# Refactoring CopperSpice Using a New C++ Signal Library

Barbara Geller & Ansel Sermersheim

ACCU- April 2016

# Introduction

- Brief Introduction to CopperSpice
- Signals & Slots
  - what are they
  - boost signals
  - CsSignal library
- CopperSpice Refactored
  - integration with CsSignal library
  - reflection using C++
- Future plans for CopperSpice

# What is CopperSpice

- CopperSpice is a collection of C++ libraries derived from the Qt framework. Our goal was to change the core design of the libraries leveraging template functionality and modern C++11 capabilities.

    - CS can be built with Autotools or CMake
    - CopperSpice is written in pure C++11
    - GPL and LGPL
    - CS can be linked directly into any C++ application
    - Meta Object Compiler (moc) is obsolete and is not required when building CS or your C++ applications

# Timeline

| | |
|---|---|
| TrollTech Qt 1.0 | Sept 1996 |
| Nokia bought Qt from TrollTech | June 2008 |
| Digia acquires Qt from Nokia | Sept  2012 |
| Qt 5.0 initial release | Dec 2012 |
| CopperSpice 1.0.0 | May 2014 |
| Qt 5.6 ( LTS release ) | March 2016 |
| CsSignal 1.0.0 | May 2016 |
| CopperSpice 1.2.2 | May 2016 |

# Why CopperSpice

- Contribute to Qt or Develop CopperSpice?
    - moc limitations
        - generated code is mostly string tables
        - does not support templates
        - every passed parameter is cast to a void *
    - bootstrap issues
        - bootstrap library is used when building moc
        - same source used for bootstrap and QtCore
    - qmake
    - CLA concerns

# Why CopperSpice

- What should CopperSpice be?
  - build system not tied to qmake
    - Autotools
    - CMake
  - moc removed
  - use native C++11 atomics & smart pointers
  - containers
    - leverage C++11 containers
    - extend the CS api functionality
    - document semantics
  - signal / slot delivery as a separate library

# What are Signals and Slots

- Signal
  - notification that something occurred
- Slot
  - an ordinary method, function, or lambda
- Connection
  - associates a signal with a slot
  - a signal can be connected to multiple slots
- Activation
  - when the signal is emitted the connected slot is called

# What are Signals and Slots

- Boost Signals 2
  - signals are objects
  - “most” of the signal classes are thread safe
  - adding or removing a signal to a class will break the ABI of this class
  - slots are called only in the current thread
  - you can not connect a signal in one thread to a slot in another thread ( thread aware  - no )

# What are Signals and Slots

- CopperSpice Signals
  - signals are methods
  - adding or removing a signal to a class will not break the ABI of this class
  - slots are called on the thread specified by the receiver
  - you can connect a signal in one thread to a slot in another thread ( thread aware - yes )

# Signal Activation

- CopperSpice
  - QPushButton::clicked() signal method
  - created by a macro located in an .h file in your program
  - function activate<Args...>(data...) is called with the complete parameter list, including all of the data types

- Qt
  - QPushButton::clicked() signal method
  - generated by moc, type information stored in a string table
  - function activate() is called with an array of void *, all of the slot data types are lost

# Declarations in your .h File

```
// signal & slot declarations in CopperSpice

public:
   CS_SIGNAL_1(Public, void clicked(bool status))
   CS_SIGNAL_2(clicked, status)

   CS_SLOT_1(Public, void showHelp())
   CS_SLOT_2(showHelp)
```

# Connections in your .cpp File

```
// ways to make a connection in CopperSpice

connect(myButton, “clicked(bool)”,
   this, “showHelp()”);

connect(myButton, &QPushButton::clicked,
   this, &Ginger::showHelp);

connect(myButton, &QPushButton::clicked,
   this, [this](){showHelp()});
```

# Runtime Activation

- QObject::activate<Args...>(data...)

    - template method
    - called every time a signal is emitted
    - compares the signal with the list of existing connections
    - when a match is found the associated slot is called

    - multiple slots can be connected to a given signal
    - queued connections can cross threads
    - blocking queued connections will wait for the slot to return

# Part II

## CsSignal Library

# CsSignal Library

- Migrated the Signal / Slot functionality out of CopperSpice and created a new standalone library
  - class SignalBase
    - inherit from this class to send a signal
  - class SlotBase
    - inherit from this class to receive a signal
  - class PendingSlot
    - function object which encapsulates the call to a slot

# CsSignal Library

- Who can use CsSignal library?

