Qt Essentials

December 2016

Based on Qt 5.8

Contents

Contents

Objectives

Learn…

- …about the history of Qt
- …about Qt's ecosystem
- …a high-level overview of Qt
- …how to create first hello world program
- …build and run a program cross platform
- …to use QtCreator IDE
- …some practical tips for developing with Qt

The Qt Company: A Brief Introduction

- Responsible for all Qt operations globally
- Worldwide leader in
  - Qt API development
  - Qt Application Development
  - Design services – UI and UX
- Trusted by over 5,000 customers worldwide
- 20+ years of Qt experience
- 200 in-house Qt experts
- Fast growing
- 27M€ revenue in year 2015

Qt History

1991
Qt conceived by Haavard Nord and Eirik Chambe-Eng on a park bench in Trondheim, Norway.

1995
Troll Tech 1st public release on 20 May - Qt 0.90 for X11/Linux
» Commercial & open source (FreeQt license)

1996
» Customer #1- European Space Agency
» Qt 1.0 - full X11 support free for free software development plus Windows
» KDE project established with Qt as its underlying library

1998
KDE Free Qt Foundation - guarantees Qt availability for free software development

1999
Qt 2.0 - Qt/X11 open source with QPL (Q Public License)

2000
» New Qt windowing system - Qt/Embedded - (a.k.a. QWS & Qtopia Core)
» Both Qt/X11 & Qt/Embedded under GPL + commercial licenses
» GPL v2 with Qt 2.2

2001
Qt 3.0 - "multiple database environments, multiple languages, multiple monitors" with Mac OS X support & a new Qt Designer GUI builder

2005
Qt 4.0 - Total makeover (a.k.a. compatibility break) under commercial & GPL 2.0 (or later) for all platforms even Windows (Qt 4 dance video published)

Qt History

2006
» Trolltech IPO
» Greenphone developer smartphone
» Qtopia in millions of devices worldwide from Sharp to Motorola

2008
Nokia acquires Trolltech – "Qt Software at Nokia"

2009
Qt Creator launched and Qt 4.5 brought LGPL v2.1

2010
Qt Quick launched, WebKit integration, & Symbian support in Qt 4.7

2011
» Digia acquires Qt commercial licensing business
» Qt Project
» Nokia N9 – Nokia MeeGo "Harmattan" smartphone
» Nokia changes smartphone platform strategy

2012
» Digia acquires all rights to Qt – "Digia, Qt"
» Qt 5.0 - Major overhaul - new modularized codebase, consolidation of QPA (Qt Platform Abstraction), Qt Quick 2, and more support for mobile (WinRT).

2013
Boot to Qt pre-built software stack & Qt WebEngine

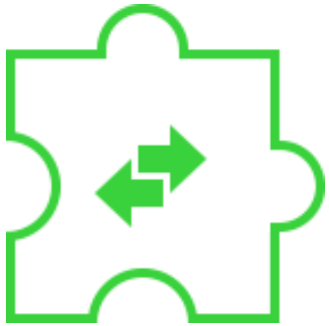
2014
"Digia, Qt" becomes its own entity "The Qt Company", subsidiary of Digia

2015
20th anniversary of Qt's 1st public release
» One Qt Site Unification Completed
» 800K+ Qt developers worldwide

What Is Qt?
The Leading C++ Cross-Platform Framework

- Integrated Development Tools
- Shorter Time-to-Market

- Cross-Platform Class Library
- One Technology for All Platforms



- Cross-Platform IDE, Qt Creator
- Productive development environment

Used by over 1 million developers in 70+ industries

Proven & tested technology – since 1994

Qt is Used for

Application Development

on Desktop,

Mobile and Embedded

Creating Powerful Devices

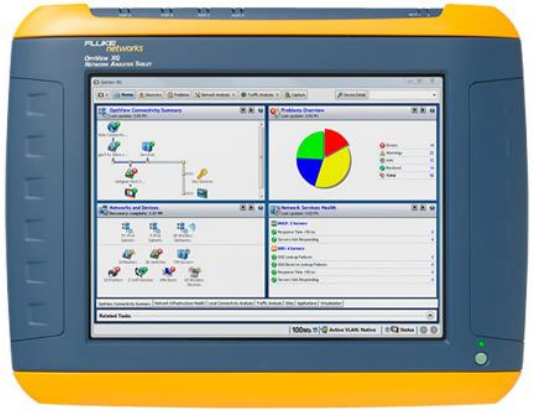
Device GUIs,

Ecosystems and whole SDKs

Where There's a User Interface, There's Qt

Automotive IVI

Refrigerators & Coffee Machines



Network Analyzers

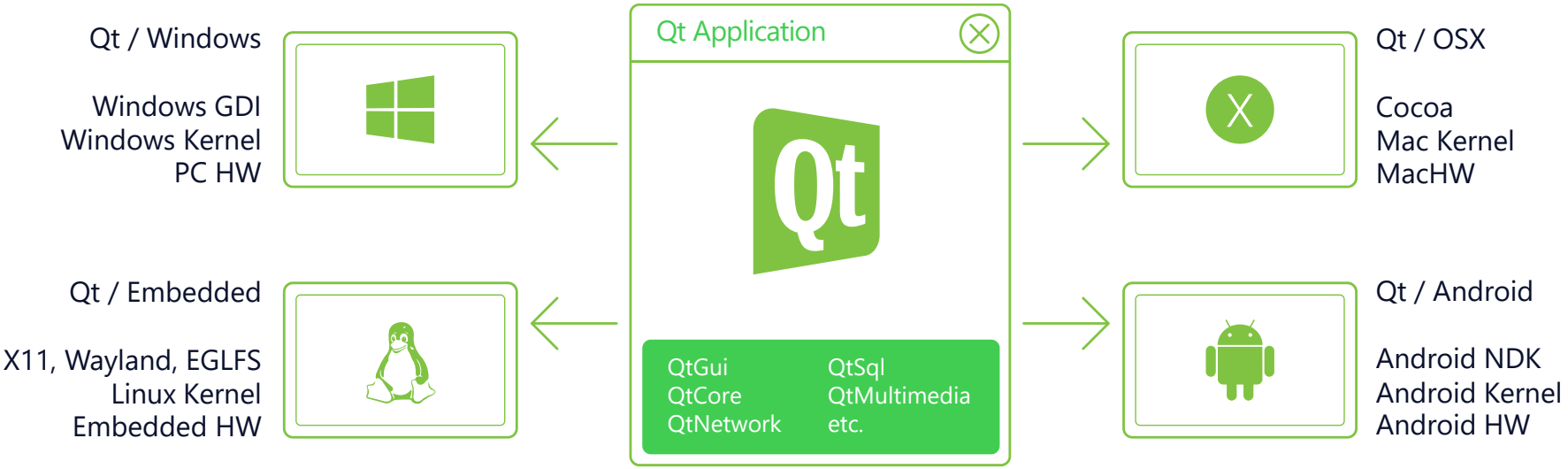
Plus:

- Medical Devices
- Home Automation
- Digital Photo Frames
- Set Top Boxes
- Industrial/UMPCS
- and many, many more …

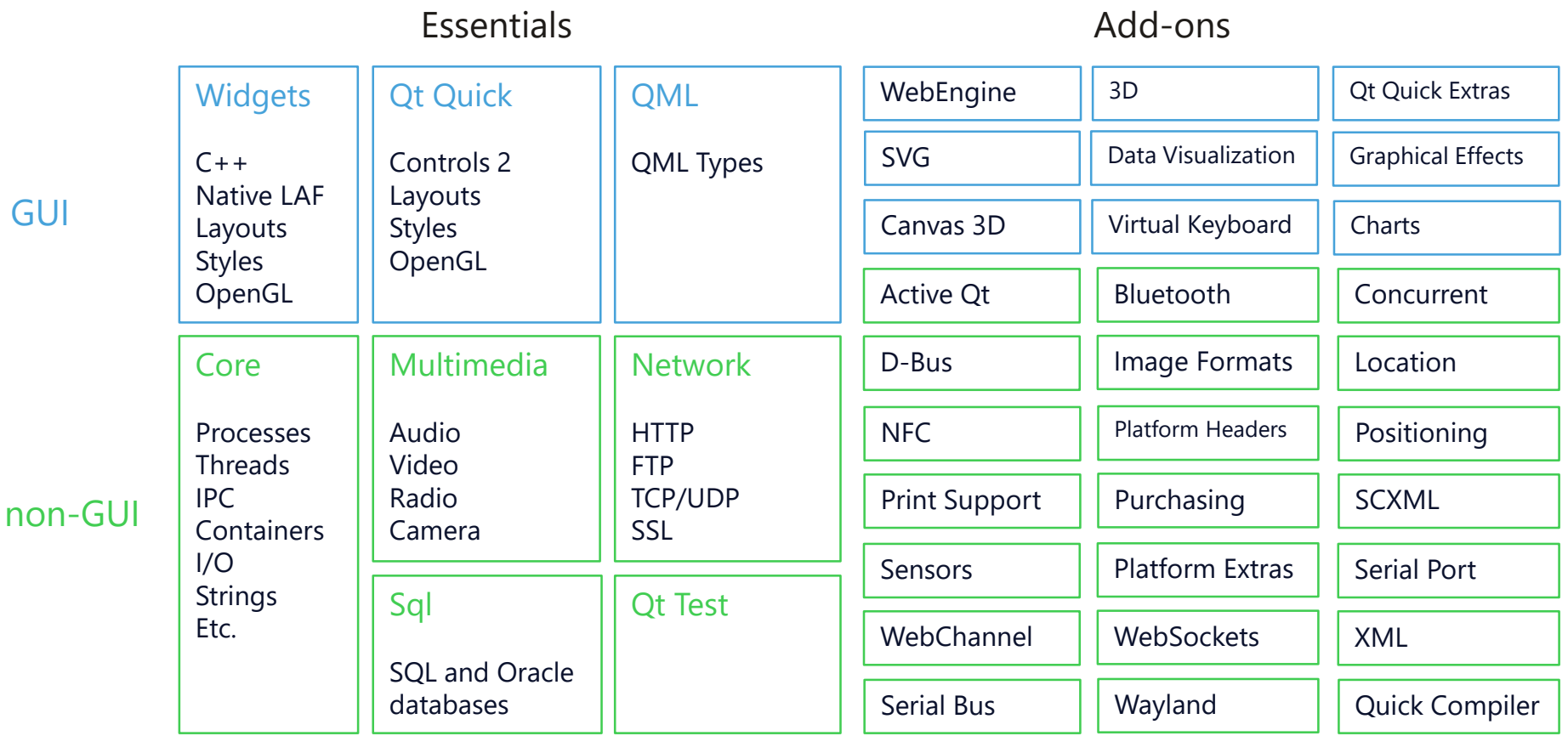
The Qt Company Trusted by 5000+ Companies from 70+ Industries

8 out of Top 10 Fortune 500 Companies Are Powered by Qt!

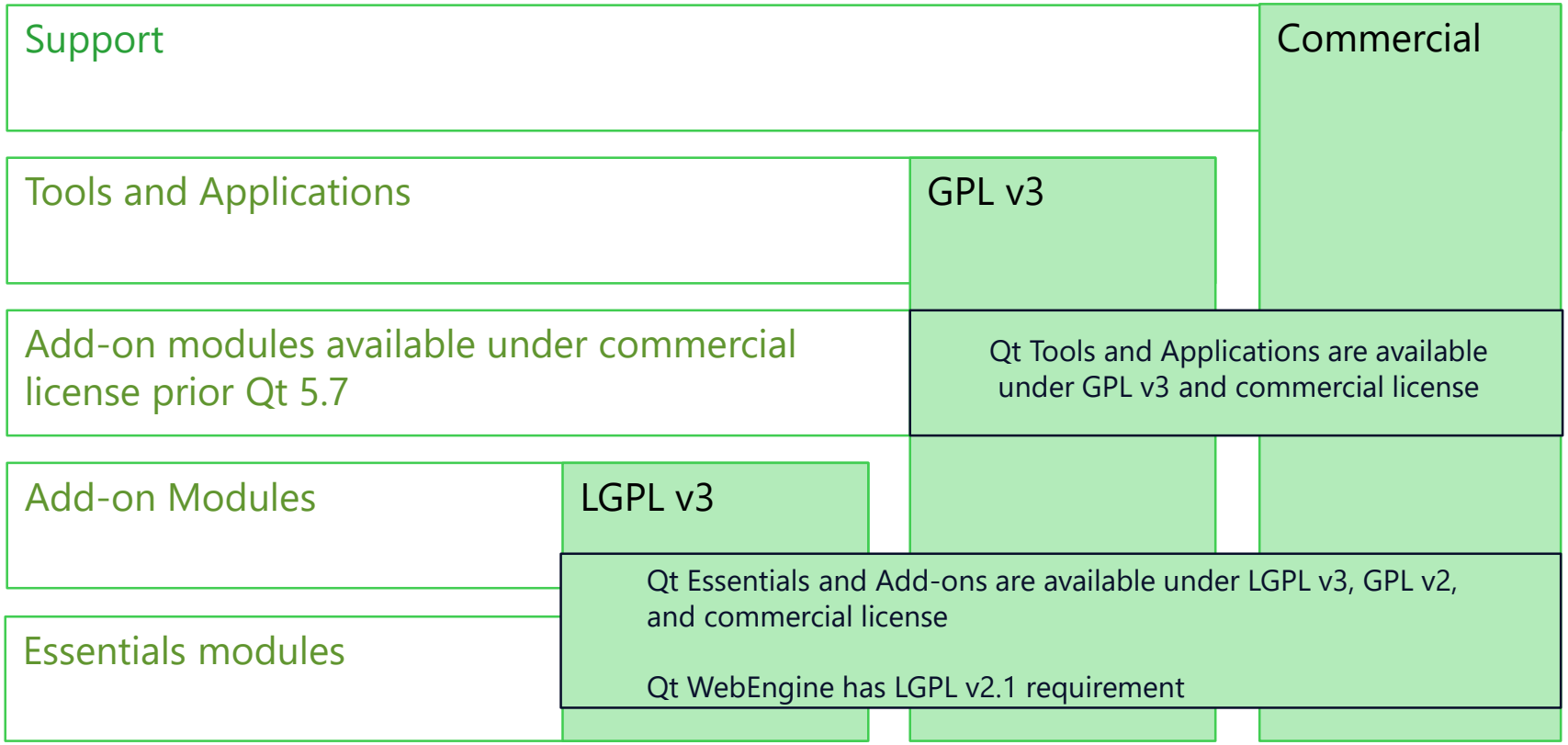
Qt Applications Are Native Applications



Qt Modules



Licensing Options



http://www.qt.io/licensing/

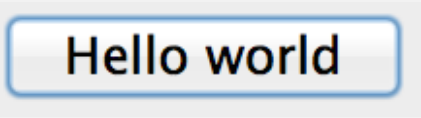
Licensing Options

- Commercial license
  - No limitations of open source licenses
  - Silver-level support included
- GPL v3
  - The original license, used in Qt
  - All proprietary code, IPR etc., will also have to be GPL licensed

