Qt5 with a touch of C++11

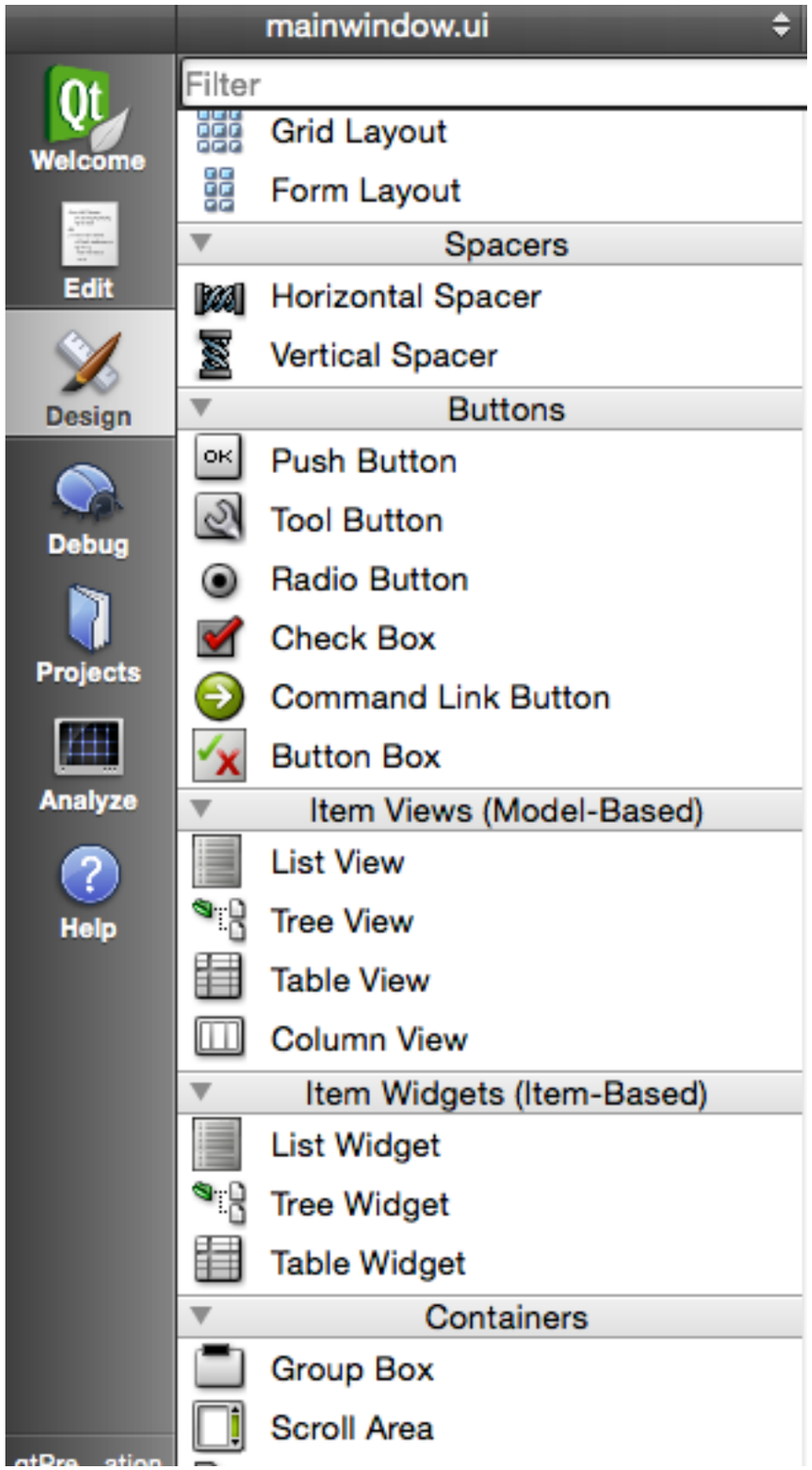
Matthew Eshleman
covemountainsoftware.com

Background - Matthew Eshleman

- 15+ years of embedded software development, architecture, management, and project planning
  - Delivered 30+ products with many more derivative projects. Millions of people have used my code. No one has threatened me yet (*except one guy in marketing*.)
- Primarily C / C++, lately discovered they joy of C#
- Recently discovered: Qt5 with C++11
- Learn more: http://covemountainsoftware.com/consulting

Qt?

