Qt is Spying on Your Types

Jędrzej Nowacki

Presentation plan

- Why Qt tries to find additional information
- What kind of informations Qt can store
  - QTypeInfo
  - Q_OBJECT
  - QMetaType

Presentation plan

- Why Qt tries to find additional information
- What kind of informations Qt can store
  - QTypeInfo
  - Q_OBJECT
  - QMetaType

The goal is to make things less magic

Why Qt needs type information

Why Qt needs type information

- Qt is a framework

Why Qt needs type information

- Qt is a framework
- Qt is a C++ framework

Why Qt needs type information

- Qt is a framework
- Qt is a C++ framework
- Qt needs to operate on custom types

Compile time vs. runtime

Compile time vs. runtime

- Compile time

Compile time vs. runtime

- Compile time
- Runtime

Compile time vs. runtime

- Compile time
- Runtime
- Compile time decisions enable, influence runtime decisions

Q_DECLARE_TYPEINFO

- Q_PRIMITIVE_TYPE
- Q_MOVABLE_TYPE

If you store a simple type in one of Qt's containers it is important

Q_DECLARE_TYPEINFO (movable)

```
struct Type {
    Type();
    Type(const Type &other);
    ~Type();
    Data *data;
};

Type::Type() : data(new Data) {}
Type::Type(const Type &other) : data(new Data(*other.data)) {}
Type::~Type() { delete data; }
```

Q_DECLARE_TYPEINFO (movable)

```
struct Type {
    Type();
    Type(const Type &other);
    ~Type();
    Data *data;
};

Q_DECLARE_TYPEINFO(Type, Q_MOVABLE_TYPE); ⚠

Type::Type() : data(new Data) {}
Type::Type(const Type &other) : data(new Data(*other.data)) {}
Type::~Type() { delete data; }
```

Q_OBJECT and friends

Q_OBJECT and friends

- Q_OBJECT

```
#define Q_OBJECT \
public: \
    Q_OBJECT_CHECK \
    static const QMetaObject staticMetaObject; \
    virtual const QMetaObject *metaObject() const; \
    virtual void *qt_metacast(const char *); \
    QT_TR_FUNCTIONS \
    virtual int qt_metacall(QMetaObject::Call, \
                            int, \
                            void **); \
private: \
    static void qt_static_metacall(QObject *, \
                                   QMetaObject::Call,\
                                   int, void **); \
    struct QPrivateSignal {};
```

Q_OBJECT and friends

- Q_OBJECT
- Friends

```
#define Q_CLASSINFO(name, value)
#define Q_PLUGIN_METADATA(x)
#define Q_INTERFACES(x)
#define Q_PROPERTY(text)
#define Q_PRIVATE_PROPERTY(d, text)
#define Q_REVISION(v)
#define Q_OVERRIDE(text)
#define Q_ENUMS(x)
#define Q_FLAGS(x)
#define Q_SCRIPTABLE
#define Q_INVOKABLE
#define Q_SIGNAL
#define Q_SLOT
```

Q_OBJECT and friends

- Q_OBJECT
- Friends
- MOC

Class example

moc output

MOC

MOC

- Signals and slots

MOC

- Signals and slots
- Scripting

MOC

- Signals and slots
- Scripting
- Introspection for debugging

How to use introspection

```
void debugDirection(QAbstractAnimation::Direction value) {
    const QMetaObject &metaObject =
         QAbstractAnimation::staticMetaObject;
    int index = metaObject.indexOfEnumerator("Direction");
    QMetaEnum enumeration = metaObject.enumerator(index);
    qDebug() << enumeration.valueToKey(value);
}
```

Q_DECLARE_METATYPE

Q_DECLARE_METATYPE

Register type in Qt and assign id to it

```
int id = qRegisterMetaType<Type>();
```

Q_DECLARE_METATYPE

Q_DECLARE_METATYPE

Wraps given type in own interface

Q_DECLARE_METATYPE

Wraps given type in own interface

```
int id = QMetaType::registerNormalizedType(normalizedTypeName,
                                QtMetaTypePrivate::QMetaTypeFunctionHelper<T>::Delete,
                                QtMetaTypePrivate::QMetaTypeFunctionHelper<T>::Create,
                                QtMetaTypePrivate::QMetaTypeFunctionHelper<T>::Destruct,
                                QtMetaTypePrivate::QMetaTypeFunctionHelper<T>::Construct,
                                int(sizeof(T)),
                                flags,
                                QtPrivate::MetaObjectForType<T>::value());

...
```

Q_DECLARE_METATYPE

Wraps given type in own interface

```
template <typename T, bool Accepted = true>
struct QMetaTypeFunctionHelper {
    static void Delete(void *t)
    {
        delete static_cast<T*>(t);
    }

    static void *Create(const void *t)
    {
        if (t)
            return new T(*static_cast<const T*>(t));
        return new T();
    }

     ...
```

