C++, QML, and static reflection

Manu Sanchez @Manu343726

$whoami

- Reflection obssesed nerd
- Self-taught in C++ in general and metaprogramming in particular
- Spanish ISO C++ national body
- Since 2015 working with access control systems and video-IP

Overview

- Why Qt?
- At the beginning, there was...
- Why QML?
- Porting models to QML
- QML integration tricks

QML?

- A declarative language for reactive UIs
- mobile and touch friendly
- built-in theme system
- No CSS, HTML, etc needed
- Part of the Qt framework

Why using Qt?

Why using Qt?: The framework

- cross platform and ABI-stable libraries
- Lots of built-in APIs: Networking, Bluetooth, NFC, containers, UI, timers, etc
- In almost 90% cases you don't have to reinvent the wheel

Why using Qt?: The runtime

- Cross-platform event loop
- Cross-thread message passing
- Object model with high level message passing: **signals and slots**
- Thread safe communication between your app modules
- Reactive design

Why using Qt?: The UI systems

- **QtWidgets**: widget based UIs for desktop
- **QtQuick**: QML, declarative UIs for mobile/automotive
- **QtWebEngine/QtWebkit**: Built-in web browsers/engines

Some context

Some context

- Access monitoring console
- Tablet-like custom touch based device
- 720p (multi)touchscreen
- ARM v7 based arch, 512MB/1GB RAM
- Yocto-like custom linux
- QtWidgets full C++ stack UI prototype

Three years ago, in a sprint far far away...

Three years ago, in a sprint far far away...

> What if... **we write the UI using a web stack!**

... web stack UI?

- Easier to get designers to work
- Frontend isolated from the rest of the firmware
- Posibility of moving the front to an external device (PC)
- Easy with Qt: webserver + Qt web browser instance

Alright, web stack here we go!

- Yes, I actually have Angular 1 experience!
- No, my work is not just writing templates and making gcc cry
- No, really, 50% of my job was developing and maintaining a web front
- Don't worry, lots of C++ on the other side

Web stack

- DB: sql
- Back: C++ business models and ops
- Web API: Google Protobuf
- Front: Angular JS (No Man's Land)

Web stack: Data model

- DB model
- C++ backend model
- Network layer/backend API model (protobuf)
- Frontend model: Javascript unicorns and AngularJS services

Two years later...

> Mmmm hey guys, we need to display four different video streams

Damn, we're f...

No, really

- No hardware graphics enabled
- The display hardware stack could not handle four streams
- stream composition in software was not an option...

What do we do now?

- Put linkedin as my landing page?
- Moving to Alaska, living in the woods?
- Give up with software engineering, consider cooking, bakery, whatever

What do we do now?

QML!!!

Why QML?

- Direct integration with Qt C++ backends
- Fully declarative, reactive UI
- Touch and mobile friendly widgets: swipe views, sliders, input boxes
- A **great** built-in multilanguage multilayout touch keyboard. Thanks Qt!

Why QML?

- Hardware acceleration out of the box!
  - *So now its just a driver problem for the linux freak of the team...*
- Simple OpenGL render target integration: We can render anything in the UI!

Anatomy of a QML file

Anatomy of a QML file

```
Window {
  id: "window"
  with: 800
  height: 600

  Button {
    id: "button"
    onClicked: {
       console.log("button clicked")
    }
  }
}
```

Anatomy of a QML file

- Hierarchical controls
- Unique IDs for each control
- Control properties
- **signal handlers**

QML property bindings

```
Window {
  id: "callWindow"
  ...

  visible: callModel.calling || callModel.talking ||
           callModel.incomingCall
}
```

QML property bindings

- The colon syntax means binding, **not assignment**
- The left side property reacts to changes on the right side expression evaluation
- The QML engine takes care of everything, monitoring changes in all operands
- Assignment is supported, **but breaks existing bindings**

QML property bindings

- No bidirectional binding supported directly

```
TextField {
  id: "ipAddressField"

  // view <-- model
  text: networkModel.ip

  // model <-- view
  Binding {
    target: networkModel
    property: "ip"
    value: ipAddressField.text
  }
}
```

QML property bindings

- Beware of binding loops
- Complext MVVM workflows could be tricky to implement

What about the models?