    - if you are using Boost Signals 2
        - want a simpler interface
        - need thread awareness
    - directly in your applications even if you have no GUI
    - multithreaded or reactive programming
    - replace your callback functions
    - license is BSD 2 Clause

    - CsSignal library does not require CopperSpice

# Review

- lvalue reference
  - caller will observe the modifications made in the called function or method
- const reference
  - called method or function can not modify the object
- rvalue reference
  - declared using &&
  - binding an rvalue to an rvalue reference prolongs the lifetime as if were an lvalue

# Review

- rvalue reference
  - in a declaration with a deduced type && is called a forwarding reference
  - if you think "rvalue reference" whenever you see && in a type declaration, you will misread C++11
  - && might actually mean &
  - a forwarding reference can be an lvalue reference or an rvalue reference

  - when a variable or parameter is declared with type T && (where T is a deduced type) that variable or parameter is a forwarding reference

# CsSignal Library

- ConnectionKind
  - QueuedConnection
    - slot is executed in the receiver's thread
  - BlockingQueuedConnection
    - slot is invoked, thread blocks until the slot returns

```
enum class ConnectionKind {
    AutoConnection,
    DirectConnection,
    QueuedConnection,
    BlockingQueuedConnection,
};
```

# CsSignal Library

- Connect function
  - sender
    - const reference to a SignalBase, `QPushButton`
  - signal
    - method pointer, `&QPushButton::clicked`
  - receiver
    - const reference to a SlotBase, `this`
  - slot
    - method pointer, function ptr, or lambda, `showHelp()`
  - connectionKind
    - enum, default is `AutoConnection`

# CsSignal Library

- Connect function
  - sender and receiver are passed by const reference
  - a const reference can bind to an lvalue or an rvalue

```
// QPushbutton{} is an rvalue
connect(QPushbutton{}, &QPushbutton::clicked,
        this, &Ginger::showHelp);
```

- connect() will bind the rvalue to the const reference, the data will be correctly stored in the connection list

# CsSignal Library

- Connect function
  - sender and receiver are passed by const reference
  - a const reference can bind to an lvalue or an rvalue

```
// QPushbutton{} is an rvalue
connect(QPushbutton{}, &QPushbutton::clicked,
        this, &Ginger::showHelp);
```

- connect() will bind the rvalue to the const reference, the data will be correctly stored in the connection list
- when the calling method “completes” the rvalue will be destroyed
- the destructor for QPushButton will disconnect this connection
- ultimately sender and receiver should be a forwarding reference

# CsSignal Library

- Disconnect function
  - sender, signal, receiver, slot
    - signal method pointer
    - slot method pointer
  - sender, signal, receiver, slot
    - signal is a method pointer
    - slot is a function pointer or a lambda

# CsSignal Library

- Activate function
  - sender
    - lvalue reference
  - signal
    - method pointer
  - data
    - variadic parameter pack

  - never call directly, not part of the API
  - activate is called from the signal method
  - to emit the signal, call the signal method
  - from our example this would be `clicked()`

# CsSignal Library

- HandleException
  - used in `activate()`
  - called if the slot throws an exception
  - the current exception is passed to `handleException()`
  - virtual method, default does nothing in CsSignal library

# CsSignal Library

- QueueSlot method
    - class SlotBase provides a virtual method called queueSlot() which can be reimplemented to override cross thread signal delivery
    - the default is to call the slot immediately

```
void SlotBase::queueSlot(PendingSlot data,
                         ConnectionKind type)
{
   data();
}
```

# CsSignal Library

- CompareThreads method
  - class SlotBase provides a virtual method called compareThreads() which can be reimplemented to override cross thread signal delivery
  - the default assumes the sender and receiver are in the same thread

```
bool SlotBase::compareThreads()
```

# Declarations in your .h File

```
// signal & slot declarations in CsSignal

public:
  SIGNAL_1(Public, void clicked(bool status))
  SIGNAL_2(clicked, status)

  void showHelp() {
    // some code for the slot
  }
```

# Connections in your .cpp File

```
// ways to make a connection in CsSignal

connect(myButton, &QPushButton::clicked,
   this, &Ginger::showHelp);

connect(myButton, &QPushButton::clicked,
   this, [this](){showHelp()});
```

# GUI Base Class

- QObject
  - main base class which all GUI classes inherit from
    - QDialog
    - QPushButton
    - QTreeView
  - too much functionality
  - too many data members
  - data members were not thread safe
  - several bit fields for boolean flags
  - signal and slot structures with redundant data members

