Qt in Depth

Bradley T Hughes
bhughes@trolltech.com

Introduction

- Bradley T Hughes
  - Senior Software Engineer
  - Leader of the Qt Platform Team
  - Open Source Programmer

- Trolltech ASA
  - Creators of Qt

Agenda

- ~~Introduction~~
- P-IMPL
- Implicit sharing
- Internal Atomic API
- QObject
- Signals and Slots
- Compiler Support

P-IMPL

- Private IMPLementation
- Used through-out Qt
  - A few exceptions
    - QColor, QModelIndex, probably a few others...
- We guarantee binary compatibility
  - Cannot add, remove, reorder members in public classes
  - Have to have a way to extend...

P-IMPL

- One pointer member in public API
  - Private access, of course
- The “d-pointer”
  - Trolltech's name for P-IMPL
  - Private classes
  - All data, private functions/slots
  - Platform dependent implementations

P-IMPL

- QObject sub-classes have Private counter-part
  - class QObject -> class QObjectPrivate
  - class QWidget -> class QWidgetPrivate
  - class QTcpSocket -> class QTcpSocketPrivate
- We will talk about P-IMPL in QObject later...

P-IMPL

- Tool classes are different
  - Many functions are inline
  - Data structure must be in public API
    - Again, with private access
  - Again, a "d-pointer" to data
- P-IMPL makes it easy to do *implicit-sharing...*

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- Implicit sharing
- Internal Atomic API
- QObject
- Signals and Slots
- Compiler Support

Implicit Sharing

- Trolltech's name for Copy-On-Write
- Used in almost all public value classes
  - Always exceptions...
    - QColor, QModelIndex, etc...
- Data contains reference count
  - Deleted when reference becomes zero
  - Copied when modified

Implicit Sharing

- Optmization: shared_null
  - Static instance of data
  - Reference count starts at one
  - Always positive, never deleted
- Rationale:
  - Data by default constructors, clear() functions
  - No need to allocate data for “empty” objects
  - No need to check if d-pointer is null

Implicit Sharing Example

- ```
  QByteArray::Data QByteArray::shared_null =
  { Q_ATOMIC_INIT(1), 0, 0, shared_null.array, {0} };
  ```

- ```
  QListData::Data QListData::shared_null =
  { Q_ATOMIC_INIT(1), 0, 0, 0, true, { 0 } };
  ```

- ```
  QString::Data QString::shared_null =
  { Q_ATOMIC_INIT(1), 0, 0, shared_null.array,
    0, 0, 0, 0, 0, {0} };
  ```

- ```
  QVectorData QVectorData::shared_null =
  { Q_ATOMIC_INIT(1), 0, 0, true };
  ```

Implicit Sharing Examples

```
inline QString::QString() : d(&shared_null)
{ d->ref.ref(); }

inline QString::~QString()
{ if (!d->ref.deref()) free(d); }
```

```
inline QMap() : d(&QMapData::shared_null)
{ d->ref.ref(); }

inline ~QMap()
{ if (!d->ref.deref()) freeData(d); }
```

Implicit Sharing

- What is Q_ATOMIC_INIT() and ref.ref()?
  - Copy-On-Write is inherently not thread-safe
  - Some protection is needed
  - Qt uses its *internal atomic API* to do reference counting...

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- ~~Implicit sharing~~
- Internal Atomic API
- QObject
- Signals and slots
- Compiler Support

Internal Atomic API

- Two classes
  - QBasicAtomic
  - QAtomic
- Why two?
  - One is POD
  - Other is convenient
- QAtomic inherits from QBasicAtomic
  - Just adds a constructor

Internal Atomic API

- bool QAtomic::ref()
- bool QAtomic::deref()
  - Increment/decrement atomically
  - Returns true if new value is non-zero
  - Returns false otherwise (new value is zero)
- Used to do reference counting

Internal Atomic API

- What else can QAtomic do?
  - Test-and-set
    - "Normal", Acquire, Release
  - Exchange/Swap
  - Basic comparison to regular integers
    - equality, inequality

Internal Atomic API

- QAtomicPointer
  - Template class
  - Typed pointers
  - Test-and-set
    - Only normal
    - No acquire, release
  - Exchange

Internal Atomic API

- I want it! Can I use it?
  - Indirectly, yes
- QSharedData and QSharedDataPointer
  - Public classes
  - Use QAtomic, QAtomicPointer

Internal Atomic API

- QMutex
  - Atomic API is not only for reference counting
- Rationale:
  - Lock overhead is high
  - Involves system call
  - Unwanted when lock is not contended

