CopperSpice and the Next Generation of Signals

Barbara Geller & Ansel Sermersheim

CppNow - May 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
  - LGPL 2.1 license
  - CS can be linked directly into any C++ application
  - Meta Object Compiler (moc) is obsolete and not required when building your C++ application

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 1.3.0	May 2016

Contribute to Qt or Develop CopperSpice

- Moc
  - 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 lib and QtCore lib
- Qmake
- CLA issues
- Concerned Qt is not focused on modern C++

What should CopperSpice be

- Build system not tied to qmake ✓
  - Autotools ✓
  - CMake ✓
- Remove moc ✓
- Use native C++ atomics ✓
- Signal / Slot delivery as a separate library ✓
- Containers
  - leverage C++ STL containers
  - extend the CS api functionality ✓
  - document container semantics
- Use native C++ smart pointers ✓
- Refactor QString

Introduction to Signals

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

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

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

```
// signal & slot method ptr
template<class Sender,
         class SignalClass, class ...SignalArgs,
         class Receiver,
         class SlotClass, class ...SlotArgs, class SlotReturn>
bool connect(const Sender &sender,
             void (SignalClass::*signalMethod)(SignalArgs...),
             const Receiver &receiver,
             SlotReturn (SlotClass::*slotMethod)(SlotArgs...),
             ConnectionKind type = ConnectionKind::AutoConnection,
             bool uniqueConnection = false);


// given Sender is QPushButton, SignalClass could be QPushButton,
QAbstractButton, QWidget, or QObject
```

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
    - const reference to a SignalBase, `QPushButton`
  - signal
    - method pointer
  - receiver
    - const reference to a SlotBase, `this`
  - slot
    - method pointer or function ptr

CsSignal Library

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

CsSignal Library

- Activate function
  - activate is not part of the CsSignal API
  - this function is called from the generated signal method
  - to emit a signal simply call the signal method, you should not call the activate function directly

```
// sample generated signal method
void clicked() {
  activate(*this,
           &std::remove_reference<decltype(*this)>::type::clicked);
}
```

CsSignal Library

- Generating the signal methods is an API convenience
- Integration with CopperSpice
  - signal methods had to be generated since there are more than 1500 in CopperSpice

```
// sample generated signal method
void windowTitleChanged(const QString &title) {
  activate(*this,
           &std::remove_reference<decltype(*this)>::type::
           windowTitleChanged, title);
}
```

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(void clicked(bool status))
  SIGNAL_2(clicked, status)

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

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 CsSignal

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

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

Integrating CsSignal with CopperSpice

GUI Base Class

- QObject
  - the main base class which all GUI classes inherit from
  - Examples: 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

Qobject (Qt 4)

```
// one structure containing all connection information


typedef void (*StaticMetaCallFunction)(QObject *, QMetaObject::Call, int, void **);
struct Connection
{
  QObject *sender;
  QObject *receiver;
  StaticMetaCallFunction callFunction;
  // The next pointer for the singly-linked ConnectionList
  Connection *nextConnectionList;
  //senders linked list
  Connection *next;
  Connection **prev;
  QBasicAtomicPointer<int> argumentTypes;
  ushort method_offset;
  ushort method_relative;
  ushort connectionType : 3; // 0 == auto, 1 == direct, 2 == queued, 4 == blocking
  ~Connection();
  int method() const { return method_offset + method_relative; }
};
```