- Qt is an application and user interface development framework primarily focused on **cross platform** UI/App development.
  - It also happens to be a great framework for more advanced embedded platforms. Especially since Qt5.
- Qt was born in 1991.
- Commercial licenses. Then GPL + Commercial
- 2009 added LGPL



Qt Apps

- Perforce Client
- VLC Media Player
- Bitcoin
- Netflix embedded player (TVs, DVDs..)
- Dropbox (Linux)

A Framework has… **modules**… lots of them…

Qt Network

Qt WebSockets

Qt Json

Qt Widgets

Qt WebKit

Qt State Machine

QtConcurrent

QFile

Qt Serial Port

QThreadPool

etc

QVector

Qt Animation

QTimer

QTouchEvent

QSettings

etc

What I love…

- Signal and slots
- Enhanced container libraries with built in memory management
- LGPL license (changing…)
- Qt5 Plugin architecture (input, graphics, touch)
- Cross platform
  - Develop UI/algorithms on PC, port to embedded target
- My boys —>

What I do not love

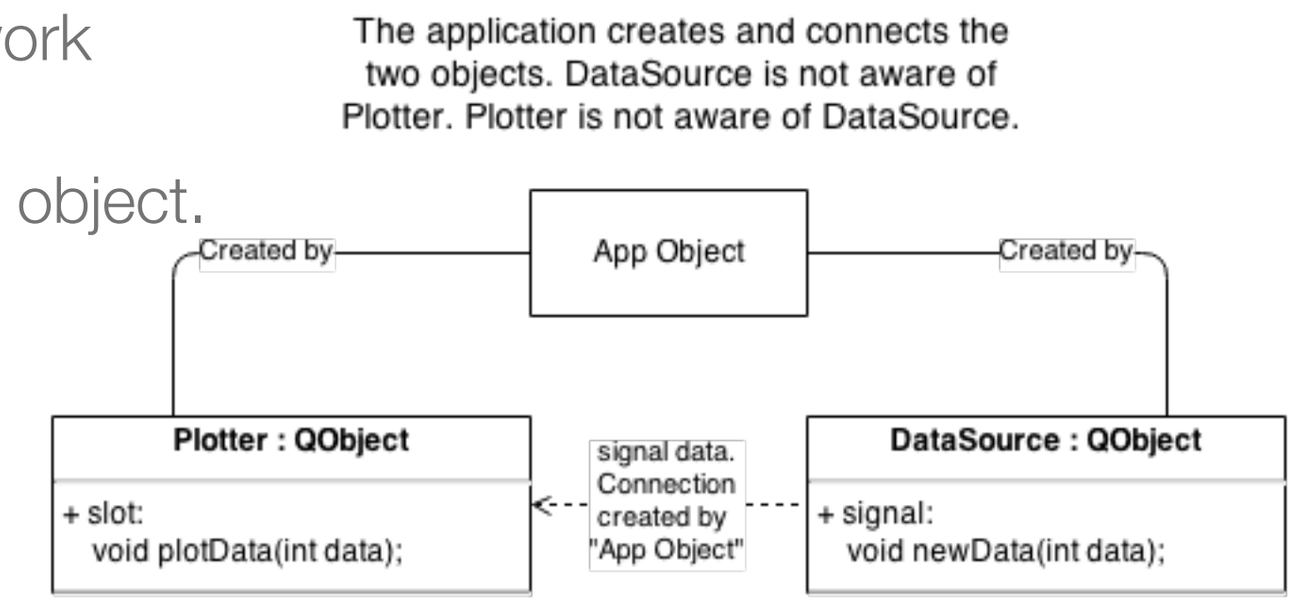
- The “meta object compiler” (discuss more later)
  - This enables the “signals and slots” functionality… which I love.
- Superset of C++. Pretty much must use their IDE if you want dynamic error highlighting and autocomplete that is “Qt aware”.

