Qt World Summit 2017

October 10-12 | Berlin, Germany

Multithreading with Qt

Giuseppe D'Angelo, Senior Software Engineer at KDAB

Agenda

KDAB

Do you know what a thread is?

QThread

- **QThread**
- Synchronization
- Thread safety in Qt
- Qt and the Standard Library threading facilities

QThread

- `QThread` is the central class in Qt to run code in a different thread
- It's a `QObject` subclass
  - Not copiable/moveable
  - Has signals to notify when the thread starts/finishes
- It is meant to *manage* a thread

QThread usage

- To create a new thread executing some code, subclass `QThread` and reimplement `run()`
- Then create an instance of the subclass and call `start()`
- Threads have priorities that you can specify as an optional parameter to `start()`, or change with `setPriority()`

QThread usage

```
1 class MyThread : public QThread {
2 private:
3     void run() override {
4         // code to run in the new thread
5     }
6 };
```

```
1 MyThread *thread = new MyThread;
2 thread->start(); // starts a new thread which calls run()
3 // ...
4 thread->wait(); // waits for the thread to finish
```

QThread usage

- Another way to create a QThread is by using `QThread::create()`
  - New in Qt 5.10
  - Similar to `std::thread`
- Returns a QThread instance
- The new thread is not started yet
  - So we can connect to its signals, set the priority, etc.

```
1 void calculate(int, int);
2
3 QThread *thread = QThread::create(calculate, arg1, arg2);
4 // ... finish the setup ...
5 thread->start(); // runs calculate in the new thread
6 // ...
7 thread->wait(); // waits for it to finish
```

QThread usage

- The thread will stop running when (some time after) returning from `run()`
- `QThread::isRunning()` and `QThread::isFinished()` provide information about the execution of the thread
- You can also connect to the `QThread::started()` and `QThread::finished()` signals
- A thread can stop its execution temporarily by calling one of the `QThread::sleep()` functions
  - Generally a *bad idea*, being event driven (or polling) is much much better
- You can wait for a `QThread` to finish by calling `wait()` on it
  - Optionally passing a maximum number of milliseconds to wait

QThread caveats

From a non-main thread you cannot:

- Perform any GUI operation
  - Including, but not limited to: using any `QWidget` / Qt Quick / `QPixmap` APIs
  - Using `QImage`, `QPainter`, etc. (i.e. "client side") is OK
  - Using OpenGL may be OK: check at runtime `QOpenGLContext::supportsThreadedOpenGL()`
- Call `Q(Core|Gui)Application::exec()`

QThread caveats

- Do not *ever* block the GUI thread
- Be sure to always destroy all the `QObjects` living in secondary threads before destroying the corresponding `QThread` object

Ensuring destruction of QObjects

- Create them on `QThread::run()` stack
- Connect their `QObject::deleteLater()` slot to the `QThread::finished()` signal
  - Yes, this will work
- Move them out of the thread

Ensuring destruction of QObjects

```
1  class MyThread : public QThread {
2  private:
3      void run() override {
4          MyQObject obj1, obj2, obj3;
5
6          QScopedPointer<OtherQObject> p;
7          if (condition)
8              p.reset(new OtherQObject);
9
10         auto anotherObj = new AnotherQObject;
11         connect(this, &QThread::finished,
12                 anotherObj, &QObject::deleteLater);
13
14         auto yetAnother = new YetAnotherQObject;
15
16         // ... do stuff ...
17
18         // Before quitting the thread, move this object to the main thread
19         yetAnother->moveToThread(qApp->thread());
20         // Somehow notify the main thread about this object,
21         // so it can be deleted there.
22         // Do not touch the object from this thread after this point!
23     }
24 };
```

QThread usage

There are two basic strategies of running code in a separate thread with QThread:

- Without an event loop
- With an event loop

QThread usage without an event loop

- Subclass `QThread` and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`

QThread usage without an event loop

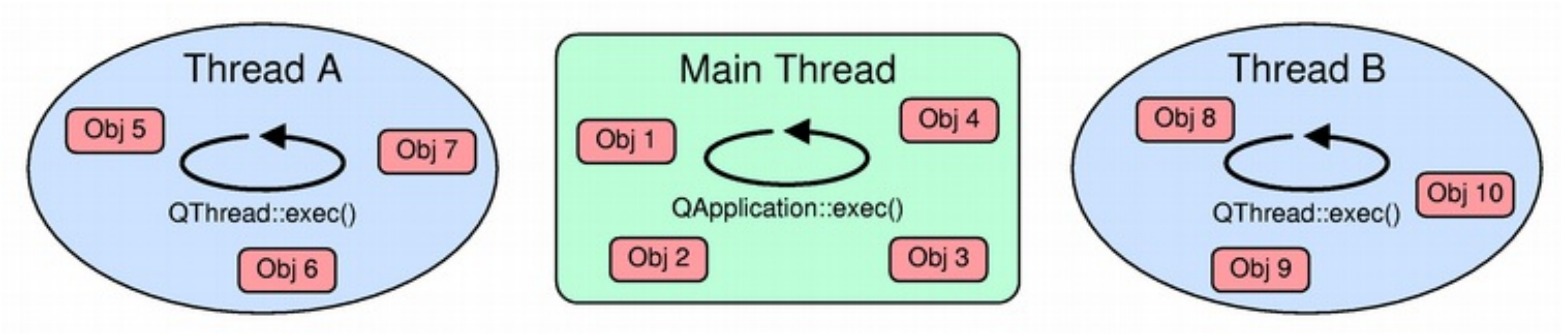
- Subclass `QThread` and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`

```
1 class MyThread : public QThread {
2 private:
3     void run() override {
4         loadFilesFromDisk();
5         doCalculations();
6         saveResults();
7     }
8 };
```

```
1 auto thread = new MyThread;
2 thread->start();
3 // some time later...
4 thread->wait();
```

QThread usage with an event loop

- An event loop is necessary when dealing with timers, networking, *queued connections*, and so on.
- Qt supports per-thread event loops:



- Each thread-local event loop delivers events for the QObjects living in that thread.

QThread usage with an event loop

- We can start a thread-local event loop by calling `QThread::exec()` from within `run()`:

```
 1 class MyThread : public QThread {
 2 private:
 3     void run() override {
 4         auto socket = new QTcpSocket;
 5         socket->connectToHost(...);
 6
 7         exec(); // run the event loop
 8
 9         // cleanup
10     }
11 };
```

- `QThread::quit()` or `QThread::exit()` will quit the event loop

- We can also use `QEventLoop`
  - Or manual calls to `QCoreApplication::processEvents()`

QThread usage with an event loop

- The default implementation of `QThread::run()` actually calls `QThread::exec()`
- This allows us to run code in other threads without subclassing QThread:

```
1  auto thread = new QThread;
2
3  auto worker = new Worker;
4
5  connect(thread, &QThread::started, worker, &Worker::doWork);
6  connect(worker, &Worker::workDone, thread, &QThread::quit);
7
8  connect(thread, &QThread::finished, worker, &Worker::deleteLater);
9
10 worker->moveToThread(thread);
11 thread->start();
```

Synchronization

- QThread
- **Synchronization**
- Thread safety in Qt
- Qt and the Standard Library threading facilities

KDAB

What is the single most important thing about threads?

Synchronization

- Any concurrent access to shared resources **must not result in a data race**
- Two conditions for this to happen:
  1. At least one of the accesses is a write
  2. The accesses are not *atomic* and no access *happens before* the other

Synchronization

Qt has a complete set of cross-platform, low-level APIs for dealing with synchronization:

- `QMutex` is a mutex class (recursive and non-recursive)
- `QSemaphore` is a semaphore
- `QWaitCondition` is a condition variable
- `QReadWriteLock` is a *shared mutex*
- `QAtomicInt` is an atomic `int`
- `QAtomicPointer<T>` is an atomic pointer to T

There are also RAII classes for lock management, such as `QMutexLocker`, `QReadLocker` and so on.

