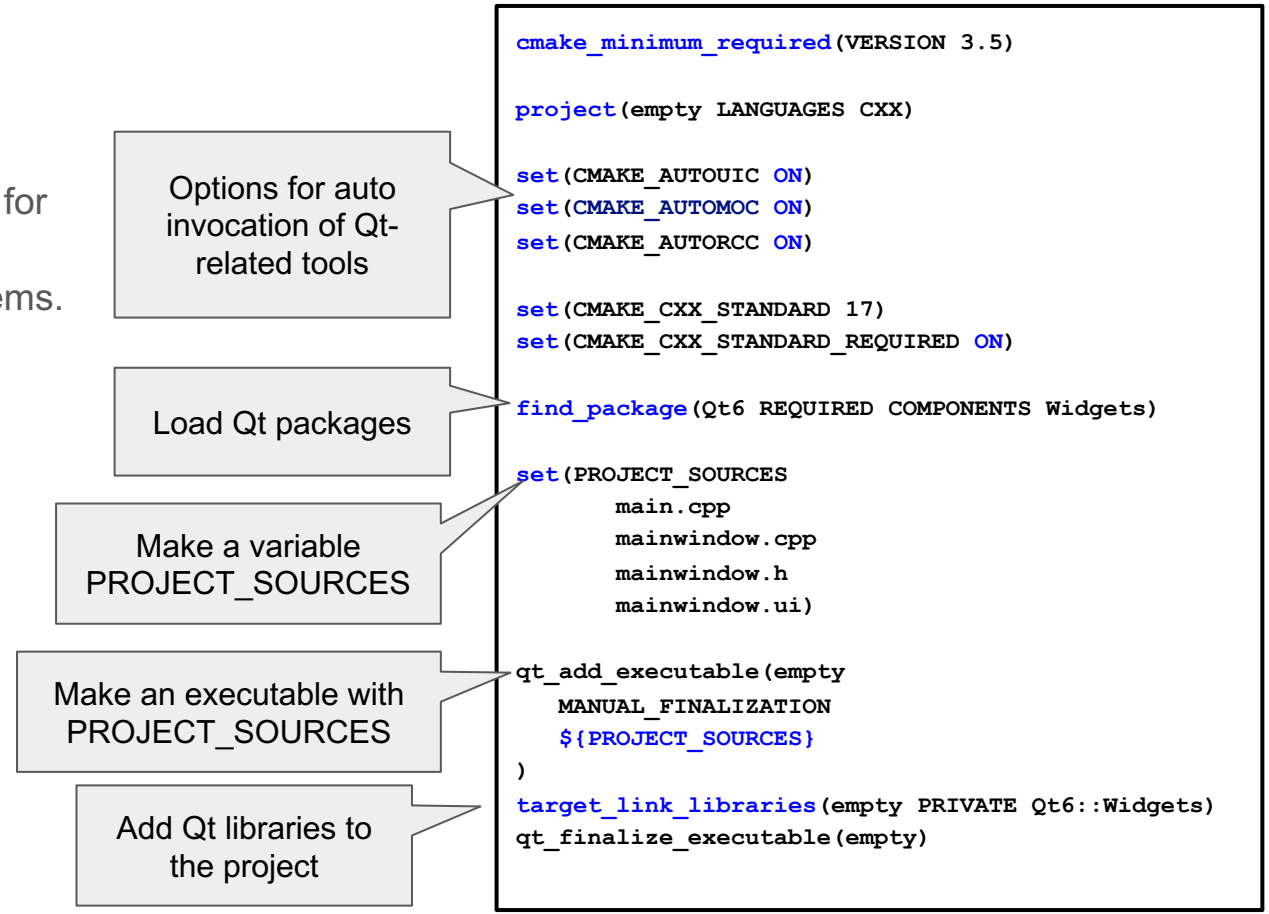
The template consists of the following components:

- CMakeLists.txt
- main.cpp
- Design file mainwindow.ui
- class MainWindow

CMakeLists.txt

The template contains code for compatibility between Qt versions (5 and 6) and systems.

Without it, a simpler CMakeLists.txt is sufficient.



https://doc.qt.io/qt-6/cmake-get-started.html

CMakeLists.txt (compatibility)

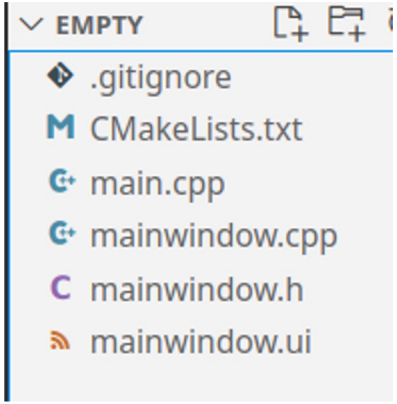
```
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS
Widgets)

if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(empty
        MANUAL_FINALIZATION
        ${PROJECT_SOURCES}
    )
# Comment
else()
    if(ANDROID)
        add_library(empty SHARED
            ${PROJECT_SOURCES}
        )
# Comment
    else()
        add_executable(empty
            ${PROJECT_SOURCES}
        )
    endif()
endif()
```

```
target_link_libraries(empty PRIVATE
Qt${QT_VERSION_MAJOR}::Widgets)

set_target_properties(empty PROPERTIES
    MACOSX_BUNDLE_GUI_IDENTIFIER my.example.com
    MACOSX_BUNDLE_BUNDLE_VERSION ${PROJECT_VERSION}
    MACOSX_BUNDLE_SHORT_VERSION_STRING
${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}
    MACOSX_BUNDLE TRUE
    WIN32_EXECUTABLE TRUE
)

install(TARGETS empty
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR})

if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(empty)
endif()
```

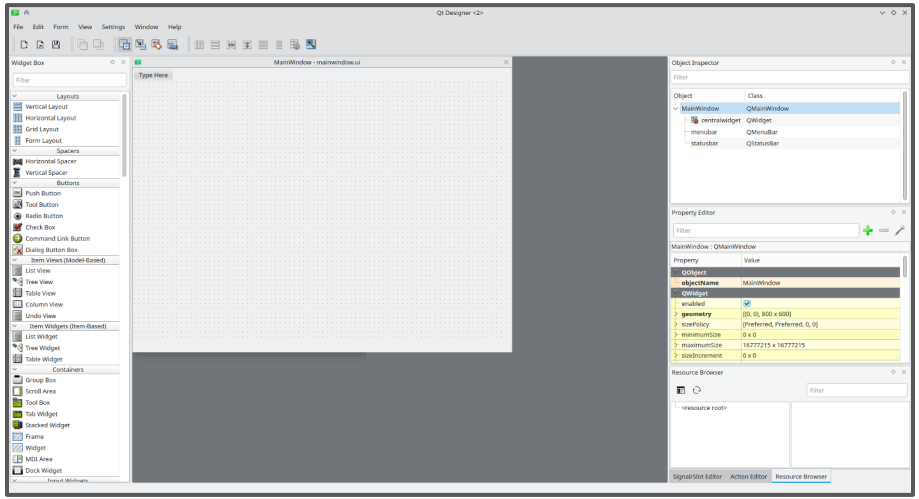
https://doc.qt.io/qt-6/cmake-qt5-and-qt6-compatibility.html

Class MainWindow

- Class MainWindow has three files that define it:
  - mainwindow.ui
    - defines UI in a XML-based format
  - mainwindow.h - header.
    - defines class MainWindow derived from QMainWindow
    - includes Q_OBJECT macro, constructor, destructor and the UI field
  - mainwindow.cpp - source file
    - defines the constructor (setting up the base class QMainWindow and the ui object)
    - defined the destructor (“delete” corresponds to the “new” from the constructor)



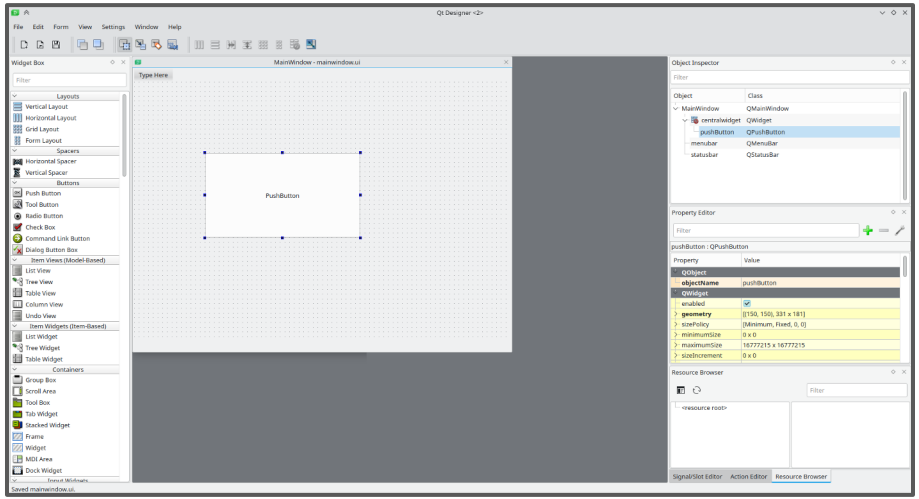
Design file mainwindow.ui



Editable with Qt Designer

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
 <property name="geometry">
  <rect>
   <x>0</x>
   <y>0</y>
   <width>800</width>
   <height>600</height>
  </rect>
 </property>
 <property name="windowTitle">
  <string>MainWindow</string>
 </property>
 <widget class="QWidget" name="centralwidget"/>
 <widget class="QMenuBar" name="menubar">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>800</width>
    <height>30</height>
   </rect>
  </property>
 </widget>
 <widget class="QStatusBar" name="statusbar"/>
</widget>
<resources/>
<connections/>
</ui>
```

Design file mainwindow.ui



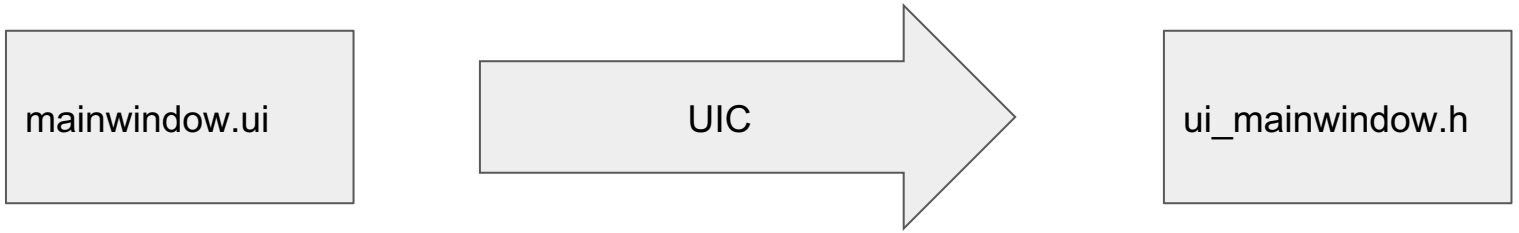
Editable with Qt Designer

Changes are reflected in the file and the generated code

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
 <property name="geometry">
  <rect>
   <x>0</x>
   <y>0</y>
   <width>800</width>
   <height>600</height>
  </rect>
 </property>
 <property name="windowTitle">
  <string>MainWindow</string>
 </property>
 <widget class="QWidget" name="centralwidget">
  <widget class="QPushButton" name="pushButton">
   <property name="geometry">
    <rect>
     <x>150</x>
     <y>150</y>
     <width>331</width>
     <height>181</height>
    </rect>
   </property>
   <property name="text">
    <string>PushButton</string>
   </property>
  </widget>
 </widget>
 <widget class="QMenuBar" name="menubar">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>800</width>
    <height>30</height>
   </rect>
  </property>
 </widget>
 <widget class="QStatusBar" name="statusbar"/>
</widget>
<resources/>
<connections/>
</ui>
```

User Interface Compiler (uic)

- Qt Designer UI files represent the widget tree as XML
- During the build process they are compiled into a regular C++ file by UIC



This is why when you include this file in your code, you should write “ui_mainwindow.h”.

Ctrl+Click the header to see the generated file.