The models

- Pure QML app
- C++ backend

C++ models for QML

- Using the Qt object model
- `QObject` includes runtime introspection, properties, signals, and slots

Anatomy of a Qt class

```
class NetworkModel : public QObject {
  Q_OBJECT

public:
  Q_PROPERTY(ip READ getIp WRITE setIp NOTIFY ipChanged)

  const QString& getIp() const;
  void setIp(const QString& newIp);

signals:
  void ipChanged(const QString& newIp);

slots:
  void onIpChange();

private:
  QString _ip;
};
```

Anatomy of a Qt class

- Classes inherit from QObject
- `Q_OBJECT` : It's a tag for the MOC
- **signals** (outgoing messages): Represented by method declarations
- **slots** (destination of a message): Represented by class methods

Emitting a signal

```
void NetworkModel::setIp(const QString& newIp)
{
  if(_ip != newIp) {
    _ip = newIp;
    emit ipChanged(newIp);
  }
}
```

Connecting two objects

```
class AppEvents : public QObject {
  Q_OBJECT

public slots:
  void ipAddressChanged(const QString& ip)
  {
      logger.info("ip address changed to {}", ip);
  }
};
```

```
QObject::connect(
  // source object                  signal
  &networkModelObject, &NetworkModel::onIpChanged,
  // dest object                 dest slot
  &appEventsObject, &AppEvents::ipAddressChanged);
```

Wait...

Wait...

- I have three years of non-Qt models
- My models were modeling the SQL database tables
- I don't like the verbosity of setter + getter + signal + PROPERTY
- What can we do now?

```
struct NetworkSettings : public TableModel
{
    DbId id;
    std::string ip;
    std::string mask;
    std::string gw;

    bool isCurrentSystemConfig() const;
};
```

Reflection!

Reflection!

- What if we can automatically translate our models to something QML understands?
- I was using reflection already, to generate the SQL queries for the models

tinyrefl

- My not-so-tiny static reflection codegen tool and API
- Cross building supported
- API agnostic metadata
- High level C++14 static reflection API

tinyrefl

- No external dependencies, all deps are gathered by cmake at config time
- Not at conan.io yet, but I'm working on it

tinyrefl: The tool

- Uses cppast by Jonathan @foonathan Muller: A C++ wrapper of libclang for dummies (like me)
- Supports introspection of user defined attributes (Clang API and libclang don't)
- It's maintained: Quick feedback, bugfixing, etc

tinyrefl: The tool

- The cli tries to follow the interface of clang and gcc

```
$ tinyrefl-tool -c include/foo/foo.hpp -std=c++14 \
  -Iinclude/ -DFOOLIB=1 -fPIC                     \
  -o include/foo/foo.hpp.tinyrefl
```

tinyrefl: The tool

- Also provides some scripts for simple CMake integration

```
add_subdirectory(thirdparty/tinyrefl)

add_library(foo foo.cpp)
target_include_directories(foo PUBLIC include/)

# This will run the tool on your headers before
# building your library
tinyrefl_tool(foo HEADERS include/foo/foo.hpp)
```

tinyrefl: The tool

- Generates API-agnostic metadata as macros
- Define the macros to get whatever you need

tinyrefl: The API

- C++14 constexpr API
- **metatype**: Type representing metadata of a given type
- **metaobject**: constexpr instance of a metatype, usually gives a higher level API

tinyrefl: The API

```
// foo.hpp
enum class Enum { A, B, C };

// main.cpp
#include <tinyrefl/api.hpp>
#include <foo/foo.hpp>
#include <foo/foo.hpp.tinyrefl>

using EnumMetatype = tinyrefl::metadata<Enum>;
static_assert(EnumMetaType::name == "Enum", "");

// But I prefer the metaobject syntax
constexpr auto enumMetaObject =
    tinyrefl::metadata<Enum>{};

int main() {
  for(const auto& value : enumMetaObject.get_values()) {
    std::cout << value.name() << ": "
              << value.underlying_value() << "\n";
  }
}
```

tinyrefl: The API

```
class MyClass : public std::vector<int>,
                public std::string {
public:
  int i = 42;
  std::string str = "hello";
};

MyClass myObject;
```

tinyrefl: The API

- Visitation API
  - Tag dispatch based
  - Visitors declare what entities they are interested in
  - Everything is resolved statically, no type-erasure behind the scenes

tinyrefl: The API

```
tinyrefl::visit_class<MyClass>([](
  const auto name,
  auto depth,
  auto metatype,
  TINYREFL_STATIC_VALUE(tinyrefl::entity::BASE_CLASS))
{
  std::cout << "base class \"" << name << "\"\n";
});
```

tinyrefl: The API

```
tinyrefl::visit_object(myObject, [](
  const auto name,
  auto depth,
  const auto& member,
  TINYREFL_STATIC_VALUE(
    tinyrefl::entity::MEMBER_VARIABLE))
{
  std::cout << "." << name << ": "
            << tinyrefl::utils::type_name<
                  decltype(member)>()
            << "[" << member << "]";
});
```

tinyrefl: The API

```
base class std::vector<int>
base class std::string
.i int [42]
.str std::string [hello]
```

tinyrefl: The API

- High level visit API:
  - `visit_object_member_variables()`
  - `visit_objects()`
- High level utilities
  - `equal()`
  - `to_string()`
  - `to_tuple()/from_tuple()`
  - `enum_cast()`
  - `to_json()/from_json()`
  - Built-in JSON For Modern C++ support

Going back to the QML world...