Elements for today

- Signals and Slots
- Container classes
- Embedded target workflow
- Strategies to use Qt’s cross platform capabilities to enable faster development

Signals and Slots

- Key/critical element of the Qt framework
- A **signal** is a message emitted by an object.
- A **slot** receives the message.
- Enables loose coupling
- Type safe (but not necessarily at compile time)
- signal and slot are Qt keywords, processed by the meta compiler
- Comparable to C#'s delegate / event



Code (defining and sending a signal)

```
//header
class DataSource : public QObject
{
    Q_OBJECT
public:
    explicit DataSource(QObject *parent = 0);
    virtual ~DataSource();

signals:
    void newData(int data);
};
```

Define the signal

```
//source
void DataSource::InternalGenerator()
{
   newValue = …;
   newData( newValue );

   //or

   emit newData( newValue );
}
```

Send or “emit” the Signal by calling the `newData()` method.
“emit” keyword is syntactic sugar.

Code (defining and using a slot)

```
//header
class Plotter : public QWidget
{
    Q_OBJECT
public:
    explicit Plotter(QWidget *parent = 0);
    ~Plotter();

public slots:
    void plotData(int data);
};
```

Define the slot
(it is just a method really)

```
//source
void Plotter::plotData(int data)
{
  //do stuff with new data pt
  …

  update(); //request paint…
}
```

There is nothing special about the slot method. It is just another method and can be used as such.

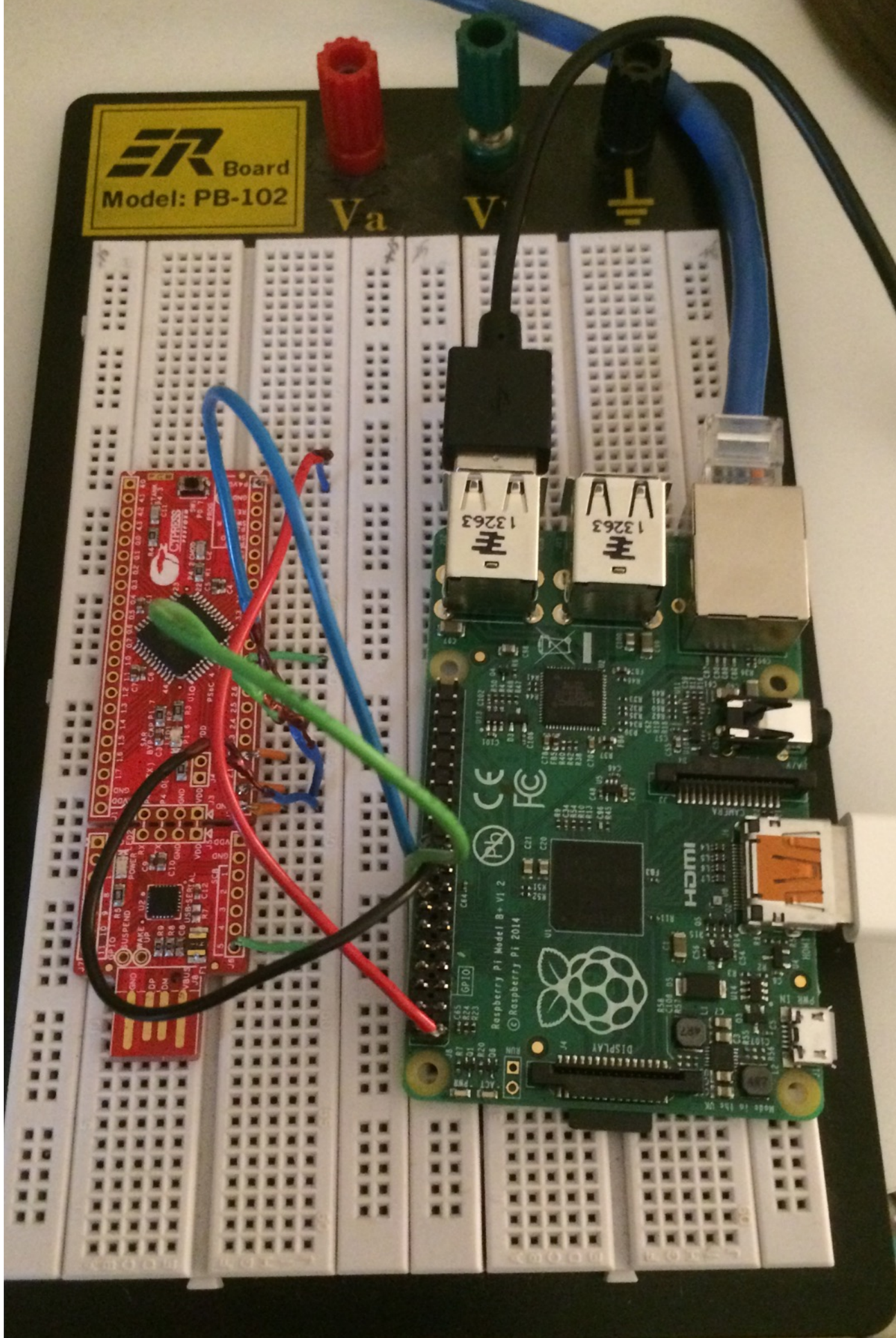
Code (connecting signals and slots, Qt5 approach)

```
{
  //need some data to plot
  mpSource = new DataSource(this);

  //connect data source and just show last value in a text label
  connect(mpSource, &DataSource::newData,
            [=](int data){
              ui->lastValueLabel->setText(QString::number(data));
            });

  //connect the data source object to the plotter object
  connect(mpSource, &DataSource::newData,
          ui->plotter, &Plotter::plotData);
}
```