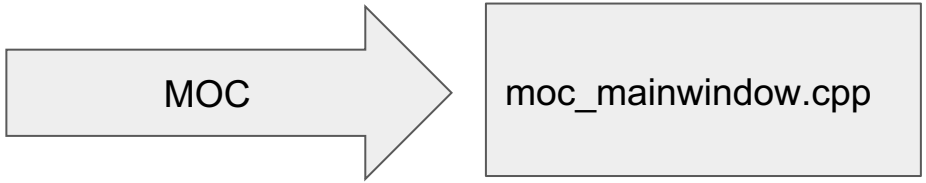
https://doc.qt.io/qt-6/designer-using-a-ui-file.html

Qt Meta-Object System

Qt's meta-object system provides the signals and slots mechanism for inter-object communication, runtime type information, and the dynamic property system.

1. The QObject class provides a base class for objects that can take advantage of the meta-object system.
2. The Q_OBJECT macro inside the private section of the class declaration is used to enable meta-object features, such as dynamic properties, signals, and slots.
3. The Meta-Object Compiler (moc) supplies each QObject subclass with the necessary code to implement meta-object features.

mainwindow.cpp	slots
Q_OBJECT	signals



https://doc.qt.io/qt-6/metaobjects.html

Class MainWindow

- **#ifndef** - so you don't include the same file twice
- **namespace Ui** - to keep UI names separate
- **{class MainWindow; }** - forward declaration. Like a function prototype, but for classes
- Class **MainWindow** inherits from **QMainWindow**
- macro **Q_OBJECT** for MOC (in the **private** section)
- constructor takes another widget as parent
  - related to the widget tree
- constructor takes another widget as parent
- destructor
- **ui** pointer implementing the idiom Pimpl
  - implementation of UI is in a separate file to prevent recompilation when just the UI changes

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private:
    Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

Class MainWindow

- Include the header of this class
- Include the generated UI class
  - not included in mainwindow.h!
  - that was only a “prototype”
- Constructor takes an argument “**parent** QWidget”
  - This is a UI element (even a window)
  - It’s a part of some other UI element, a container
  - Which one? (nullptr)
- Base class **QMainWindow** is initialized
- Field **ui** is initialized
  - and the setup function is called
- Destructor removes UI elements

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

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

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

Class Ui_MainWindow

- Automatically generated
- Includes all necessary libraries for present widgets
- Uses Qt name space macro
- UI elements are class members (pointers)
- Objects are created in setupUi
- Initial properties (from Qt Designer) are set in setupUi
- Class is separated into its own namespace, so you don’t confuse it with your own objects

```
// warning
#ifndef UI_MAINWINDOW_H
#define UI_MAINWINDOW_H

#include <QtCore/QVariant>
#include <QtWidgets/QApplication>
#include <QtWidgets/QMainWindow>
#include <QtWidgets/QMenuBar>
#include <QtWidgets/QStatusBar>
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:
    QWidget *centralwidget;
    QMenuBar *menubar;
    QStatusBar *statusbar;

    void setupUi(QMainWindow *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName("MainWindow");
        MainWindow->resize(800, 600);
        centralwidget = new QWidget(MainWindow);
...
};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_MAINWINDOW_H
```

main.cpp

- include main window
  - everything else is included in the main window itself
- include QApplication as the only required module
- process input arguments
- create the application object **a**
- create the MainWindow object **w**
  - default constructor, no parent
- show the main window
- run (execute) the application
- return the application's return code

```
#include "mainwindow.h"

#include <QApplication>

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

Next time

Qt basics and objects

Types of windows

Layouts

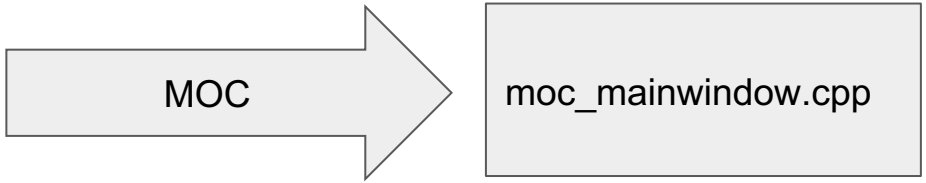
Qt. Project.

Qt Meta-Object System

Qt's meta-object system provides the signals and slots mechanism for inter-object communication, runtime type information, and the dynamic property system.

1. The QObject class provides a base class for objects that can take advantage of the meta-object system.
2. The Q_OBJECT macro inside the private section of the class declaration is used to enable meta-object features, such as dynamic properties, signals, and slots.
3. The Meta-Object Compiler (moc) supplies each QObject subclass with the necessary code to implement meta-object features.

mainwindow.cpp	slots
Q_OBJECT	signals



https://doc.qt.io/qt-6/metaobjects.html

Class MainWindow

- **#ifndef** - so you don't include the same file twice
- **namespace Ui** - to keep UI names separate
- **{class MainWindow; }** - forward declaration. Like a function prototype, but for classes
- Class **MainWindow** inherits from **QMainWindow**
- macro **Q_OBJECT** for MOC (in the **private** section)
- constructor takes another widget as parent
  - related to the widget tree
- constructor takes another widget as parent
- destructor
- **ui** pointer implementing the idiom Pimpl
  - implementation of UI is in a separate file to prevent recompilation when just the UI changes

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
   Q_OBJECT

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

private:
   Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

Class MainWindow

- Include the header of this class
- Include the generated UI class
  - not included in mainwindow.h!
  - that was only a “prototype”
- Constructor takes an argument “**parent** QWidget”
  - This is a UI element (even a window)
  - It’s a part of some other UI element, a container
  - Which one? (nullptr)
- Base class **QMainWindow** is initialized
- Field **ui** is initialized
  - and the setup function is called
- Destructor removes UI elements

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

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

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

Class Ui_MainWindow

- Automatically generated
- Includes all necessary libraries for present widgets
- Uses Qt name space macro
- UI elements are class members (pointers)
- Objects are created in setupUi
- Initial properties (from Qt Designer) are set in setupUi
- Class is separated into its own namespace, so you don’t confuse it with your own objects

```
// warning
#ifndef UI_MAINWINDOW_H
#define UI_MAINWINDOW_H

#include <QtCore/QVariant>
#include <QtWidgets/QApplication>
#include <QtWidgets/QMainWindow>
#include <QtWidgets/QMenuBar>
#include <QtWidgets/QStatusBar>
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:
    QWidget *centralwidget;
    QMenuBar *menubar;
    QStatusBar *statusbar;

    void setupUi(QMainWindow *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName("MainWindow");
        MainWindow->resize(800, 600);
        centralwidget = new QWidget(MainWindow);
...
};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_MAINWINDOW_H
```

main.cpp

- include main window
  - everything else is included in the main window itself
- include QApplication as the only required module
- process input arguments
- create the application object **a**
- create the MainWindow object **w**
  - default constructor, no parent
- show the main window
- run (execute) the application
- return the application's return code

```
#include "mainwindow.h"

#include <QApplication>

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

This time

Qt basics and objects

Types of windows

Layouts

“Main” Window Types

- QMainWindow
  - designed to be a main window of an application
  - has a menu bar, a status bar, a tool bar, can have docked widgets
  - **best**
- QWidget
  - base class of all widgets and windows
  - doesn’t have anything predefined but is extremely **customizable**
  - suitable, but hard
- QDialog
  - window based on QWidget
  - has special dialog functionality (built-in buttons, can be modal)
  - not suitable

What is next after an empty project?

- Input
  - Buttons
  - Text fields
  - Selection boxes
- Output
  - Text fields
  - Labels
  - Files
- Logic
  - Slots
  - QObject, QString, QList

More new problems

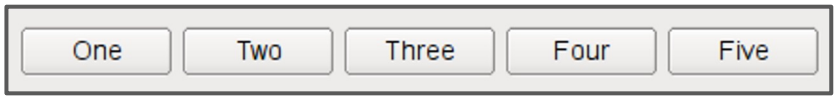
Layouts

No layout

- Widgets are placed at precise coordinates
- Nothing is adjusted automatically
- Simple to start, hard to continue using

1D layout: QHBoxLayout, QVBoxLayout

- Widgets are placed in a single row or column
- Very simple and intuitive
- Can be nested (row of columns)



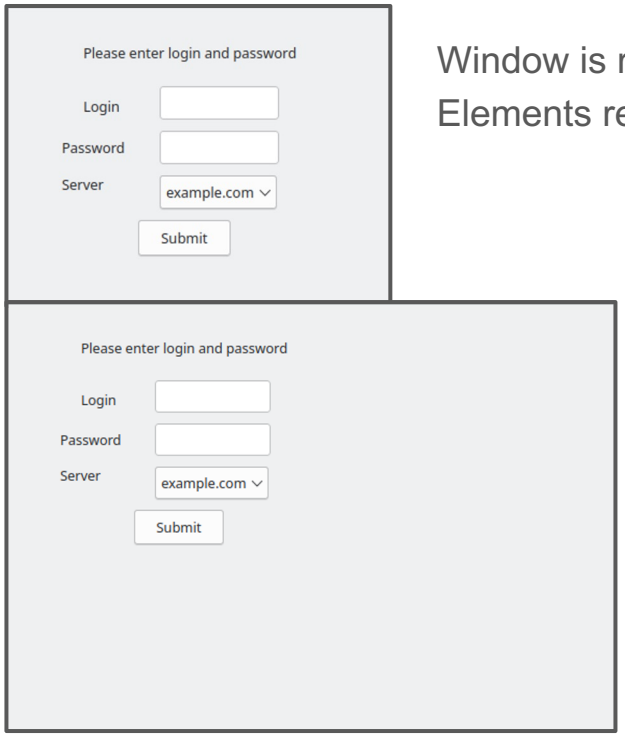
QGridLayout

- Widgets are placed in a two-dimensional grid
- Less intuitive, allows complex configurations

