Qt Signals and Slots

Olivier Goffart

October 2013
About Me

- QStyleSheet
- Itemviews
- Animation Framework
- QtScript (porting to JSC and V8)
- QObject, moc
- QML Debugger
- Modularisation
- ...
About Me

Offering Qt help and services: Visit http://woboq.com

C++ Code browser: http://code.woboq.org
1 History
2 Pointer to member function
3 Lambda functions
4 New syntax in Qt5
5 Under The Hood
Technical Talk
1 History
2 Pointer to member function
3 Lambda functions
4 New syntax in Qt5
5 Under The Hood
Reference Documentation

The Qt API:
- Alphabetical Class List
- Annotated Class List
- Structure Overview
- Inheritance Hierarchy
- Alphabetical Function Index (long)
- Header File Index
- Widget Screenshots

Using Qt:
- Tutorial
- Example Programs
- Introduction to Signals and Slots
- Using the Meta Object Compiler (mod)
- Debugging Techniques

Qt Extensions:
- OpenGL 3D Graphics Support
- Netscape/Explorer Plugin Support
- Image File Formats Support
- X/11 Legacy Code Support

Further Information:
- TrollTech Contact Information
- Qt Mailing Lists
- Credits
- Developers' Homepage [external]

About Qt

Qt™ is a multi-platform C++ GUI toolkit. It is a product of TrollTech. It is supported on all major variants of Microsoft Windows and Unix/X Windows.

Qt Professional Edition is provided for commercial software development. It is provided with upgrades and technical support. For the latest prices, please see the TrollTech web site, Pricing and Availability page, or contact sales@trolltech.

Qt Free Edition is the Unix/X11 version of Qt available for development of free software only. It is provided free of charge under the Qt Free Edition License. The latest version is available for download.

Copyright © 1998 TrollTech

Trademarks

Qt version 1.41
A Small Example

A minimal C++ class declaration might read:

```cpp
class Foo
{
    public:
        int value() const { return val; }
        void setValue( int );
    private:
        int val;
};
```

A small Qt class might read:

```cpp
class Foo : public QObject
{
    Q_OBJECT
    public:
        Foo();
        int value() const { return val; }
    public slots:
        void setValue( int );
    signals:
        void valueChanged( int );
    private:
        int val;
};
```

Slots are implemented by the application programmer (that’s you). Here is a possible implementation of Foo::setValue():

```cpp
void Foo::setValue( int v )
{
    if ( v != val )
    {
        val = v;
        emit valueChanged(v);
    }
}
```

The line emit valuechanged(v) emits the signal valuechanged from the object. As you can see, you emit a signal by using emit signal(arguments).

Here is one way to connect two of these objects together:

```cpp
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
b.setValue( 11 );
a.setValue( 79 );
b.setValue(); // this would now be 79, why?
```
// The following macros are our "extensions" to C++
// They are used, strictly speaking, only by the moc.

#define slots   » » » »   // slots: in class
#define signals protected » »   // signals: in class
#define emit     » » » »   // emit signal

/* tmake ignore Q_OBJECT */
#define Q_OBJECT   » » » » » » » »   \
   public:» » » » » » » »   
   QMetaObject *metaObject() const { return metaObj; }» » » » » » » »   
   const char *className() const; » » » » » » » »   
protected:» » » » » » » »   
   void initMetaObject(); » » » » » » » »   
private:» » » » » » » »   
   static QMetaObject *metaObj;

/* tmake ignore Q_OBJECT */
#define Q_OBJECT_FAKE Q_OBJECT

#if defined(_OLD_CPP_)
define METHOD(a) "0""a"
define SLOT(a) » "1""a"
define SIGNAL(a) » "2""a"
define METHOD_CODE 0
#define SLOT_CODE 1
#define SIGNAL_CODE 2
#else
#define METHOD(a) "0"#a
#define SLOT(a) » "1"#a
#define SIGNAL(a) » "2"#a
#endif
Qt 2, Qt 3

- Q_PROPERTY
- No major changes in signals and slot
- Thread support
- QueuedConnection
- Meta type registration
- Several major internal changes
- Added file and line number information in debug mode
- But still no changes in the syntax
How Does it Work?

```cpp
bool connect(const QObject *sender,
             const char *signal,
             const QObject *receiver,
             const char *member);
```
How Does it Work?