# CopperSpice Integrated with CsSignal

- QObject now uses multiple inheritance

```
class QObject : public virtual SignalBase,
                public virtual SlotBase
```

- QObject
  - removed class members which became obsolete and members which moved to SignalBase or SlotBase
  - destructor refactored
  - improved readability
  - CopperSpice libraries 10-15% smaller

# CopperSpice Integrated with CsSignal

- Wrote wrappers in CopperSpice to call the CsSignal library and maintain our existing API

- CopperSpice calls connect(), disconnect(), and activate() which are now in CsSignal

- Your class in CopperSpice can inherit directly from SignalBase

# QObject / Signal Example

- What are the ways to leverage the changes we have made by refactoring CopperSpice, shrinking QObject, and adding our new CsSignal library?

# QObject / Signal Example

- QFuture<T>
  - does not inherit from anyone, including QObject
  - can not emit signals

- QFutureWatcher<T>
  - inherits from QFutureWatcherBase
  - QFutureWatcherBase inherits from QObject
  - allows monitoring a QFuture using signals & slots

  - QFutureWatcherBase emits a signal when a QFuture becomes ready
  - signals and slots can only exist in QFutureWatcherBase

# QObject  / Signal Example

- CopperSpice will resolve this by changing the inheritance and removing QFutureWatcher and QFutureWatcherBase

```
class QFuture<T> : public SignalBase, public SlotBase
```

  - this can not be done in Qt 5 due to moc limitations

# Part III

## Registration

# Registration in CopperSpice

- CopperSpice allows strings to be used for signal or slot methods
- Allowing string names means there must be a mechanism to look up the name at run time and retrieve the method pointer
- In CopperSpice the signal or slot name and the corresponding method pointer are saved in a map at run time

# Reflection in CopperSpice

- Reflection is the ability of a program to examine its own structure or data
- C++ does not have built in reflection
- CopperSpice registration would be unnecessary or simplified if C++ supported reflection natively

# What is Reflection

- RTTI (run time type information)
    - dynamic_cast<T> and typeid

- Introspection
    - **examine** data, methods, and properties at runtime

- Reflection
    - **modify** data, methods, and properties at runtime

*A "property" is similar to a class data member*

# Reflection in CopperSpice

- At compile time, the registration process is initialized by macros in your .h file

- At run time, the registration methods are called automatically to set up the meta data

- Registration of class meta data occurs the first time a specific class is accessed

# Techniques used to Implement Reflection

- Random number of signals or slots scattered in a class declaration
- How do you automate the process of registering the meta data for each method?

  - macros
  - constexpr
  - method overloading
  - inheritance
  - templates
  - decltype

# Our Goal

- Ideally, we would like to have the cs_register()  method  do something and then call the “next cs_register” method
- This is not valid C++ code

```
cs_register(0) {
    //  do something
    cs_register(1);
}

cs_register(1) {
    //  do something
    cs_register(2);
}
```

# Review

- method overloading

```
void foo(int data1)  {
    //  do something with int
}

void foo(std::string data2)  {
    //  do something with the string
}
```

# Review

- constexpr expressions evaluated at compile time
- foo is initialized to 42 at compile time
- without constexpr the array size would be invalid

```
static constexpr int foo = 30 + 12;
char data[foo];
```

# Review

```
// macro expansion
// CS_TOKENPASTE2(value_, __LINE__)

41
42  CS_SLOT_1(Public, void showHelp())
43  CS_SLOT_2(showHelp)
44

41
42  . . .  value_42
43  . . .  value_43
44
```

# Implementation

- “zero” and “one” are integer values
- method overloading is based on data types
- how can you make a value a data type?

```
cs_register(0)  {
  // do something
  cs_register(1);
}
```

# Templates

- Templates allow you to pass a data type as a parameter to a class, method, or function

- Can you pass an integer value as a template parameter?

    - yes, passing an integer to a template creates a unique data type (by instantiating the template)

- So how do you create a class template to “wrap” the integer value as a new data type?