Internal Atomic API

- QMutex – How do we do it?
  - Check lock first with testAndSet()
  - Make system call only if lock is contended
- What about fairness?
  - Can a thread steal the mutex from a waiting thread?
  - Not in our implementation

Internal Atomic API

```
void QMutex::lock()
{
    ulong self = d->self();

    int sentinel;
    forever {
        sentinel = d->lock;
        if (d->lock.testAndSetAcquire(sentinel,
                                      sentinel + 1))
            break;
    }
    ...
```

Internal Atomic API

- Each contender increases d->lock
- Use previous value of d->lock
  - Indicates number of contenders ahead of current thread

Internal Atomic API

```
    if (sentinel != 0) {
        if (!d->recursive || d->owner != self) {
            if (d->owner == self) {
                qWarning("QMutex::lock: Deadlock "
                         "detected in thread %ld",
                         d->owner);
            }
            // didn't get the lock, wait for it
            d->wait();
        }
        // don't need to wait for the lock anymore
        d->lock.deref();
    }
    ...
```

Internal Atomic API

- If thread could not get lock
  - First, check for recursive lock, deadlock
  - Go to sleep
  - When woken up, lock has been passed to thread
  - Decrease d->lock (current thread is no longer a contender)

Internal Atomic API

```
▶       d->owner = self;
        ++d->count;
        Q_ASSERT_X(d->count != 0, "QMutex::lock",
                   "Overflow in recursion counter");
    }
```

Internal Atomic API

- Thread now has lock
  - Set owner, lock count

Internal Atomic API

```
void QMutex::unlock()
{
    Q_ASSERT_X(d->owner == d->self(),
               "QMutex::unlock()",
               "A mutex must be unlocked in the "
               "same thread that locked it.");

    if (!--d->count) {
        d->owner = 0;
        if (!d->lock.testAndSetRelease(1, 0))
            d->wakeUp();
    }
}
```

Internal Atomic API

- Decrease lock count
  - If zero, release the lock
  - Can only release the lock if no other contenders
    - `d->lock.testAndSetRelease(1, 0)`
  - If contenders, lock is passed to first waiting thread
    - This is fair since lock is FIFO

Internal Atomic API

- Yes, that is really QMutex
  - Only platform code not shown

Internal Atomic API

- No other uses currently
  - Some possibilities in the future
  - QReadWriteLock
  - Internal lock-free data structures

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- ~~Implicit sharing~~
- ~~Internal Atomic API~~
- QObject
- Signals and slots
- Compiler Support

QObject

- The interesting parts of QObject
  - QObjectPrivate
  - Thread affinity
  - Signals and slots
    - Not here, next section

QObject

- QObjectPrivate
  - Inherits from QObjectData
- QObjectData?!
  - Remember P-IMPL?
  - Inline implementation of trivial functions
    - isWidgetType(), signalsBlocked(), children(), parent()
  - QObject::d_ptr

QObject

- Why QObject:d_ptr?
  - Why not QObject::d?
  - QObjectPrivate inherits QObjectData
  - Need to cast to QObjectPrivate
- QObject::d_func()
  - Returns QObjectPrivate pointer
  - `static_cast<QObjectPrivate *>(d_ptr);`

QObject

- So you type d_func() all the time?
  - No, we have a Q_D(Class) macro
  - ClassPrivate *d = Class::d_func();

QObject

```
void QObject::setObjectName
                    (const QString &name)
{
    Q_D(QObject);
    d->objectName = name;
}
```

QObject

- Why all the trouble?
  - QObject subclasses also have QObjectPrivate subclasses
  - Only one instance of the Private object
    - Instead of one per subclass

QObject

- Most derived class creates Private instance
  - Passes it to protected base class constructor
  - `QObject(QObjectPrivate &dd, QObject *parent);`
  - All QObject subclasses in Qt have this constructor.
- Q_DECLARE_PRIVATE()
  - Macro to declare Class::d_func()
  - Does appropriate static_cast

QObject

- The QObjectPrivate mechanism is internal API
  - P-IMPL, remember?

QObject

- Thread affinity
  - Each QObject “belongs” to a thread
  - Thread delivers events to object
  - Used by signal-slot mechanism
- QThreadData
  - The real identity of a thread
  - Each QObjectPrivate holds reference to one
  - This includes QThread

QObject

- QThreadData
  - Posted event list, thread local storage, event dispatcher
  - Pointer to QThread it represents

QObject

```
QThread *QObject::thread() const
{
    Q_D(QObject);
    return d->threadData->thread;
}
```

QObject

- Thread affinity can be changed
  - Change the QThreadData reference
  - Move posted events
  - Post an event to restart timers, socket-notifiers

QObject

- Adding thread affinity to QObject gave us the possibility to add thread support to *the signal and slot* mechanism...