  ⇒No commercial possibilities without opening own source code
- LGPL v3
  - Unlike LGPL v2, forbids creation of closed devices
  - Adds additional requirements making it less attractive in commercial usage
    - Mainly DRM restrictions, patent retaliation limitations, and Tivoization
- LGPL v2.1
  - Used prior Qt 5.7
  - Allows dynamic linking to Qt libraries with any LGPL v2.1 compatible license
  - Still possible to use with Qt 5.6 LTS (Long Term Support) version

“Hello World” in Qt – Widget-Based

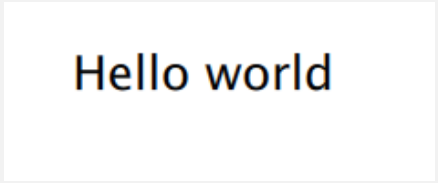
```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);       // Loads the QPA plugin
    QPushButton button("Hello world"); // Creates a widget
    button.show();
    return app.exec();                  // Starts the event loop
}
```



- Program consists of
  - `main.cpp` – application code
  - `helloworld.pro` – project file

“Hello World” in Qt – Qt Quick -Based

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    // Window created in QML
    return app.exec();
}
```



- Program consists of
  - `main.cpp` – creation and startup of the QML engine
  - `helloworld.pro` – project file
  - `qml.qrc` – resource files, consisting main.qml
  - `main.qml` – application code

Project File - helloworld.pro

- `helloworld.pro` file
  - lists source and header files
  - provides project configuration

```
# File: helloworld.pro

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TEMPLATE = app
SOURCES = main.cpp
```

- Assignment to variables
  - Possible operators =, +=, -=, *=

Documentation: qmake Manual

Using qmake

- `qmake` tool
  - Creates cross-platform make-files
- Build project using `qmake`

```
cd helloworld
qmake helloworld.pro     # creates target-dependent Makefile
make                     # compiles and links application

./helloworld             # executes application
```

- Tip: `qmake -project`
  - Creates default project file based on directory content

Qt Creator IDE does it all for you

Useful qmake Variables

Hint! Additional variables and values may be defined in the command line:

```
qmake "CONFIG += debug"
```

- `TEMPLATE` - Defines the project type
  - `app, vcapp`
  - `lib, vclib`
  - `subdirs`
- `TARGET`
  - Executable name (by default equals to the **.pro** file name)
- `QT`
  - Modules, used in the project
  - Defined in t`mkspecs/modules`
  - `QT += webkit sql network charts`
- `CONFIG`
  - Specifies a project configuration or compiler option
  - May refer to a project feature file (.prf) in `mkspecs/features`
  - E.g. a custom library with headers
- `INCLUDEPATH` and `DEPENDPATH`
  - Sets the include search path (–I option)
- `RESOURCES`
  - Specifies resource collection (**.qrc**) files to include in build
- `LIBS`
  - Library path `–L` and library `–l`
  - Omit platform-dependent prefixes and extensions
- `DEFINES`
  - Compiler pre-processor macors `–D`
- `INSTALLS`
  - Files and folders to be copied after building (`make install`)
  - `target.path += $$[QT_INSTALL_qml]/module`
  - `INSTALLS += target`

Using qmake Variables

- Scopes
  - `win32:debug { SOURCES += paintwidget_win.cpp }`
- Single line conditional assignment
  - `win32:DEFINES += QT_DLL`

- Environment variable reference
  - `DESTDIR = $$PWD # Evaluated when the .pro file is processed`
  - `DESTDIR = $PWD  # Evaluated when the Makefile is executed`
- Variable reference
  - `TARGET = myproject_$${TEMPLATE}`
- qmake configuration options
  - You may ask the options with the command `qmake -query`
  - `message(Qt version: $$[QT_VERSION])`

Alternatives to qmake: cmake

› Configuration file CMakeLists.txt

```
project(ex-cmake-project)
cmake_minimum_required(VERSION 2.8.11)
# Define sources
aux_source_directory(. SRC_LIST)
# Find headers
set(CMAKE_INCLUDE_CURRENT_DIR ON)
# Define Qt module locations
# Qt install folder defined with CMAKE_PREFIX_PATH
find_package(Qt5Widgets)
# Executable to be created
add_executable(${PROJECT_NAME} ${SRC_LIST})
# Link executable to any libraries
target_link_libraries(${PROJECT_NAME} Qt5::Widgets)
```


ex-cmake-project, documentation: cmake Manual

Alternatives to qmake: Qt Build Suite (QBS)

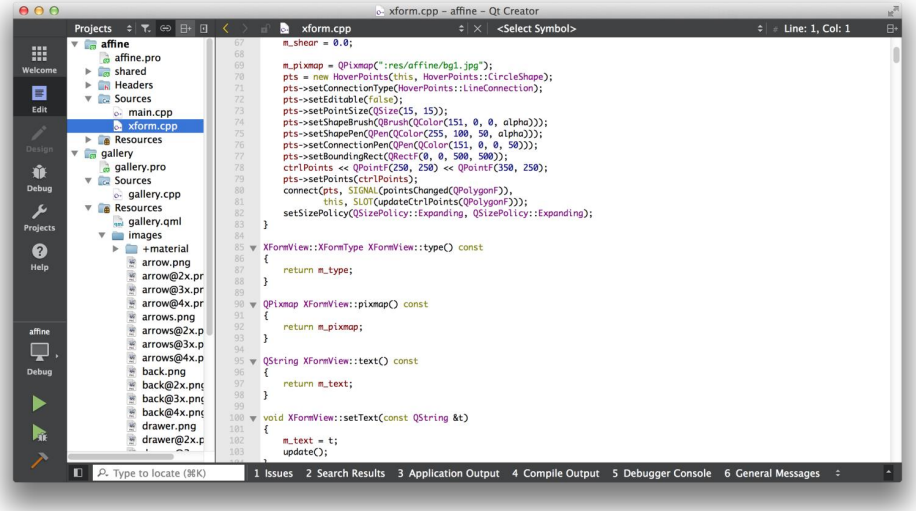
- Declarative way to define the build graph
  - Syntax similar to QML
- A project consists of one or more products
  - Modules define, how source code files are handled

