Domain Specific Debugging Tools

Volker Krause
volker.krause@kdab.com

KDAB
What's the Problem?
Invalid read of size 1
at 0x4C2D9B0: bcmp (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x6B33AA6: QOpenGLFunctions::glLinkProgram(unsigned int) (qopenglfunctions.h:1098)
by 0x6B2FD15: QOpenGLShaderProgram::link() (qopenglshaderprogram.cpp:826)
by 0x4F83AE1: QSGDefaultDistanceFieldGlyphCache::createBlitProgram() (qsgdefaultdistancefieldglyphcache_p.h:118)
by 0x4F82D50: QSGDefaultDistanceFieldGlyphCache::resizeTexture(QSGDefaultDistanceFieldGlyphCache::TextureInfo*, int, int)
by 0x4F8262A: QSGDefaultDistanceFieldGlyphCache::storeGlyphs(QHash<unsigned int, QImage> const&)
by 0x4F77A81: QSGDistanceFieldGlyphCache::update() (qsgadaptationlayer.cpp:169)
by 0x4F86449: QSGDistanceFieldGlyphNode::preprocess() (qsgdistancefieldglyphnode.cpp:167)
by 0x4F690E8: QSGRenderer::preprocess() (qsgrenderer.cpp:378)
by 0x4F68A06: QSGRenderer::renderScene(QSGBindable const&) (qsgrenderer.cpp:229)
by 0x4F68975: QSGRenderer::renderScene() (qsgrenderer.cpp:229)
by 0x4F7B48E: QSGContext::renderNextFrame(QSGRenderer*, unsigned int) (qsgcontext.cpp:270)
by 0x4FBE833: QQuickWindowPrivate::renderSceneGraph(QSize const&) (qquickwindow.cpp:346)
by 0x50D0217: QQuickTrivialWindowManager::renderWindow(QQuickWindow*) (qquickwindowmanager.cpp:263)
by 0x50D076F: QQuickTrivialWindowManager::event(QEvent*) (qquickwindowmanager.cpp:351)
by 0x5B37DB7: QApplicationPrivate::notify_helper(QObject*, QEvent*) (qapplication.cpp:3619)
by 0x5B354A9: QApplication::notify(QObject*, QEvent*) (qapplication.cpp:3050)
by 0x79B9479: QCoreApplication::notifyInternal(QObject*, QEvent*) (qcoreapplication.cpp:748)
by 0x79BCC86: QCoreApplication::sendEvent(QObject*, QEvent*) (in /home/vkrause/dev/qt5/inst/lib/libQtCore.so.5.0.0)
by 0x79BA53D: QCoreApplicationPrivate::sendPostedEvents(QObject*, int, QThreadData*) (qcoreapplication.cpp:1349)
by 0x79BA0FE: QCoreApplication::sendPostedEvents(QObject*, int) (qcoreapplication.cpp:1209)
Address 0x7fa847b93a81 is not stack'd, malloc'd or (recently) free'd
GDB is not enough

- Increasing abstraction
- Asynchronous API
- Distributed architecture
- Runtime interpreted code
- JIT compilers
General Purpose Tools

• Instruction-level debuggers
• printf
• Profilers measuring:
  • CPU ticks
  • malloc calls
Increasing abstraction

• Examples:
  • Qt Model/View
  • QStateMachine

• Instruction-level view is too far below semantics
• Debug output triggered too often
Asynchronous API

- Examples:
  - QNetworkAccessManager/QNetworkReply
  - Job-based APIs
- Hard to follow control flow
Distributed Architecture

- Examples:
  - D-Bus

- Even harder to follow control flow
- Profilers don't analyze complexity in IPC protocol
• Examples:
  • QtQuick
  • QtWebKit

• Debuggers and profilers analyze interpreter code

• Hard to correlate issues in interpreter to issues in interpreted code
JIT-Compiled Code

- Examples:
  - QtScript
  - QtQuick
- Debuggers and profilers see generated code
- Even harder to correlate issues in generated code to issues in original QML/JavaScript.
General Purpose Tools

- Inefficient/cumbersome
- Require knowledge of framework internals
  - Up to the point of a JIT compiler!
- Can lead to erroneous conclusions
  - “JavaScript is slow!”
What can we do about it?
Can we have better tools?

