Advanced, cross-platform logging for Qt

Introduction

This article presents the classic Qt logging technique and then describes a cross-platform, open source logging library called QsLog. The library is licensed under the BSD license and can be used both in commercial and Free software.

Classic logging - custom message handlers

A capable logging library is an essential aid when it comes to debugging. It's not always possible to have a debugger attached, especially when it comes to mobile devices. Sometimes the bug can't be reproduced in the release version, sometimes GDB can't read the information from the device, but a log message can be useful even after the application has been shipped and is running on the customer's hardware.

Qt doesn't include a logging library by default, however there's a simple mechanism that allows one to use a message handler function to process the output from qDebug-like function calls. A message handler is nothing more than a free function that receives a string and a message type parameter. You write `qDebug() << "Parameter out of range:" << 1.0 << "-" << 2.0` and the message handler will receive the array of char `"Parameter out of range: 1.0 – 2.0"` and the message type, which is one of debug, warning, critical and fatal. Custom handlers can be installed with the qInstallMsgHandler function.

There are quite a few code samples and tutorials available that demonstrate the message handler technique, including on Nokia Developer. This is a very popular approach, but unfortunately it has some subtle issues and is not so convenient.

Developers that have used other C++ logging libraries will quickly notice that there's no easy way to toggle what messages are shown – there is no logging level support. All the logging calls you've made are executed on each run of the application, which discourages leaving logging messages in potentially interesting, but performance sensitive spots. Furthermore, the four available message functions - debug, warning, critical and fatal - don't provide fine-grained control over the message type. All non-error messages will be grouped under debug, while most libraries offer multiple debug levels and a trace level.

Perhaps the biggest issue is that when you install a custom message handler, all messages from qDebug / qWarning and so on will be sent to your handler. This means that Qt's warnings or asserts would also end up in the handler, and that the logger itself could trigger another log call, which in turn could result either in a deadlock or an endless loop.

Finally, a message handler by itself is not thead-safe, and few if any examples take this into consideration.

Advanced logging - QsLog

Message handlers are tricky to get right, and that is precisely why QsLog was created. It was designed as a cross-platform solution that's easy to add to any application and can understand Qt's types.

QsLog works on Windows, Linux and Symbian – including the simulator. Most of the library's code is Qt code, with a few exceptions.

Adding it to a project is really easy; include the QsLog.pri file in your project file and you're ready to go. Using it is no problem either, just call one of the logging macros like this: `QLOG_INFO() << „Hello" << „logging world" << ,!';` And the really nice thing is that you can log to a file of your choice, or to the Qt Creator debugger / output pane or both. You can even create your own destinations.

Last but not least, you have access to six logging levels, and the active level can be set at runtime. You can write as many trace messages as you like in that tight loop, but they only evaluate to a compare plus a couple of simple function calls if the current logging level is higher than the level requested by the log call.

Step by step example

The header files that are of interest are QsLog and QsLogDest. The latter only has to be included when setting up the logger, QsLog.h is enough when just logging.

```
#include "QsLog.h"
#include "QsLogDest.h"
#include <QtCore/QCoreApplication>
#include <QDir>
#include <iostream>
```

The logger is a singleton object that can be accessed through its `instance()` member function. Calling `instance()` guarantees that the logger has been created. It is good practice to explicitly set the logging level, but if it is not set it defaults to InfoLevel.

```
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);

   // init the logging mechanism
   QsLogging::Logger& logger = QsLogging::Logger::instance();
```


(C) Copyright Nokia 2013. All rights reserved.

```
    logger.setLoggingLevel(QsLogging::TraceLevel);
```

Destinations are targets for the log messages. We're creating and registering two destination: the file destination outputs messages to the `log.txt` file in the application's directory, while the debug destination outputs messages in Qt Creator's output pane or debug pane.

```
    const QString sLogPath(QDir(a.applicationDirPath()).filePath("log.txt"));
    QsLogging::DestinationPtr fileDestination(
       QsLogging::DestinationFactory::MakeFileDestination(sLogPath) );
    QsLogging::DestinationPtr debugDestination(
       QsLogging::DestinationFactory::MakeDebugOutputDestination() );
    logger.addDestination(debugDestination.get());
    logger.addDestination(fileDestination.get());
```