```
QtGuiApplication { // Product subtype
    targetName: "ex-qbs-project"
    files: "main.cpp"
    Depends { name: "cpp", "Qt.widgets" } // 2 modules
    // Possible to define any rules, how files are handled
    Group {
        condition: qbs.targetOS.contains("linux")
        fileTagsFilter: "application"
        qbs.install: true
        qbs.installDir: ".."
    }
}
```

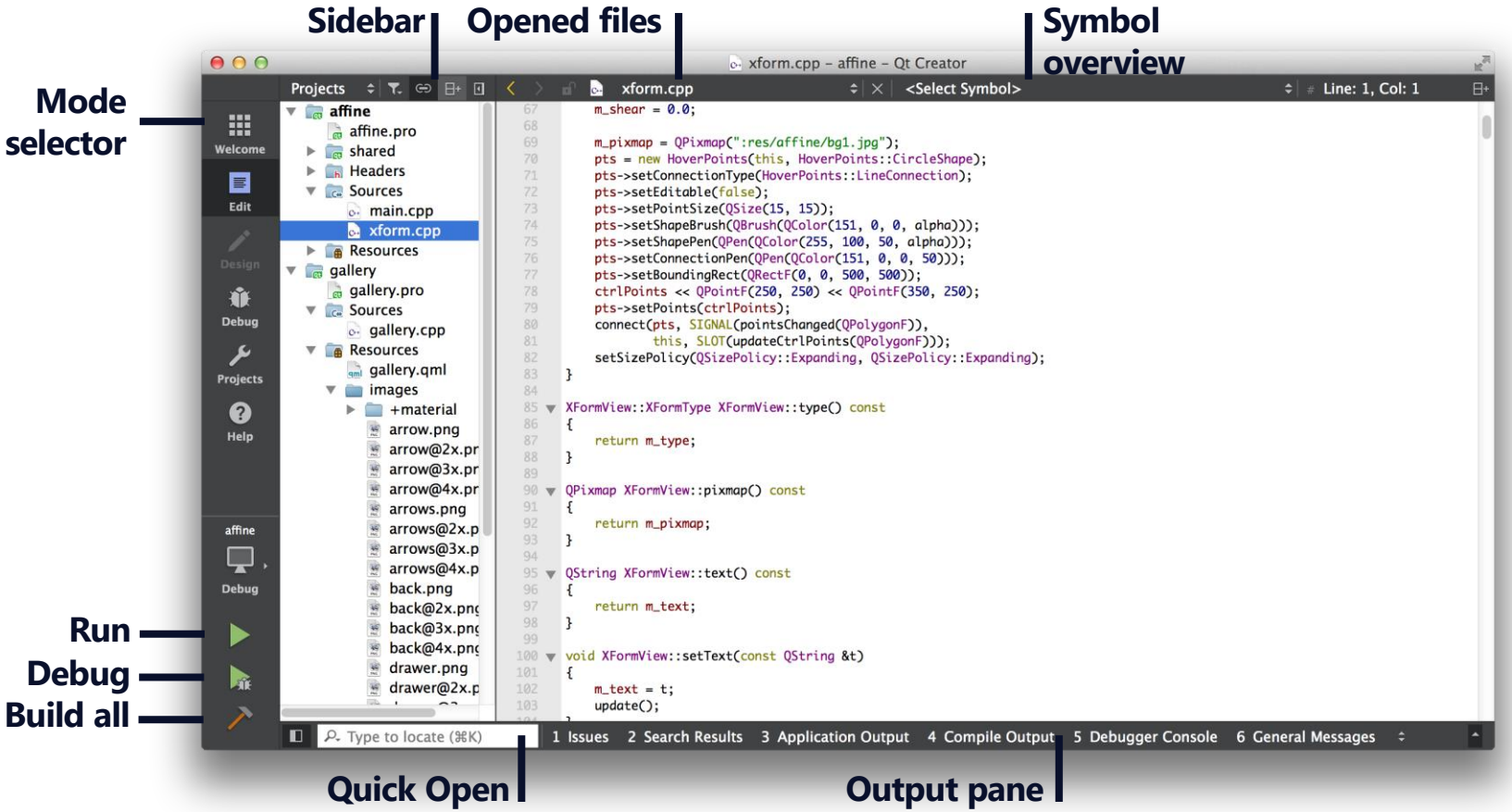
ex-qbs-project, documentation: Qbs Manual

Creating, Building, and Debugging Applications with QtCreator

- Advanced C++/QML code editor
- Integrated GUI layout and forms designer for widgets and Qt Quick items
- Project and build management tools – qmake, cmake, and qbs integrated
- Integrated, context-sensitive help system
- Visual C++, QML and JavaScript debugger
- Rapid code navigation tools
- Code analysis tools
- QML profiler tool
- Supports multiple platforms

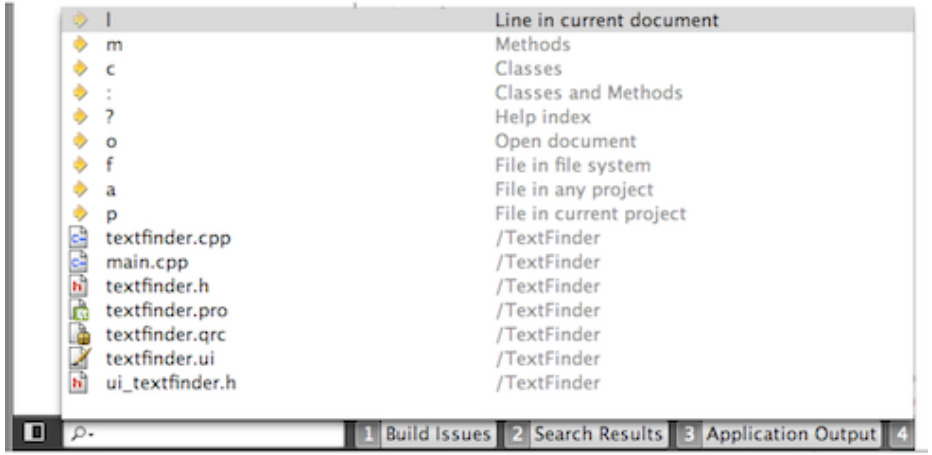


Qt Creator IDE



Finding Code – Locator

- Click on Locator or press Ctrl+K (Mac OSX : Cmd+K)
- Type in the file name
- Press Return

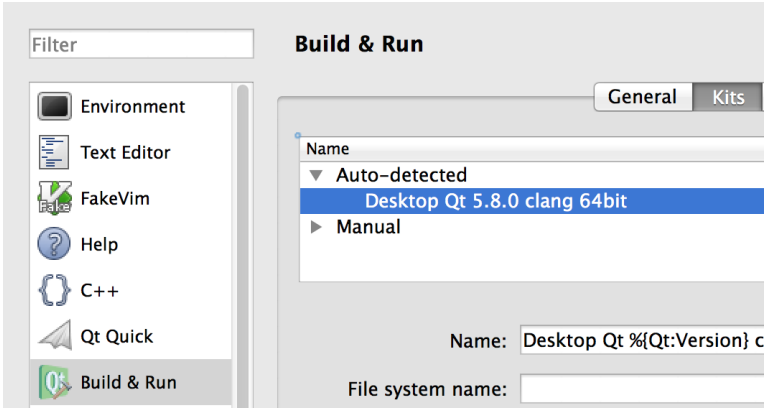


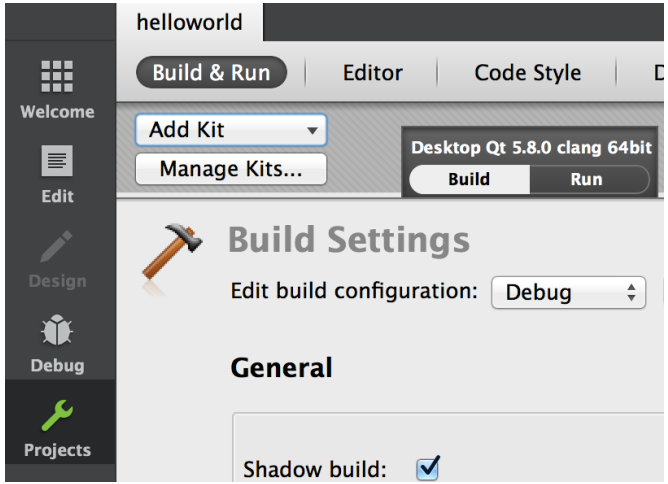
- Locator Prefixes
  - **:<class name>** - Go to a symbol definition
  - **l<line number>** - Go to a line in the current document
  - **?<help topic>** - Go to a help topic
  - **o<open document>** - Go to an opened document

Kits

- Build and run targets defined as kits
  - *Tools > Options – Build & Run: Kits*
- A kit defines, which
  - Qt version is used (Qt versions sheet – *qmake* location)
  - compiler is used (Compilers sheet)
  - debugger is used (Debuggers sheet)

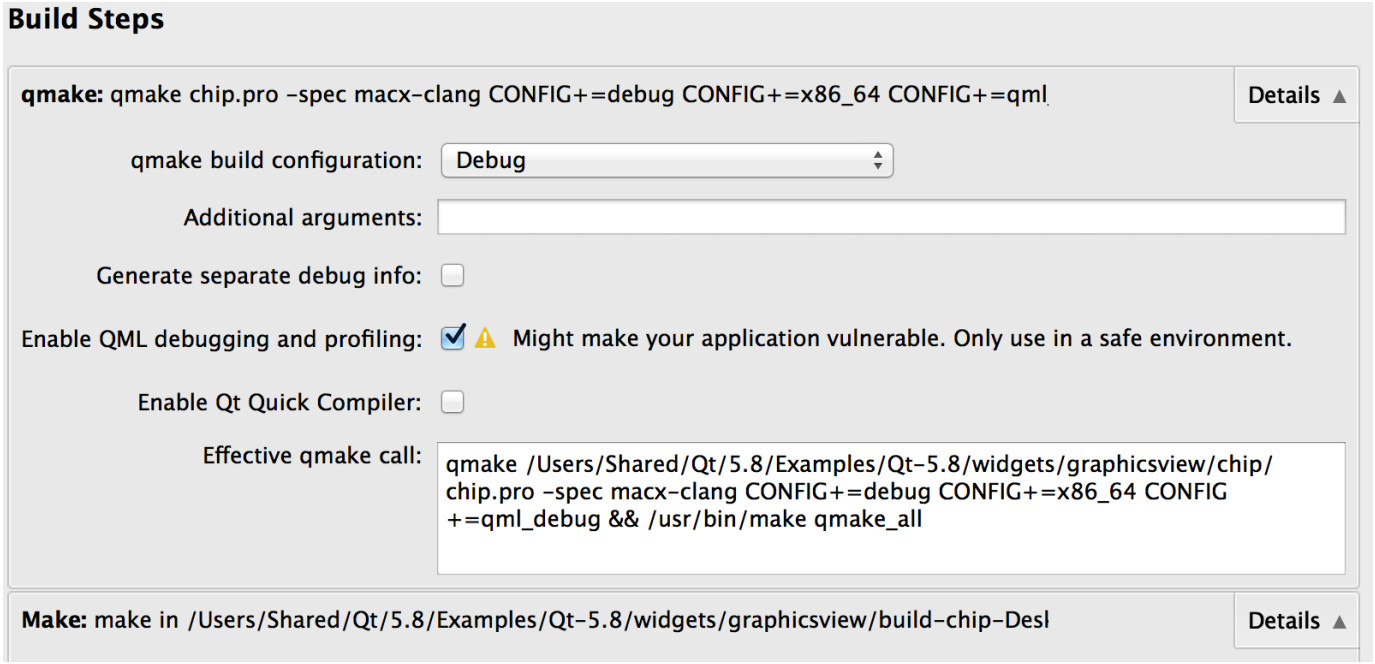
- In *Projects* mode kits may be dynamically added/removed
  - Kit and build type selectable above run/debug buttons
- Deployment and on-device debugging easy to configure
  - *Tools > Options – Devices*





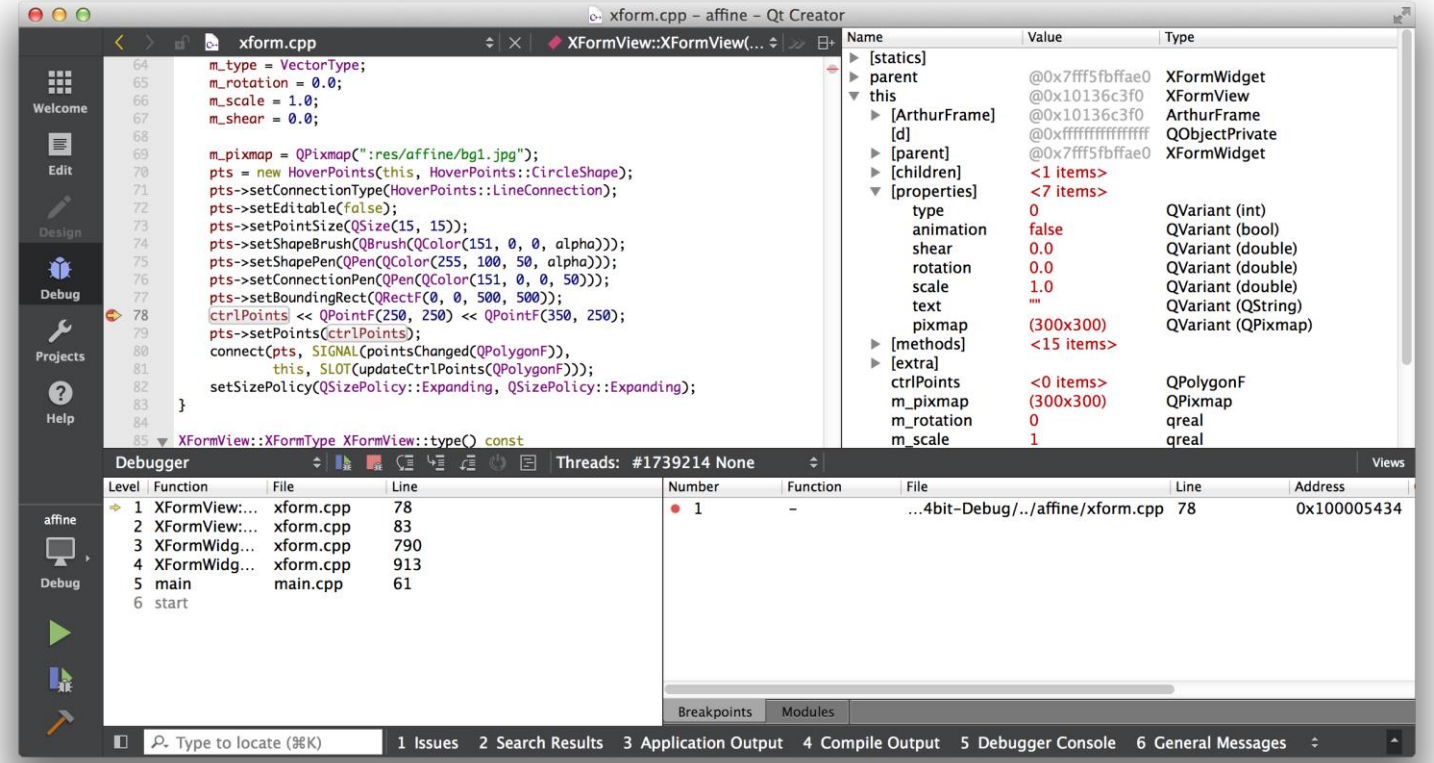
Build and Run Settings – Qt Creator Project Mode

- Customize build steps
- Add build steps
- Add command line arguments
- Add/edit environment variables



Debugging an Application – Locally or Remotely

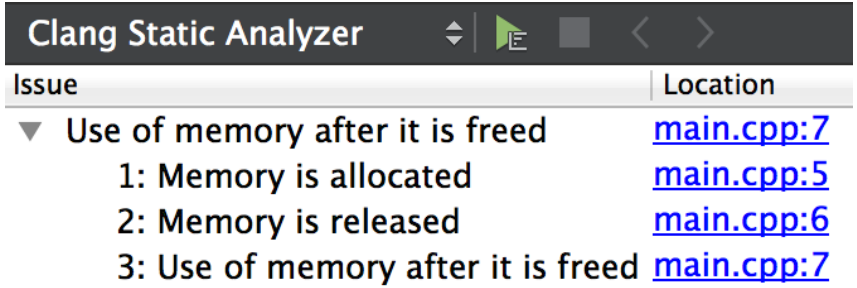
› Debug –> Start Debugging (o rF5)

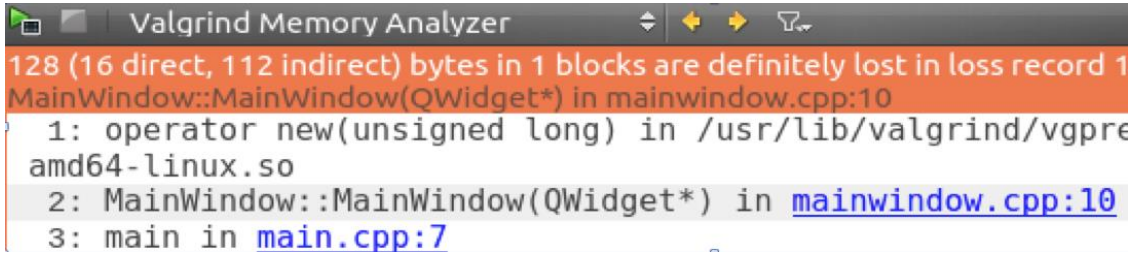


Code Analysis

- C++ code analysis require a backend
- QML analysis can be done with Qt tools

- Clang static analyzer
  - Uses clang open source library as a backend
  - Detects more than a compiler (e.g. use of dangling pointers)
- Valgrind memory analyzer and profiler
  - Detects, e.g., memory leaks and most frequently executed functions

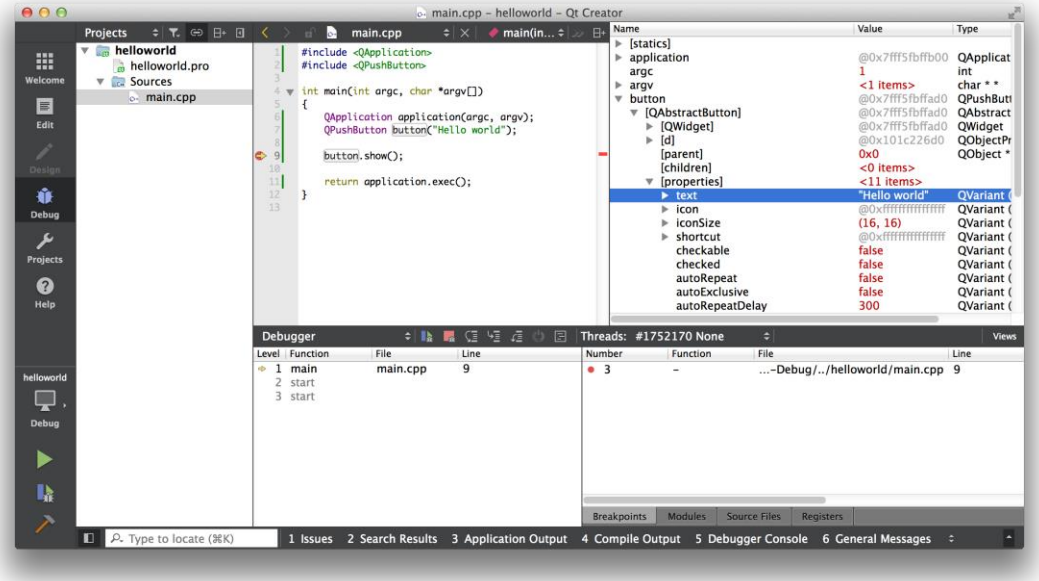




Qt Creator Demo "Hello World"

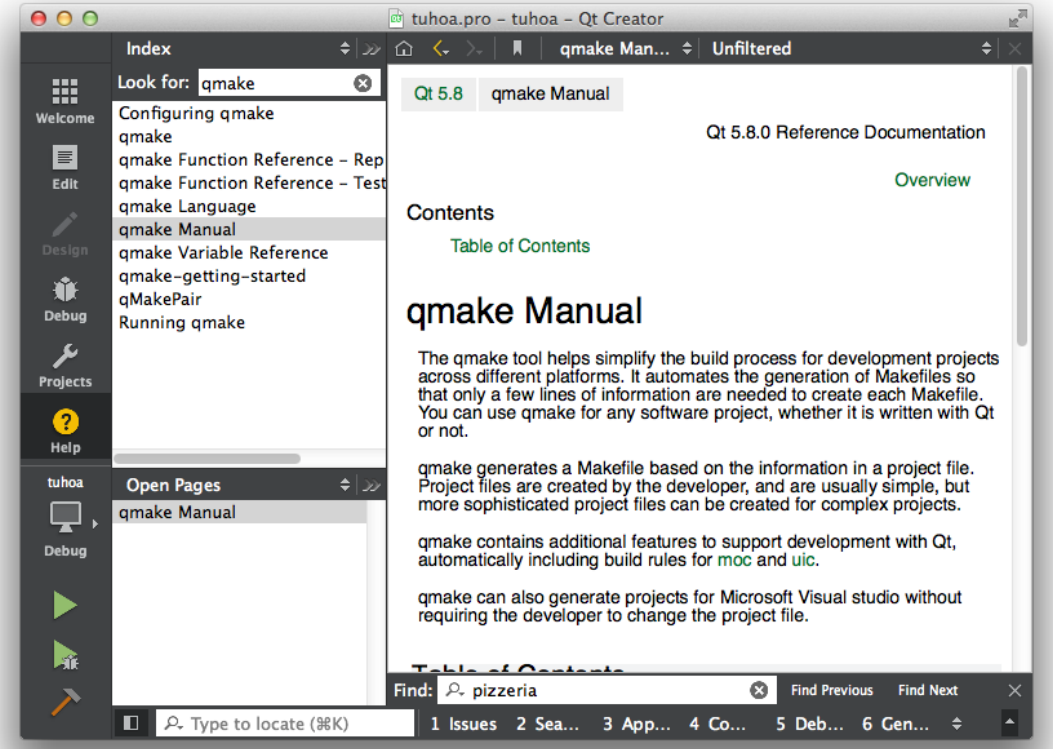
What we'll show:

- Creation of an empty Qt project
- Adding the `main.cpp` source file
- Writing of the Qt Hello World Code
  - Showing Locator Features
- Running the application
- Debugging the application
  - Looking up the `text` property of our button



Qt Documentation

- Reference Documentation
  - All classes documented
  - Contains tons of examples
- Collection of Howto's and Overviews
- A set of Tutorials for Learners



Finding the Answers

- I need answers ASAP
  - Use Qt support http://www.qt.io/support/
  - Meet Qt professional in IRC https://wiki.qt.io/Online_Communities
  - Questions and answers forum – Qt Centre http://www.qtcentre.org/content/
- It does not work
  - Bug reporting and finding https://bugreports.qt.io/
  - Stack Overflow http://stackoverflow.com
- I have some time to study Qt myself
  - Qt Developer Guides http://wiki.qt.io/Developer-Guides
  - Learning videos https://www.youtube.com/user/QtStudios
  - Books, tools, guides, documentation - http://wiki.qt.io/Main_Page
- I want to know the latest Qt updates
  - Qt Blog http://blog.qt.io
- I want to have some Qt apps for free
  - http://qt-apps.org

Qt's source code is easy to read, and can answer questions the reference manual cannot answer!

Qt Coding Convention

