QObject Life-Cycle

things learned when implementing
a general purpose QtContacts engine

Mathias Hasselmann - Openismus

Introduction

- Working on qtcontacts-tracker since Feb. 2010
- First serious Qt project for me,
  but strong GNOME background.
- The life cycle issues hits us badly...
- Surprised to see such issues in Qt...
- Seems everyone knows the problems...

...so maybe time to finally address them with Qt5?

Smart Pointers vs. Threads

```
if (ptr.isNull()) {
    ptr->doSomeThing();
}
```

Smart Pointers vs. Threads

```
{
    QMutexLocker locker(&mutex);

    if (not ptr.isNull()) {
        ptr->doSomeThing();
    }
}
```

Smart Pointers vs. Threads

```
QMutexLocker locker(&mutex);
{
    if (not ptr.isNull()) {
        ptr->doSomeThing();
    }
}
```

...how to force API users to lock that mutex before causing object destruction?

QSharedPointer failing

- QSharedPointer<Foo>(...)->start();
- Foo::start() { engine()->start(this); }
- Engine::start(Foo *req)

```
{
    register(QSharedPointer<Foo>(req)); // autsch!
}
```

Things that ~QObject() breaks

```
QObject::~QObject()
{
    ...
    QObjectPrivate::clearGuards(this);
    ...
    if (d->sharedRefcount) { ... }
    ...
```

Things that ~QObject() breaks

...

emit destroyed(this);

…

QAbstractDeclarativeData::destroyed()

…

// disconnect all receivers

…

Things that ~QObject() breaks

```
    …
    // unregister pending timers
    …
    d->deleteChildren();
    …
}
```

Things that ~QObject() breaks

- **Summary:** Every single life-cycle mechanism*) only kicks in, when the affected object is reduced to a plain QObject and any subclass aspect has been removed already!!!
- **Implication:** Relying on Qt4 object management causes memory corruption, crashes.

*) QPointer, QSharedPointer, QWeakPointer, QML, signals, parent ownership, ...

Our Hackish Workarounds

```
QContactFetchRequest::~QContactFetchRequest()
{
    …
    engine->destroyNotify(this);
    …
}
```

Our Hackish Workarounds

```
QctEngine::requestDestroyed(... *request)
{
    QMutexLocker l(d->m_requestLifeGuard);
    ...
}
```

Our Hackish Workarounds

```
QctRequestLocker QctEngine::request(... *worker)
{
    return QctRequestLocker
        (m_requests.value(worker),
         QMutexLocker(&d->m_requestLifeGuard));
}
```

Our Hackish Workarounds

```
void QctSomeWorker::run()
{
    …
    engine()->updateStatus
        (ugly_casts<>(engine()->request(this)), …)
    …
}
```

An IMHO Nicer Solution

```
void QObject::unref()
{
    if (not d->refCount.unref()) {
        emit disposing();
        dispose(); // maybe also use signal to cleanup
    }                  // from proper thread...
}
```

Session Conclusions

- ideally QObject should have two-phase construction and destruction, but it is cumbersome to implement in C++:
  - wrapper objects break polymorphism
  - explicit construction, reference and unref methods radially break Qt API and C++ paradigms (stack allocation, delete keyword, …)
  - language support would be needed, but unrealistic in Qt5 time timescale

Session Conclusions

- Thiago will finish explicit QObject support in QWeakPointer::toStrongRef() - ABI constraints stopped that effort in Qt4
- provide mixin in spirit of QSharedData to properly support toStrongRef() for abritary objects
- maybe provide something like QctRequestLocker to keep objects fully alive in threaded context