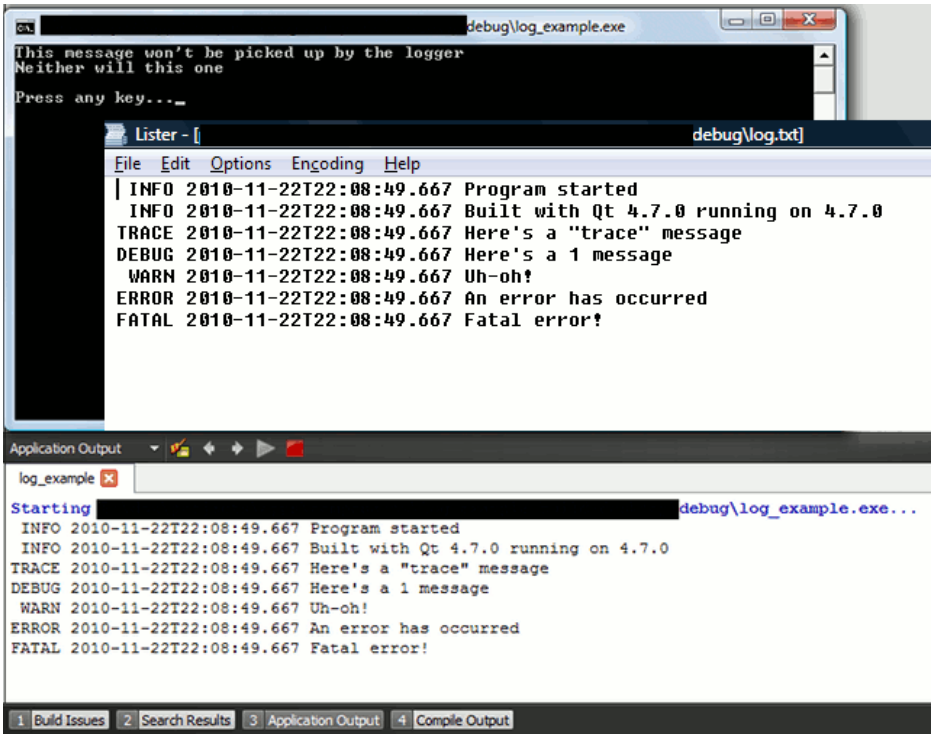
Each logging level has an associated logging macro. Macros are used to minimize the performance impact when logging is disabled.

```
    QLOG_INFO() << "Program started";
    QLOG_INFO() << "Built with Qt" << QT_VERSION_STR << "running on" << qVersion();

    QLOG_TRACE() << "Here's a" << QString("trace") << "message";
    QLOG_DEBUG() << "Here's a" << static_cast<int>(QsLogging::DebugLevel) << "message";
    QLOG_WARN()  << "Uh-oh!";
    qDebug() << "This message won't be picked up by the logger";
    QLOG_ERROR() << "An error has occurred";
    qWarning() << "Neither will this one";
    QLOG_FATAL() << "Fatal error!";

    const int ret = 0;
    std::cout << std::endl << "Press any key...";
    std::cin.get();
    QLOG_INFO() << "Program exited with return code" << ret;
    return ret;
}
```

After running the example, the log output for multiple destinations should resemble the following snapshot:



Implementation details

QsLog tries to make maximum use of the power of Qt. For instance, it uses the QDebug object for advanced formatting. This is a public object that is also used internally by the qDebug function family. By taking advantage of it, QsLog allows you to log most Qt types: QVector, QHash, QPoint – whatever you want to log – will be nicely formatted into a string.

After all the variables are transformed into a string, they are sent to the destinations registered with the logger. A destination is the target of the log call; currently there are only two destinations available – file and debugger. Destinations receive the messages in the order that they were registered with the logger. Before being sent to the destination, the message is locked behind a mutex, so you can log from multiple threads.

The complete source code for the logger can be downloaded at the QsLog bitbucket repository. Bugs and suggestions can also be submitted at the repository.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.