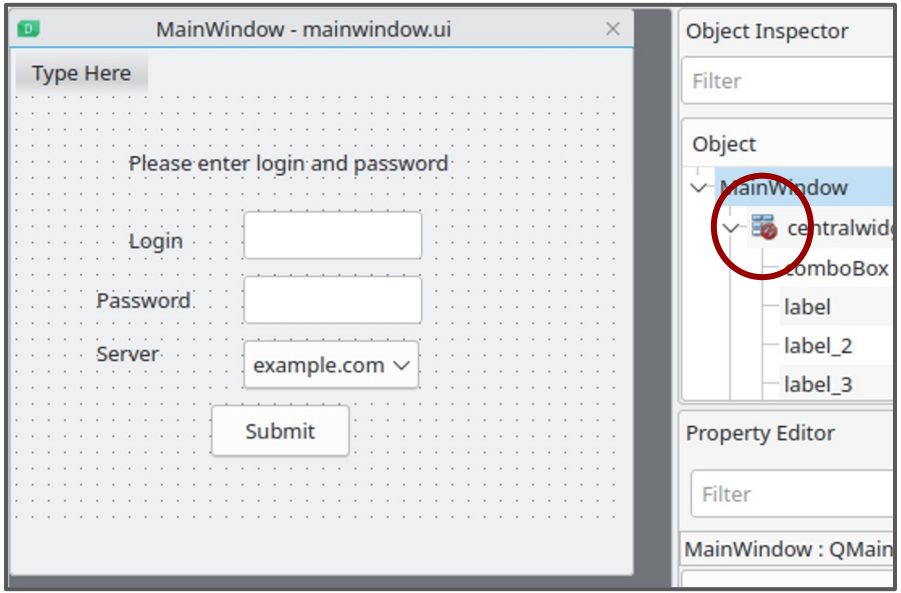
QFormLayout

- Widgets are placed in a 2-column label-field style

No Layout

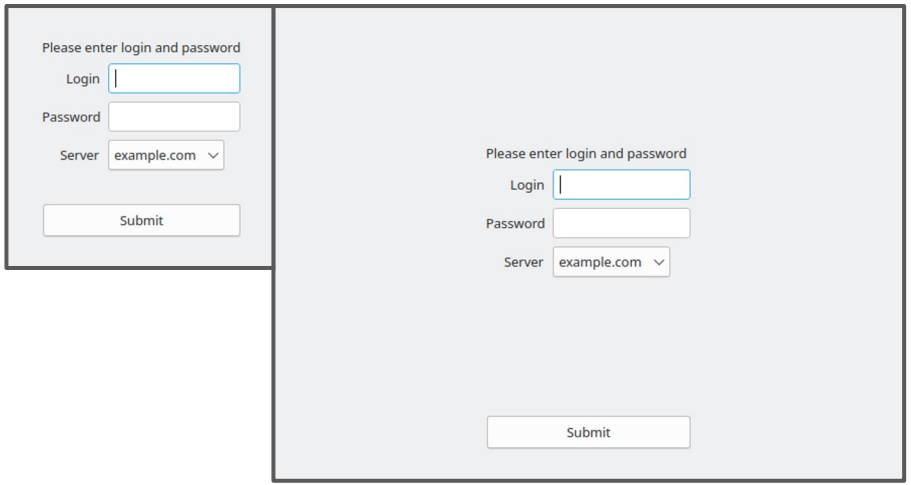


Window is resized
Elements remain in place



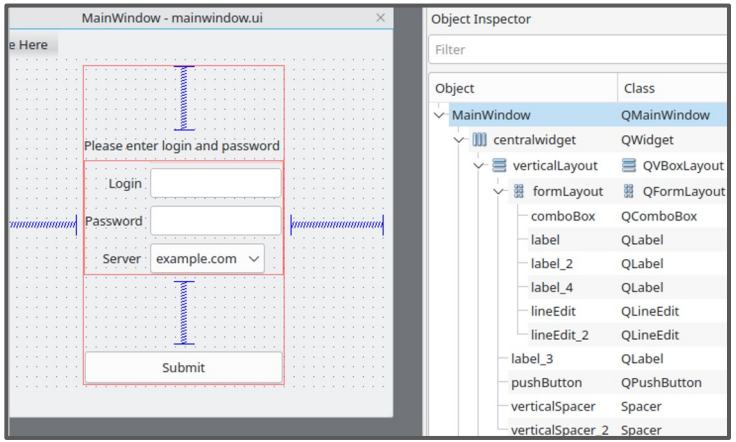
No layout selected

Layouts



Window is resized.

UI elements remain in the center.



Nested layouts and spacers allow UI elements to be placed precisely at any window size.

To set main window layout, right-click “MainWindow” and select a layout.

Adding Logic To Widgets

- Elements added to the UI file in Qt Designer will appear inside the **ui** object
- Some operations, like reading properties, are available right away
  - Read documentation about the UI element to learn how to get its data
  - You can also browse properties in Qt Designer
  - All properties have the same function names:
    - propery() - reading
    - setProperty() - writing
- Operations requiring user input are implemented using the **signal and slot** system

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"
#include <QDebug>

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

    QString labelText = ui->label->text();
    qInfo() << labelText;
}

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

Qt Modules

- **QtCore**, a base library that provides containers, thread management, event management, and much more
- **QtGui** and **QtWidgets**, a GUI toolkit for Desktop, that provides a lot of graphical components to design applications.
- **QtNetwork** for network communications
- **QtWebkit**, the webkit engine, enables the use of web pages and web apps in a Qt application.
- **QtSQL**, a full featured SQL RDBM abstraction layer, support for some databases is available out of the box
- Other (QtXML, QtXmlPatterns)

Qt Core Features

Standard C++:

- Fast, efficient
- Inflexible, rigid. Too static in some problem domains
- Built-in solutions (inheritance, polymorphism, heap-memory) are hard to use

GUI requires:

- Speed
- Runtime flexibility (closing tabs, opening files, loading data)
- Preferably easier to use solutions

Qt Core Features

- seamless object communication with signals and slots
- queryable and designable object properties
- events and event filters
- contextual string translation for internationalization (i18n)
- hierarchical and queryable object trees that organize object **ownership**
- guarded pointers (QPointer) that are automatically set to `nullptr` when the referenced object is destroyed
- etc

Ownership and communication

Ownership

- An object is created in the heap
- Initially you only have a pointer

- This pointer may be moved, value transferred and copied
- Have to keep track of pointers and whether the object is still needed
  - Maybe smart pointers
- Have to keep track of the **data** behind the pointers - which pointers provide access to (or **own**) the data.

Communication

- An object is created in the heap
- You only have a pointer

- To send “messages” to different objects you need to keep track of all the pointers
- A “message” is a function call
  - Button is pressed - submit info
  - Data is changed - update widget
- Having access to everything at all times makes the program more confusing and error-prone (like if you never use *const*)

QObject

The QObject class is the base class of all Qt objects

- Helps solve ownership by organizing objects in object trees
- Helps solve communication using signals and slots

QObject doesn’t have a copy constructor or copy assignment operator - each object has only one instance which can’t be transferred.

Object Trees and Ownership

- When a QObject is created, a parent object may be added to it
  - this adds the object to the list of the parent's children()
  - when the parent is deleted, the child is deleted, too
  - "if you close a tab, you automatically delete all widgets inside that tab"
- Creation/deletion order matters!
  - The parent must be initialized before the child

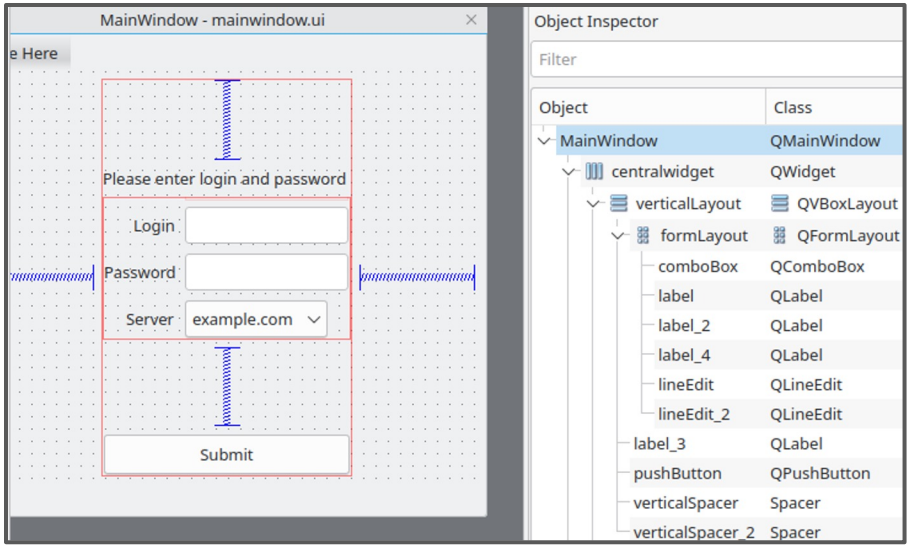
```
int main()
{
    QWidget window;
    QPushButton quit("Quit", &window);
    ...
}
```

```
int main()
{
    QPushButton quit("Quit", &window);
    QWidget window;
    ...
}
```

Object Tree: Example

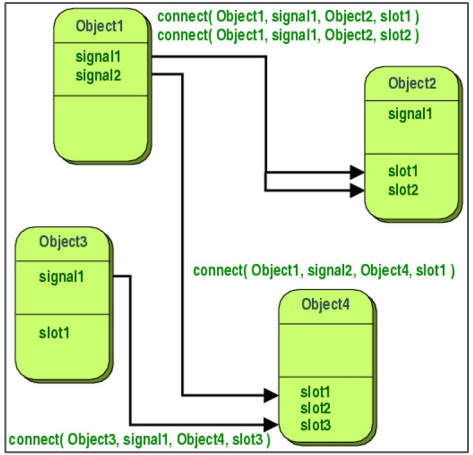
- The layouts act as parents to their child objects
- Other container widgets act this way, too

The tree structure is useful not only for object deletion, but for organization, as well



Signals and Slots

- Signals and slots are used for communication between objects
- Components of GUI applications have many interactions for each action
  - a button is clicked - a dialog is shown
  - a dialog window is opened - its controls are initialized
  - a data view widget is scrolled to a new position - new data is loaded and displayed
- An alternative communication system is *callbacks*: calling functions directly
  - in function button.onClick() - call showDialog()
  - in dialog.open() - call controls.initialize()
  - etc.
- Signals are better because they are more flexible

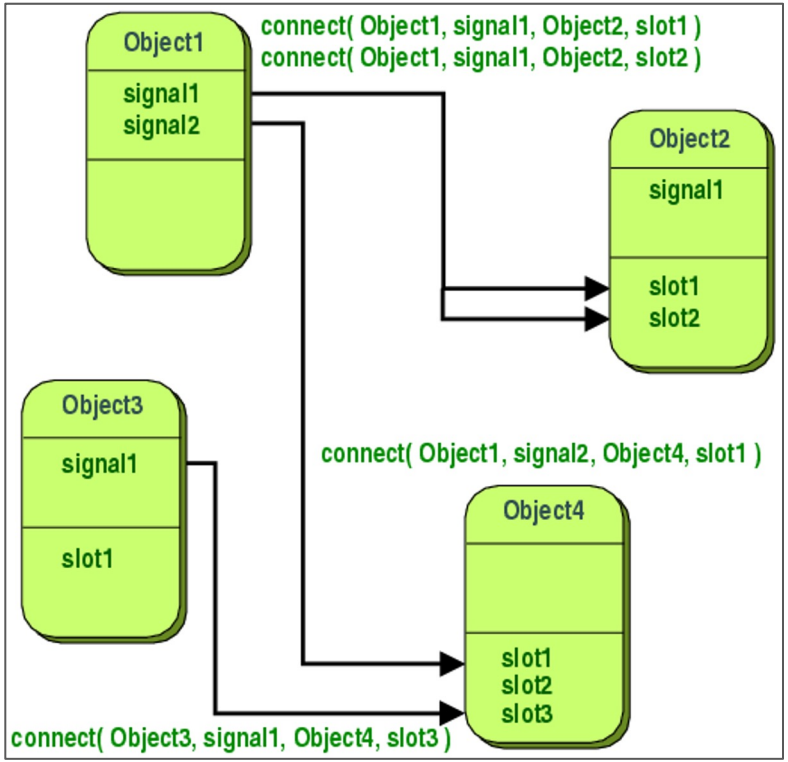


Signals and slots

A **signal** is **emitted** when a particular event occurs.

A **slot** is a function that is **called** in response to a particular signal.

A single signal can be connected to multiple slots.



Signals and Slots: Function Signatures

- Signals and slots are type safe:
  - The signature of a signal must match the signature of the receiving slot
  - A slot may drop some arguments and have a shorter signature
    - Extra arguments are ignored

You will mostly define **slots**.

```
signals:
   void clicked();
```

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

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

```
signals:
   void clicked(int);
```

```
private slots:
   void on_buttonAdd_clicked(bool);
```

Connecting Signals to Slots

There are severals ways how you can connect signals to slots

1. Function name
2. QObject::connect()

Qt Designer has UI tools that let you automatically generate code connecting signals to slots

```
signals:
   void clicked();
```

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

```
   QObject::connect(_model, &QStringListModel::dataChanged, _reminders, &Reminders::doMyAction);
```

Next time

Creating widgets

Loading data

More on signals and slots

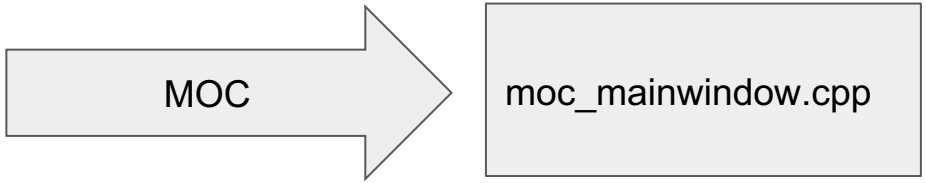
Qt. Project. Layouts. Signals.

Qt Meta-Object System

Qt's meta-object system provides the signals and slots mechanism for inter-object communication, runtime type information, and the dynamic property system.