- Move knowledge about framework internals to the tool
- Visualize results at the same semantic level
- Downside: Tools become specific to one framework
Where do I get these tools?

- Some tooling exist for Qt
  - cf. Romain Pokrzywka, Volker Krause, “Effective Debugging and Profiling for Qt and Qt Quick”, Qt Dev Days 2011

- Often no tooling exist for your own frameworks
Should I build my own?

- Struggling with complex control flow
- Repeatedly adding the same debug code or printf statements
- Complex internal structures that benefit from dedicated visualization
- Performance metrics lacking correlation to the actual cost cause
Approaches

- Built-in diagnostics
- External observers
- Emulators
- IDEs
- API tracing
- Binary instrumentation
- Qt Introspection
• `QDebug()` operator<< overloads

```c++
QDebug operator <<( QDebug d,
                const MyType &myObj )
{
    return d << myObj.foo()
        << myObj.bar();
}
```

• Declare outside of namespaces

• Needs to be exported or inline if provided by a library
• Minimal increase in code size
• No runtime impact when not used
• Can be disabled completely at compile time
  • QT_NO_DEBUG
  • QT_NO_DEBUG_OUTPUT
  • QT_NO_WARNING_OUTPUT
Built-In Diagnostics

- Enable at compile time or runtime
  - preprocessor define
  - environment variable
  - config file/QSettings
  - triggered via IPC
- Typically perform extra checks or provide verbose diagnostic output
Built-in Diagnostics Examples

- Preprocessor defines
  - QIODEVICE_DEBUG, QSSL_SOCKET_DEBUG, ...
  - grep for _DEBUG
- Environment Variables
  - QT_FLUSH_PAINT
  - QDBUS_DEBUG
  - grep for getenv
• Compile-time conditional debug output

```cpp
#ifndef FOO_DEBUG
  # define myDebug qDebug
#else
  # define myDebug if (false) qDebug
#endif

... myDebug(“printf style\n”);
myDebug() << “stream style”;
```
• Runtime conditional output

    static const int debugLevel = qgetenv("MY_DEBUG").toInt();
    ...
    if (debugLevel > 3)
        dumpInternalState();

• Requires application restart to activate
• D-Bus triggered diagnostics

```cpp
class MyClass : public QObject {
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", "com.kdab.debug")
public:
    MyClass()
    {
        QDBusConnection::sessionBus().
            registerService("com.kdab.MyApp");
        QDBusConnection::sessionBus().
            registerObject("/Debug", this,
                QDBusConnection::ExportScriptableSlots);
        ...
    }
public slots:
    Q_SCRIPTABLE void dumpInternalState() const
    {
        ...
    }
}
```
• Compile-time diagnostics
  • Can be disabled completely
  • Ideal for very expensive features
• Runtime diagnostics
  • Minimal runtime overhead
  • Diagnostics always available
• Not built into framework, but provided separately
• Has no access to framework internals
• Example: ModelTest
• Useful for non-trivial diagnostics performed using official API
External Observers

- Tools using public interfaces to observe what your application is doing
- Requires communication or other externally visible effects
- Example: qdbusviewer
- Also useful (but not Qt-specific):
  - Network sniffer
  - Database logging/viewers
• No changes required in your application
• Don't require application restart but can be used on-demand
• Requires interceptable communication channels
  • Problematic with e.g. TLS/SSL
• Example for DIY project: QDataStream viewer
Emulators

- Simulate the real environment your application runs in
- Makes you independent of hardware or physical constraints
- Example: qvfb
Emulators

- Allows replay of recorded input
- Allows easy testing of corner cases and “that should never happen” conditions
- Very useful for CI systems
• Find the right interface
  • API-compatible drop-in replacement DLL
  • Using existing backend abstractions (e.g. QtSensors)
  • IPC or network protocols