QObject (Qt 5)

```
typedef void (*StaticMetaCallFunction)(QObject *, QMetaObject::Call, int, void **);
struct Connection
{
  QObject *sender;
  QObject *receiver;
  union {
     StaticMetaCallFunction callFunction;
     QtPrivate::QSlotObjectBase *slotObj;
  };
  // The next pointer for the singly-linked ConnectionList
  Connection *nextConnectionList;
  //senders linked list
  Connection *next;
  Connection **prev;
  QAtomicPointer<const int> argumentTypes;
  QAtomicInt ref_;
  ushort method_offset;
  ushort method_relative;
  uint signal_index : 27; // In signal range (see QObjectPrivate::signalIndex())
  ushort connectionType : 3; // 0 == auto, 1 == direct, 2 == queued, 4 == blocking
  ushort isSlotObject : 1;
  ushort ownArgumentTypes : 1;
  Connection() : nextConnectionList(0), ref_(2), ownArgumentTypes(true) { }
  ~Connection();
  int method() const { Q_ASSERT(!isSlotObject); return method_offset + method_relative; }
  void ref() { ref_.ref(); }
  void deref() {
    if (!ref_.deref()) {
      Q_ASSERT(!receiver);
      delete this;
    }
  }
};
```

CsSignal Library

- class SignalBase

```
struct ConnectStruct {
  std::unique_ptr<const Internal::BentoAbstract> signalMethod;
  const SlotBase *receiver;
  std::unique_ptr<const Internal::BentoAbstract> slotMethod;
  ConnectionKind type;
};

// list of connections from my Signal to some Receiver
mutable std::vector<ConnectStruct> m_connectList;
```

CsSignal Library

- class SlotBase

```
// list of possible Senders for this Receiver
mutable std::vector<const SignalBase *> m_possibleSenders;
```

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
  - improved readability
  - destructor refactored

CopperSpice Integrated with CsSignal

- Wrote wrappers in CopperSpice to call the CsSignal library and maintain the existing API
- CopperSpice calls connect(), disconnect(), and activate() which are now in CsSignal
- A class in a CopperSpice application can inherit directly from SignalBase

Results of Refactoring

- Other ways we leveraged the changes made by refactoring CopperSpice, shrinking QObject, and adding our new CsSignal library

  - ran Clang Thread Sanitizer
  - hidden issues in other libraries like Webkit and Networking surfaced

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

Registration

Registration in CopperSpice

- CopperSpice allows strings to be used to identify the signal or slot method
- Allowing string names requires a mechanism to look up the name at run time to retrieve a method pointer
- In CopperSpice, the method 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 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

- Signals / Slots are scattered in a class definition with a random number in any given class
- 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 is based on a data type

```
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

```
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>{} );
}
```

Registration Costs

- compile time
  - improved static checking
- program start up
  - dynamic linking
    - relocations
      - not a good benchmark
      - most methods, template instantiations
  - static initialization
    - optimized out
- run time
  - activate<T> can be optimized

Challenges with CopperSpice

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

- Benefits of the CS 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) {
    Q_ASSERT(staticMetaObject.cast(_o));
    QPushButton *_t = static_cast<QPushButton *>(_o);
    Q_UNUSED(_t)
    switch (_id) {
    case 0: _t->clicked((*reinterpret_cast< bool(*)>(_a[1]))); break;
    default: ;
    }

// continued . . .
```

Sample Moc Code

```
  } else if (_c == QMetaObject::IndexOfMethod) {
    int *result = reinterpret_cast<int *>(_a[0]);
    void **func = reinterpret_cast<void **>(_a[1]);
    {
      typedef void (QPushButton::*_t)(bool );
      if (*reinterpret_cast<_t *>(func) ==
              static_cast<_t>(&QPushButton::clicked)) {
          *result = 0;
          return;
      }
    }
  }
}
```

Wrap up

Current Advantages of CopperSpice

- Uses CMake or Autotools
- Template classes can inherit from QObject
- Compound data types are supported
- Signal activation does not lose type information
- Signals and Slots refactored
- Obsolete source code removed
- Uses modern C++
- Atomics improved
- Improved API documentation

KitchenSink Application

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

Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- CsSignal Library
  - standalone thread aware signal / slot library
- LibGuarded
  - standalone multithreading library for shared data

Libraries & Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CS libraries

- DoxyPress & DoxyPressApp
  - an application for generating documentation

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?