The Art of Logging

How to get helpful error reports

Andreas Cord-Landwehr  
cordlandwehr@kde.org
Next up...

1. Part 1: Introduction

2. Part 2: The Qt Logging Framework in KDE

3. Part 3: Log Sinks

4. The End
About me

- I am Andreas (nick: CoLa) and I am with KDE for about a decade
- In my day job, I am working on software for big agriculture vehicles

My Motivation for this Talk

- Getting a bug report without proper logs is frustrating, especially when you cannot reproduce it
- I get annoyed when I start an application and see tons of non useful (for me in a user perspective) log messages
- Time is valuable and you should not waste it by spending too much time on reading logs to analyze a problem
- ... and I think that doing logging correctly is easy :D
About me

- I am Andreas (nick: CoLa) and I am with KDE for about a decade
- In my day job, I am working on software for big agriculture vehicles

My Motivation for this Talk

- Getting a bug report without proper logs is frustrating, especially when you cannot reproduce it
- I get annoyed when I start an application and see tons of non useful (for me in a user perspective) log messages
- Time is valuable and you should not waste it by spending too much time on reading logs to analyze a problem
- ... and I think that doing logging correctly is easy :D
Developer Logs

Log messages are text lines that are printed by the a program when something “meaningful” happens

Examples:

- something happened that should not happen
- output the helps to follow program logic and where a developer can analyze if steps fit to his thinking
- tracking of user interactions in order to make an error analysis possible
- tracing points (mostly out of scope here)

Usually, logs are printed to stdout – at the end of the talk I will show alternatives
Log Message Severity Levels

**Fatal** Something so critical happened that you only print a log message and then die.

**Critical** Something bad happened and an operation could not be performed; possibly with data loss.

**Warning** Something unexpected/unwanted happened but the program can resume and handle this situation.

**Info** A low frequent change in the programs business state done by user or by data.

**Debug** Messages that print information for “interesting” program locations and help you to see what happened.

**Trace** High frequent debug messages that can be activated to deeply analyse specific operations.
Log Message Severity Levels

**Fatal** Something so critical happened that you only print a log message and then die.

**Critical** Something bad happened and an operation could not be performed; possibly with data loss.

**Warning** Something unexpected/unwanted happened but the program can resume and handle this situation.

**Info** A low frequent change in the programs business state done by user or by data.

**Debug** Messages that print information for “interesting” program locations and help you to see what happened.

**Trace** High frequent debug messages that can be activated to deeply analyse specific operations.
Log Message Severity Levels

**Fatal**  Something so critical happened that you only print a log message and then die.

**Critical**  Something bad happened and an operation could not be performed; possibly with data loss.

**Warning**  Something unexpected/unwanted happened but the program can resume and handle this situation.

**Info**  A low frequent change in the programs business state done by user or by data.

**Debug**  Messages that print information for “interesting” program locations and help you to see what happened.

**Trace**  High frequent debug messages that can be activated to deeply analyse specific operations.
Log Message Severity Levels

**Fatal**  Something so critical happened that you only print a log message and then die.

**Critical**  Something bad happened and an operation could not be performed; possibly with data loss.

**Warning**  Something unexpected/unwanted happened but the program can resume and handle this situation.

**Info**  A low frequent change in the programs business state done by user or by data.

**Debug**  Messages that print information for “interesting” program locations and help you to see what happened.

**Trace**  High frequent debug messages that can be activated to deeply analyse specific operations.
Log Message Severity Levels

**Fatal**  Something so critical happened that you only print a log message and then die.

**Critical**  Something bad happened and an operation could not be performed; possibly with data loss.

**Warning**  Something unexpected/unwanted happened but the program can resume and handle this situation.

**Info**  A low frequent change in the programs business state done by user or by data.

**Debug**  Messages that print information for “interesting” program locations and help you to see what happened.

**Trace**  High frequent debug messages that can be activated to deeply analyse specific operations.
Log Message Severity Levels

- **Fatal**: Something so critical happened that you only print a log message and then die.
- **Critical**: Something bad happened and an operation could not be performed; possibly with data loss.
- **Warning**: Something unexpected/unwanted happened but the program can resume and handle this situation.
- **Info**: A low frequent change in the programs business state done by user or by data.
- **Debug**: Messages that print information for “interesting” program locations and help you to see what happened.
- **Trace**: High frequent debug messages that can be activated to deeply analyse specific operations.
Next up...