1. The QObject class provides a base class for objects that can take advantage of the meta-object system.
2. The Q_OBJECT macro inside the private section of the class declaration is used to enable meta-object features, such as dynamic properties, signals, and slots.
3. The Meta-Object Compiler (moc) supplies each QObject subclass with the necessary code to implement meta-object features.

mainwindow.cpp	slots
Q_OBJECT	signals



https://doc.qt.io/qt-6/metaobjects.html

Class MainWindow

- **#ifndef** - so you don't include the same file twice
- **namespace Ui** - to keep UI names separate
- **{class MainWindow; }** - forward declaration. Like a function prototype, but for classes
- Class **MainWindow** inherits from **QMainWindow**
- macro **Q_OBJECT** for MOC (in the **private** section)
- constructor takes another widget as parent
  - related to the widget tree
- constructor takes another widget as parent
- destructor
- **ui** pointer implementing the idiom Pimpl
  - implementation of UI is in a separate file to prevent recompilation when just the UI changes

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
   Q_OBJECT

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

private:
   Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

Class MainWindow

- Include the header of this class
- Include the generated UI class
  - not included in mainwindow.h!
  - that was only a “prototype”
- Constructor takes an argument “**parent** QWidget”
  - This is a UI element (even a window)
  - It’s a part of some other UI element, a container
  - Which one? (nullptr)
- Base class **QMainWindow** is initialized
- Field **ui** is initialized
  - and the setup function is called
- Destructor removes UI elements

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

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

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

Class Ui_MainWindow

- Automatically generated
- Includes all necessary libraries for present widgets
- Uses Qt name space macro
- UI elements are class members (pointers)
- Objects are created in setupUi
- Initial properties (from Qt Designer) are set in setupUi
- Class is separated into its own namespace, so you don’t confuse it with your own objects

```
// warning
#ifndef UI_MAINWINDOW_H
#define UI_MAINWINDOW_H

#include <QtCore/QVariant>
#include <QtWidgets/QApplication>
#include <QtWidgets/QMainWindow>
#include <QtWidgets/QMenuBar>
#include <QtWidgets/QStatusBar>
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:
    QWidget *centralwidget;
    QMenuBar *menubar;
    QStatusBar *statusbar;

    void setupUi(QMainWindow *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName("MainWindow");
        MainWindow->resize(800, 600);
        centralwidget = new QWidget(MainWindow);
...
};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_MAINWINDOW_H
```

main.cpp

- include main window
  - everything else is included in the main window itself
- include QApplication as the only required module
- process input arguments
- create the application object **a**
- create the MainWindow object **w**
  - default constructor, no parent
- show the main window
- run (execute) the application
- return the application's return code

```
#include "mainwindow.h"

#include <QApplication>

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

This time

Qt basics and objects

Types of windows

Layouts

“Main” Window Types

- QMainWindow
  - designed to be a main window of an application
  - has a menu bar, a status bar, a tool bar, can have docked widgets
  - **best**
- QWidget
  - base class of all widgets and windows
  - doesn’t have anything predefined but is extremely **customizable**
  - suitable, but hard
- QDialog
  - window based on QWidget
  - has special dialog functionality (built-in buttons, can be modal)
  - not suitable

What is next after an empty project?

- Input
  - Buttons
  - Text fields
  - Selection boxes
- Output
  - Text fields
  - Labels
  - Files
- Logic
  - Slots
  - QObject, QString, QList

More new problems

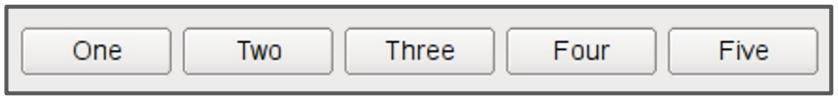
Layouts

No layout

- Widgets are placed at precise coordinates
- Nothing is adjusted automatically
- Simple to start, hard to continue using

1D layout: QHBoxLayout, QVBoxLayout

- Widgets are placed in a single row or column
- Very simple and intuitive
- Can be nested (row of columns)



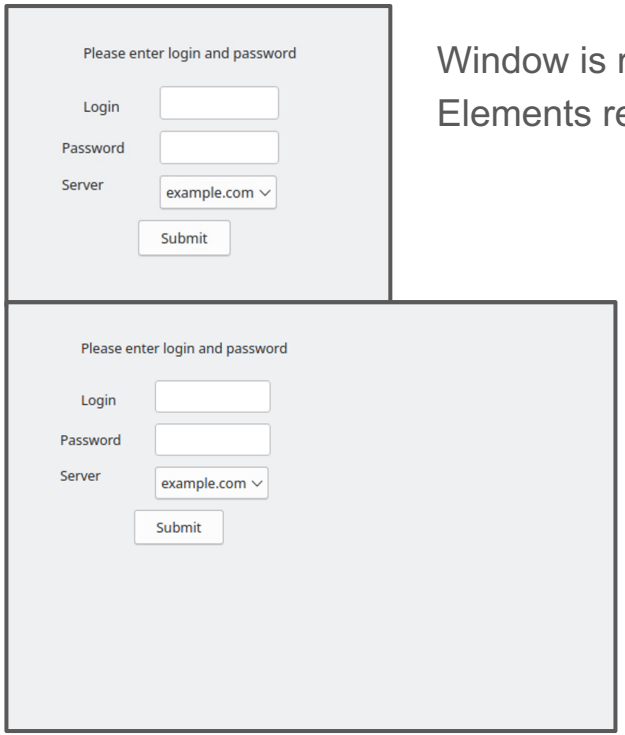
QGridLayout

- Widgets are placed in a two-dimensional grid
- Less intuitive, allows complex configurations

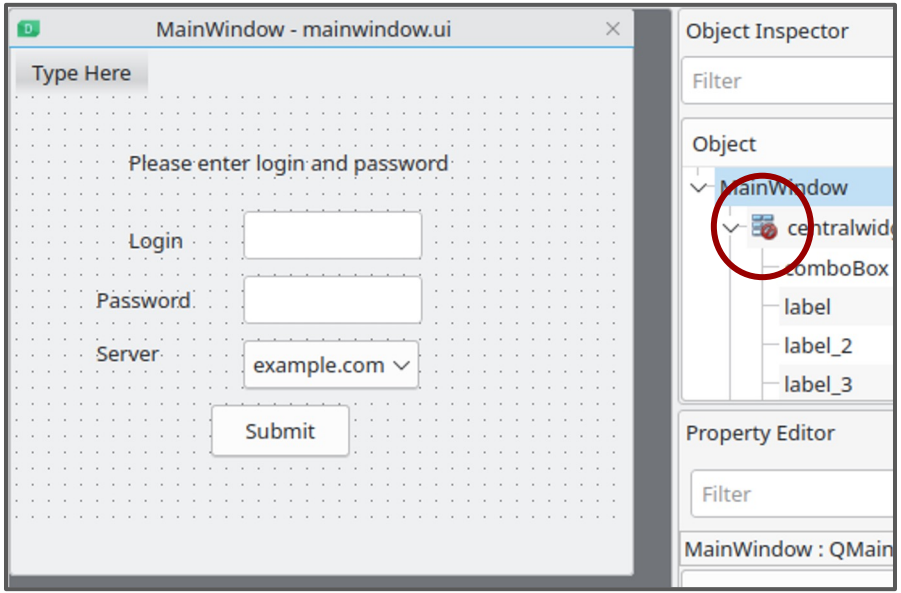
QFormLayout

- Widgets are placed in a 2-column label-field style

No Layout

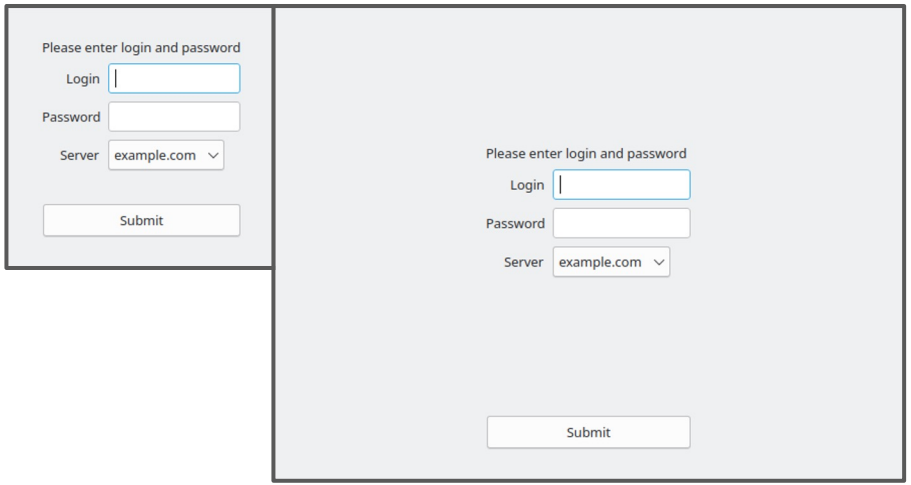


Window is resized
Elements remain in place



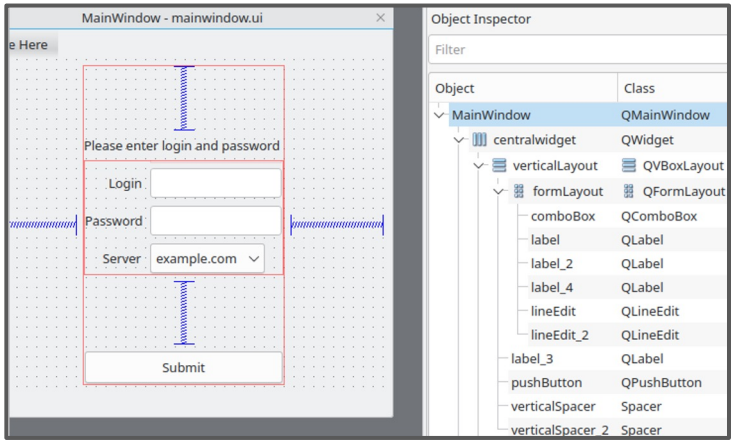
No layout selected

Layouts



Window is resized.

UI elements remain in the center.



Nested layouts and spacers allow UI elements to be placed precisely at any window size.

To set main window layout, right-click “MainWindow” and select a layout.

Adding Logic To Widgets

- Elements added to the UI file in Qt Designer will appear inside the **ui** object
- Some operations, like reading properties, are available right away
  - Read documentation about the UI element to learn how to get its data
  - You can also browse properties in Qt Designer
  - All properties have the same function names:
    - propery() - reading
    - setProperty() - writing
- Operations requiring user input are implemented using the **signal and slot** system

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"
#include <QDebug>

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

    QString labelText = ui->label->text();
    qInfo() << labelText;
}

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

Qt Modules

- **QtCore**, a base library that provides containers, thread management, event management, and much more
- **QtGui** and **QtWidgets**, a GUI toolkit for Desktop, that provides a lot of graphical components to design applications.
- **QtNetwork** for network communications
- **QtWebkit**, the webkit engine, enables the use of web pages and web apps in a Qt application.
- **QtSQL**, a full featured SQL RDBM abstraction layer, support for some databases is available out of the box
- Other (QtXML, QtXmlPatterns)

Qt Core Features

Standard C++:

- Fast, efficient
- Inflexible, rigid. Too static in some problem domains
- Built-in solutions (inheritance, polymorphism, heap-memory) are hard to use

GUI requires:

- Speed
- Runtime flexibility (closing tabs, opening files, loading data)
- Preferably easier to use solutions

Qt Core Features

- seamless object communication with signals and slots
- queryable and designable object properties
- events and event filters
- contextual string translation for internationalization (i18n)
- hierarchical and queryable object trees that organize object **ownership**
- guarded pointers (QPointer) that are automatically set to `nullptr` when the referenced object is destroyed
- etc

Ownership and communication

Ownership

- An object is created in the heap
- Initially you only have a pointer

- This pointer may be moved, value transferred and copied
- Have to keep track of pointers and whether the object is still needed
  - Maybe smart pointers
- Have to keep track of the **data** behind the pointers - which pointers provide access to (or **own**) the data.

Communication

- An object is created in the heap
- You only have a pointer

- To send “messages” to different objects you need to keep track of all the pointers
- A “message” is a function call
  - Button is pressed - submit info
  - Data is changed - update widget
- Having access to everything at all times makes the program more confusing and error-prone (like if you never use *const*)

QObject

The QObject class is the base class of all Qt objects

- Helps solve ownership by organizing objects in object trees
- Helps solve communication using signals and slots

QObject doesn’t have a copy constructor or copy assignment operator - each object has only one instance which can’t be transferred.

Object Trees and Ownership

- When a QObject is created, a parent object may be added to it
  - this adds the object to the list of the parent's children()
  - when the parent is deleted, the child is deleted, too
  - "if you close a tab, you automatically delete all widgets inside that tab"
- Creation/deletion order matters!
  - The parent must be initialized before the child

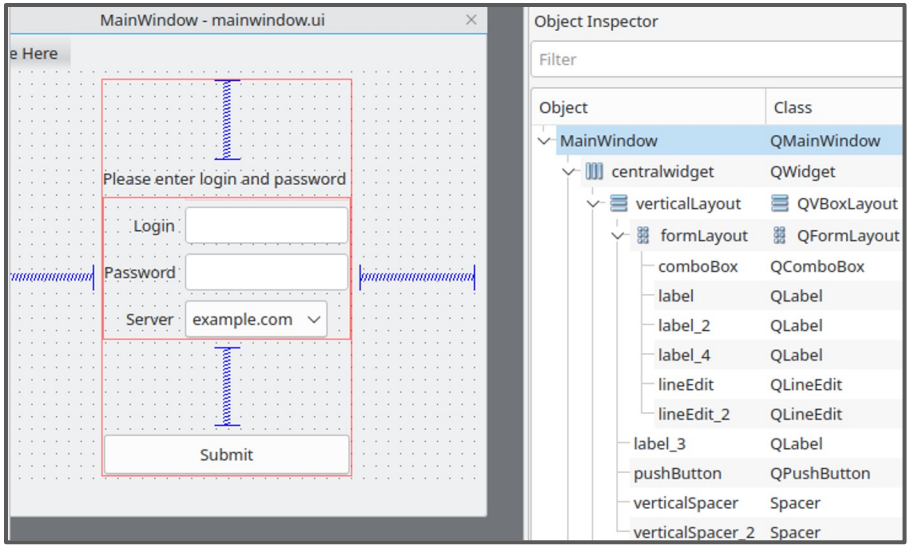
```
int main()
{
    QWidget window;
    QPushButton quit("Quit", &window);
    ...
}
```