C++11 Lambda Usage!

Normal typesafe member function reference

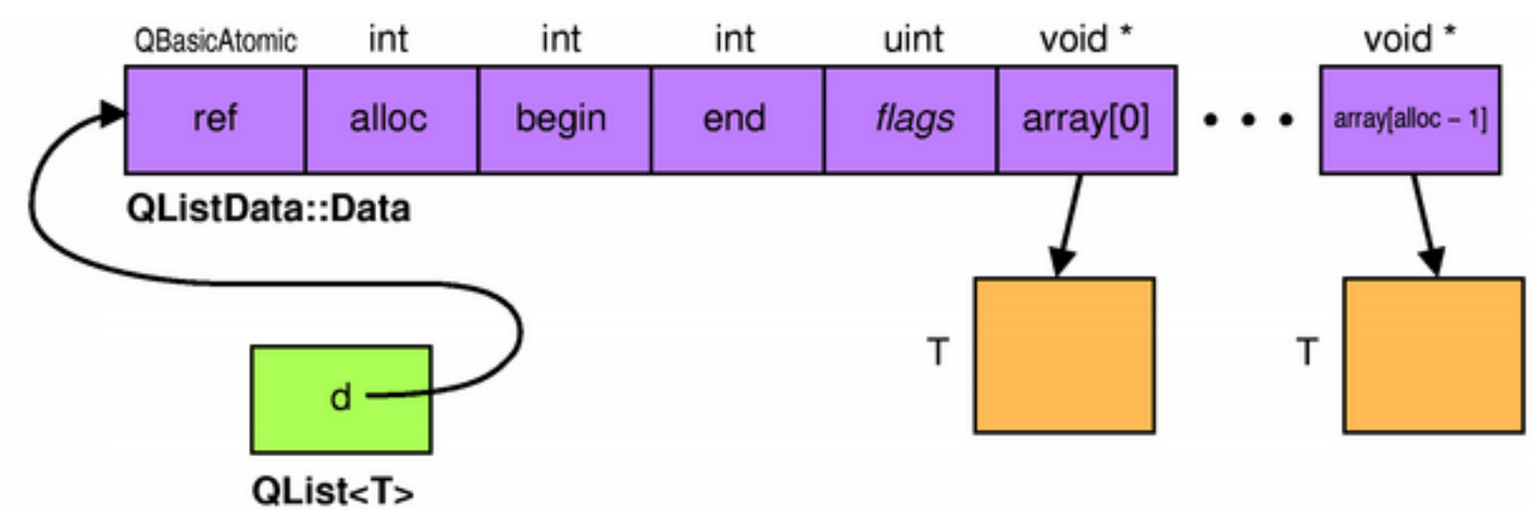
Demo Time!