- Compare the signature string to see if the arguments match.
- Use the information provided by the moc to find the index of the signal and of the slot.
- Keep in an internal map which signal is connected to what slots.
- When emitting a signal, QMetaObject::activate is called.
- It calls qt_metacall (generated by moc) with the slot index which call the actual slot.
```cpp
1  connect(button, SIGNAL(clicked()),
2       this, SLOT(slotCliked()));
```
connect(button, SIGNAL(clicked()),
        this, SLOT(slotCliked()));

connect(socket, SIGNAL(infoReceived(const Info &)),
        this, SLOT(slotInfoReceived(const MyFramework::Info &)));
Problems

1. `connect(button, SIGNAL(clicked()),
   this, SLOT(slotCliked()));`

2. `connect(socket, SIGNAL(infoReceived(const Info &)),
   this, SLOT(slotInfoReceived(const MyFramework::Info &)));`

3. `connect(button3, SIGNAL(clicked()),
   this, SLOT(buttonClicked(3)));`
Problems

1. `connect(button, SIGNAL(clicked()),
          this, SLOT(slotCliked()));`

2. `connect(socket, SIGNAL(infoReceived(const Info &)),
          this, SLOT(slotInfoReceived(const MyFramework::Info &)));`

3. `connect(button3, SIGNAL(clicked()),
          this, SLOT(buttonClicked(3)));`

4. `connect(comboBox, SIGNAL(valueChanged(int)),
           settings, SLOT(updateValue(QVariant)));`
Problems

1. `connect(button, SIGNAL(clicked()),
       this, SLOT(slotClicked()));`

2. `connect(socket, SIGNAL(infoReceived(const Info &)),
       this, SLOT(slotInfoReceived(const MyFramework::Info &)));`

3. `connect(button3, SIGNAL(clicked()),
       this, SLOT(buttonClicked(3)));`

4. `connect(comboBox, SIGNAL(valueChanged(int)),
       settings, SLOT(updateValue(QVariant)));`

5. `connect(model, SIGNAL(modelReset()),
       this, SLOT(oneLineSlot()));`
connect(action, SIGNAL(selected(QString)),
    receiver, SLOT(actionSelected(QString)));

connect(action, &QAction::selected,
    receiver, &Receiver::actionSelected);

connect(action, &QAction::selected,
    [] (const QString &act) {
        qDebug() << "Action selected:" << act;
    });
struct Point { int x; int y; };

int Point::*coordinate = 0;

if (orientation == Qt::Horizontal)
    coordinate = &Point::x;
else if (orientation == Qt::Vertical)
    coordinate = &Point::y;

Point p = /* ... */
Point *pp = /* ... */

if (coordinate)
    pp->*coordinate = p.*coordinate;
struct Point { int x() const; int y() const;
    void setX(int); void setY(int);  
};

int (Point::*getter)() const = 0;
void (Point::*setter)(int) = 0;

if (orientation == Qt::Horizontal) {
    getter = &Point::x;
    setter = &Point::setX;
}

Point p = /* ... */
Point *pp = /* ... */

if (getter && setter) {
    int c = (p.*getter)();
    (pp->*setter)(c);
}
Fun facts

```cpp
int Point::*coordinate = 0;
int (Point::*getter)() = 0;
```

Fun facts

```
1    int Point::*coordinate = 0;
2    int (Point::*getter)() = 0;
3    qDebug() << sizeof(coordinate) << sizeof(getter);
```
Fun facts

1. \texttt{int Point::*\texttt{coordinate} = 0;}
2. \texttt{int (Point::*\texttt{getter})(\texttt{}) = 0;}
3. \texttt{qDebug() << sizeof(\texttt{coordinate}) << sizeof(\texttt{getter});}