```
#include <QApplication>
#include <QLabel>
#include "customobject.h"

int main(int argc, char *argv[])
{
    QApplication application(argc, argv);
    QLabel label;
    CustomObject customObject;
    customObject.setString("Hello world");
    label.setText(customObject.string());
    label.show();
    return application.exec();
}
```

```
// Avoid acronyms, use camel-case
// Class names with capital first letter
// Function and variable names lower case first letter

#include <QObject>

class CustomObject : public QObject
{
    Q_OBJECT
public:
    explicit CustomObject(QObject *parent = 0);
    void setString(const QString &string);
    QString string() const;
protected:
    bool event(QEvent *event) Q_DECL_OVERRIDE;
private:
    QString m_string;
};
```

```
#include "customobject.h"

CustomObject::CustomObject(QObject *parent) : QObject(parent) { }

void CustomObject::setString(const QString &string)
{
    if (string != m_string) m_string = string;
}
bool CustomObject::event(QEvent *event)
{
    // Dummy example
    return QObject::event(event);
}
```

Includes and Compilation Time

- Class includes

```
#include <QLabel>
```

- Module includes

```
#include <QtGui>
```

- Reduce compilation time
  - Use class includes
    - `#include <QLabel>`
  - Forward declarations
    - `QT_FORWARD_DECLARE_CLASS(QLabel)`

- *Place module includes before other includes*

Qt Debugging Aids – Logging

- `QDebug` allows streaming debug information to QString or any `QIODevice`
  - `QIODevice` subclasses: `QBuffer`, `QFileDevice`, `QProcess`

```
QDebug debug(aDebugDevice);
debug << "Something happened";
```

- Often more convenient to use macros `qInfo()`, `qDebug()`, `qWarning()`, `qCritical()`, and `qFatal()`
  - Macros expand to corresponding function calls in `QMessageLogger`
  - Message logger uses `QtMessageHandler`
  - Suppress messages by adding `DEFINES += QT_NO_DEBUG_OUTPUT QT_NO_WARNING_OUTPUT QT_NO_INFO_OUTPUT` to your .pro file or installing a custom message handler

```
// #include <QtDebug>
qDebug("Method computed: %d", intVariable);
qDebug() << "Mouse was clicked at " << mouseEvent->pos();
QMessageLogger(__FILE__, __LINE__, Q_FUNC_INFO).debug() << "Oh no”;
```

Logging Message Categories

- Allows controlling which messages to log and which to ignore
- Message types (debug, warning etc.) and categories can be enabled/disabled

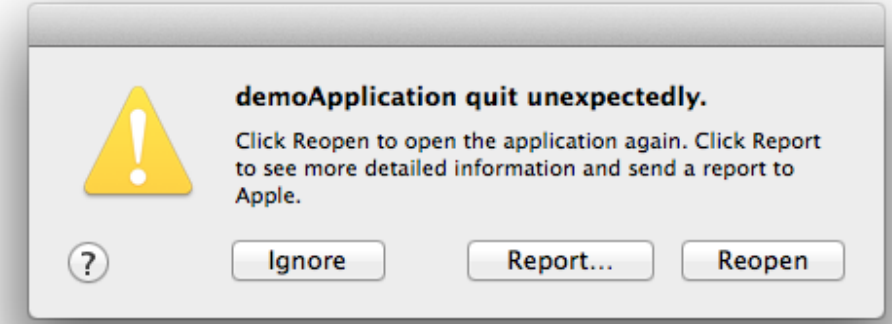
- Define a category using `QLoggingCategory` or use macros to declare and define categories
  - `Q_DECLARE_LOGGING_CATEGORY(aCategory)`
  - `Q_LOGGING_CATEGORY(aCategory, "com.theqtcompany.application", QtWarningMsg)`

```
QLoggingCategory category("com.theqtcompany.app");
category.setEnabled(QtDebugMsg, false);
qCWarning(category) << "A warning message";
qCDebug(category) << "This is not logged";
qCWarning(aCategory) << "Another warning message";

// Outputs:
// com.theqtcompany.app: A warning message
// com.theqtcompany.application: Another warning message
```

Qt Debugging Aids – Asserts

- `Q_ASSERT` and `Q_ASSERT_X`
- `Uses qFatal()`, which aborts the application, when the default message logger used

```
Q_ASSERT_X(1 > 2, Q_FUNC_INFO, "False condition");
// Q_ASSERT(false);

// Outputs:
ASSERT failure in int main(int, char **): "False condition",
file ../demoApplication/main.cpp, line 17
The program has unexpectedly finished.
```



Qt Debugging Aids – Debug Dumps

- Each QObject can be named with `QObject::setObjectName()`
  - The name can be retrieved with `QObject::objectName()`
- As a lot of the debugging information is only really helpful if these names are set, it is good Qt programming style to do so

- Debug dumps
  - Work only in debug builds
  - `QObject::dumpObjectInfo()` dumps information about object internals, like signals/slots
  - `QObject::dumpObjectTree()` dumps the parent/child relationships of all descendant objects

```
OBJECT QApplication::unnamed
  SIGNALS OUT
    signal: destroyed(QObject*)
    signal: destroyed()
    signal:
objectNameChanged(QString)
```

```
QApplication::
  QCocoaEventDispatcher::
  QSessionManager::
  QMacStyle::macintosh
```

Questions And Answers

- What is Qt?
- What licenses can be used with Qt?
- Which code lines do you need for a minimal Qt application?
- What is a .pro file?
- What is qmake, and when is it a good idea to use it
- What is a Qt module and how to enable it in your project
- How can you include a `QLabel` from the QtGui module
- Name places where you can find answers about Qt problems

Summary

- Qt is a cross-platform framework, allowing the same code to be built and run in desktop, embedded, and mobile platforms
- Qt has a large selection of libraries, providing developers with some 1,500 C++ classes
  - In addition to class libraries, Qt framework contains several development tools, like Qt Creator IDE
- Qt is available under commercial and open source licenses
- GUI application can be written with C++ widgets, with QML or with web technologies
- There is a large and active developer community around Qt

Contents

Objectives

Learn…

- … about Qt objects
- … memory management with object trees and Qt smart pointers
- … how objects can communicate
- … what signals & slots are
- … how to use signals & slots for object communication
- … which variations for signal/slot connections exist
- … how to create custom signals & slots
- … how Qt handles events

Qt's Object Model

- Qt's 1,500 classes can be divided into two groups
  - Identity types
  - Value types

- Identity types
  - derive from `QObject`
  - extend C++ with many dynamic features using a meta-object system
  - cannot be copied as the copy constructor and assignment operator equal to delete
  - `QWidget`, `QWindow`, `QApplication`, `QEventLoop`, `QThread`, `QFile`, `QTcpSocket`

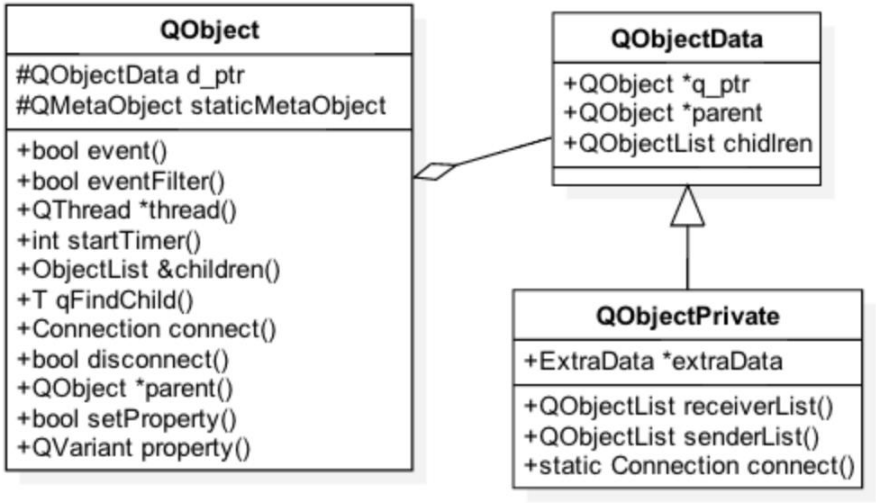
- Value types are standard C++ classes
  - `QColor`, `QEvent`, `QDataStream`, `QMetaType`
  - ~100 value types use copy-on-write pattern – implicitly shared
    - `QString`, `QByteArray`, `QList`, `QVector`, `QHash`, `QCache`, `QDir`, `QPixmap`, `QImage`, `QBrush`, `QPen`

QObject – Heart of Qt's Object Model

Extends Qt with dynamic features, useful in GUI programming

- Object tree
- Thread-safe and type-safe object communication – signals and slots
- Event handling
- String localization
- Object properties
- Safe static cast – `qobject_cast`
- Internal timer

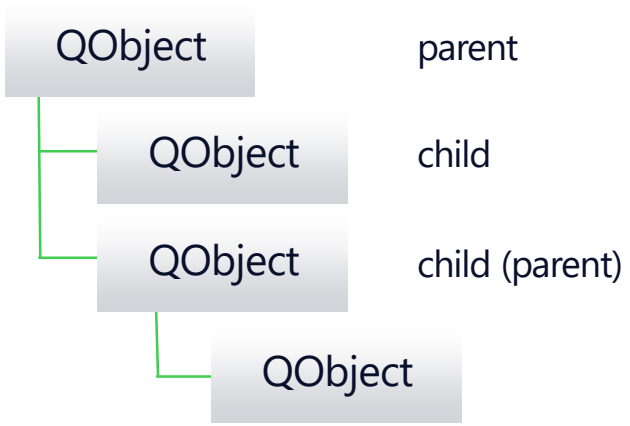
- Qt objects use P-IMPL pattern to guarantee binary compatibility



Object Tree

- `QObjects` organize themselves in object trees
  - Based on parent-child relationship
- `QObject(QObject *parent = Q_NULLPTR)`
  - Parent adds object to list of children
    - `const QObjectList &children();`
    - `QList<QWidget> findChildren();`
  - Parent owns children
- Construction/Destruction
  - Tree can be constructed in any order
  - Tree can be destroyed in any order
    - if object has parent: object first removed from parent
    - if object has children: deletes each child first
    - No object is deleted twice

- *Note: Parent-child relationship is NOT inheritance*



Object Creation

- **On Heap** – `QObject` with parent:
  - `QObject *object = new QObject(parent);`
- **On Stack** - `QObject` without parent:
  - `QFile`, usually local to a function
  - `QApplication`, local to `main()`
  - Top level widgets : `QMainWindow window; // No parent`
- **On Stack** - value types