Mutex Example

```
1  class Thread : public QThread
2  {
3      bool m_cancel;
4  public:
5      explicit Thread(QObject *parent = nullptr)
6        : QThread(parent), m_cancel(false) {}
7
8      void cancel()   // called by GUI
9      {
10         m_cancel = true;
11     }
12
13 private:
14     bool isCanceled() const   // called by run()
15     {
16         return m_cancel;
17     }
18
19     void run() override { // reimplemented from QThread
20         while (!isCanceled())
21             doSomething();
22     }
23 };
```

Mutex Example (cont'd)

```
1  class Thread : public QThread
2  {
3      mutable QMutex m_mutex; // protects m_cancel
4      bool m_cancel;
5  public:
6      explicit Thread(QObject *parent = nullptr)
7        : QThread(parent), m_cancel(false) {}
8
9      void cancel() { // called by GUI
10         const QMutexLocker locker(&m_mutex);
11         m_cancel = true;
12     }
13
14 private:
15     bool isCanceled() const { // called by run()
16         const QMutexLocker locker(&m_mutex);
17         return m_cancel;
18     }
19
20     void run() override { // reimplemented from QThread
21         while (!isCanceled())
22             doSomething();
23     }
24 };
```

QThread's built-in cancel

KDAB

QThread actually has this already built-in:

- `QThread::requestInterruption()`, to set the flag
- `QThread::isInterruptionRequested()`, to check the flag

```
1  void run() override { // reimplemented from QThread
2      const int checkAtNthIteration = 10;
3
4      int iteration = 0;
5      while (true) {
6          ++iteration;
7          if (iteration == checkAtNthIteration) {
8              iteration = 0;
9              if (isInterruptionRequested())
10                 return;
11         }
12
13         doSomething();
14     }
15 }
```

Quick Quiz: Mutex Example

In this code:

```
explicit Thread(QObject *parent = nullptr)
  : QThread(parent), m_cancel(false) {}
```

don't you need to protect

```
m_cancel(false)
```

with `m_mutex`, too, like in `cancel()`?

```
1 void cancel() { // called by GUI
2     const QMutexLocker locker(&m_mutex);
3     m_cancel = true;
4 }
```

Thread safety in Qt

- QThread
- Synchronization
- **Thread safety in Qt**
- Qt and the Standard Library threading facilities

Reentrancy definitions

A function is:

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

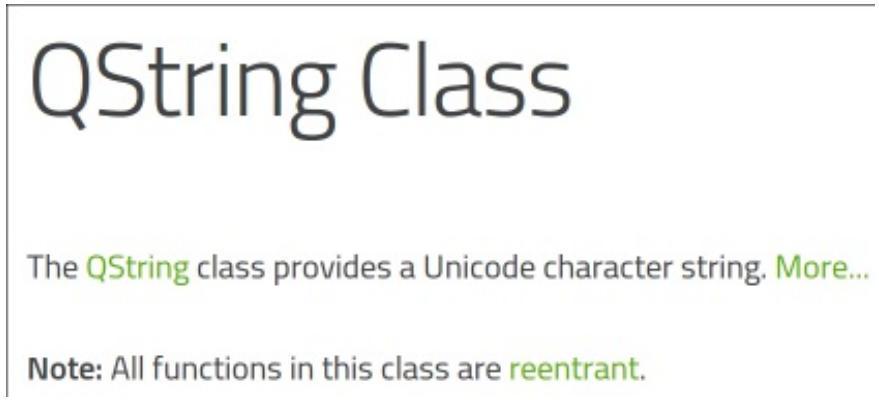
For classes, the above definitions apply to non-static member functions when invoked on the same instance. (In other words, considering the `this` pointer as an argument.)

Examples

- **Thread safe**:
  - `QMutex`
  - `QObject::connect()`
  - `QCoreApplication::postEvent()`
- **Reentrant**:
  - `QString`
  - `QVector`
  - `QImage`
  - value classes in general
- **Non-reentrant**:
  - `QWidget` (including all of its subclasses)
  - `QQuickItem`
  - `QPixmap`
  - in general, GUI classes are usable only from the main thread

Thread safety for Qt classes/functions

The documentation of each class / function in Qt has notes about its thread safety:



Unless otherwise specified, classes and functions are **non-reentrant**.

QObject: thread affinity

What about QObject?

QObject: thread affinity

What about QObject?

- `QObject` itself is thread-aware.
- Every `QObject` instance holds a reference to the thread it was created into (`QObject::thread()`)
  - We say that the object *lives in*, or *has affinity with* that thread
- We can move an instance to another thread by calling `QObject::moveToThread(QThread *)`

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance
- You can only call `moveToThread()` on a `QObject` from the same thread the object has affinity with (`moveToThread()` is non-reentrant)

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance
- You can only call `moveToThread()` on a `QObject` from the same thread the object has affinity with (`moveToThread()` is non-reentrant)

In practice: **it's easier to think of `QObject` as non-reentrant**, as it will make you avoid many mistakes.