8 16
Fun facts

1. `int Point::*coordinate = 0;`
2. `int (Point::*getter)() = 0;`
3. `qDebug() << sizeof(coordinate) << sizeof(getter);`
4. `qDebug() << *reinterpret_cast<int*>(&coordinate);`
Fun facts

1. `int Point::*coordinate = 0;`
2. `int (Point::*getter)() = 0;`
3. `qDebug() << sizeof(coordinate) << sizeof(getter);`
4. `qDebug() << *reinterpret_cast<int*>(&coordinate);`
struct Struct {
    int foo(int);
    int bar(int);
    int bar(double);
};

int (Struct::*barP1)(int) = &Struct::bar;

auto fooP = &Struct::foo;
// decltype (fooP): int (Struct::*)(int)
auto barP2 = &Struct::bar;
error: variable 'barP2' with type 'auto' has incompatible initializer of type '<overloaded function type>'
error: unable to deduce 'auto' from '&Struct::bar'
struct Struct {
    int foo(int);
    int bar(int);
    int bar(double);
};

int (Struct::*barP1)(int) = &Struct::bar;
auto fooP = &Struct::foo;
```cpp
struct Struct {
    int foo(int);
    int bar(int);
    int bar(double);
};

int (Struct::*barP1)(int) = &Struct::bar;
auto fooP = &Struct::foo;

// decltype(fooP): int (Struct::*)(int)
```
struct Struct {
    int foo(int);
    int bar(int);
    int bar(double);
};

int (Struct::*barP1)(int) = &Struct::bar;
auto fooP = &Struct::foo;
// decltype(fooP): int (Struct::*)(int)
auto barP2 = &Struct::bar;
struct Struct {
    int foo(int);
    int bar(int);
    int bar(double);
};

int (Struct::*barP1)(int) = &Struct::bar;
auto fooP = &Struct::foo;

// decltype(fooP):: int (Struct::*)(int)
auto barP2 = &Struct::bar;

error: variable ‘barP2’ with type ‘auto’ has incompatible initializer of type ‘<overloaded function type>’

error: unable to deduce ‘auto’ from ‘& Struct::bar’
Lambda

[foo] (int a) -> int { return a + foo; }

- **Capture:** Variables that you capture
- **Parameter list:** The parameters of the function
- **Return type (optional)**
- **Function body**
[foo] (int a) -> int { return a + foo; }

struct {
    double foo;
    int operator()(int a)
    { return a + foo; }
}

int a = 1, b = 2, c = 3;

// 'a' by value, 'b' by reference
auto f1 = [a, &b]() { b = a; };

// everything by reference
auto f2 = [&]() { b = a; };

// everything by value
auto f3 = [=]() { return a + c; };

// everything by value, 'b' by reference
auto f4 = [=,&b]() { b = a + c; };
Examples

```cpp
1 connect(button3, &QPushButton::clicked,
2     [=] { this->buttonClicked(3); });
3
4 connect(model, &QAbstractItemModel::modelReset,
5     [] { qDebug() << "model has been reset"; });
```
Outline

1. History

2. Pointer to member function

3. Lambda functions

4. New syntax in Qt5

5. Under The Hood
New connect Overloads

1. `QObject::connect(const QObject *sender, const char *signal, const QObject *receiver, const char *slot, Qt::ConnectionType type)`

2. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, const QObject *receiver, PointerToMemberFunction slot, Qt::ConnectionType type)`

3. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, Functor method)`

4. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, const QObject *context, Functor method, Qt::ConnectionType type) (since Qt 5.2)`
New connect Overloads

1. `QObject::connect(const QObject *sender, const char *signal, const QObject *receiver, const char *slot, Qt::ConnectionType type)`

2. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, const QObject *receiver, PointerToMemberFunction slot, Qt::ConnectionType type)`

3. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, Functor method)`

4. `QObject::connect(const QObject *sender, PointerToMemberFunction signal, const QObject *context, Functor method, Qt::ConnectionType type)` (since Qt 5.2)
There is no "receiver" when connecting to a lambda.