# Template Class with an Integer Argument

```
template<int N>
class CSInt : public CSInt<N - 1> {
   public:
      static constexpr const int value = N;
};

template<>
class CSInt<0> {
   public:
      static constexpr const int value = 0;
};


// inheritance relationship, “3” inherits from “2”,
”2” inherits from “1”, and “1” inherits from “0”
```

# Class Ginger Expansion (after pre-processing)

```
class Ginger : public QObject
{
 public:
   template<int N>
   static void cs_register(CSInt<N>) { }

   static constexpr CSInt<0> cs_counter(CSInt<0>);


// this code is expanded from a macro which is called
// at the beginning of your class
```

# Example Class ( after preprocessing )

```
// macro expansion from line 42
static constexpr const int value_42 =
   decltype(cs_counter(CSInt<255>{}))::value;

static constexpr CSInt<value_42 + 1> cs_counter(CSInt<value_42 + 1>);
// additional code . . .


// macro expansion from line 43
static constexpr const int value_43 =
    decltype(cs_counter(CSInt<255>{}))::value;

static constexpr CSInt<value_43 + 1> cs_counter(CSInt<value_43 + 1>);
// additional code . . .


// what is value_42 ?  what is value_43 ?
```

# Using the Counter Value

```
// retrieve current counter value of “zero”
static constexpr const int value_42 =
    decltype(cs_counter(CSInt<255>{}))::value;

static constexpr CSInt<value_42 + 1> cs_counter(CSInt<value_42 + 1>);

// setup “cs_register(0)”
static void cs_register(CSInt<value_42>)
{
  cs_class::staticMetaObject().register_method(“showHelp",
    &cs_class::showHelp, QMetaMethod::Slot, "void showHelp()",
    QMetaMethod::Public);

  cs_register(CSInt<value_42 + 1>{} );
}

// retrieve current counter value of “one” . . .
```

# Using the Counter Value

```
// cs_counter() can only “see” above this point
static constexpr const int value_42 =
   decltype(cs_counter(CSInt<255>{}))::value;

static constexpr CSInt<value_42 + 1> cs_counter(CSInt<value_42 + 1>);

// cs_register() can “see” the entire class
static void cs_register(CSInt<value_42>)
{
  cs_class::staticMetaObject().register_method(“showHelp",
    &cs_class::showHelp, QMetaMethod::Slot, "void showHelp()",
    QMetaMethod::Public);

  cs_register(CSInt<value_42 + 1>{} );
}
```

# Challenges with CopperSpice

- Registration process
  - signals, slots, properties, and invokable methods
  - obtaining the values of an enum

- Benefits of the CopperSpice Registration System
  - cleaner syntax
  - improved static type checking
  - no lost data type information
  - no string table comparisons
  - no limit on parameter types or number of parameters

# Sample Moc Code

```
void QPushButton::clicked(bool _t1) {
  void *_a[] = { Q_NULLPTR, const_cast<void*>(
      reinterpret_cast<const void*>(&_t1)) };
  QMetaObject::activate(this, &staticMetaObject, 0, _a);
}

void QPushButton::qt_static_metacall(QObject *_o, QMetaObject::Call _c,
      int _id, void **_a)
{
  if (_c == QMetaObject::InvokeMetaMethod) {
    QPushButton *_t = static_cast<QPushButton *>(_o);
    Q_UNUSED(_t)
    switch (_id) {
      case 0: _t->clicked((*reinterpret_cast< bool*(*)>(_a[1])));
        break;
      default: ;
     }
  }
  // ...
```

# Part IV

## Future Plans

# Current Advantages of CopperSpice

- Template classes can inherit from QObject
- Compound data types are supported
- Signal activation does not lose type information
- Signal / Slots refactored
- Obsolete source code removed
- Build system improvements
- Container library reimplementation
- Atomics improved
- Improved API documentation

# Why CopperSpice requires C++11

- type traits
- enable_if
- decltype with an expression (expression SFINAE)
- tuples, templates to deconstruct a tuple
- constexpr
- lambda functions
- variadic templates
- templates to build a variadic parameter list

# KitchenSink Application

- Music Player
- HTML Viewer
- Font Selector
- Standard Dialogs
- XML Viewer
- Calendar Widget
- Sliders
- Tabs
- Analog Clock
- And More. . .

# How to contribute

- Developers
  - any C++ enthusiast who would like to contribute
  - help us improve the documentation

- Using CopperSpice
  - if your C++ application requires a GUI we encourage you to use CopperSpice
  - binary files available for Linux, OS X, and Windows

# Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- KitchenSink
  - over 30 CopperSpice demos in one application
- Diamond
  - programmers editor which uses the CS libraries
- DoxyPress & DoxyPressApp
  - documentation program
- CsSignal Library
  - standalone thread aware signal / slot library

# Where to find our libraries

- download.copperspice.com/cs_signal/source/

- www.copperspice.com
- download.copperspice.com
- forum.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- Questions?  Comments?