• Feed data
  • manually, with custom UI
  • manually, from code
  • from previously recorded file
• Fully integrated suite for the entire development workflow, including debugging and profiling
• Example: QtCreator for QML
• Usually overkill, but worth considering when providing a complex domain specific language
  • Existing IDEs (QtCreator, KDevelop, ...) can be extended by plug-ins
import QQuick 1.0

Rectangle {
  id: root
  width: 600
  height: 400
  property int pixelsize: 10
  property int nwidth: width / pixelsize
  property int nheight: height / pixelsize
  property real angle: 0.0
  Repeater {
    id: rowRepeater
    model: root.height / root.pixelsize
    Repeater {
      id: columnRepeater
      model: root.width / root.pixelsize
      property int index2: index
      Rectangle {
        property int iy: columnRepeater.index2
        property int ix: index
        property int nx: iy * (root.nwidth / 2)
        property int ny: ix * (root.nheight / 2)
        x: ix * width
        y: iy * height
        width: root.pixelsize
        height: root.pixelsize
        color: {
          var r = Math.sin(root.nwidth, root.nheight) / 2;
          if (ny*ny + nx*nx < r*r & Math.atan2(ny, nx) < Math.PI*angle & Math.atan2(ny, nx) > -Math.PI*angle & (nx*nx + (ny + r/2)*(ny + r/2)) > 1) {
            return "yellow";
          }
          return "blue";
        }
      }
    }
  }
}

GWL Printer
API Tracing

• Trace all calls (and arguments) to a specific API
• Visualization for the massive amount of data gathered

• Approach:
  • Intercept API call
  • Record call and its arguments
  • Call the original method
API Tracing Examples

- strace
  - Traces all system calls
- apitrace
  - Traces OpenGL/Direct3d calls
  - http://github.com/apitrace/apitrace
  - Qt visualization UI for OpenGL state at an arbitrary point in time
1318) `glXSwapBuffers(dpy = 0x663010, drawable = 109051937)`
API Tracing DIY

- OS-level system-wide tracing tools:
  - DTrace
  - SystemTap, perf, uprobes
- POSIX ptrace
- Library pre-loading and forwarding
  - LD_PRELOAD, dl sympt(RTLD_NEXT, ...)
  - even more ugly on Windows
API Tracing Impact

- Overhead usually comparable to one extra function call
- Be prepared to handle large amounts of data
- Requires no modifications on traced code
- Also works if no source code is available
• Interpret or JIT rewrite binary code
• Example: Valgrind suite
• Requires in-depth knowledge of binary code execution
• Allows analysis of very low-level details, e.g. for memory profiling
• Existing frameworks for binary instrumentation
  • Valgrind (http://www.valgrind.org/)
  • Pin (http://www.pintool.org/)
• Example use-case: runtime attachable Massif
Qt Introspection

- QObject Introspection
  - QMetaObject
  - signals, slots, properties, enums, object types
- Global hooks
  - object creation/destruction
  - application start
- Examples: Squish, GammaRay
• `qt_startup_hook()`
• Triggered from QCoreApplication constructor
• Allows you to run your diagnostics code early inside any Qt application
• Use event filter or object creation hooks to wait for interesting events
• Overwriting the hook is platform-specific
• `qt_[add|remove]Object(QObject*)`
• Triggered fromQObject constructor/destructor
  • Too early/late for the virtual table to be complete
  • Consider multi-threading
  • Only covers QObjects
• Powerful, but slightly dangerous.
GammaRay provides comprehensive visualization for various Qt frameworks

- http://www.kdab.com/gammaray
- Free Software (GPL)
- Introspection from start or runtime attaching
- Framework for building Qt introspection tools
Extending GammaRay

- Plug-in based
- Hides the nasty details of the Qt hooks
- Simple API
  - thread-safe object creation/destruction notifications, delayed until the virtual table exists
  - flat or hierarchical object models
  - built-in filtering by object types
• Increased complexity requires better tooling
• Time invested in tooling easily pays off
• Don't be scared about overhead
• Consider turning your repeatedly added debug output into something more reusable :-(
Questions?

Volker Krause
volker.krause@kdab.com
KDAB