```
QString name;
QStringList list;
QColor color;
```

  - Passed by value everywhere
  - `QString` and `QStringList` implicitly shared

- **Stack or Heap** - `QDialog` - depending on lifetime

Object Deletion – deleteLater()

- An object must not be deleted, while it is handling an event
- `QObject::deleteLater()` slot schedules the object for deletion
  - The object deleted, when the control returns to the event loop
  - If the thread does not have an event loop, the object deleted when the thread finishes

```
connect(threadPtr, &QThread::finished, threadPtr, &QThread::deleteLater);
```

- At least one event loop must be running or started later
  - Otherwise the object is not deleted
- Calling `deleteLater()` more than one for the same object does not cause double deletion

Object Deletion – QPointer

- Object tree does not solve the dangling pointer problem
  - `QPointer` provides a guarded pointer for `QObject`
  - Sets pointer to 0, when the referenced object destroyed
  - Easy to mix guarded and normal pointers
    - Guarded pointer automatically cast to the pointer type

- Qt objects may also notify observers just before their destruction

```
// ExampleObject is just QObject subclass
QPointer<ExampleQObject> object(new ExampleQObject);
delete object;
if (object)
    qDebug() << ”Dangling pointer";
else
    qDebug() << "No dangling pointer";
```

Object Deletion – QScopedPointer

- Deletes the referenced object, when the pointer goes out of the scope
- Unlike `QPointer`, can be used for any type
  - Four different cleanup handlers
    - `QScopedPointerDeleter`
    - `QScopedPointerArrayDeleter`
    - `QScopedPointerArrayDeleter`
    - `QScopedPointerDeleteLater`

```
QScopedPointer<int, QScopedPointerArrayDeleter<int>> intArrayPointer(new int(100));
```

- Useful in functions with several exit paths
  - E.g. exceptions
  - Simplifies the code

- Often used to delete dynamically allocated member variables as well

Data Sharing

1.Data sharing

- Shared data pointer does not get destroyed, if there are any references to it
- No need to worry who deletes the data and when
- Strong and weak data pointers

```
QSharedPointer<MyObj> sharedPointer =
    QSharedPointer<MyObj>(new MyObj, &QObject::deleteLater);
QWeakPointer<MyObj> weakPointer(sharedPointer);
if (weakPointer)
// Referenced object still exists
```

2.Implicit sharing

- Similar to data sharing, but data gets automatically copied, if any referencing object changes the data

3.Explicit sharing

- Similar to implicit sharing, but data is never copied implicitly

Communication between Qt Objects

- **Between objects**
  - Signals & Slots
- **Between Qt and the application**
  - Events
- **Between Objects on threads**
  - Signal & Slots + Events
- **Between Applications**
  - D-Bus, sockets, shared files, process standard input/output, `QSharedMemory`

Callbacks

- How do you get from "the user clicks a button" to your business logic?

- Possible solutions
  - Callbacks
    - Based on function pointers
    - Not type-safe
  - Observer Pattern(Listener)
    - Based on interface classes
    - Needs listener registration
    - Many interface classes

- Qt uses
  - Signals and slots for high-level (semantic) callbacks
  - Virtual methods for low-level (syntactic) events

Object Communication with Signals and Slots

- A *signal* is a way to notify an observer that something has happened
  - A mouse pressed on a UI control, for example
  - Signals are member functions that are *automatically implemented in the meta-object*
  - Signal is sent, or *emitted*, using the keyword `emit` or `Q_EMIT` - actually does not do anything
    - `Q_EMIT someSignal(7, "Hello");`

- A *slot* is a function that is to be executed after a signal has been *emitted*
  - On a mouse press, start an animation
  - A slot function may be a member function, a non-member function or a lambda function

- Signals are connected to other signals or slots
  - Many-to-many relationship
  - Signal and slot parameter types must match, but the slot may omit any number of trailing parameters

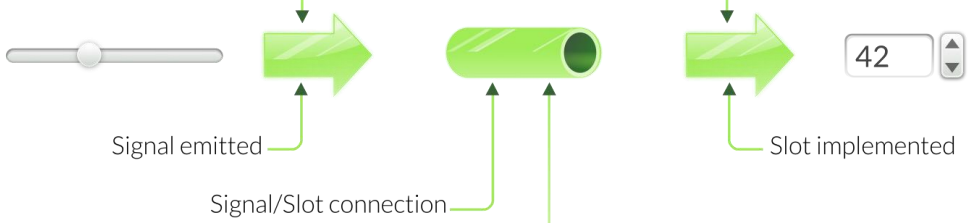
Qt Object Class Declaration with Signals and Slots

```
class CustomObject : public QObject
{
    // Q_OBJECT macro defines among other things a static meta object
    // Required for signals and string-based slots
    Q_OBJECT

public:
    explicit CustomObject(QObject *parent = 0) : QObject(parent), m_value(42) { }

Q_SIGNALS: // Macro expands to public void
    simpleSignal(int, const QString &); // Or alternatively Q_SIGNAL prefix

// Slots are often setters
public Q_SLOTS: // Or alternatively Q_SLOT prefix
    void setValue(int value) { if (value != m_value) m_value = value; }
private:
    int m_value;
};
```

Connecting Signals to Slots

```
void QSlider::mousePressEvent(...)
{
   ...
   emit valueChanged( newValue );
   ...
}
```

```
void QSpinBox::setValue( int value )
{
   ...
   m_value = value;
   ...
}
```




```
QObject::connect( slider, &QSlider::valueChanged,
                  spinbox, &QSpinBox::setValue )
```

Signal Slot Connection Variants

- String-based connection:
  - `connect(slider, SIGNAL(valueChanged(int)), spinbox, SLOT(setValue(int)));`
- Using pointers to member functions:
  - `connect(slider, &QSlider::valueChanged, spinbox, &QSpinBox::setValue);`
- Using non-member function:
  - `static void printValue(int value) {...}`
  - `connect(slider, &QSignal::valueChanged, &printValue);`
- Using lambda functions:
  - `connect(slider, &QSlider::valueChanged, [=] (int value) {...} );`

Connect – Pointers to Member Functions

› Qt5 components

```
connect(slider, &QSlider::valueChanged, spinbox, &QSpinBox::setValue);
```

› Overloaded functions:

```
connect(spin, qOverload<int>
      (&QSpinBox::valueChanged), slider, &QSlider::setValue);
connect(spin, static_cast<void (QSpinBox::*)(int)>
      (&QSpinBox::valueChanged), slider, &QSlider::setValue);
```

› Primary choice when connecting objects

√ Compile time errors
√ No special syntax for slots
√ `Q_OBJECT` not need for slots

 ex-connect-function-pointers

Connect – String-based Connections

› Used in Qt4, still used in connecting to slots in QML:

```
connect(slider, SIGNAL(valueChanged(int)), spinbox, SLOT(setValue(int)));
```

› Receiving object:

✗ Need to declare the slot in a `Q_SLOTS` section

✗ Need the `Q_OBJECT` macro

✗ Need to have moc run on it

✗ Only run-time errors

√ Overloaded slots are easy

√ Existing Qt4 code do not need to be rewritten

Connect – Non-Member

› Using non-member functions:

```
static void printValue(int value) {
        qDebug( "value = %d", value );
}
connect(slider, &QSignal::valueChanged, &printValue);
```

√ No slots syntax, no `Q_OBJECT`, no moc
√ Compile time errors
√ Any function, e.g. the return value of `std::bind`

Connect – Lambda Functions

- Using lambda functions:
  - Add `CONFIG += c++11` to your project file, if working prior Qt 5.7