Q_DECLARE_METATYPE

Stores values given by Q_DECLARE_TYPEINFO

```
template<typename T>
struct QMetaTypeTypeFlags
{
    enum { Flags = (!QTypeInfo<T>::isStatic ? QMetaType::MovableType : 0)
        | (QTypeInfo<T>::isComplex ? QMetaType::NeedsConstruction : 0)
        | (QTypeInfo<T>::isComplex ? QMetaType::NeedsDestruction : 0)
        | (IsPointerToTypeDerivedFromQObject<T>::Value ?
                                    QMetaType::PointerToQObject : 0)
        | (IsSharedPointerToTypeDerivedFromQObject<T>::Value ?
                                    QMetaType::SharedPointerToQObject : 0)
        | (IsWeakPointerToTypeDerivedFromQObject<T>::Value ?
                                    QMetaType::WeakPointerToQObject : 0)
        | (IsTrackingPointerToTypeDerivedFromQObject<T>::Value ?
                                    QMetaType::TrackingPointerToQObject : 0)
        | (Q_IS_ENUM(T) ? QMetaType::IsEnumeration : 0)
    };
};
```

Q_DECLARE_METATYPE

Q_DECLARE_METATYPE

- We can construct / destruct instances

Q_DECLARE_METATYPE

- We can construct / destruct instances
- We know size of a type

Q_DECLARE_METATYPE

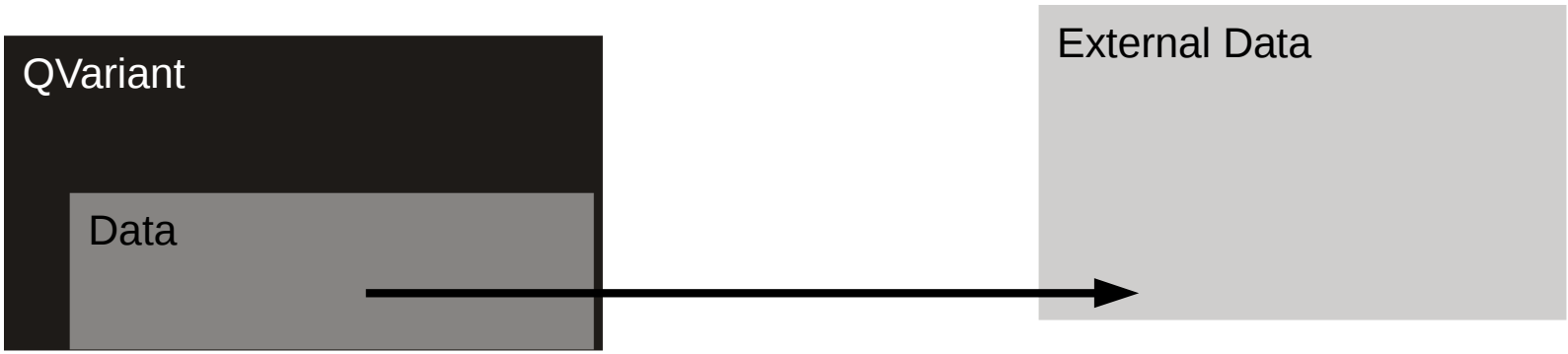
- We can construct / destruct instances
- We know size of a type
- We can determine if a type is movable

Q_DECLARE_METATYPE

- We can construct / destruct instances
- We know size of a type
- We can determine if a type is movable
- That is exactly what we need to construct QVariant

Q_DECLARE_METATYPE

- We can construct / destruct instances
- We know size of a type
- We can determine if a type is movable
- That is exactly what we need to construct QVariant



Q_DECLARE_METATYPE

- We can construct / destruct instances
- We know size of a type
- We can determine if a type is movable
- That is exactly what we need to construct QVariant

It can do more!

It can do more!

- Register some types and type traits automatically

It can do more!

- Register some types and type traits automatically
- Convert between types

```
static bool registerConverter(MemberFunction function);
static bool registerConverter(MemberFunctionOk function);
static bool registerConverter(UnaryFunction function);

static bool convert(const void *from, int fromTypeId, void *to, int toTypeId);
```

It can do more!

- Register some types and type traits automatically
- Convert between types
- Compare instances of the same type

```
template<typename T>
static bool registerComparators()

static bool compare(const void *lhs, const void *rhs, int typeId, int* result);
```

It can do more!

- Register some types and type traits automatically
- Convert between types
- Compare instances of the same type
- Iterate over sequences and mappings

It can do more!

- Register some types and type traits automatically
- Convert between types
- Compare instances of the same type
- Iterate over sequences and mappings
- Access QMetaObject pointer

It can do more!

- Register some types and type traits automatically
- Convert between types
- Compare instances of the same type
- Iterate over sequences and mappings
- Access QMetaObject pointer
- Serialize objects to QDataStream and QDebug

Summary

Summary

- Qt needs meta-type information to work with custom types

Summary

- Qt needs meta-type information to work with custom types
- There is no clear cut between compile time and runtime traits

Summary

- Qt needs meta-type information to work with custom types
- There is no clear cut between compile time and runtime traits
- Meta-type information describe and wrap custom API

Thank you!

www.qt.io

www.qt.io

See you there!