```
int main()
{
    QPushButton quit("Quit", &window);
    QWidget window;
    ...
}
```

Object Tree: Example

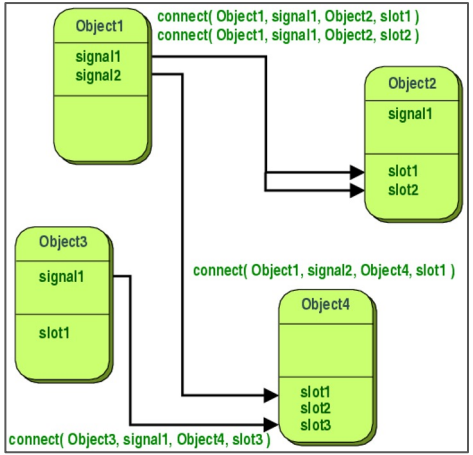
- The layouts act as parents to their child objects
- Other container widgets act this way, too

The tree structure is useful not only for object deletion, but for organization, as well



Signals and Slots

- Signals and slots are used for communication between objects
- Components of GUI applications have many interactions for each action
  - a button is clicked - a dialog is shown
  - a dialog window is opened - its controls are initialized
  - a data view widget is scrolled to a new position - new data is loaded and displayed
- An alternative communication system is *callbacks*: calling functions directly
  - in function button.onClick() - call showDialog()
  - in dialog.open() - call controls.initialize()
  - etc.
- Signals are better because they are more flexible

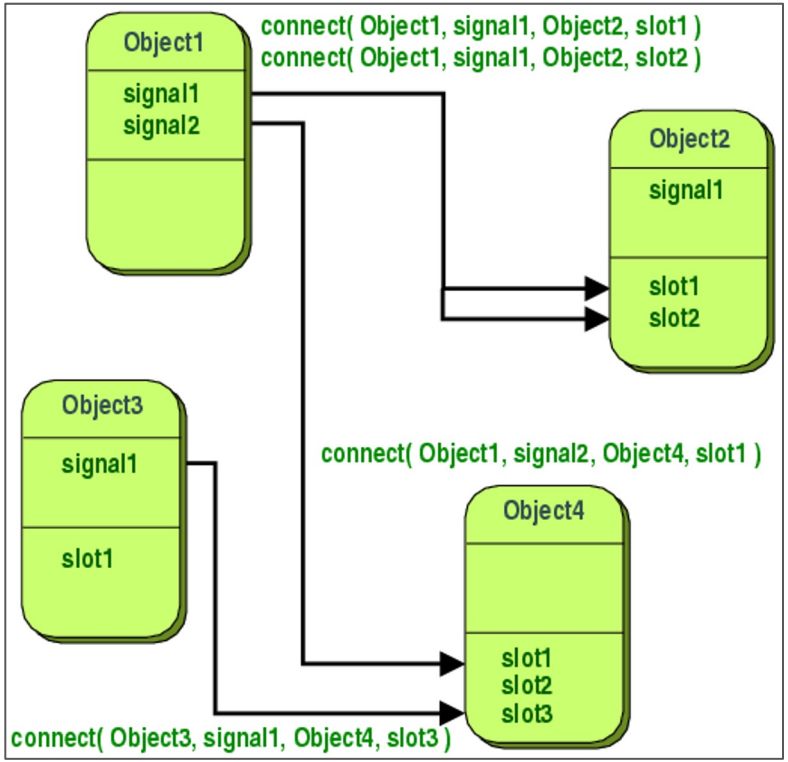


Signals and slots

A **signal** is **emitted** when a particular event occurs.

A **slot** is a function that is **called** in response to a particular signal.

A single signal can be connected to multiple slots.



Signals and Slots: Function Signatures

- Signals and slots are type safe:
  - The signature of a signal must match the signature of the receiving slot
  - A slot may drop some arguments and have a shorter signature
    - Extra arguments are ignored

You will mostly define **slots**.

```
signals:
   void clicked();
```

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

Auto connection

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

Manual connection

```
signals:
   void clicked(int);
```

```
private slots:
   void on_buttonAdd_clicked(bool);
```

Signature mismatch

Connecting Signals to Slots

There are severals ways how you can connect signals to slots

1. Function name
2. QObject::connect()

Qt Designer has UI tools that let you automatically generate code connecting signals to slots

```
signals:
   void clicked();
```

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

```
   QObject::connect(_model, &QStringListModel::dataChanged, _reminders, &Reminders::doMyAction);
```

Signals

- Signals are emitted by an object when its internal state has changed
- When a signal is emitted, the slots connected to it are usually executed immediately

```
class Reminders : public QObject
{
    Q_OBJECT
public:
    explicit Reminders(QStringList reminderList);
    ~Reminders();

signals:
    void fileOpen();
...
};
```

```
void Reminders::openFile(const QString& filePath)
{
    _fileName = fileName;
    readFile();
    _active = true;
    ...
    emit fileOpen();
    ...
}
```

Now all slots connected to the signal **`Reminders::fileOpen`** will be executed.
So, you can add functions “when the reminder file is opened, do this”.

Slots

- A slot is called when a signal connected to it is emitted.
- Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them.

```
class Reminders : public QObject
{
   Q_OBJECT
public:
   explicit Reminders(QStringList
reminderList);
   ~Reminders();

public slots:
   void update();
```

`update()` is a regular function
now it is connected to the signal
`QStringListModel::dataChanged`.

Meaning “when data changes in the object **_model**, update object **_reminders**”.

```
   QObject::connect(_model, &QStringListModel::dataChanged, _reminders, &Reminders::update);
```

Next time

Creating widgets

Loading data

More on signals and slots

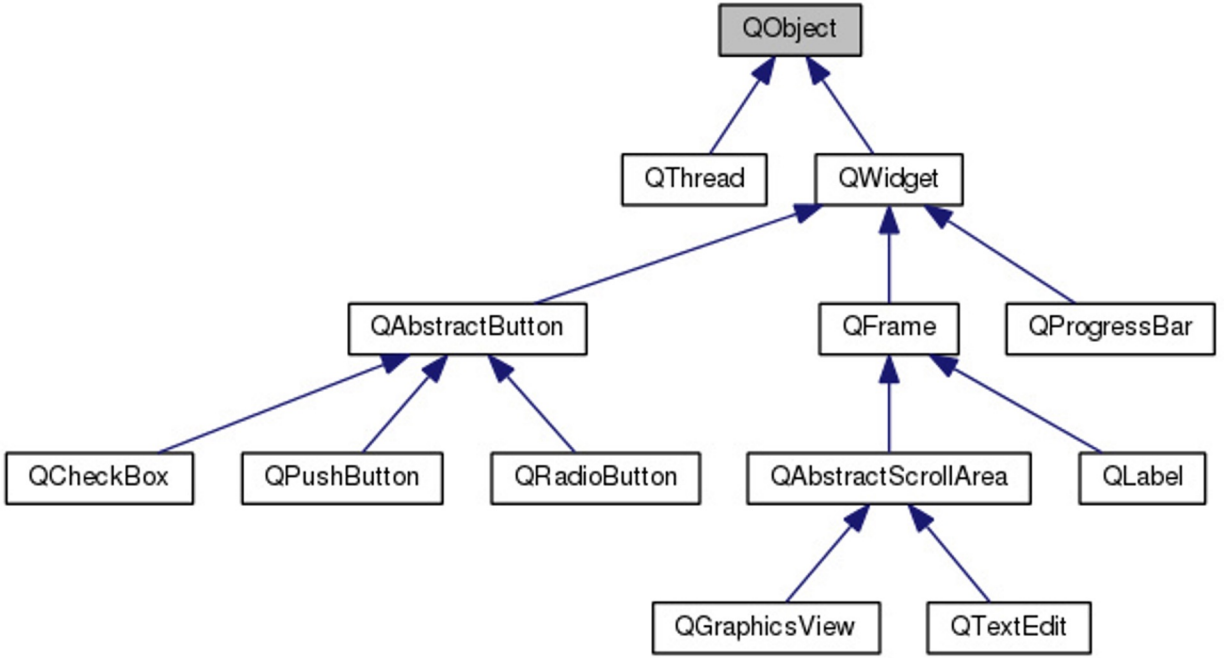
Qt Classes.

Qt Classes

- Core (QObject derivatives)
  - QString, QList, QVariant, QVector, etc
- Widgets (QWidget derivatives)
  - QLabel, QTextEdit, QPushButton, etc
- Other
  - XML readers, styles, pointers, images

1000+ classes in total

Qt Classes



https://wiki.qt.io/Qt_for_Beginners

Implicit Sharing Approach

- Resource sharing is used to minimize resource copying
- Reference counting based approach:
  - a shared class consists of a pointer to a resource
  - counts number of “users” (“references”)
  - similar to *shared pointers* from the STL
- There are two ways of copying an object:

fast
  - **shallow copy** is getting a pointer to an existing object, increment counter of “references”
    - `operator=` makes a shallow copy of an object
    - copying a pointer/reference

slow
  - **deep copy** clones an object, creates an independent instance
    - when an object is about to change, it's “detached” from its “original” (only if the reference counter is > 1)

<- means passing objects by value is cheap

QString

The **`QString`** class provides a Unicode character string;

- a sequence of 16-bit **`QChar`**-s (similar to **char16_t**),
  - each corresponds to a UTF-16 code unit;
  - Unicode characters with code values above 65535 are stored using surrogate pairs.
- Uses *implicit sharing*:
  - can be easily passed by value, but it might be better to still use const&
- Can be null, which is different from empty ← unique feature compared to std::string.
- Initializing:
  - passing **const char*** to a constructor
  - **fromUtf8(data)** static method (deep copy)
  - **fromRawData(data)** static method (shallow copy)
  - **fromStdString(string)** static method (uses fromUtf8())

```
QString s("Text goes here");
```

- Converting to a C/C++-string:
  - s.toLocal8Bit().constData()
  - s.toStdString()

Types for Representing Characters (Reminder)

Qt uses **UTF-16**: 2 or 4 bytes per character.

std::string uses char - 1 byte per character.

std::wstring uses 2 bytes - closer to Qt

Conversion requires reencoding:

- **qs.toUtf8()**
- **qs.toLocal8Bit()**

qs.toStdString() uses **qs.toUft8()**

Type	Size	Range	Codepages (examples)
char	8 bit	0..255	Fixed width: ASCII (ext), CP1251, KOI8-r Variable width: UTF-8
char16_t	16 bit	0..65535	Fixed width: UCS-2 “Variable width” (surrogate pairs): UTF-16
char32_t	32 bit	0.. 4294967295	UTF-32
wchar_t	16 or 32 bit		UCS-2
char8_t (since C++20)	8 bit		UTF-8

QFile

- **QFile** provides an interface for reading from and writing to files
  - Input/Output device interface for reading and writing text and binary files and resources.
- Can be used by itself or with:
  - **QTextStream** class providing a convenient interface for reading and writing text, similar to C++ streams
  - **QFileDevice** class providing an interface for reading from and writing to open files
- Setting a file name:
  - in a constructor;
  - by calling the setFileName() method.
- Preparing:
  - open(), close(), flush() for flushing buffer.

QFile Example - Reading Simple CSV Files

- Set file path in the constructor
- Open the file and select modes
  - **ReadOnly** and **Text** in this case
- Connect a text stream to the file
- Use **stream.readLine()** to read lines
- Use **string.split(",")** to split rows into cells
- Don't forget to **close the file** at the end!

This method doesn't handle extra commas, quotation marks and newline characters.

It shows basic QFile/QTextStream interactions. For more complex readers, refer to the StackOverflow link below.

