Modern practices for Qt Development

Luca Ottaviano, Develer Srl, <lottaviano@develer.com>

Short bio

Software developer at Develer

Using Qt since 3.3

Mentor and trainer for the past few years

Agenda

- Sharpen your tools
- Modernize your style
- Avoid pitfalls
- Old but gold

Qt day

develer

Sharpen your tools

Qt day

https://pixabay.com/it/cat-mieze-gattino-zero-2320318/

Qt Creator and the code model

Code model: what the IDE understands of your code

The code model drives autocompletion and refactoring options (among other things)

C++ is so complex that a full compiler is needed

Enter ClangCodeModel: a model built on Clang

Much more useful options, just in time analysis

“Wow, my builds are warning free!”

Actual quote from a colleague

Qt Creator will signal all errors and warnings inline in your code

You can fix the code in the file you’re working on, no more distracting warnings from other parts of the code

Faster than launching a build

Clang tidy and Clazy integration

Clang tidy

It’s a “linter” (AKA static analysis tool) for C++

It can check and in some cases automatically fix:

- Interface misuse
- Performance issues
- Modern C++11 features (eg. override or nullptr)

Clazy

It’s a static analyzer specific for Qt code

Warnings include old style connect, operations that detach a container, QString related anti-patterns

It’s really helpful and you should enable it now!

Code beautifier

Too much energy has already been spent on code style, let’s do it automatically!

Enter clang-format: format your C++ code with a given style

Beautifier plugins will enable auto-format on save

Many styles are present by default, you can define your own

Modernize your style

https://commons.wikimedia.org/wiki/File:Dog_dressed_in_an_airplane_outfit.jpg

Qt day

Choose your container wisely

In standard C++ the default container is std::vector<>

Other containers may be used for particular use cases

In C++/Qt applications there are many containers to choose from: std::vector<>, QVector, QList and many more

The wrong container may lead to slow performance from many little inefficiencies

QList considered harmful

It has different allocation strategies depending on many factors

It impacts binary compatibility and other guarantees (eg. references to elements)

Factors include the platform (32-bit, 64-bit), the size of the contained type, Q_DECLARE_TYPEINFO for the contained type

For “good” types it works as QVector<>, for “bad” types it works as QVector<void*>

“Bad” types include QVariant, QModelIndex, QImage, QPixmap, QString and QByteArray (from Qt 6) and by default every new type

Containers take away

QList is presented as the default type, but that is questionable. A more reasonable default:

Use std::vector<> or QVector for business logic and local variables

Prefer QVector<> for new Qt-ish APIs, std::vector<> for C++-ish APIs.

Use QList if you need to interact with APIs that require QList or for code you don't want to touch (converting QList to QVector isn't free)

Always Q_DECLARE_TYPEINFO for your own (non-QObject) types! (Helps QVector too)

Let Q_FOREACH go

Q_FOREACH() is a macro to iterate over a Qt container

Also called foreach() if you have enabled Qt keywords

Always takes a copy of the container

Ok for Qt containers (implicit sharing), not ok for std containers

The devil is in the details

```
Q_FOREACH(const QString &lang, languages)
    languages += getSynonymsFor(lang);
```

Anything wrong with this code?

It’s changing the container it’s iterating over

The rest of the code may assume the container now contains extra elements!

This is completely non standard C++ and will surprise many

Enter range-for

foreach() and Q_FOREACH() have been deprecated with Qt 5.7, replace with range-for

Range-for may force a detach in Qt containers, porting tips:

- If you have a std container, just use for() instead of foreach()
- If the loop modifies the container, take a copy
- If you have const containers, they don’t detach => use for()
- If you have non-const rvalues, use a temporary value
- If you have non-const lvalues, use qAsConst() (since Qt 5.7) or std::as_const() (since C++17)

Gotchas

`for (auto v : values)`, v is a copy. It's inefficient and changes to v are not into `values`

Better use for `(const auto &v : values)` for read-only loops

This is standard C++ anyway

Some classes don't have STL iterators (eg. QDomNodeList)

Looking for something to contribute? :)

New style connect

Introduced with Qt5, it allows you to connect pointer to member functions

The connection is checked at compile time → no more runtime warnings swamped in the general application output!

You can connect to any method, not just the ones declared as “slots”

```
QLabel *label = new QLabel;
QLineEdit *lineEdit = new QLineEdit;
QObject::connect(lineEdit, &QLineEdit::textChanged, label, &QLabel::setText);
```

New connect with 3 parameters

Pros: connect to a free function or lambda allowed

Cons: no automatic disconnect if the lambda captures a QObject

Use the 4 parameter overload, the one with a context object

```
connect(
    sender, &Sender::valueChanged,
    [=]( const QString &newValue ) {
        receiver->updateValue( "senderValue", newValue );
    }
);
```

Avoid pitfalls

Ph. natasia.causse on Flickr

Qt day

Don’t derive from QThread

```
class WorkerThread : public QThread {
   QVector<int> vec;
public:
   WorkerObject() {}
   void setInt() { vec.append(666); }
protected:
   virtual void run() override {
       while (true) {
           vec.append(42);
       }
   }
};
```

Don’t derive from QThread (Quiz time!)

In which thread the WorkerThread object is created?

In the thread that calls “new WorkerThread”.

In which thread WorkerThread::run() is executed?

In a new thread, not the one in which the WorkerThread is created

What happens if you access private members in WorkerThread::run()?

You have a race condition if you don’t use synchronization primitives

Concurrent code paths

Creation thread

```
class WorkerThread : public QThread {
   QVector<int> vec;
public:
   WorkerObject() {}
   void setInt() { vec.append(666); }
protected:
```

```
   virtual void run() override {
       while (true) {
           vec.append(42);
       }
   }
```

New thread

Correct usage

Create worker objects, bare QThreads and use moveToThread() on your worker object

Connect() after you have moved your worker object to the thread or use Qt::QueuedConnection

Beware! Create any QTimers in the worker thread

Qt Quick animations pitfalls

Running an animation will change the property of the object to the target value

You lost the source value unless you saved it

- Keep it in mind when creating “free” animations

Animators will update the property only at the end of the animation

- If two objects are linked with a binding, one will not animate

Old but gold

Photo:

https://en.wikipedia.org/wiki/National_Numismatic_Collection

Qt day

QPointer is your friend

Present since Qt 4.0

Automatically tracks any QObject lifetime (template class)

Becomes nullptr when the object is destroyed

Use when storing pointer to an object whose lifetime is beyond your control (eg. QNetworkReply, network services from libconnman-qt…)

Questions?