- We have type safe introspection of member variables
- But Qt uses its Qt types...

Qt <--> std translation layer

- Basically, translate strings, integers, containers, to a common model
- An string in the std side is a QString in the Qt/QML side
- `QtType<T>`: Trait specialized to do the conversions
- `Optional<T>` <--> `Optional<QtType<T>>`
- `fromQt()/toQt()` utility conversion functions

Qt <--> std translation layer

- Since Qt introspection and property system is dynamic, uses QVariant for the values
- QVariant: Is not an std::variant, **but an std::any**
- `toQVariant()/fromQVariant()` on top of `toQt()/fromQt()`

Qt <--> std translation layer

```
struct Foo
{
   std::string a, b;
};
```

gets translated to

```
struct QFoo
{
   QString a, b;
};
```

Exposing the translated values to QML

- Property system is done at runtime, but the registration API is hidden
- MOC generates code to register all the signals, slots, properties, getters, etc
- QObject provides a dynamic property system: `setProperty()` and `getProperty()`

Exposing the translated values to QML

- **QML does not support binding to the QObject dynamic property system**

What do we do now?

- Put linkedin as my landing page?
- Moving to Alaska, living in the woods?
- Give up with software engineering, consider cooking, bakery, whatever

thanks_for_comming(); }

There's a solution!

QQmlPropertyMap

QQmlPropertyMap to the rescue

- It has a QString key -> QVariant value map interface
- Implements all the magic behind the scenes
- Whenever a new property is assigned, it tells the Qt runtime to add new changed signals, properties, etc

QQmlPropertyMap to the rescue

- The binding is bidirectional

```
class MyPropertyMap : public QQmlPropertyMap {
   QVariant updateValue(const QString& key,
                        QVariant& value) override {
     // "key" was changed to "value" on the QML side

     return value;
   }
};

MyPropertyMap map;

// A new property i is exposed
map.insert("i", toQVariant(42));
// The i changed signal is sent to QML
map.insert("i", toQVariant(43));
```

QmlPropertyMap

- Because I find the extra Q redundant...
- High level type safe wrapper of the QQmlPropertyMap

QmlPropertyMap

```
class QmlPropertyMap : public QQmlPropertyMap
{
  Q_OBJECT
public:
  template<typename T>
  T value(const std::string& key) const {
    return fromQVariant<T>(
      QQmlPropertyMap::value(
        QString::fromStdString(key)
        )
      );
  }

  template<typename T>
  void value(const std::string& key, const T& value) {
    QQmlPropertyMap::insert(
      QString::fromStdString(key),
      toQVariant(value)
    );
  }
};
```

Model proxies

- Take an existing model by ref, and return QmlPropertyMap proxies to it

Model proxies

```
template<typename Model>
class ModelProxy : public QmlPropertyMap {
public:
  ModelProxy(Model& model) :
    _model{&model},
    _cache{model}
  {
    reload();
  }

public slots:
  void save();
  void reload();

private:
  Model* _model;
  Model _cache;

  QVariant updateValue(
    const QString&  key,
    const QVariant& value) override final;
};
```

Model proxies

- One reference to the model, and a cached copy
- Whenever the QMl side changes a property, the cache is updated
- `save()` saves the contents of the cache in the true model
- `reload()` invalidates the cache and reloads it from the model
- Implements a common CRUD pattern for config dialog windows

Model proxies: reload()

```
void ModelProxy::reload() {
  _cache = *_model;

  tinyrefl::visit_object_member_variables(_cache,
  [&](const std::string& name, const auto& value) {
    QmlPropertyMap::value(name, value);
  });
}
```

Model proxies: save()

```
void ModelProxy::save() {
  *_model = _cache;
}
```

Model proxies: updateValue()

```
QVariant ModelProxy::updateValue(
  const QString& key,
  const QVariant& value)
{
  tinyrefl::visit_object_member_variables(_cache,
  [&](const std::string& name, auto& member) {
    if(name == key.toStdString()) {
      member = fromQVariant(value);
    }
}
```

Putting all together

```
NetworkSettings networkSettings;
networkSettings.id = 1;
networkSettings.ip = "192.168.1.1";

QQmlEngine qmlEngine;
// load qml sources, etc

ModelProxy<NetworkSettings> proxy{networkSettings};
qmlEngine.setContextVariable(
  "networkSettingsModel", &proxy);
```

Putting all together

```
Window {
 id: "networkSettingsWindow"

 TextField {
   id: "ipAddressField"
   text: networkSettingsModel.ip

   Binding {
     target: networkSettingsModel
     property: "ip"
     value: ipAddressfield.text
   }
 }

 Button {
   id: "saveSettingsButton"
   onClicked: {
     networkSettingsModel.save()
   }
 }
```

thanks_for_comming(); }