```
void ExampleModel::fillDataTableFromFile(QString path)
{
   QFile inputFile(path);
   if (!inputFile.open(QFile::ReadOnly | QFile::Text))
      throw std::runtime_error("Could not open file")

   QTextStream inputStream(&inputFile);

   QString firstline = inputStream.readLine();

   while(!inputStream.atEnd())
   {
       QString line = inputStream.readLine();

       QList<QString> dataRow;
       for (const QString& item : line.split(",")) {
           dataRow.append(item);
       }
       dataTable.append(dataRow);
   }
   inputFile.close();
}
```

https://stackoverflow.com/questions/27318631/parsing-through-a-csv-file-in-qt
https://github.com/dsba-z/week18cpp2021-modelview/blob/213-1/src/ex_modelview/examplemodel.cpp

Class QVariant

- **QVariant** acts like a *union* for the most common Qt data types.
- A **QVariant** object holds a single value of a single **type()** at a time.
- Getting a value interpreted as a specific type T via **to*T*()** methods:
  - toInt();
  - toString();
  - …
- Getting the stored value converted to the template type T:
  - value();
  - canConvert() determines whether a type can be converted.

Basic Types	Qt Types	GUI Types
int, uint, bool, const char*, double, …	QObject, QString, QChar, QList, QMap, QDate, ...	QColor, QImage, QPen, QBrush, QPixmap ...

Containers (1)

QList<T>	This is by far the most commonly used container class. It stores a list of values of a given type (T) that can be accessed by index. Internally, it stores an array of values of a given type at adjacent positions in memory. Inserting at the front or in the middle of a list can be quite slow, because it can lead to large numbers of items having to be moved by one position in memory.
QVarLengthArray<T, Prealloc>	This provides a low-level variable-length array. It can be used instead of QList in places where speed is particularly important.
QStack<T>	This is a convenience subclass of QList that provides "last in, first out" (LIFO) semantics. It adds the following functions to those already present in QList: push(), pop(), and top().
QQueue<T>	This is a convenience subclass of QList that provides "first in, first out" (FIFO) semantics. It adds the following functions to those already present in QList: enqueue(), dequeue(), and head().

https://doc.qt.io/qt-6/containers.html#the-container-classes

Containers (2)

QSet<T>	This provides a single-valued mathematical set with fast lookups.
QMap<Key, T>	This provides a dictionary (associative array) that maps keys of type Key to values of type T. Normally each key is associated with a single value. QMap stores its data in Key order; if order doesn't matter QHash is a faster alternative.
QMultiMap<Key, T>	This is a convenience subclass of QMap that provides a nice interface for multi-valued maps, i.e. maps where one key can be associated with multiple values.
QHash<Key, T>	This has almost the same API as QMap, but provides significantly faster lookups. QHash stores its data in an arbitrary order.
QMultiHash<Key, T>	This is a convenience subclass of QHash that provides a nice interface for multi-valued hashes.

https://doc.qt.io/qt-6/containers.html#the-container-classes

GUI + Qt Classes. Example. Text Finder.

As you can see, Qt Code library has basically all functions and classes needed to create programs. It has so much that you can **write programs using exclusively Qt features**.

How does this work in practice?

Qt has a good collection of tutorials and examples you can use to learn about building applications.

This one uses the tutorial here - https://doc.qt.io/qtcreator/creator-writing-program.html

Code

old slides next (for reference)

Object Trees and Ownership

- When a QObject is created, a parent object may be added to it
  - this adds the object to the list of the parent's children()
  - when the parent is deleted, the child is deleted, too
  - "if you close a tab, you automatically delete all widgets inside that tab"
- Creation/deletion order matters!
  - The parent must be initialized before the child

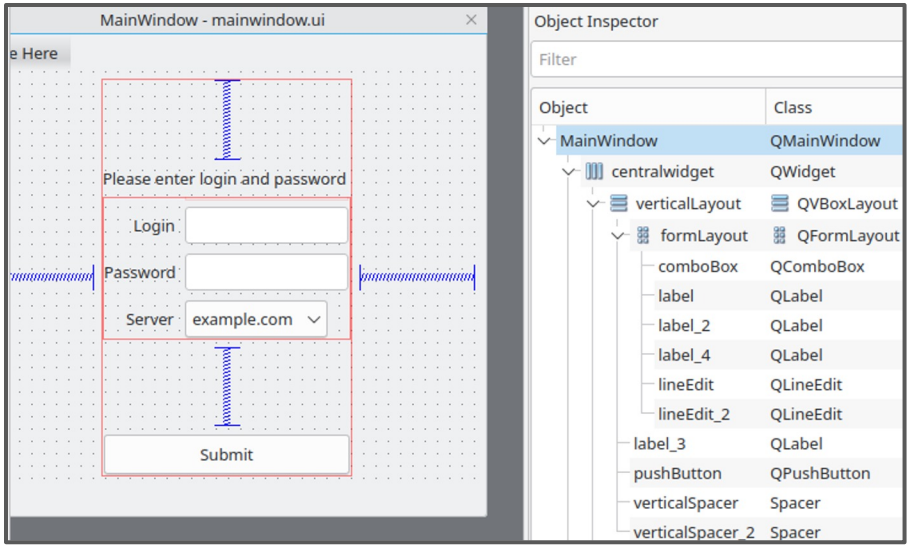
```
int main()
{
    QWidget window;
    QPushButton quit("Quit", &window);
    ...
}
```

```
int main()
{
    QPushButton quit("Quit", &window);
    QWidget window;
    ...
}
```

Object Tree: Example

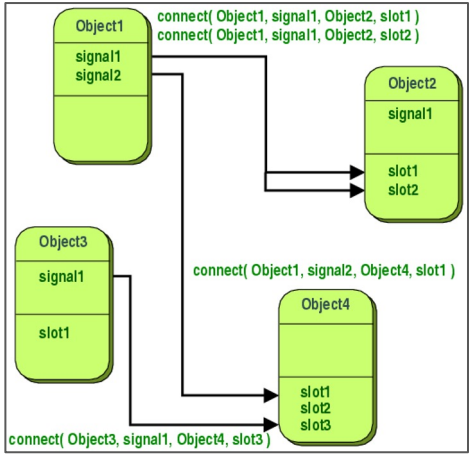
- The layouts act as parents to their child objects
- Other container widgets act this way, too

The tree structure is useful not only for object deletion, but for organization, as well



Signals and Slots

- Signals and slots are used for communication between objects
- Components of GUI applications have many interactions for each action
  - a button is clicked - a dialog is shown
  - a dialog window is opened - its controls are initialized
  - a data view widget is scrolled to a new position - new data is loaded and displayed
- An alternative communication system is *callbacks*: calling functions directly
  - in function button.onClick() - call showDialog()
  - in dialog.open() - call controls.initialize()
  - etc.
- Signals are better because they are more flexible

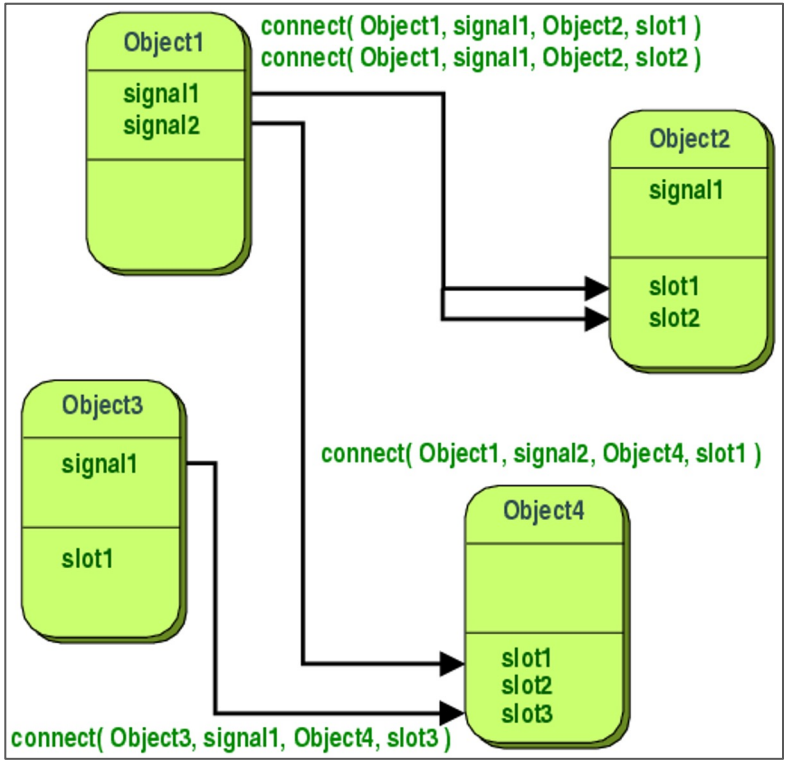


Signals and slots

A **signal** is **emitted** when a particular event occurs.

A **slot** is a function that is **called** in response to a particular signal.

A single signal can be connected to multiple slots.



Signals and Slots: Function Signatures

- Signals and slots are type safe:
  - The signature of a signal must match the signature of the receiving slot
  - A slot may drop some arguments and have a shorter signature
    - Extra arguments are ignored

You will mostly define **slots**.

```
signals:
   void clicked();
```

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

Auto connection

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

Manual connection

```
signals:
   void clicked(int);
```

```
private slots:
   void on_buttonAdd_clicked(bool);
```

Signature mismatch

Connecting Signals to Slots

There are severals ways how you can connect signals to slots

1. Function name
2. QObject::connect()

Qt Designer has UI tools that let you automatically generate code connecting signals to slots

```
signals:
   void clicked();
```

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

```
   QObject::connect(_model, &QStringListModel::dataChanged, _reminders, &Reminders::doMyAction);
```

Signals

- Signals are emitted by an object when its internal state has changed
- When a signal is emitted, the slots connected to it are usually executed immediately

```
class Reminders : public QObject
{
    Q_OBJECT
public:
    explicit Reminders(QStringList reminderList);
    ~Reminders();

signals:
    void fileOpen();
...
};
```

```
void Reminders::openFile(const QString& filePath)
{
    _fileName = fileName;
    readFile();
    _active = true;
    ...
    emit fileOpen();
    ...
}
```

Now all slots connected to the signal **`Reminders::fileOpen`** will be executed.
So, you can add functions “when the reminder file is opened, do this”.

Slots

- A slot is called when a signal connected to it is emitted.
- Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them.

```
class Reminders : public QObject
{
    Q_OBJECT
public:
    explicit Reminders(QStringList reminderList);
    ~Reminders();

public slots:
    void update();
```

`update()` is a regular function
now it is connected to the signal
`QStringListModel::dataChanged`.

Meaning “when data changes in the object **_model**, update object **_reminders**”.