```
connect(slider, &QSlider::valueChanged,
        [] (int value) { qDebug("%d", value); });
```

√ No slots syntax, no `Q_OBJECT`, no moc
√ Compile time errors
√ No need for an extra function

Variations of Signal/Slot Connections

Signal	Connect to	Slot(s)
one	√	many
many	√	one
one	√	another signal
`rangeChanged(int,int)`	√	`setRange(int,int)`
`rangeChanged(int,int)`	√	`update()`
`valueChanged(int)`	√	`setValue(int)`
`valueChanged(int)`	√	`update()`
`valueChanged(int)`	✗	`setRange(int,int)`
`valueChanged(int)`	√	`setValue(float)`
`textChanged(QString)`	✗	`setValue(int)`

Connection Types

- `Qt::AutoConnection` – the default connection type
  - Actual connection type determined, when a signal is emitted
  - Connection type depends on signaling and receiving objects' thread affinity
- `Qt::DirectConnection` - An immediate slot function call
- `Qt::QueuedConnection` – A delayed function call
  - Signaling object sends an event to the receiving object
- `Qt::BlockingQueuedConnection` – Signaling thread blocks, until the slot is executed
- `Qt::UniqueConnection` – Prevents multiple instances of the same connection

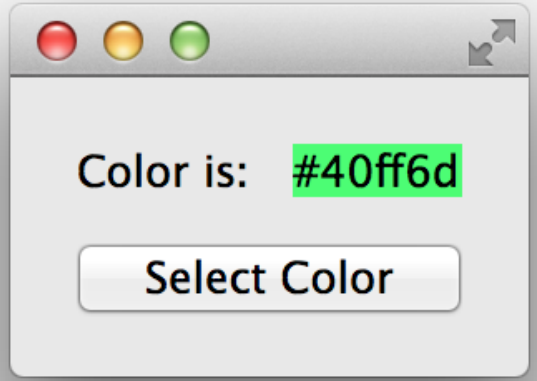
Disconnecting Signals

- Disconnect everything connected to an object's signals
  - `disconnect(senderObject, 0, 0, 0);`
  - `// sender (cannot be 0), signal, receiver, slot`
- Disconnect everything connected to a specific signal
  - `disconnect(senderObject, &Object::theSignal, 0, 0);`
- Disconnect a specific receiver
  - `disconnect(senderObject, 0, receiver, 0);`
- Disconnect a specific connection
  - `disconnect(senderObject, &Object::theSignal, receiver, &Object::theSlot);`

- If connected to lambda expressions, use the following overloaded version of the `disconnect()` function:
  - `bool QObject::disconnect(const QMetaObject::Connection &connection);`

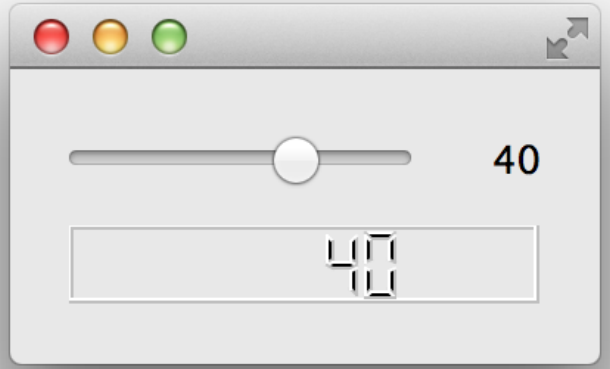
Lab – Connect to Click

- **Create an application as shown here**
  - Clicking on "SelectColor" updates label with color's name
- **Hints**
  - `QColorDialog::getColor()` to fetch a color
  - `QColor::name()` to get the color name
- **Optional**
  - In `QColorDialog`, honor the user clicking "cancel"', and provide it with the current color to start from
- Set the selected color as the label's background
  - Hint: see `QPalette`
  - Hint: see `QWidget::setAutoFillBackground()`



Lab – Source Compatibility

- **Implement custom slider**
  - API compatible with `QSlider`
  - Shows current value of slider
- **To create custom slider**
  - Use `QSlider` and `QLabel`
- **To test slider**
  - `main.cpp` provides test code
  - `QLCDNumber` is part of test code
- **Optional:**
  - Discuss pros and cons of inheriting from `QSlider` instead of using an instance in a layout



Event Processing

Qt is an event-driven UI toolkit

`QCoreApplication::exec()` runs the event loop

1.Generate Events

- By input devices: keyboard, mouse, etc.
- By Qt itself (e.g. timers)

2.Queue Events

- By event loop

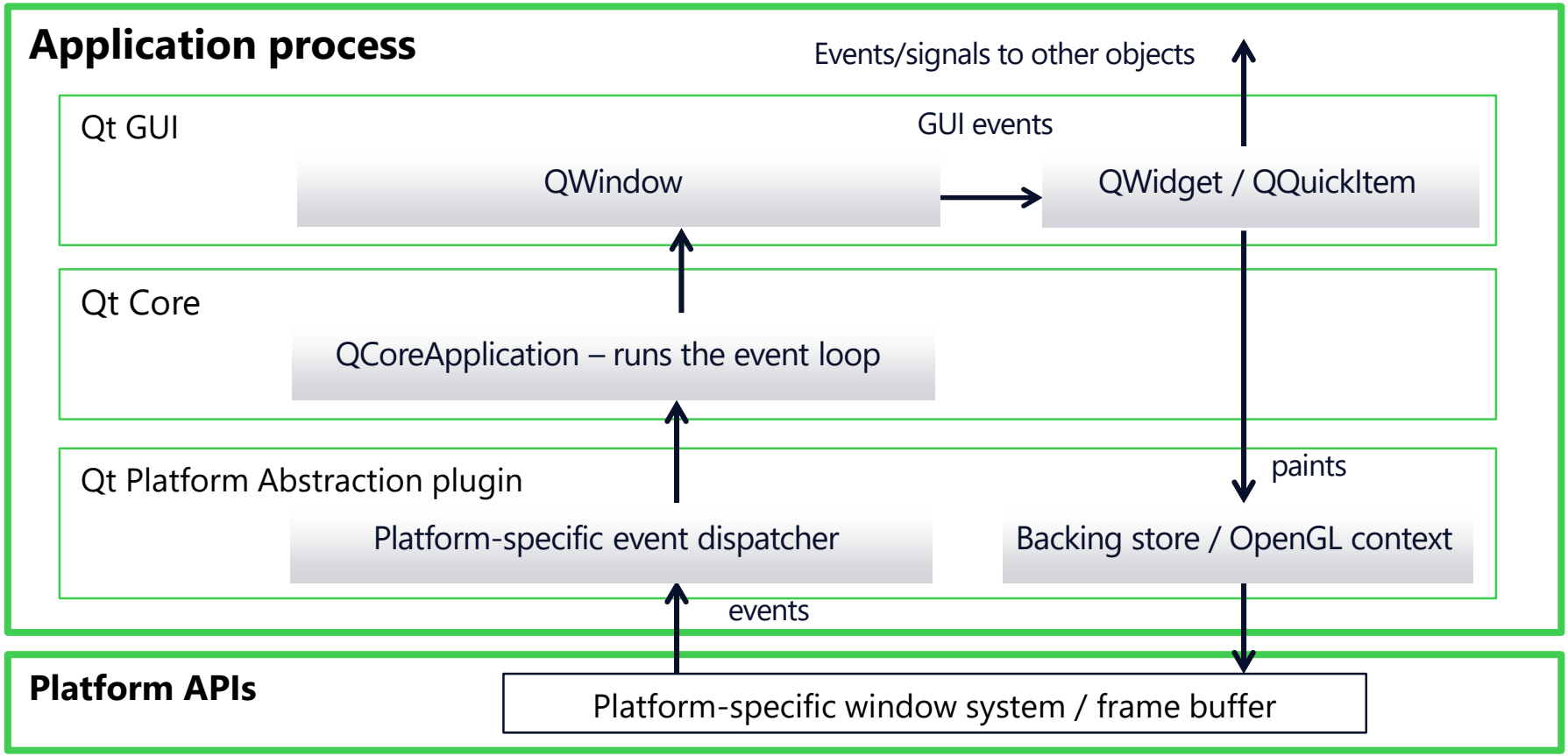
3.Dispatch Events

- By `QApplication` to receiver: `QObject`
- *Key events sent to widget/Qt Quick item with focus*
- *Mouse events sent to widget/Qt Quick item under cursor*

4.Handle Events

- By `QObject` event handler methods

Event Processing



1. Generate Events

- Spontaneous events
  - Asynchronous, e.g. mouse, touch or key press events
  - Generated by the underlying window system
  - Read and queued by a QPA plug-in
  - Processed by the event loop
- Synthetic events
  - Created in Qt program
  - Synchronous or asynchronous

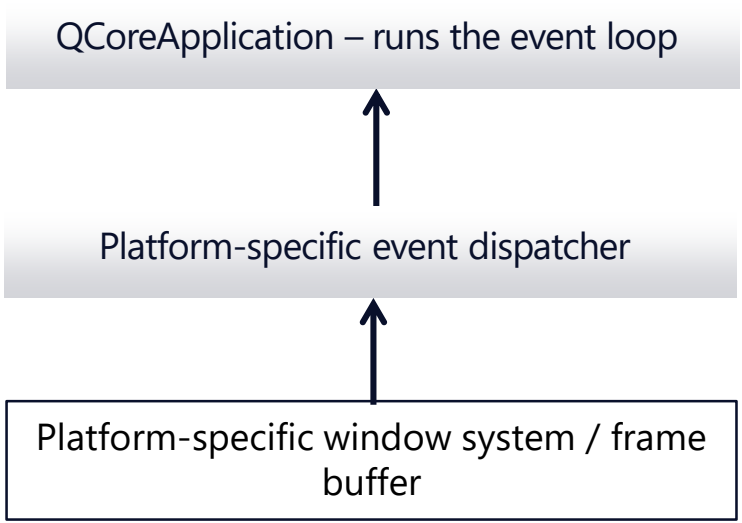
Synchronous and Asynchronous Events

- Asynchronous synthetic events
  - Queued by Qt and processed by the event loop
  - *In some cases Qt can compress several posted events into one!*
  - E.g., queued connections, `QWidget::update()`, `QQuickItem::update()`
  - Queued with `QCoreApplication::postEvent(QObject *receiver, QEvent *event, int priority = Qt::NormalEventPriority)`
- Synchronous synthetic events
  - Sent directly to the target object
  - E.g., `QWidget::repaint()`
  - Sent with `QCoreApplication::sendEvent(QObject *receiver, QEvent *event)`
- In some cases it is simpler to use signals and slots than sending and posting signals

ex-synthetic-events

2. Queue Events

- Spontaneous and asynchronous synthetic events queued by the event loop
- `QCoreApplication::exec()` runs the main event loop
- The event loop tells the event dispatcher what to do and how to do it
  - For example: wait for an event, exclude user input events



Event Loops

- All event loops in Qt are represented by the class `QEventLoop`
  - `QEventLoop` can be used to launch *local event loops*
- Event loop is modal
  - `QApplication::exec()`, `QEventLoop::exec()`, etc. are blocking
  - `QDialog`, `QMenu` also have `exec()` functions
  - Continues processing until told to stop by calling `exit()`
- Event-loop is recursive
  - Any of the `exec()` functions can re-curse
  - Except for `QApplication::exec()`
  - Use with caution, unexpected recursion can happen

ex-event-loop

3. Dispatch Events

- Platform-specific event dispatchers derived from `QAbstractEventDispatcher`
- Event dispatcher receives events from the window system and other sources via the event loop
  - Delivers events to `QCoreApplication`

Event Processing in Long-Running Tasks

- It is possible to process events during long-running tasks
  - `QAbstractEventDispatcher::processEvents(QEventLoop::ProcessEventsFlag flags)`
- For example, show progress indicator, blocking user interaction, in a single-threaded application
  - `QCoreApplication::processEvents(QEventLoop::ExcludeUserInputEvents)`
- Or check, if a timer event occurred in a worker thread running a task
  - `QAbstractEventDispatcher *dispatcher = QThread::eventDispatcher();`
  - `dispatcher->processEvents(QEventLoop::AllEvents);`

4. Event Handling

1. Dispatcher delivers all events of all application threads to `QCoreApplication::notify()`
   - Can be used to check, if a certain event exists, for example
2. After notify all GUI thread events are delivered to application-global event filters, if there are any
3. Object-specific event filters receive the event after application-global event filters
4. If none of the event filters remove the event, receiving object's `event()` function gets called
5. In case of a custom event, receiver object's `customEvent()` receives the event

ex-event-handling

Event Handling

- bool QObject::event(QEvent *) handles all events for this object
  - Re-implement for custom event handling
  - For example, to handlle touch events and gestures in widgets
  - Returning false results that the event is propagated to a parent

- Specialized event handlers for `QWidget` and `QQuickItem`
  - `void mousePressEvent(QEvent *)` for mouse clicks
  - void keyPressEvent(`QEvent *`) for key presses
  - `void touchEvent(QEvent *)` for touch handling in `QQuickItem` only
  - Event accepted by default
  - Ignored events propagated to a parent
  - `QEvent::accept()/QEvent::ignore()`

ex-all-events

Accepting and Ignoring QCloseEvent

- `QCloseEvent` delivered to top level widgets (windows)
- Accepting event allows window to close
- Ignoring event keeps window open

```
void Widget::closeEvent(QCloseEvent *event)
{
    if (windowShouldClose())
        event->accept();
    else
        event->ignore();
}
```

Event Filters

- Sometimes you need to add the same functionality to a number of different widgets
  - All widgets reacting on a certain event the same way
- ...or you might want block/hijack events aimed at a certain widget
  - E.g. a `QPushButton` on the screen
- The usual way to do this is to subclass each widget and re-implement the required event handler function(s)
  - May be cumbersome if all you try to obtain is the possibility to e.g. add common mouse movement handling to widgets or block certain events
- The solution: use one or more event filters

Event Filters

- Any Qt object, re-implementing the `eventFilter()` function and installed to one or more Qt objects
  - `bool eventFilter(QObject *receiver, QEvent *event)`
  - This method returns `true` if the filter handles the given event, `false` otherwise
  - `QObject::installEventFilter(QObject *targetObject)`
- An application-global event filter is installed on the `QCoreApplication` instance
  - Avoid in general because of increased event handling time
  - Good for debugging – e.g. is there a certain event type or not?

Further Notes

- Events are not handled at all for objects whose thread affinity is NULL (i.e. do not belong to any thread)
- An object and its event filter need to be in the same thread (i.e. have the same thread affinity) - otherwise `QObject::installEventFilter()` does nothing
  - Furthermore, if one of them is moved to another thread after installation, the event filter will not be called before both are in the same thread again
  - The filter is *not* removed by Qt

Questions And Answers

- How do you connect a signal to a slot?
- How would you implement a slot?
- How would you emit a signal?
- Can you return a value from a slot?
- When do you need to run `qmake`
- Where do you place the `Q_OBJECT` macro and when do you need it?
- What is the purpose of the event loop
- How does an event make it from the device to an object in Qt?

Summary

- **Between objects**
  - Signals & Slots
- **Between Qt and the application**
  - Events
- **Between Objects on threads**
  - Signal & Slots + Events

Lab – Event Handling

- You may build and run the lab project
- Add the following functionality without sub-classing
  - If the left mouse button is clicked on the quote button, the window is moved to the left
  - Right click on the button moves the window to the right
- In your event handler
  - Check the event type – `QEvent::type()`
  - In case of mouse press event, cast the event to mouse event
  - Cast the receiver Qt object to `QWidget`
  - Check, which mouse button is pressed (`Qt::LeftButton` or `Qt::RightButton`)
  - Get the widget position with `QPoint QWidget::pos()` and move it with `move(int x, int y)`
  - Add any other functionality, possibly required by your event handler

Contents

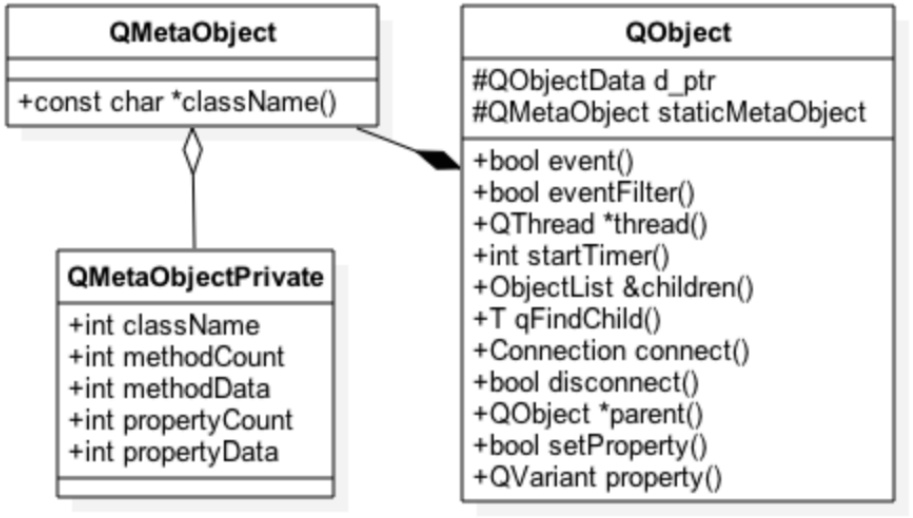
Objectives

Learn…

- …Qt object properties and their usage
- … about `QVariant` and meta-type system
- … how to register custom value types
- …benefits of using `QVariant`

Meta-Object System

- Adds dynamic features to Qt objects
- Features can be introspected via `QMetaObject`
  - Class and parent class names
  - Class info
  - Method information
    - Type: constructor, signal, slot, method
    - Signature
  - Properties
  - Enumerations and flags



- `Q_OBJECT` macro defines a static meta-object
  - Actual object data generated with the `moc` tool

Qt Object Class Declaration – QWindow

- `Q_OBJECT` macro
- One `enum` added to the meta-object
- Signals and slots
- Properties

```
class Q_QUICK_EXPORT QQuickWindow : public QWindow
{
    Q_OBJECT // Defines a static meta-object for the class
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
    Q_PROPERTY(QQuickItem* contentItem READ contentItem CONSTANT)

public:
    enum SceneGraphError { ContextNotAvailable = 1 };
    Q_ENUM(SceneGraphError)

Q_SIGNALS:
    void frameSwapped();
    void openglContextCreated(QOpenGLContext *context);
    void sceneGraphInitialized();

public Q_SLOTS:
    void update();
```

QMetaObject

- Contains meta-information about Qt objects
- Allows object introspection
  - `const char *QMetaObject::classname();`
  - `bool QMetaObject::inherits(const QMetaObject *metaObject)`
- Qt object may contain any custom information in `<name, value>` pairs
  - E.g., `Q_CLASSINFO("author", "Qt Developer");`
  - `QMetaClassInfo QMetaObject::classInfo(int index) const`

Object Introspection

```
QString objToString(const QObject *obj)
{
    QStringList result;
    const QMetaObject *meta = obj->metaObject();
    result += QString("class %1 : public %2 {")
        .arg(meta->className())
        .arg(meta->superClass()->className());
    for (auto i=0; i < meta->methodCount(); ++i)
    {
        const QMetaMethod method = meta->method(i);
        const QMetaMethod::MethodType methodType = method.methodType();
        if (methodType == QMetaMethod::Signal)
            signalPrefix = QStringLiteral("Q_SIGNAL");
        else if (methodType == QMetaMethod::Slot)
            signalPrefix = QStringLiteral("Q_SLOT");
        result += QString(" %1 %2 %3;")
                .arg(signalPrefix)
                .arg(QVariant::typeToName(method.returnType()))
                .arg(QString(method.methodSignature()));
    }
```


ex-object-introspection

QMetaObject – Method Invocation

- Any Qt object method marked as `Q_SLOT` or `Q_INVOKABLE` can be invoked
  - Provided allowed by method access specifiers