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- ~~Implicit sharing~~
- ~~Internal Atomic API~~
- ~~QObject~~
- Signals and slots
- Compiler Support

Signals and Slots

- Connections represented by QConnection struct
  - Internal, found in `qobject.cpp`
  - Sender, signal number
  - Receiver, member number
    - Can be signal or slot
  - Qt::ConnectionType
  - Argument marshalling information
- Signals, slots represented by integers
  - Fast comparisons during emission

Signals and Slots

- Connections stored in QConnectionList
  - Global list
  - Indexing on sender, receiver using QMultiHash
  - Connection removed if sender, receiver deleted

Signals and Slots

- Signal emitting done by QMetaObject::activate()
  - Called by moc generated code
  - Arguments are sender, signal number, slot arguments

Signals and Slots

- Example: QAbstractButton::clicked()
  - Has one argument, `bool checked = false`
  - Overloaded by moc because of default value
  - Really two signals instead of one

Signals and Slots

```
▶ void QAbstractButton::clicked(bool _t1)
  {

▶     void *_a[] = {
          // return value
          0,
          // argument
          const_cast<void*>
          (reinterpret_cast<const void*>(&_t1)) };

▶      QMetaObject::activate(this, &staticMetaObject,
                             // 2 = clicked()
                             // 3 = clicked(bool)
                             2, 3,
                             _a);
  }
```

Signals and Slots

- QMetaObject::activate() does its job
  - Looks in sender index
  - Goes through all connections
  - Activates those that match signal number(s)

Signals and Slots

- Activating a connection
  - Looks at ConnectionType
  - if Auto
    - currentThread == sender->thread == receiver->thread?
    - if so, use Direct, otherwise Queued
  - if Direct
    - call immediately
  - if Queued
    - post event to receiver

Signals and Slots

- Activations done through qt_metacall()
  - Virtual function
  - Defined by Q_OBJECT macro
  - Moc generated code calls slot implementation
- Example: QLineEdit::setText() slot

Signals and Slots

```
int QLineEdit::qt_metacall(QMetaObject::Call _c,
                           int _id, void **_a)
{
    ...
    if (_c == QMetaObject::InvokeMetaMethod) {
        switch (_id) {
        ...
        case 7:
        setText((*reinterpret_cast
                 <const QString(*)>(_a[1])));
        break;
        ...
```

Signals and Slots

- An interesting side-effect
  - Slots are virtual
  - Even if not declared virtual
  - Backdoor for keeping binary compatibility
  - Add a “virtual” function
    - Declare new, non-virtual slot in base class
    - Override it in subclasses
- `QStyle::standardIconImplementation()`

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- ~~Implicit sharing~~
- ~~Internal Atomic API~~
- ~~QObject~~
- ~~Signals and slots~~
- Compiler Support

Compiler Support

- GCC
  - Lots of nice extensions
    - typeof() - makes foreach() simple
  - Very complete implementation
- Intel C++ Compiler for Linux
  - Supports many GCC extensions
  - Binary compatible with GCC

Compiler Support

- MSVC.NET 2003, MVSC++ 2005
  - Very complete implementation as well
  - No support for GCC extensions
  - Have to do foreach() in “proper” C++

Compiler Support

- So, C++ compilers are pretty good
  - There are always exceptions
- The Party Crashers
  - MSVC 6.0
  - Borland
  - Commercial UNIX compilers

Compiler Support

- **MSVC 6.0**
  - for() **scoping is wrong:**

```
for (int i = 0; i < count; ++i)
    break;
// i is still accessible
done = i;
```

  - **Arguments in template functions must include template arguments:**

```
template <typename T>
void function(T arg);
```

Compiler Support

- **MSVC 6.0**
  - No partial template specialization
  - **WARNING:**
    - Code is accepted
    - No warnings, no errors
    - *It never picks the specialization*
    - Instantiates the original template declaration instead.

Compiler Support

- Borland
  - Not supported by Qt 4
  - Often problems with templated code
    - Full template specialization buggy
    - Normal functions overloaded with template functions
      - Never picks template function
  - We never could get it to work...

Compiler Support

- Commercial UNIX compilers
  - Usually not a problem, but they do occur
  - Optimizer bugs are the worst
  - Code works debugging
  - Final “release” build breaks horribly
  - Worst yet – the compiler itself breaks
    - Not going to name names...

Agenda

- ~~Introduction~~
- ~~P-IMPL~~
- ~~Implicit sharing~~
- ~~Internal Atomic API~~
- ~~QObject~~
- ~~Signals and slots~~
- ~~Compiler Support~~

Qt in Depth

Bradley T Hughes
bhughes@trolltech.com