```
    QObject::connect(_model, &QStringListModel::dataChanged, _reminders, &Reminders::update);
```

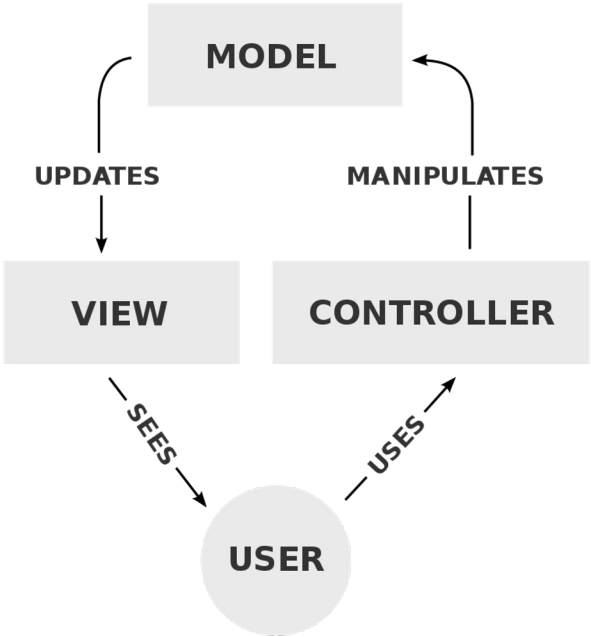
Next time

Model-View

Qt Models.

Model - View - Controller Design Pattern

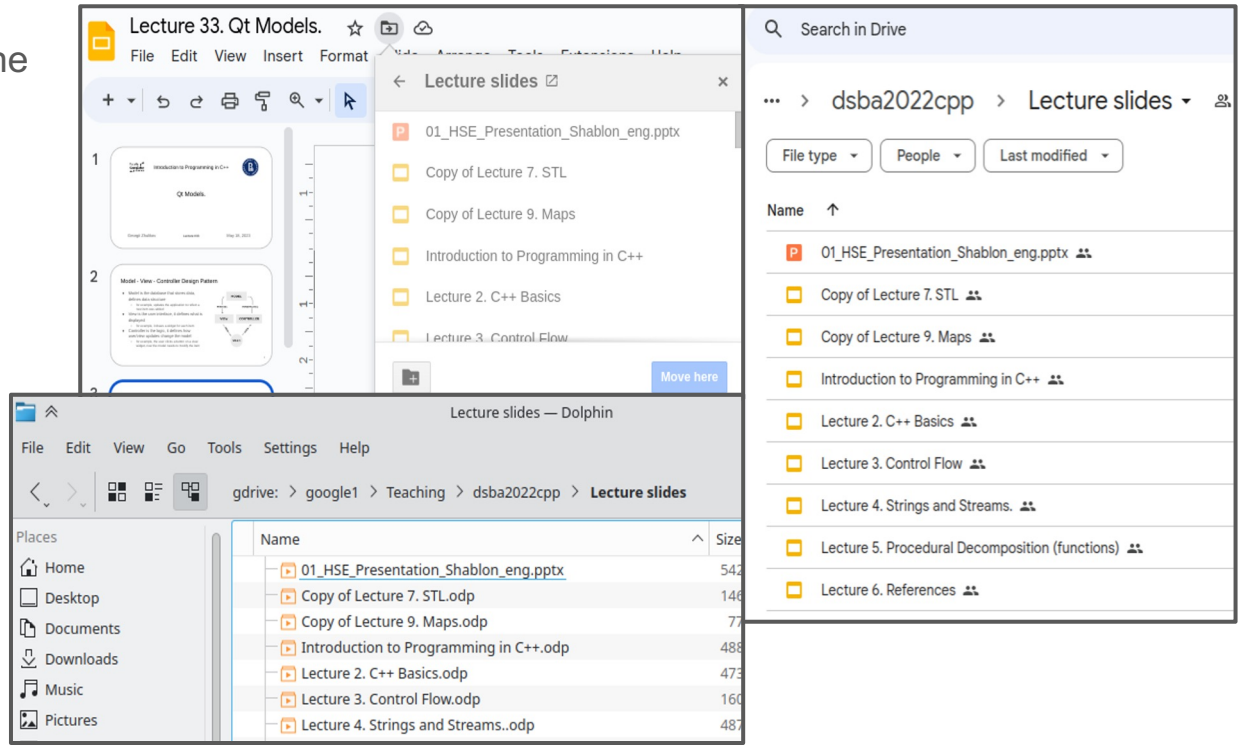
- Model is the database that stores data, defines data structure
  - for example, updates the application to reflect a new item was added
- View is the user interface, it defines what is displayed
  - for example, it draws a widget for each item
- Controller is the logic, it defines how user/view updates change the model
  - for example, the user clicks a button on a view widget, now the model needs to modify the item



MVC Web Example: Google Drive

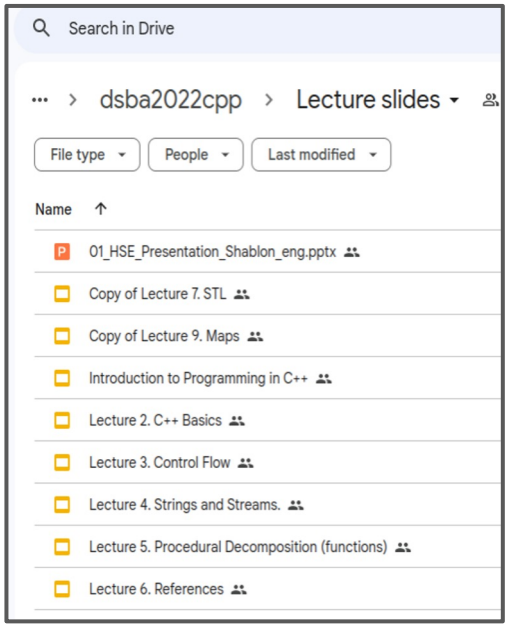
Different views have access to the same underlying file system with presentations.

Model here is the filesystem, the Google database that stores all the files.

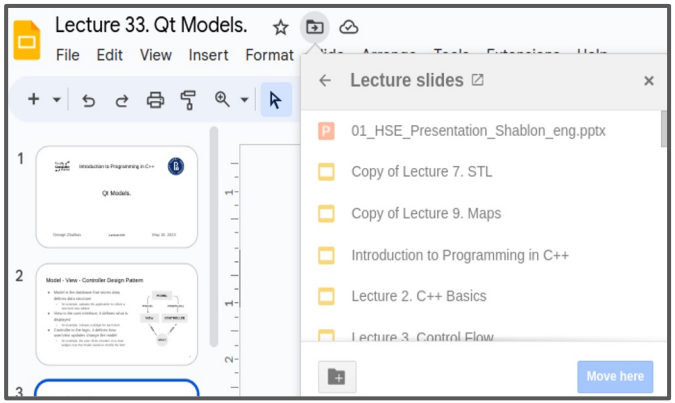


MVC Web Example: Google Drive

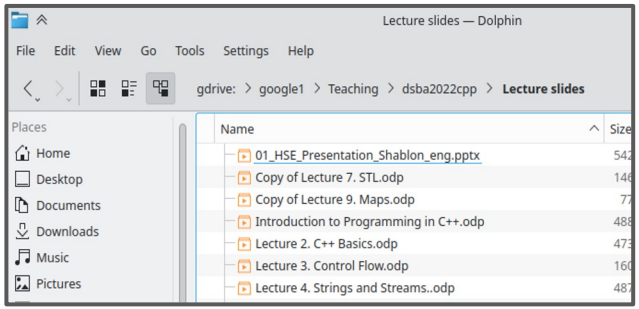
View 1: web interface



View 2: "Move to" pop-up



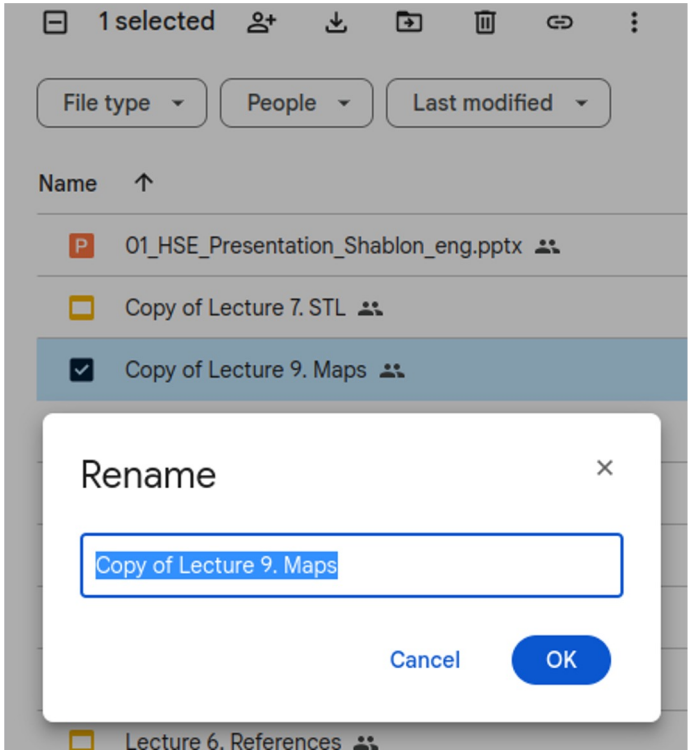
View 3: Local file manager



MVC Web Example: Google Drive

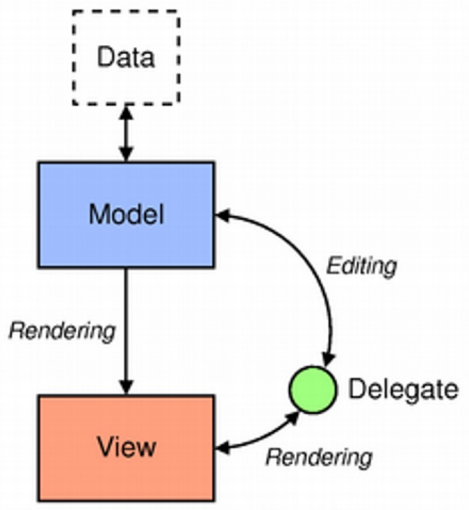
The controller is the code connecting view buttons (like “Rename”) and changes in data - renaming of files.

Note: Google Drive does not necessarily work as a regular MVC application. It’s probably more complex internally, but similar interface can be made with MVC.



Model-View in Qt

- No Controllers - they are combined with Views
  - table, list, tree views
  - delegates handle complex data input
- Many built-in model classes
  - QAbstractItemModel and its derivatives
  - Table model, File system model, etc



Model-View Tutorials

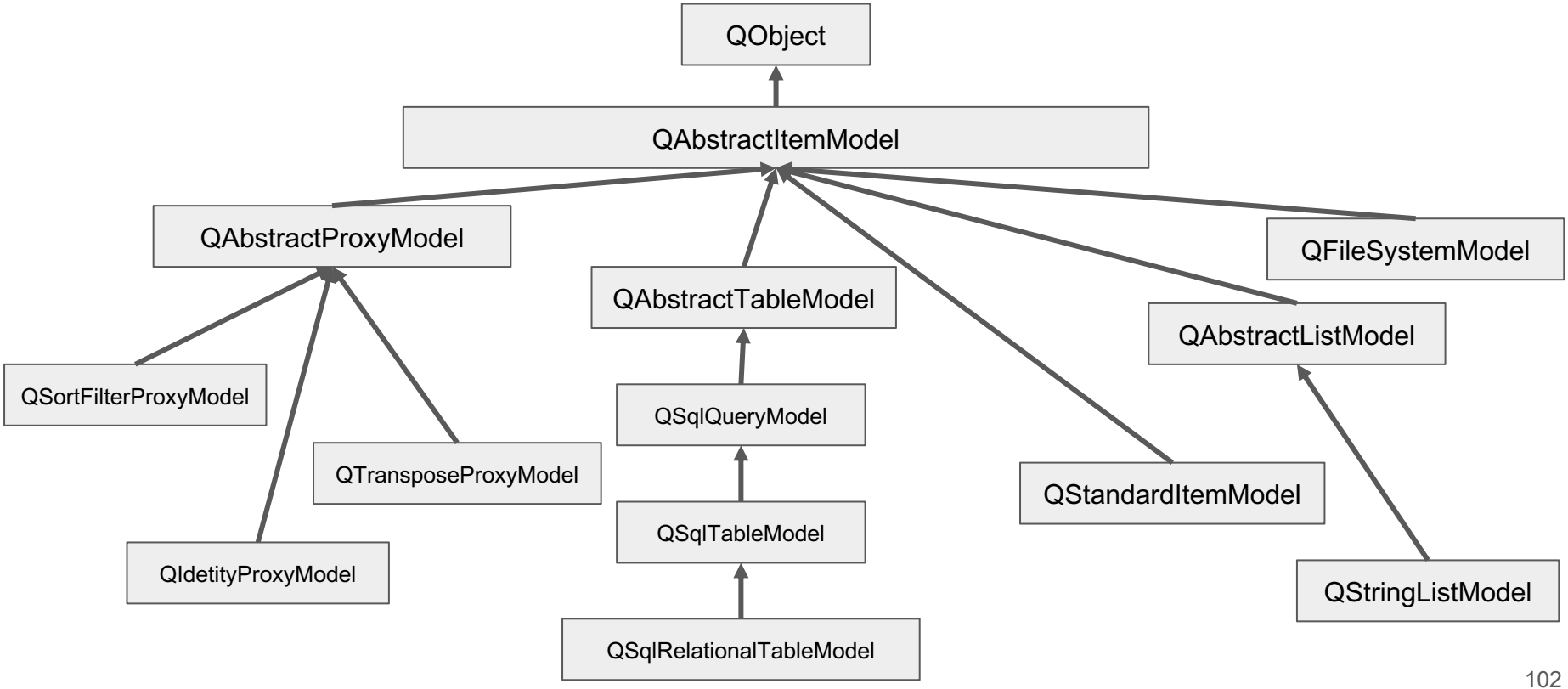
https://doc.qt.io/qt-6/modelview.html - basics of Model-View, practice

https://doc.qt.io/qt-6/model-view-programming.html - theory, going in depth

https://doc.qt.io/qt-6/examples-itemviews.html - examples

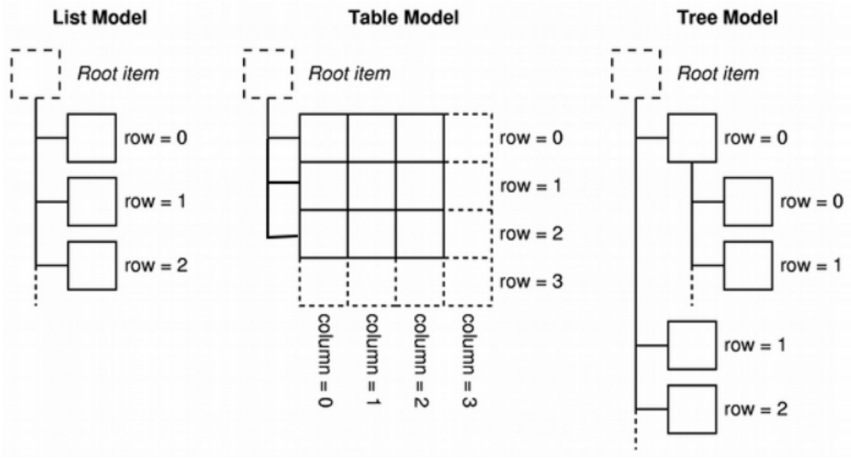
Qt has a very extensive documentation with many examples and tutorials.

Standard Model Classes



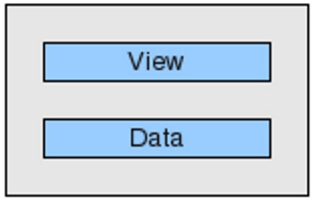
Basic Models

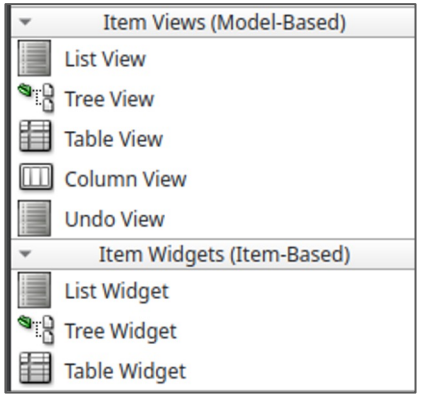
- **QAbstractItemModel** defines the standard model interface