Containers

Qt's Container Types

- **vector**, **list**, linked list, stack, **queue**, set, **map**, multi map, hash, multi hash
- Template based
- **Implicitly Shared - reference counted with copy on write**
  - **Pass by value (easy, atomic reference counted)**
  - **Copy on write (pass it around. If anyone modifies, they get a "deep copy" automatically)**
- Iterators (STL style too)
- use Qt's foreach macro OR use C++11's new "for each" style



Go ahead, pass it around…

- Qt encourages pass by value of containers
- The data is internally using a pointer.
- Feels like C#
- Especially useful when emit'ing a QVector or other container.

Code (sending a vector)

```
//header
class DataSource : public QObject
{
    Q_OBJECT
public:
    explicit DataSource(QObject *parent = 0);
    virtual ~DataSource();

signals:
    void newData(QVector<double> data);
};
```

Notice this is not a pointer, it is pass by value.

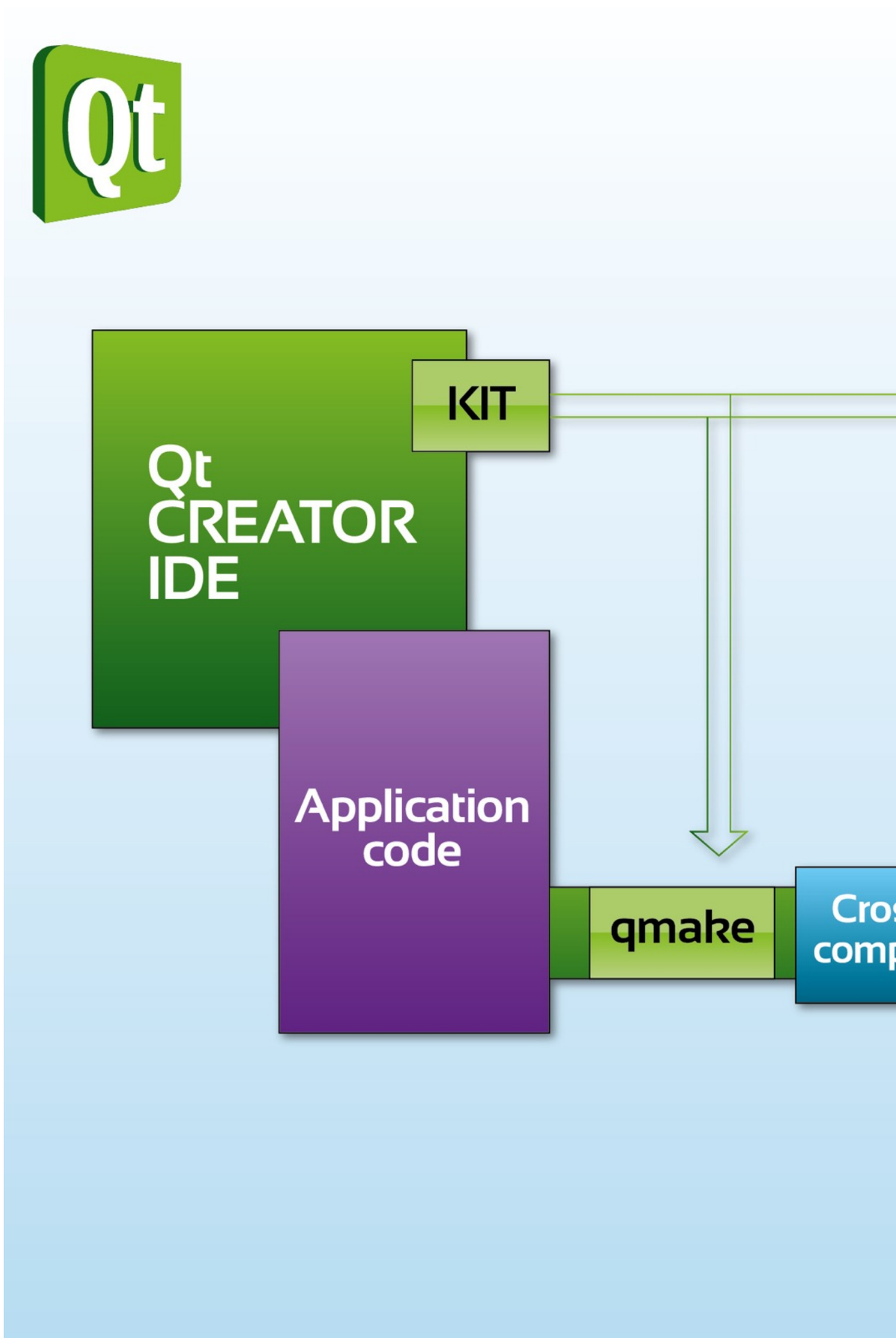
```
//source
void DataSource::InternalGenerator()
{
   QVector<double> newValue = …;
   emit newData( newValue );
}
```