Receiver can be used for:

1. Thread affinity (QueuedConnection)
2. Automatic disconnection when the receiver is destroyed
3. sender()

In Qt 5.2 you can use a context with your lambda function

```cpp
connect(button3, &Button::clicked,
        bar, [=]{ bar->buttonClicked(3); });
```
Remember QSignalMapper?

```cpp
for (int i = 0; i < texts.size(); ++i) {
    QPushButton *button = new QPushButton(texts[i], this);
    // A C++11 lambda
    connect(button, &QPushButton::clicked, [=]{ this->select(texts[i]); });
    // OR, without C++11, using tr1/boost bind
    connect(button, &QPushButton::clicked, bind(&MyWidget::select, this, texts[i]));
}
```
Qt 5 syntax

Design Goals

- Detect as many errors as possible at compile time
- Be easy and intuitive
- Do not require users to understand templates or function pointers
- Do not require C++11
Qt 5 syntax

History

- Research started in august 2009 (~ Qt 4.6)
- First trial:

```cpp
connect(QSignal(s, &SenderObject::signal1),
    QSlot(r1, &ReceiverObject::slot1));
```
Protected Signals

In Qt4 (and before)

```cpp
#define signals protected
```

```cpp
connect(MyObject, &QObject::destroyed, //...
```

`main.cc:37:22: error: 'destroyed' is a protected member of 'QObject'`
Can we change?

```
#define signals public
```
Protected Signals

Can we change?

```cpp
#define signals public
```

Two problems

- Binary compatibility
- Everybody can emit a signal from another object
Is there a way to avoid typing the type of the object?

1 connect(QSIGNAL(button, clicked),
2 QSLOT(receiver, buttonClicked));
Typing the type name

Is there a way to avoid typing the type of the object?

```cpp
connect(QSIGNAL(button, clicked),
       QSLIGHT(receiver, buttonClicked));

// C++11 Only
#define QSIGNAL(OBJ, FUNC) OBJ, [&](){
    typedef std::remove_reference<
        decltype(*OBJ)>::type Type;
    return &Type::FUNC();
}()
```
class Obj : public QObject {
    Q_OBJECT
    signals:
    void valueChanged(int);
    void valueChanged(const QString &);
};

QObject::connect(obj, &Obj::valueChanged, [[]]{});
class Obj : public QObject {
    Q_OBJECT
    signals:
    void valueChanged(int);
    void valueChanged(const QString &);
};
QObject::connect(obj, &Obj::valueChanged, []{});

error: no matching function for call to ‘QObject::connect(Obj* const, <unresolved overloaded function type>, __lambda0)’
```cpp
class Obj : public QObject {
    Q_OBJECT
    signals:
    void valueChanged(int);
    void valueChanged(const QString &);
};

QObject::connect(obj, &Obj::valueChanged, []({}));

error: no matching function for call to 'QObject::connect(Obj* const, <unresolved overloaded function type>, __lambda0)'

QObject::connect(obj,
    static_cast<void(Obj::*) (int)>(&Obj::valueChanged),
    []({}));
```
Overloads

Avoid overloading signals!
5 Under The Hood

- Moc
- Connections
- Emiting a Signal
- New Syntax
Under The Hood

- Moc
- Connections
- Emiting a Signal
- New Syntax
Outline

5 Under The Hood
- Moc
- Connections
- Emiting a Signal
- New Syntax
Outline

5 Under The Hood
- Moc
- Connections
- Emiting a Signal
- New Syntax
New Syntax
Summary

- Compile time checks
- Not problems in arguments with namespaces or typedef
- Automatic type conversions
Summary

- Compile time checks
- Not problems in arguments with namespaces or typedef
- Automatic type conversions

With C++11 you benefit from

- No 6 arguments limit
- Better error messages `static_assert`
- Lambda functions
Questions

olivier@woboq.com

visit http://woboq.com