```
bool QMetaObject::invokeMethod(QObject *object,
                               const char *member,
                               Qt::ConnectionType type,
                               QGenericReturnArgument ret,
                               QGenericArgument val0, …)
```

  - Generic return values and arguments can be defined with corresponding macros
  - `Q_RETURN_ARG(Type, const Type &value) // Q_RETURN_ARG(int, x)`
  - `Q_ARG(Type, const Type &value)`

Property System

- Allows Qt objects to expose data as properties
  - Exposed properties are not limited to data members
- Platform and compiler independent
- Property access does not require any knowledge of the Qt object type
- Used by many Qt frameworks
  - Animation framework animates Qt object properties
  - Style sheets can be used to assign values to properties
  - QML engine manages property bindings between Qt objects

Providing Properties from QObject

```
Q_PROPERTY(Type name
           (READ getFunction [ WRITE setFunction ] |
           MEMBER memberVariable [( READ getFunction | WRITE setFunction )])
           // MEMBER is straightforward mapping to a member variable
           // If any processing is required, use READ / WRITE
           [ RESET resetFunction ] // Implemented, if reset is different from WRITE
           [ NOTIFY notifySignal ] // Emitted, when the property value changes
           [ REVISION int ]        // Used often in QML to define API revision
           [ DESIGNABLE bool ]     // Property accessible in Qt Designer
           [ SCRIPTABLE bool ]      // Property accessible in Qt Script
           [ STORED bool ]         // False, if property not stored by object
           [ USER bool ]           // One essential property can be marked USER
           [ CONSTANT ]            // Constant property, no WRITE or NOTIFY
           [ FINAL ]               // Property not overridden in subclasses
```

Providing Properties from QObject

```
class ExampleQtObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Employee employee MEMBER m_employee NOTIFY employeeChanged)
    Q_PROPERTY(AQtObject *objectProperty MEMBER m_object NOTIFY
objectPropertyChanged)
public:
    explicit ExampleQtObject(QObject *parent = 0);

Q_SIGNALS:
    void employeeChanged();
    void objectPropertyChanged ();

private:
    Employee m_employee;
    AQtObject *m_object;
};
```

Accessing Properties

- Property access methods
  - Take `QVariant` argument and return `QVariant` type

```
QVariant property(const char* name) const;
void setProperty(const char* name, const QVariant &value);
```

- If name is not declared as a `Q_PROPERTY`
  - ->**dynamic property**
  - Not accessible from Qt Quick

- Note:
  - `Q_OBJECT` macro required for properties to work
  - `QMetaObject` knows nothing about dynamic properties

Enumerations

- Enumerations and flags can be made accessible via the meta object
  - Can be used as property types as well
  - `Q_ENUM` – registers the enum type with both the meta-object and meta-type system
  - `Q_FLAG` – Registers flags (defined as `Q_DECLARE_FLAGS`) using `Q_ENUM`

```
class ExampleQtObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Regions regions MEMBER m_regions NOTIFY regionsChanged)
public:
    explicit ExampleQtObject(QObject *parent = 0);
    enum Region { Africa = 0x01, Americas = 0x02, Asia = 0x04, Australia = 0x08 };
    Q_DECLARE_FLAGS(Regions, Region)
    Q_FLAG(Region) // Q_FLAG registers the enumeration using Q_ENUM
Q_SIGNALS:
    void regionsChanged();
private:
    Regions m_regions;
};
```

Variants

- `QObject` property types can be any type, supported by `QVariant`
- Union for common Qt "value types" (copyable, assignable)
- Value type, supporting implicit sharing
- Custom types can be added
- Based on the meta-type system
  - Extends dynamic features to value types

Using Variants

- Easy to create `QVariant` objects as the inverse conversion is automatic for all supported data types

```
QVariant variant = QColor(Qt::red);
QVariant anotherVariant = QPixmap(QStringLiteral("image.png"));
qDebug() << anotherVariant.typeName(); // Logs "QPixmap"
CustomType customValueType;
QVariant yetAnotherVariant = QVariant::fromValue(customValueType);
```

- Conversion functions exist for QtCore types
  - For other types use template functions

```
QString string = variant.toString(); // QString is a type in Qt Core
// QSettings store <key, value> pairs and QSettings::value() returns QVariant
QFont textFont = settings.value("TextFont").value<QFont>());
```

Meta-Type System

- `QMetaType` manages named types
  - Types known by `QMetaType` can be accessed using `QVariant`
  - Provides registered types a `char *` name and `int` identifier
  - Allows creating, destroying, copying, and serializing assignable types
  - Custom types can be added to the meta-type system with `Q_DECLARE_METATYPE(Type)`
  - Custom streaming operators, type converter, and equality comparison functions may be registered

- Makes value types dynamic
  - Completes introspection functionality to non-`QObject` types by associating a type name to a type

```
#include <QMetaType>

class Contact {
// Class declaration
};

Q_DECLARE_METATYPE(Contact);
```

Custom Types and QVariant

```
Contact contact;
contact.setName("Peter");

QVariant variant = QVariant::fromValue(contact); // Custom type
Contact contact2 = variant.value<Contact>();

qDebug() << contact2.name();     // "Peter"
qDebug() << variant.typeName(); // prints "Contact"
int type(variant.userType());   // 1025 (QMetaType::User + 1)
qDebug() << qRegisterMetaType<Contact>() << qMetaTypeId<Contact>();

// New object created using int identifier
Contact *contact3(static_cast<Contact *>(QMetaType::create(type)));
contact3->setName("Ann");
```

Limitations on Custom Types

- Parameters in queued connections must be known by the meta type system
  - Parameters serialized into an event, which is passed between a signalling and receiving objects
  - Parameter objects re-created in the receiving object and parameters de-serialized from the event
- Add your custom type to the meta-object system using `Q_DECLARE_METATYPE()`
- Additionally, call `qMetaTypeId<MyType>()` or `qRegisterMetaTyoe<TypeDef>()` before making the first queued connection, if the signalling and receiving objects are in separate threads
  - This will enable run-time name resolution

Questions and Answers

- What benefits are there in Qt property system?
- Are there any drawbacks?
- Where properties are used in Qt?
- What kind of classes can be used as properties?
- Is it possible to use Qt objects as properties? Why or why not?
- How custom types can be used as properties?
- What is `QVariant`?
- How and where `QVariant` is used?
- How QVariant is related to meta-type system?
- When is it required to register custom types with `qMetaTypeId()` or `qRegisterMetaType()`?

Summary

- Qt Object properties allow accessing object data without knowing the actual type of the object
- Several frameworks in Qt use properties
  - Property animations
  - Style sheets
  - QML
- Any type, known by `QVariant` (meta-type system) can be used as a property
- `QVariant` extends object introspection to value types
  - Value types are given an char pointer name and integer identifier
- Custom assignable types can be registered to the meta-type system

Contents

Objectives

Learn…

- …how to use and manipulate string efficiently in Qt
- …file usage and data streaming
- …Qt item containers
- …best practices with item containers
- …Java-style iterators

QString and QByteArray

- Container for 16-bit Unicode 4.0 `QChars`
  - Support for 32-bit unicode characters
  - Uses `QTextCodec` to interpret characters
- `QByteArray` provides a container for raw data and 8-bit characters
  - May be used in programs with strict memory requirements
- Both classes implicitly shared
  - `QString string1("abc");  // Makes a deep copy of “abc”`
  - `QString string2(string1);  // Makes a shallow copy of string1`
  - Both strings share the same data, until either string changes the shared data

String Creation

Strings can be created in a number of ways:

- Conversion constructor and assignment operators:

```
QString emptyAndNullString; QString emptyString(“”);
QString str("abc"); // Uses QString::fromUtf8() function
str = "def”;
```

- From a char pointer using the static functions:

```
QString text = QString::fromLatin1("Hello Qt");
QString text = QString::fromUtf8(inputText);
QString text = QString::fromLocal8Bit(cmdLineInput);
QString text = QStringLiteral("Literal string"); (Assumed to be UTF-8)
```

String Creation

- From a number overloaded static functions

```
QString n = QString::number(1234); // (u)int, u(long), double
```

- From char pointer with translations:

```
QString text = QObject::tr("Hello Qt");
```

- From standard strings

```
QString string(QString::fromStdString(std::string("hello")));
QString string(QString::fromStdU16String(u"Europe:\u20ac Japan:\u00a5"));
QString string(QString::fromStdU32String(U"Europe:\u20ac Japan:\u00a5"));
```

String Creation and Performance

- `QString` is “expensive” to create from double-quoted ASCII literals
  - String allocation, deep copy of the data, decoding using the selected codec
  - Automatic conversion can be disabled with `DEFINES += QT_NO_CAST_FROM_ASCII`
  - In this case, `QLating1String` can be used as a thin wrapper to char pointer string
- Avoid creating temporary `QString` objects
  - Many member functions (assignment, comparison, insert, replace etc.) work directly with `const char *` without a conversion to `QString`
  - Faster to compare `QString` as `aStr == “hello”` or `aStr == QLatin1String(“hello”)` than `aStr == QString(“hello”)`
- `QStringLiteral` macro generates data for a `QString` at compile-time
  - Light-weight to create a `QString` using `QStringLiteral`

Text Processing with QString

- operator+ and operator+=
  - `QString str = str1 + str2;`
  - `fileName += ".txt";`

- `simplified()` // removes duplicate whitespace
- `left(), mid(), right()` // part of a string

- `leftJustified(), rightJustified()` // padded version

```
QString s = "apple";
QString t = s.leftJustified(8, '.'); // t == "apple..."
```

- For heavy string processing, like a parser, use `QStringRef`
  - Reduces the number of memory allocations, but may result to more complicated code

Extracting Data from the Strings

- Numbers:

```
QString text = ...;
nt value = text.toInt();
float value = text.toFloat();
```

- Strings:

```
QString text = ...;
QByteArray bytes = text.toLatin1();
QByteArray bytes = text.toUtf8();
QByteArray bytes = text.toLocal8Bit(); // bytes = qPrintable(text);
```

- Characters:

```
const QChar char = text.at(42);
QCharRef char = text[42];   // Allows modifying text
```

QByteArray

- Obtaining raw character data from a `QByteArray`:

```
char *str = bytes.data();
const char *str = bytes.constData();
```

WARNING:

- Character data is only valid for the life time of the byte array
- Calling a `non-const` member of bytes also invalidates the pointer
- Either copy the character data or keep a copy of the byte array

Formatted Output with QString::arg()

```
int i = ...;
int total = ...;
QString fileName = ...;
QString status = tr("Processing file %1 of %2: %3").arg(i).arg(total).arg(fileName);
double d = 12.34;
QString str = QString::fromLatin1("delta: %1").arg(d,0,'E',3);
// str == "delta: 1.234E+01"
```

- Safer: `arg(QString,...,QString)` ("multi-arg()")
  - But: only works with `QString` arguments
- Strings may be formatted using similar syntax to `printf() – QString::asprintf();`
  - Prefer `QString.arg()` in new code, because it supports unicode seamlessly and is type-safe

Text Processing with QString

- `length()`
- `endsWith()` and `startsWith()`
- `contains()`, `count()`
- `indexOf()` and `lastIndexOf()`

Expression can be characters, strings, or regular expressions

Text Processing with QStringList

- `QString::split()`,`QStringList::join()`
- `QStringList::replaceInStrings()`
- `QStringList::filter()`

Text Processing with Regular Expressions

- `QRegularExpression` supports
  - Regular expression matching
  - Wildcard matching

```
QRegularExpression rx("^\\d\\d?$"); // match integers 0 to 99
rx.match("123"); // QRegularExpressionMatch(Valid, no match)
rx.match("-6");  // QRegularExpressionMatch(Valid, no match)
rx.match("6");   // QRegularExpressionMatch(Valid, has match: 0:(0, 1, "6"))
```

Essential Classes

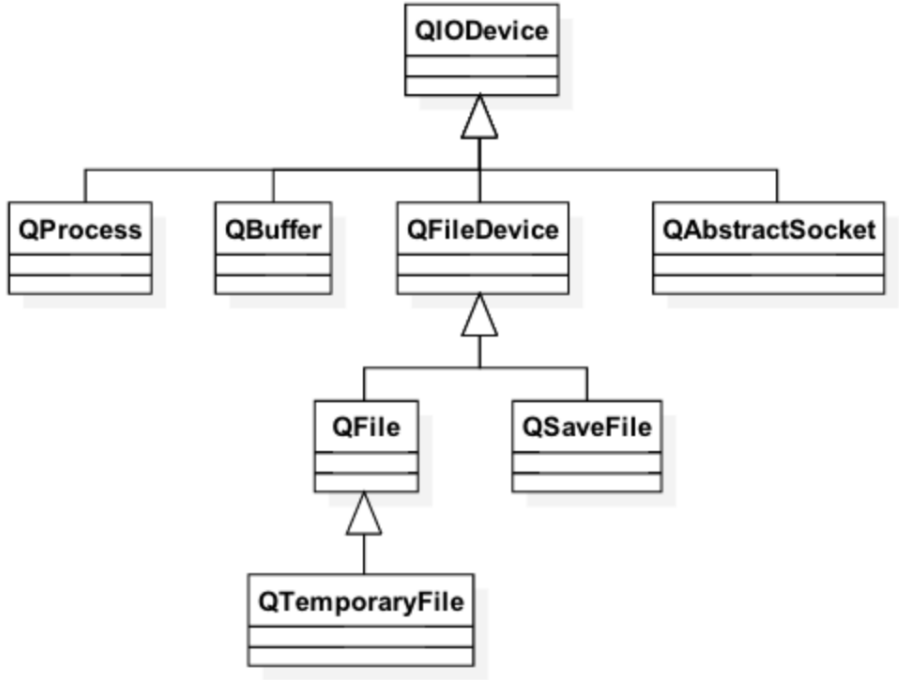
- `QDir`
  - Navigate between folders – `setPath(const QString &)`
    - Qt uses "/" as a universal directory separator
  - Create, remove, and rename folders
  - Get file info – `QFileInfoList QDir::entryInfoList()`
  - Get files in the directory – `QStringList QDir::entryList()`

- `QFileInfo`
  - File name, owner, group
  - Last modified, last read
  - Permissions

- `QFile`
  - Create, remove, rename files and create links
  - Read and write data

File I/O

- `QFile` inherits `QIODevice`
- `QIODevice`
  - Abstract class, used as an interface to provide device independent I/O features
- By default file data is buffered
  - Buffering can be bypassed with `Unbuffered` open mode flag
  - Except in Windows, where data is always buffered



Reading and Writing Data with QIODevice

- Variety of functions available
  - `bool getChar(char *c), bool putChar(char c)`
  - `QByteArray readLine(qint64 maxSize)`
  - `qint64 read(char *data, qint64 maxSize)`
  - `qint64 write(const QByteArray *byteArray)`
  - `QByteArray readAll()`

- Position where data is written to or read from
  - Returned by `qint64 pos()`
  - `qint64 bytesAvailable()` returns bytes available after the position
  - Position may be changed with `seek(qint64 pos)` or `reset()`
  - `bool atEnd()` returns true in the end of the file

File I/O