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?
- Qt has a solution: **cross-thread signals and slots**

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?
- Qt has a solution: **cross-thread signals and slots**
- You can emit a signal from one thread, and have the slot invoked by another thread
  - Not just any thread: the thread the receiver object is living in

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
- Also, `qRegisterMetaType()` is required for the argument types passed

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
- Also, `qRegisterMetaType()` is required for the argument types passed
- We can force any connection to be queued:

```
connect(sender, &Sender::signal, receiver, &Receiver::slot, Qt::QueuedConnection);
```

QObject: queued connections example

```
 1 class MyThread : public QThread {
 2     Producer *m_producer;
 3 public:
 4     explicit MyThread(Producer *p, QObject *parent = nullptr)
 5         : QThread(parent), m_producer(p) {}
 6
 7     void run() override {
 8         Consumer consumer;
 9         connect(m_producer, &Producer::unitProduced,
10                 &consumer, &Consumer::consume);
11         exec();
12     }
13 };
14
15 // in main thread:
16 auto producer = new Producer;
17 auto thread = new MyThread(producer);
18 thread->start();
19 producer->startProduction();
20
21 // Producer::unitProduced gets emitted some time later from the main thread,
22 // Consumer::consume gets run in the secondary thread
```

QObject: queued connections example (2)

```
1  // Same as before, but without the race
2
3  auto producer = new Producer;
4  auto consumer = new Consumer;
5  auto thread = new QThread;
6
7  connect(m_producer, &Producer::unitProduced,
8          consumer, &Consumer::consume);
9  connect(thread, &QThread::finished,
10         consumer, &QObject::deleteLater);
11
12 consumer->moveToThread(thread);
13
14 thread->start();
15
16 // Producer::unitProduced gets emitted some time later from the main thread,
17 // Consumer::consume gets run in the secondary thread
```

QObject: queued connections example (3)

```
1  class MyThread : public QThread {
2  public:
3      explicit MyThread(QObject *parent = nullptr)
4          : QThread(parent) {}
5
6  private:
7      void run() override {
8          emit mySignal();
9      }
10
11 signals:
12     void mySignal();
13 };
14
15 // in main thread:
16 auto thread = new MyThread;
17 connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
18 thread->start();
```

QObject: queued connections example (3)


```
1  class MyThread : public QThread {
2  public:
3      explicit MyThread(QObject *parent = nullptr)
4          : QThread(parent) {}
5
6  private:
7      void run() override {
8          emit mySignal();
9      }
10
11 signals:
12     void mySignal();
13 };
14
15 // in main thread:
16 auto thread = new MyThread;
17 connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
18 thread->start();
```

- It is perfectly OK to add signals to `QThread`
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
- `someSlot()` gets invoked by the main thread's event loop

QObject: queued connections example (4)

```
1  class MyThread : public QThread {
2      Socket *m_socket;
3  public:
4      explicit MyThread(QObject *parent = nullptr)
5          : QThread(parent) {}
6
7  private:
8      void run() override {
9          m_socket = new Socket;
10         connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
11         m_socket->connectToHost(...);
12         exec();
13     }
14
15 private slots:
16     void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
17 };
```

QObject: queued connections example (4)

```
1  class MyThread : public QThread {
2      Socket *m_socket;
3  public:
4      explicit MyThread(QObject *parent = nullptr)
5          : QThread(parent) {}
6
7  private:
8      void run() override {
9          m_socket = new Socket;
10         connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
11         m_socket->connectToHost(...);
12         exec();
13     }
14
15 private slots:
16     void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
17 };
```

- QThread is a QObject and as such has its own thread affinity (it's the thread that created the MyThread instance, *not itself!*)
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
  - *This is not what we wanted!*
- Huge recommendation: **avoid adding slots to QThread**

Qt and the Standard Library threading facilities

- QThread
- Synchronization
- Thread safety in Qt
- **Qt and the Standard Library threading facilities**

General remarks

- It is perfectly possible to mix'n'match Qt and `std` threading classes.
- The Standard Library is moving extremely fast and Qt will not (and should not) catch up with all of its new developments:
  - parallel algorithms, continuations, latches, barriers, atomic smart pointers, executors, concurrent queues, distributed counters, coroutines, ...
- More and more tooling will start checking for correct usages of `std` APIs, but not Qt ones (unless they get reimplemented on top of the `std` ones).
- QThread is still more convenient when dealing with QObjects and event loops.
- A comparison of the APIs is in the next slides.

