CopperSpice:
A Pure C++ GUI Library

Barbara Geller & Ansel Sermersheim

CPPCon - September 2015

Introduction

- What is CopperSpice
  - Why we developed CopperSpice
  - Drawbacks of Qt
  - Advantages of CopperSpice
- CopperSpice Internals
  - Implementing Reflection in C++11
  - Signals & Slots
- Future plans for CopperSpice
  - Developers & Users
  - Where is CopperSpice headed

Why we Developed CopperSpice

- Many C++ applications need a GUI
- Started using Qt 4 libraries in 2009

- Nokia bought Qt from TrollTech (June 2008)
- Nokia major reorganization (Feb 2011)
- Nokia sold Qt licensing to Digia (March 2011)
- Digia acquires Qt from Nokia (Sept 2012)
- Qt 5.0 initial release Dec 2012
- Qt 5.6 LTS estimated release Dec 2015

Why we Developed CopperSpice

- Contributing to Qt Development
  - CLA concerns - Qt Company can use your Open Source contributions for their closed source product
  - Summit Conferences have been invitation only
  - Qt Company develops both the Closed & Open Source versions of Qt
- Qt 4 support ending Dec 2015
  - support may be available by paid contract only

Drawbacks of Qt 4 and Qt 5

- Meta Object Compiler (moc)
  - moc runs on your .h files and produces .cpp files which are compiled and linked into your application
  - moc is a code generator

  - no native support in standard build systems
  - compound types like QMap do not work in Properties
  - typedefs do not work
  - code moc generates is mostly string tables
  - does not support templates
  - every passed parameter is cast to a void *

Building Qt 4 or Qt 5

- Moc
  - must be built before building QtCore
- Bootstrap Library
  - bootstrap is a library used when building moc
  - same source used for bootstrap and QtCore
  - #ifdef used to decipher if building bootstrap or QtCore
- QMake (build system)
  - requires “bootstrap” version of the QtCore library
  - required to build Qt

What is CopperSpice

- On May 21 2012 we forked Qt 4.8
- GNU Autotools Build System
- CMake Build System is under development

- CopperSpice is written in pure C++11
- CS can be linked directly into any C++ application

- Qt Meta Object Compiler (moc) is obsolete and is not required when building CopperSpice or your C++ applications

Advantages of CopperSpice

- Template classes can inherit from QObject
- Compound data types are supported
- Your application can use any build system
- Container library improvements
- Obsolete source code removed
- Signal activation does not lose type information
- Improved API documentation

- CopperSpice is not Qt 4, it is better
- CopperSpice is not Qt 5, it is better

CopperSpice Libraries

- CsCore
- CsGui
- CsNetwork
- CsOpenGL
- CsSql
- CsXml
- CsWebKit
- Phonon
- And more. . .

Libraries begin with Cs, classes still use Q for API compatibility

Moving from Qt to CopperSpice

- PepperMill Utility
  - we used PM to convert the Qt library header files
  - Qt syntax was changed to CS syntax
  - you can modify your Qt header files by hand or use PepperMill to automate the process
  - PepperMill is only used one time

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

CopperSpice Supported Compilers

- GCC 4.7.2 or greater
  - tested on gcc 4.7.2, 4.8, 4.9, 5.1
- Windows - MinGW 32 bit and 64 bit
  - numerous versions of MinGW exist
  - links for MinGW located on our website
- Clang 3.4 or greater
  - tested on clang 3.4, 3.5, 3.6

CopperSpice Unsupported Compilers (For Now)

- MSVC 15
  - missing expression SFINAE
  - partial support for initializer lists
  - limited support for constexpr

- MSVC 13
  - no support for expression SFINAE
  - no support for initializer lists
  - no support for constexpr

We value MSVC and will continue to monitor their progress.

- What problem did moc solve?

- Moc solved the problem that C++ does not implement or natively support Reflection
  - ISO C++ study group for Reflection exists
  - very unlikely Reflection will be added in C++17

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

Where is Reflection Used

- Signals
- Slots
- Properties
- Enums
- Invokable Constructors
- UI Designer
- ...