Again, just pass the actual vector object to the signal/emit method.

Pros/cons of sending a QVector, QList, etc, versus traditional “C” pointer approach

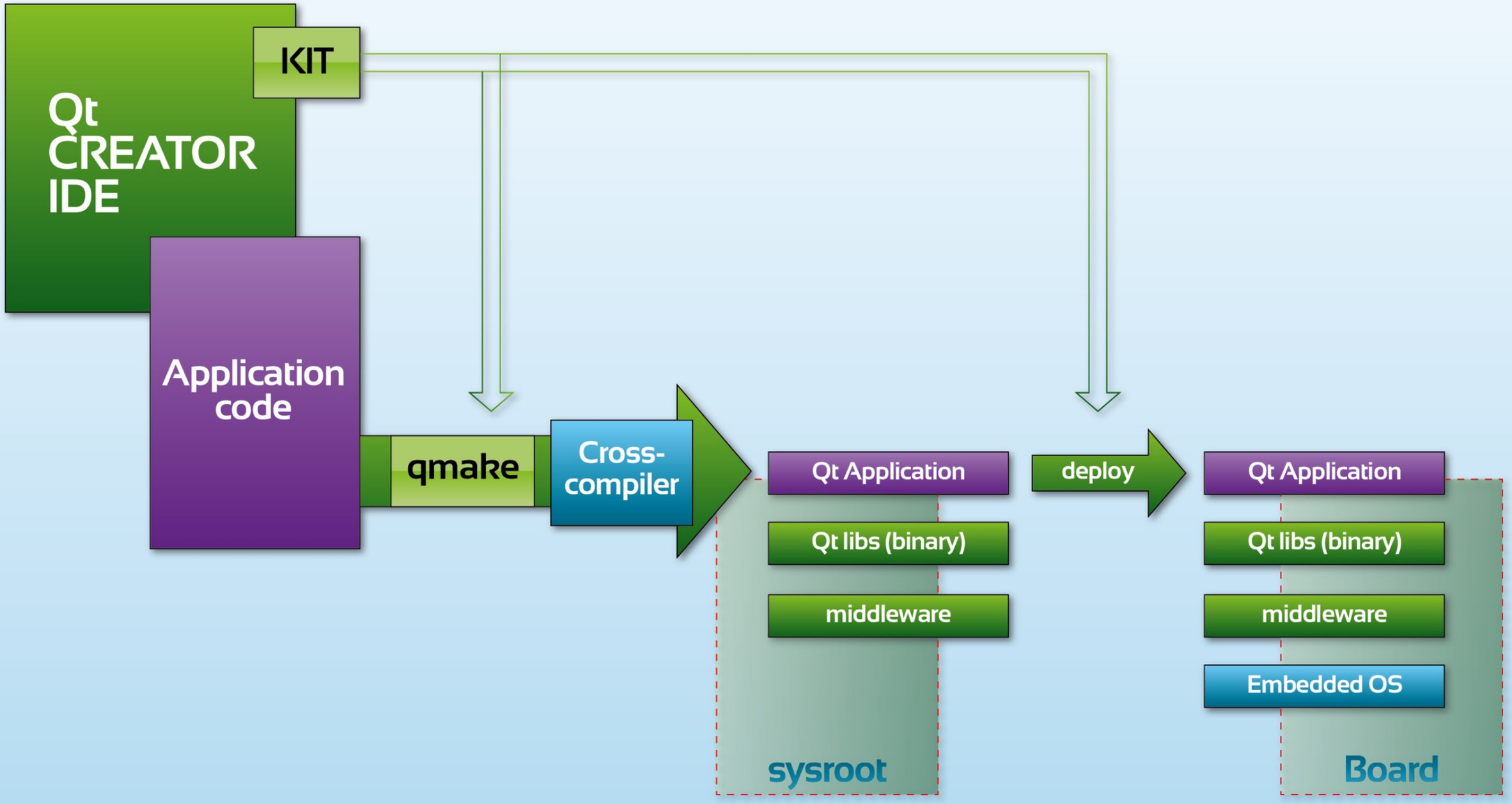
- Pros
  - Easy mental model. The object has everything needed (size, memory, etc). This will result in more reliable code.
  - Very safe memory model - no leaking thanks to reference counting.
- Cons
  - Will be slightly slower (atomic reference counting)
  - Increased heap usage versus a ring buffer or memory pool approach.