```
QFile file("data.txt");
if (!file.open(QFile::ReadWrite | QIODevice::Text)) ; // Handle error

const auto fileSize(file.size());
QByteArray readBuffer;
// Read file character by character
char c;
while (file.getChar(&c))
    readBuffer.append(c);

// Read data by chunks
readBuffer.clear();
file.reset();
const int bufferSize(10); // Intentionally extremely small for demo purposes
char chunk[bufferSize];
auto readBytes(0);
while ((readBytes = file.read(chunk, bufferSize)) > 0)
    readBuffer.append(chunk, readBytes);
```

Streaming

- Data is often read and written using `QDataStream` or `QTextStream`
  - Add extra information about the data to the stream
  - Can operate with any `QIODevice`, a `QByteArray` or a `QString`

- `QTextSream`
  - Convenient for reading and writing words, lines, and numbers
  - Commonly used to read console input and write to console output
  - Locale-aware

- `QDataStream`
  - Portable between hardware architectures and operating systems
  - Format depends on the stream version used
  - Not human-readable
  - Complex types broken to primitive types, which are then serialized to the stream

Data Stream

```
Data data; // Custom assignable type, registered with the meta-type system
data.setString("Hello World");
data.setValue(42);

QTemporaryFile file;
if (!file.open()) { qDebug() << "Temp file opening failed."; return EXIT_FAILURE; }

QDataStream stream(&file);
// If a custom type is searilzed using QVariant,
// serialization operators need to be provided
#ifndef USE_VARIANT
stream << data;
#else
qRegisterMetaTypeStreamOperators<Data>("Data");
QVariant variantData(QVariant::fromValue(data));
stream << variantData;
#endif
```

Memory-Mapped Files

- Data manipulated via a memory pointer
- Changes stored into a file
  - Unless the file is open in mode: `QFileDevice::MapPrivateOption`
- Memory keeps mapped, until unmapped or file object destroyed

```
if (!file.open(QFile::ReadWrite | QIODevice::Text))
    ; // Error handling
uchar *mappedPointer = file.map(0, file.size());
QByteArray array(reinterpret_cast<char *>(mappedPointer));
array.replace("Dickens", "DICKENS");
memcpy(mappedPointer, array.constData(), file.size());
```

Container Classes

General purpose template-based container classes

- `QList<QString>` - *Sequence Container*
- Other: `QVector, QLinkedList, QStack, QQueue...`
- `QMap<int, QString>` - *Associative Container*
- Other: `QHash, QSet, QMultiMap, QMultiHash, QCache`

Qt's Container Classes compared to STL

- Lighter, safer, and easier to use than STL containers
- If you prefer STL, feel free to continue using it.
- Methods exist that convert between Qt and STL
  - e.g. you need to pass `std::list` to a Qt method

Using Containers

› Using `QList`

```
QList<QString> list({ "one", "two", "three" }); // list << "one" << "two"...
QString item1 = list[1]; // "two"
for (const QString &item : list) {
    qDebug() << item; // Do somethinh with item reference
}
int index = list.indexOf("two"); // returns 1
```

› Using `QMap`

```
QMap<QString, int> map({ { "Norway", 47 }, { "Italy", 39 } });
auto value = map["France"]; // inserts key if it does not exist
if (map.contains("Norway")) {
    int value2 = map.value("Norway"); // recommended lookup
}
```

Algorithm Complexity

How fast is a function when number of items grow?

› Sequential Container (*all complexities are amortized*)

	Lookup	Insert	Append	Prepend
`QList`	O(1)	O(n)	O(1)	O(1)
`QVector`	O(1)	O(n)	O(1)	O(n)
`QLinkedList`	O(n)	O(1)	O(1)	O(1)

› Associative Container (*all complexities are amortized*)

	Lookup	Insert
`QMap`	O(log(n))	O(log(n))
`QHash`	O(1)	O(n)

Storing Classes in Qt Container

- Class must be an *assignable data type*
- Class is *assignable*, if:

```
class Contact {
public:
    Contact() {} // default constructor
    Contact(const Contact &other); // copy constructor
    // assignment operator
    Contact &operator=(const Contact &other);
};
```

- *If copy constructor or assignment operator is not provided*
  - C++will provide one (uses member copying)
- *If no constructors provided*
  - Empty default constructor provided by C++

Requirements on Container Keys

- Type `K` as key for `QMap`:

```
bool K::operator<(const K &)
bool operator<(const K &, const K &)

bool Contact::operator<(const Contact &c);
bool operator<(const Contact &c1, const Contact &c2);
```

- Type `K` as key for `QHash` or `QSet`:

```
bool K::operator==(const K &)
bool operator==(const K &, const K &)
uint qHash(const K &)
```

- Pay attention to the key type
  - It's heavier to calculate hash value from `QString` than `long`

Memory Usage and Performance

- `QVector<T>`, `QString`, and `QByteArray` store their items contiguously in memory
- `QList<T>` maintains an array of pointers to the items it stores
  - Fast index-based access
  - Very memory efficient as an object
  - `QList` is beneficial for movable types of size `void *`

- To avoid re-allocations, it is possible to reserve space for a list and vector
  - Still list data may require extra allocation for non-movable types
  - By default, a list allocates 4kB chunks for movable types and 50% more for non-movable types
  - A vector doubles its memory in re-allocation

Custom Types and Performance

- Provide type information for your custom types
  - Allow Qt containers to choose appropriate storage methods

- `Q_DECLARE_TYPEINFO(Type, Flags)`
  - `Q_PRIMITIVE_TYPE`
    - No constructor or destructor, typically enums and flags
    - `memcpy()` creates a valid independent copy of the object
  - `Q_MOVABLE_TYPE`
    - Type has constructor and/or destructor
    - Can be moved using `memcpy()`
  - `Q_COMPLEX_TYPE` (the default)
    - Type has constructor and/or destructor
    - May not be moved in memory

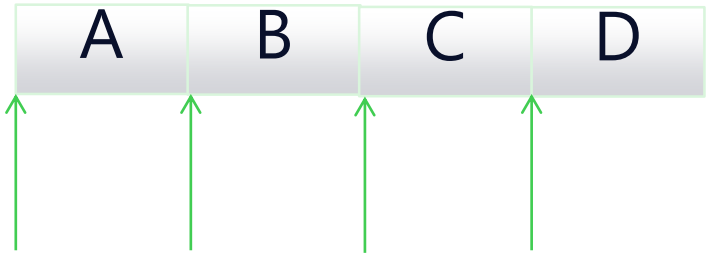
Iterators

- Allow reading a container's content sequentially
- Java-style iterators: simple and easy to use
  - `QListIterator<...>` for read
  - `QMutableListIterator<...>` for read-write
- STL-style iterators slightly more efficient
  - `QList::const_iterator, QList::const_reverse_iterator` for read
  - `QList::iterator(), QList::reverse_iterator` for read-write
- Same works for `QSet`, `QMap`, `QHash`, ...

Iterators Java style

- Example `QList` iterator

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> iterator(list);
```



- Forward iteration

```
while(iterator.hasNext()) {
    qDebug() << iterator.next(); // A B C D
}
```

- Backward iteration

```
iterator.toBack(); // position after the last item
while(iterator.hasPrevious()) {
   qDebug() << iterator.previous(); // D C B A
}
```

Modifying During Iteration

- Use *mutable* versions of the iterators (e.g. `QMutableListIterator`)

```
QList<int> list({ 1, 2, 3, 4});
QMutableListIterator<int> iterator(list);
while (iterator.hasNext()) {
    if (iterator.next() % 2 != 0)
        iterator.remove();
}
// list contains now 2, 4
```

- `remove()` and `setValue()`
  - Operate on items just jumped over using `next()` / `previous()`
- `insert()`
  - Inserts item at current position in sequence
  - `previous()` reveals just inserted item

Iterating Over QMap and QHash

- `next()` and `previous()`
  - Return Item class with `key()` and `value()`

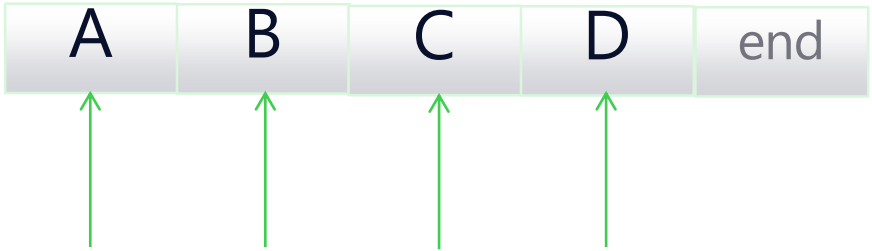
- Alternatively use `key()` and `value()` from iterator

```
QMap<QString, QString> map;
map["Paris"] = "France";
map["Guatemala City"] = "Guatemala";
map["Mexico City"] = "Mexico";
map["Moscow"] = "Russia";

QMutableMapIterator<QString, QString> iterator(map);
while (iterator.hasNext()) {
    if (iterator.next().key().endsWith("City"))
        iterator.remove();
}
// map now "Paris", "Moscow"
```

STL-style Iterators

- Example `QList` iterator

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator iterator;
```



- Forward mutable iteration

```
for (iterator = list.begin(); iterator != list.end(); ++iterator) {
    *iterator = (*iterator).toLower();
}
```

- Backward mutable iteration

```
iterator = list.end();
while (iterator != list.begin()) {
    --iterator;
    *iterator = (*iterator).toLower();
}
```

The Q_FOREACH Keyword

- It is a macro, feels like a keyword

```
Q_FOREACH (const QString& string, list) {
    if (string.isEmpty())
        break;
    qDebug() << string;
}
```

- Modifying the container while iterating
  - Results in container being copied
  - Iteration continues in unmodified version
  - Not possible to modify item in the container
  - Do not copy the items –use const & to optimize performance

- Prefer using the range-loop as `Q_FOREACH` will be removed in the future

```
for (const QString &string : list) { }
```

Algorithms

- STL-style iterators are compatible with the STL algorithms
  - Defined in the STL `<algorithm>` header
- Qt has own algorithms
  - Defined in `<QtAlgorithms>` header
  - Mostly deprecated!
- For parallel (i.e. multi-threaded) algorithms use `QtConcurrent` name space
- *If STL is available on all your supported platforms you can choose to use the STL algorithms*
  - The collection is much larger than the one in Qt

Algorithms

- Sort items in a range: `std::sort`
- Stable sort: `std::stable_sort`
- Search for a value: `std::find`
- Check if two ranges are the same: `std::equal`
- Copy items from one range to another: `std::copy`
- Copy backwards: `std::copy_backward`
- Count the number of items matching the search criteria: `std::count`

Algorithms

```
QList<int> list({ 3, 3, 2, 2, 7, 2, 8 });
int countOf2(std::count(list.begin(), list.end(), 2));

QList<QString> stringList({ "one", "two", "three" });
QVector<QString> vector(3);
std::copy(stringList.begin(), stringList.end(), vector.begin());
// vector: [ "one", "two", "three" ]

std::sort(vector.begin(), vector.end(), qLess<QString>());
```

Questions And Answers

- How `QString` stores data?
- Does Qt support 4-byte Unicode?
- What is the complexity of copying a `QString` with n characters?
- What options are there to use string literals?
- What is the difference between `QByteArray` and `QString`?
- What are the differences between `QTextStream` and `QDataStream`? Which one should be preferred in general? Why?
- What item containers are there in Qt?
- Which item containers should be preferred? Why?
- What should be considered, when inserting items into the container?
- Performance wise, should you prefer Java-style or STL-style iterators?
- Are there any limitations using STL-style iterators with Qt containers? Why or why not?

Summary

- `QString` is a container of 16-bit Unicode `QChars`
  - 32-bit Unicode is supported as well
- There are plenty of ways to optimize string usage
  - Avoid creating unnecessary temporary `QString` objects from char pointers
  - Avoid copying `QStrings`, even though it has constant complexity – use references
  - Consider using string literals or `QLatin1String`
  - When handling raw or 8-bit data, consider using `QByteArray`
- Item containers are implicitly shared
  - Avoid copying/changing the container, when iterated with an STL-style iterator
- As default choice, `QVector` and `QHash` should be used
- STL-style iterators may have better performance than Java-style iterators

Lab – Item Containers

- Add 10,000 `QString` items into `QMap` using `QPoint` as a key
  - `QPoint` x and y values should both loop 100 times
  - Performance wise, is it a good idea to have `QPoint` as a key?
  - Comment out `operator<` for `QPoint`. Try to compile the program. What do you observe?
- Use either Java-style or STL-style iterators to go through all the items in the container
  - `qDebug() << "Takes a stream, which can be shown in the debug console"`
- Remove every third item in the container
  - Print out the result
- Modify every fifth item in the container
  - Print out the result
- Use either Qt or STL algorithms to calculate the number of modified items

Thank You!

www.qt.io