1. Part 1: Introduction
2. Part 2: The Qt Logging Framework in KDE
3. Part 3: Log Sinks
4. The End
QDebug
Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>
qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1,2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
    << "Akademy" << year;
// Akademy2021
qDebug() << Qt::hex
    << Qt::uppercasedigits << year;
// 7E5
```

- **log severity**: `qDebug, qWarning, qInfo, qCritical, qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

**QDebug**

Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>
qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1,2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
    << "Akademy" << year;
// Akademy2021
qDebug() << Qt::hex
    << Qt::uppercasedigits << year;
// 7E5
```

- **log severity**: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- **support for many Qt data types with pretty printing**
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

QDebug
Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>
qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1,2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
    << "Akademy" << year;
// Akademy2021
qDebug() << Qt::hex
    << Qt::uppercasedigits << year;
// 7E5
```

- **log severity**: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- **support for many Qt data types with pretty printing**
- you can also register your own debug printers for your data type
- **simple output assembly**
- `QDebugStateSaver` for resetting stream format state

QDebug Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>
qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1,2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
    << "Akademy" << year;
// Akademy2021
qDebug() << Qt::hex
    << Qt::uppercasedigits << year;
// 7E5
```

- **log severity**: qDebug, qWarning, qInfo, qCritical, qFatal
- **support for many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- **simple output assembly**

"QDebugStateSaver for resetting stream format state"

Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>
qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1,2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
// Akademy2021
qDebug() << Qt::hex
<< Qt::uppercasedigits << year;
// 7E5
```

- **log severity:** `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- **support for many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- **simple output assembly**
- `QDebugStateSaver` for resetting stream format state

QDebug
Stream-API (there is also printf API, but I will only use stream API in the following)

```cpp
#include <QDebug>

qDebug() << "welcome!";
// welcome!
qWarning() << QDate::currentDate();
// QDate("2021-06-20")
qCritical() << QRect(1, 2, 3, 4);
// QRect(1, 2 3x4)
int year = 2021;
qDebug() << "Akademy" << year;
// Akademy 2021
qDebug().nospace().noquote()
    << "Akademy" << year;
// Akademy2021
qDebug() << Qt::hex
    << Qt::uppercasedigits << year;
// 7E5
```

- **log severity**: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- **support for many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- **simple output assembly**
- **QDebugStateSaver** for resetting stream format state

Categorized Logs
The better way

Group log messages by hierarchical categories:

category.subcategory.subsubcategory.[...]

Log outputs can be enabled/disabled at runtime by category and message type.

Define Global Logging Categories

Q_DECLARE_LOGGING_CATEGORY(name) // declare it in header

// define category and make it usable with string identifier
Q_LOGGING_CATEGORY(name, string)
// optional msgType sets enabled minimal severity
Q_LOGGING_CATEGORY(name, string, msgType)
Categorized Logs

The better way

Group log messages by hierarchical categories:

category.subcategory.subsubcategory.[...]

Log outputs can be enabled/disabled at runtime by category and message type.

Define Global Logging Categories

Q_DECLARE_LOGGING_CATEGORY(name) // declare it in header

// define category and make it usable with string identifier
Q_LOGGING_CATEGORY(name, string)
// optional msgType sets enabled minimal severity
Q_LOGGING_CATEGORY(name, string, msgType)
Categorized Logs

Example

loggingcategories.h

```cpp
#pragma once
#include <QLoggingCategory>
Q_DECLARE_LOGGING_CATEGORY(mycoolcategory);
```

loggingcategories.cpp

```cpp
#include "loggingcategories.h"
Q_LOGGING_CATEGORY(mycoolcategory, "myapp.foo", QtWarningMsg);
```

Just using it:

```cpp
#include "loggingcategories.h"
qCDebug(mycoolcategory) << "log message in category myapp.foo";
```
Log Filtering

Assume we have an app `myapp` with logging categories:

1. `<QLibraryInfo::DataPath>/qtlogging.ini`, `[Rules]` section
2. `.config/QtProject/qtlogging.ini`, `[Rules]` section
3. `setFilterRules(const QString &rules)`
4. `[Rules]` section of file set in `QT_LOGGING_CONF`
5. `QT_LOGGING_RULES` environment variable

Output filtering with `QT_LOGGING_RULES` variable:

1. `QT_LOGGING_RULES=*=true ./myapp` # we see everything
2. `QT_LOGGING_RULES=myapp.lib=true,myapp.backend=false ./myapp` # only lib, no backend
Logging Rules Format

category[.type] = true|false

1 * = true
2 kate. *= false
3 kwrite.debug = true
4 *.critical = true
5 *.strangethings. *= true

Rule Formats
- type can be any of: debug, info, warning, critical
- * wildcard only allowed as first and/or last character
- rules evaluated from first to last
User Friendly Log Filtering

Use `kdebugsettings` to edit `./config/QtProject/qtlogging.ini`:

![Log Filtering Interface](image)

**Note:** To allow configuration, you have to install the logging category names though! (/usr/share/qlogging-categories5/)
Using Extra-CMake-Modules

api.kde.org/ecm/module/ECMQtDeclareLoggingCategory.html

1 ecm_qt_declare_logging_category( MYPROJECT_SRCS
2     HEADER "myproject_debug.h"
3     IDENTIFIER "MYPROJECT_DEBUG"
4     CATEGORY_NAME "myproject"
5     OLD CATEGORY NAMES "myprojectlog"
6     DESCRIPTION "My project"
7     EXPORT MyProject
8 )
9 ecm_qt_export_logging_category( IDENTIFIER "MYPROJECT_SUBMODULE_DEBUG"
10     CATEGORY_NAME "myproject.submodule"
11     DESCRIPTION "My project - submodule"
12     EXPORT MyProject
13 )
14 ecm_qt_install Logging_categories( EXPORT MyProject
15     FILE myproject.categories
16     DESTINATION \"${KDE_INSTALL_LOGGINGCATEGORIESDIR}\"
Next up...

1. Part 1: Introduction
2. Part 2: The Qt Logging Framework in KDE
3. Part 3: Log Sinks
4. The End
Logging Sinks
Perspective = we look at a full system with many apps

**Question:** How to store logs and serialize async log outputs?

- Qt supports several log backend integrations apart from stdout
- Backend selected at Qt configure time (e.g., Fedora enables journald)
- When enabled, such log databases can be super helpful
- ... until there are too many logs

**Note:** Per default journald forwards system session logs to syslog
journald
I like it

- Internal log rotation and file consistency checks
- Pretty robust database files with internal index handling for fast usage
- Awesome CLI tool: `journalctl`:
  
  1. `journalctl -b 1`: only last boot’s logs
  2. `journalctl -u sddm`: only log of `sddm` systemd service
  3. `journalctl -u sddm -p 1..4`: only `sddm` output up to warning
  4. ...and much more

Remarks for Integration
- for Qt only one backend must be configured at once (default = none)
- when backend is configured, you do not see console output unless:
  
  QT_FORCE_STDERR_LOGGING=1
- systemd services (thanks @Plasma team!) are nicely integrated
journald
I like it

- Internal log rotation and file consistency checks
- Pretty robust database files with internal index handling for fast usage
- Awesome CLI tool: `journalctl`:
  1. `journalctl -b 1`: only last boot's logs
  2. `journalctl -u sddm`: only log of sddm systemd service
  3. `journalctl -u sddm -p 1..4`: only sddm output up to warning
  4. ...and much more

Remarks for Integration
- for Qt only one backend must be configured at once (default = none)
- when backend is configured, you do not see console output unless:
  `{QT_FORCE_STDERR_LOGGING=1}`
- systemd services (thanks @Plasma team!) are nicely integrated
journald with colors and filters
Suggestion for nicer integration

https://invent.kde.org/libraries/kjournald

- goal: QAbstractItemModel abstraction of journald’s C-API
- allows trivial integration into applications: I am looking for a nice use case in KDE :)
- ships a reference implementation for a QtQuick based journald browser
  1. filter by boots, systemd services and severity
  2. rainbow colors for services
- my main use case right now: offline analysis of journald databases from embedded devices → could this be helpful for Plasma mobile, too?
Next up...

1. Part 1: Introduction
2. Part 2: The Qt Logging Framework in KDE
3. Part 3: Log Sinks
4. The End
Key Messages

Please take this home

- Only use categorized logging
- Add all meaningful log messages you want
- Disable verbose logs per default (set level to e.g. Warning or Info)
- For bug reports you can tell users to enable more logs
- Start `journalctl` or `kjournald-browser` and look at your own system

Skipped due to time: you can format your output messages as you like
The End

Question Time

Contact

mail: cordlandwehr@kde.org
irc: CoLa