Workflow



Embedded work flow

- What is needed
  - The target device (raspberry pi, etc, etc)
  - Host PC (typically Linux) with:
    - Cross compiler for target
    - Qt libraries cross compiled for the target
    - Target file system with development headers, libraries, etc
      - Often called the “rootfs” or “sysroot” of the target
    - Qt “qmake” setup for target on host
    - Typically using Qt Creator for the IDE. Works great with remote/embedded targets.



Qt with remote embedded target workflow

Image Source

Helpful sites for embedded Linux with Qt

- These sites help or provide the necessary libraries, compilers, and/or filesystems needed for embedded Linux. In many cases they have Qt compiled and ready for you as well.
  - Build root (http://buildroot.uclibc.org/)
  - emdebian (http://www.emdebian.org/)
  - Linaro, ARM specific (http://www.linaro.org/)
  - Raspberry pi… etc.

Quick Tip

Hide the hardware from the app

- Use abstract base class (similar to C# interface)
- Use the factory pattern to create appropriate concrete object for the current hardware. This might be a “fake” object when building for a desktop.
- Especially with Qt, this allows the developer to use a desktop PC for developing and debugging the application’s custom graphics or algorithms.

Example

```
class IntegerDataSource : public QObject
{
    Q_OBJECT
public:

    virtual void Activate() = 0;
    virtual void Deactivate() = 0;

signals:
    void newData(int newDataSample);
};
```

Define the interface

Factory returns only pointer to abstract base class.

```
class DataSourceFactory
{
public:
    static IntegerDataSource*
                GetNewIntegerSource(… typeOfSource,…);
};
```

Parting comments…

- Qt's threading model is great (and the framework supports signals/slots across threads)
- Lots of communications options:
  - serial port, websockets, general networking, json, xml, etc…
- CSS like style sheets
- OpenGL
  - Alternate application domain: QtQuick (QML). Requires OpenGL
- Explore… Qt has a great deal to offer!

Thank you!

Any questions?
matthew@covemountainsoftware.com

Useful links

- http://stackoverflow.com/questions/16948382/how-to-enable-c11-in-qt-creator
- https://qt-project.org/
- http://qt-project.org/wiki/New_Signal_Slot_Syntax
- http://doc.qt.digia.com/qq/qq19-containers.html
- Build root (http://buildroot.uclibc.org/)
- emdebian (http://www.emdebian.org/)
- Linaro, ARM specific (http://www.linaro.org/)
- https://github.com/metrological/buildroot
    - Qt5/RaspberryPi optimized buildroot config