What are Signals and Slots

- Signal
  - notification something occurred
- Slot
  - an ordinary method
- Connection
  - associates a Signal with a Slot
  - when the Signal is emitted the Slot is called
  - a given Signal can be connected to multiple Slots

What are Signals and Slots

- Boost Signals
  - each signal is an object
  - adding or removing a signal breaks ABI
  - slots are called only on the emitting thread

How is a Signal Processed

- User presses a mouse button
- Mouse button event is processed
- Signal QPushButton::clicked() is emitted

- Qt 4 or Qt 5 QPushButton::clicked()
  - method is generated by moc, stored in a string table
  - all passed parameters are cast to void *
  - activate() is called with an array of void *
- CopperSpice QPushButton::clicked()
  - method is created by a macro
  - full parameter list with complete data types
  - activate<Args…>() is called

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: ;
     }
  }
  // ...
}
```

Reflection in CopperSpice

- Signal & Slot meta data must be registered
- 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

Why is Registration Required

- When a Signal / Slot connection is made, you can specify either method by name
- connectSlotsByName()
  - called by generated code from the UI Designer
  - automatically connects Signals with Slots
- Plugins used in the UI Designer
- Query any child of QObject for a list of methods or properties belonging to the object

Declarations in your .h File

```
// signal & slot declarations
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

```
// 3 different ways to make the same connection
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...>()
  - 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

Techniques used to Implement Reflection

- Slot macro
  - `CS_SLOT_1(Public, void showHelp())`
  - `CS_SLOT_2(showHelp)`

- counter is used to “chain” methods which register the actual Slot meta data
- template class wraps an integer value
- method overloading
- constexpr
- decltype

Our Goal

- cs_register() will do something and then call the "next cs_register" method

```
cs_register(0) {
   cs_register(1);
}

cs_register(1) {
   cs_register(2);
}
```

Implementation

- “zero” and “one” are integer values
- method overloading is based on data types
- create a class template to wrap the int value

```
cs_register(0) {
   cs_register(1);
}
```

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_regTrigger(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   CS_TOKENPASTE2(value_, __LINE__)
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   CS_TOKENPASTE2(value_, __LINE__)
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 ?
```

Macro SLOT Expansion (after pre-processing)

```
// macro expansion from line 42   CS_TOKENPASTE2(value_, __LINE__)
void showHelp();

static constexpr const int value_42 =
    decltype(cs_counter(CSInt<255>{}))::value;

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

static void cs_regTrigger(CSInt<value_42>)
{
  cs_class::staticMetaObject().register_method("showHelp",
    &cs_class::showHelp, QMetaMethod::Slot, "void showHelp()",
    QMetaMethod::Public);

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

Challenges with CopperSpice

- Registration process
  - Signals, Slots, Properties, and Invokable methods
- Store method pointer for Signal & Slot methods
- Obtaining the values of an Enum

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

CopperSpice Work in Progress

- CS Container Classes
  - thin wrappers around the STL C++11 containers
  - we will maintain CS API

- Benefits
  - reverse iterators, which have been missing
  - QList has performance issues and the Qt dev team recommends avoiding this container
  - difficult to avoid QList since it is the return type for many numerous methods
  - many of the containers have exception safety problems

Future Plans for CopperSpice

- Use the C++11 threading library
- Back port additional classes from Qt 5
- Add support for smart pointers
- Optimize QVariant
- Investigate switching from WebKit to Chromium
- Android support
- Stand alone library containing Signals & Slots
- Add cmake / ninja to our CI system

How to contribute to CopperSpice

- Developers
  - we welcome C++ enthusiasts who would like to contribute to CopperSpice
  - help us improve the documentation

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

KitchenSink Application

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

Libraries & Applications

- CopperSpice
  - Libraries for developing GUI applications
- PepperMill
  - Converts old headers to CS standard C++ header files
- KitchenSink
  - Over 30 CopperSpice demos in one application
- Diamond
  - Programmers Editor which uses the CopperSpice libraries
- DoxyPress & DoxyPressApp
  - Documentation program, works with C++11

Where to find CopperSpice

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

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

- Questions? Comments?