- All subclasses of **QAbstractItemModel** represent data as a hierarchical structure containing tables of items
- Views use this *convention* to access data items in the model
- Models notify attached views about changes to data through the signals and slots system



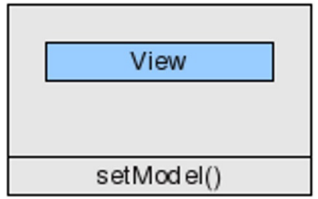
X-Widget vs X-View

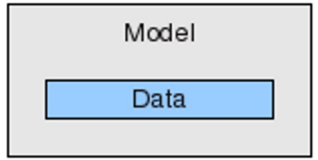
Standard widgets, item based.
QListWidget, QTableWidget, etc.



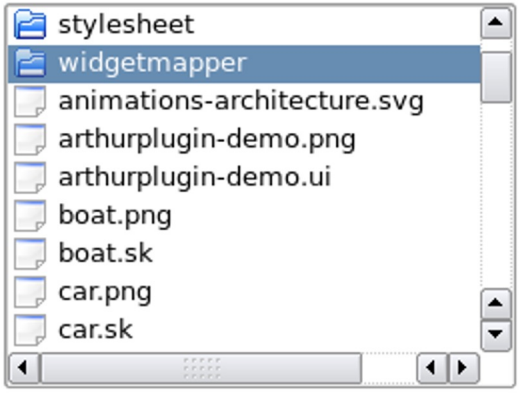


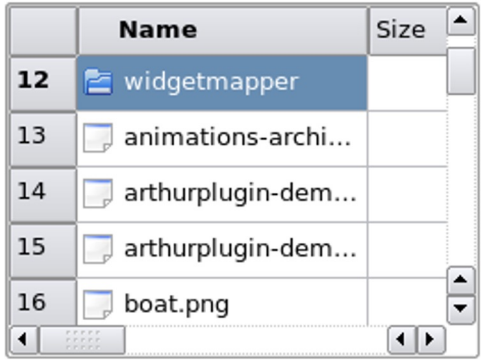
Model-View widgets.
QListView, QTableView, etc.

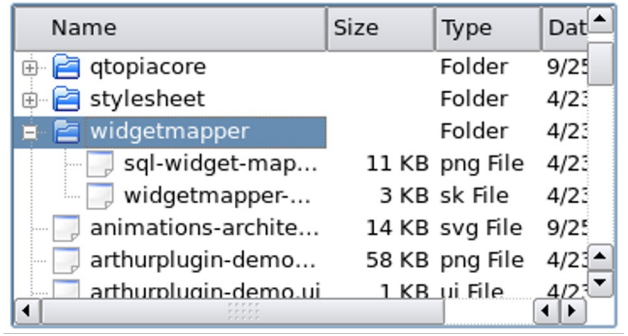


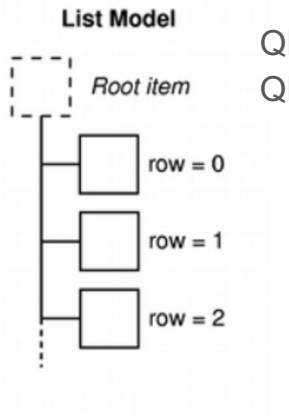


Model/View Widgets and Classes

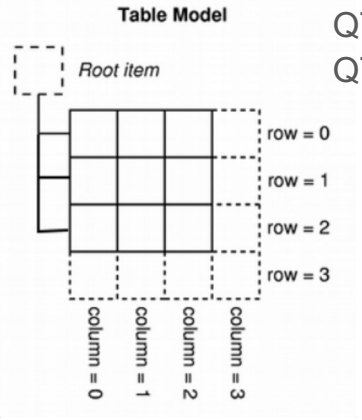




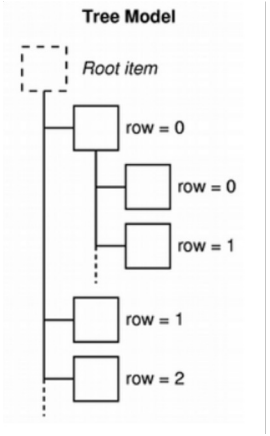




QListView
QListWidget



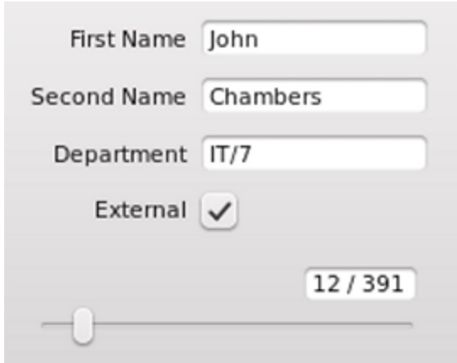
QTableView
QTableWidget



QTreeView
QTreeWidget

Forms

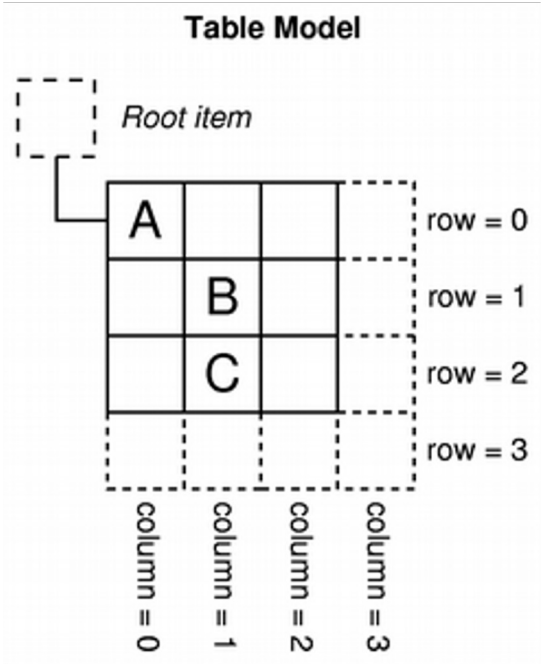
- Useful for entering data
- Have their own layout
- Make data modifications *atomic*
  - either everything is modified at once and correct
  - or nothing is modified at all



- Connected to Models via **Adapters**
  - **QDataWidgetMapper**
  - **QCompleter**

Model Indexes

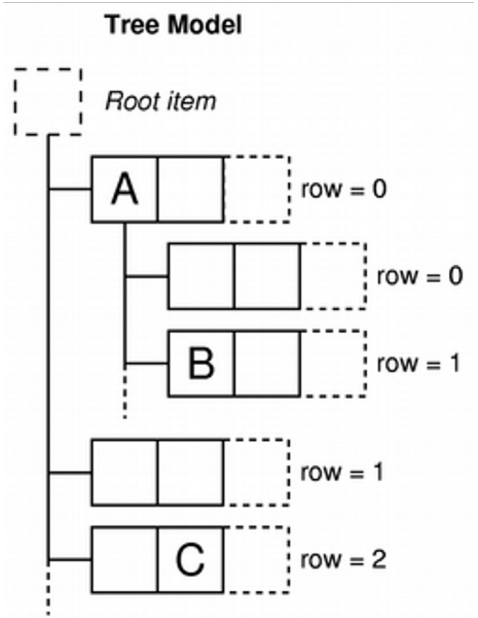
- **Model index** represents the position (index) of the data in a model
- *Views* and *Delegates* interact with data through **indexes.**
- Indexes are similar to iterators and DB cursors (and integer indexes)
  - Use Qt indexes instead of references/pointers/iteratora here
- Implemented in **QModelIndex**




```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexB = model->index(1, 1, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

Class **QModelIndex**

- **QModelIndex** is used to *locate* data in a model
  - Represents a model index - a position, an iterator
- Valid indices are created by **QAbstractItemModel::createIndex()** function
- Constructor **QModelIndex()** creates an *invalid* index
  - Useful in hierarchical models as a parent of top-level elements
  - Or as a default value
- Locating an element:
  - **row()**, **column()**, **parent()**




```
QModelIndex index = model->index(row, column, parent);
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
QModelIndex indexB = model->index(1, 0, indexA);
```

Next time

More on Models