QThread vs. std::thread

	QThread	std::thread
No need to subclass it in order to use it	✓	✓
Function (job/task) runner	✗	✓
Detach support	✗ [1]	✓
Interruption request	✓	✗ [2]

[1] we can emulate that by connecting `QThread::finished()` to `QThread::deleteLater()`

[2] as shown before, it's trivial to emulate

QThread vs. std::thread (2)

	QThread	std::thread
Event loop support	✓	✗ [1]
QObjects can be created into	✓	✓
QObjects can be moved to	✓	✓ [2]
Signals can be emitted from	✓	✓
Slots work in direct connections	✓	✓
Slots work in queued connections	✓	✓

[1] But we can use `QEventLoop` to run a thread-local event loop

[2] We can use `QThread::currentThread()` to get a `QThread *` (to move a `QObject` to, etc.)

Synchronization primitives

Qt	Standard Library
QMutex	std::mutex std::timed_mutex std::recursive_mutex std::recursive_timed_mutex
QSemaphore	✗
QReadWriteLock	std::shared_mutex std::shared_timed_mutex
QWaitCondition	std::condition_variable
✗	std::call_once
Q_GLOBAL_STATIC	✗

Synchronization primitives: remarks

- `QMutex` and `QReadWriteLock` are faster than the `std` equivalents
- A non-recursive `QMutex` never allocates nor throws exceptions on Linux
- `QMutex` in 5.8 models the TimedLockable concept
  - Can be used together with `std` lock managers
- No `std` compatibility functions in `QReadWriteLock` (yet)
- `std::condition_variable(_any)` more generic / convenient than `QWaitCondition`
  - Supports any BasicLockable
  - Pass predicate to test in `wait()` call, instead of using the mandatory while loop
- `Q_GLOBAL_STATIC` is superseded by C++11's semantics for thread-safe function statics (and/or `std::call_once`)

Lock management

Qt	Standard Library
`QMutexLocker`	`std::lock_guard`
`QReadLocker`	`std::shared_lock`
`QWriteLocker`	`std::lock_guard`
⊗	`std::unique_lock`
⊗	`std::lock()`

Lock management: remarks

Standard Library lock management is much more powerful and flexible

- Movable lock guards (`std::unique_lock`) to return a managed lock
- Lock managers also have timed `try_lock()`s
- Tag classes to decide what a lock manager should do with the lock
- In C++17 `std::lock_guard` manages multiple locks (in a deadlock-free fashion)
  - `QOrderedMutexLocker` is C++17's `std::lock_guard` for two `QMutexes`
    - private API

Unless you're dealing with `QReadWriteLock`, prefer the `std` alternatives

Atomics

Qt	Standard Library

`QBasicAtomicInteger<T>`
`QAtomicInteger<T>`
`QAtomicInt`
`QBasicAtomicPointer<T>`
`QAtomicPointer<T>`
	`std::atomic<T>`

	`std::atomic_`*`operation`*`()`

Atomics: remarks

- Starting with Qt 5.7, Qt atomics actually uses C++11 atomics under the hood
  - Except on MSVC, since it doesn't (properly) implement them yet
- The `std` atomics support extra (advanced) features compared to the Qt ones
  - Consume, acq+rel memory ordering
  - Different memory orderings available for success/failure in read-modify-write operations
- The non-member atomic operations allow for generic code and specializations

```
std::atomic_store(std::shared_ptr<T> *p, std::shared_ptr<T> q)
```

- If you do use atomics, start thinking to move towards the Standard Library

Thread-local storage

Qt	Standard Library
`QThreadStorage`	`thread_local`

- Same functionality, different syntaxes
- Both lazy initialized
- `QThreadStorage` allows checking / skipping initialization

KDAB

Questions?

Thank you!

www.kdab.com

giuseppe.dangelo@kdab.com