Diagnostics

Description

This tutorial describes several aspects of diagnostic support during development and post-development phases.

Assertions

For using assertions include *FeatStd/Diagnostics/Debug.h* in your program.

Compile Time Checks

Best time to catch an error is at compile time before the program starts. If the assertion is a constant expression, then the compiler is able to check it.

Use

- FEATSTD_COMPILETIME_ASSERT(assertion)

Runtime Checks

These kinds of assertions should be used to document logically impossible situations and discover programming errors.

Use

- FEATSTD_DEBUG_ASSERT(condition)
- FEATSTD_DEBUG_FAIL()
- FEATSTD_DEBUG_REENTRANCE_GUARD()
- FEATSTD_DEBUG_BREAK()

FEATSTD_DEBUG_ASSERT() and FEATSTD_DEBUG_FAIL() allow the user to check **pre-conditions** and also **post-conditions** as well as **invariants**, sometimes known as "embedded testing".

```
bool CgiApp::LoadAsset(const Char* filePath)
{
    FEATSTD_DEBUG_ASSERT(filePath != 0);

    ClearLoadedAsset();
    m_assetConfig.AddFileRepository(filePath);
```

```
                if ((*tempName >= '0') && (*tempName <= '9')) {
                    hash += (ToUInt32(*tempName) - static_cast<UInt32>('0'));
                }
                else if ((*tempName >= 'A') && (*tempName <= 'F')) {
                    hash += (ToUInt32(*tempName) - (static_cast<UInt32>('A') - 10U));
                }
                else if ((*tempName >= 'a') && (*tempName <= 'f')) {
                    hash += (ToUInt32(*tempName) - (static_cast<UInt32>('a') - 10U));
                }
                else {
                    // failure
                    FEATSTD_DEBUG_FAIL();
                }
```

FEATSTD_DEBUG_REENTRANCE_GUARD() allows to check that a part of code isn't entered twice:

```
void Appender::Append(const LogEvent& logEvent)
{
    FEATSTD_DEBUG_REENTRANCE_GUARD();
```

Be aware that all FEATSTD_**DEBUG**_... macros are expanded in Debug-Versions only. This avoids codesize and performance penalty in Release-Versions.

For a distinction between ErrorHandling and Assertions please see

http://en.wikipedia.org/wiki/Assertion_(computing)

Customizing

The behavior can be adapted by implementing

- either a platform specific FeatStd::Platform::Diagnostic class
- or a **DebugControlHook** class derived from FeatStd::Diagnostics::IDebugControlHook, activated via FeatStd::Diagnostics::DebugControl::SetHook.

See UnitTests\Diagnostics\DebugControlTest.cpp and the FeatStd::UnitTestHelper::DebugOutputChecker implementation for details.

Logging

Description

Logging is an important feature while programming (debugging), but also during the post-development phase to detect the reasons for system and software crashes.

In contrast to classical "printf" debugging, the FeatStd implementation is "two-dimensionally" configurable.

- First, the framework is structured in various **realms**.
- Second, for every realm the **level of detail** (FeatStd::Diagnostics::LogLevel::Enum, Candera::Diagnostics::LogLevel::Enum) can be chosen separately.

Free configurable **appenders** decide which channel(s) should be used to output the log messages.

See FeatStd::Diagnostics::Appender (and subclasses) for details.

Configure Log Output

Default Log Output

By default, Candera produces logging output with LogLevel "Warning" for all realms on the console. This chapter explains how to adapt those default logging configuration in an application.

To configure logging support include *Log.h* in your program.

Set LogRealm and LogLevel

To modify LogLevel for one, more, or even all realms, use

- FeatStd::Diagnostics::LogControl::SetLogLevel<LogRealm>().
- The convenience function FeatStd::Diagnostics::LogControl::SetLogLevel() sets the requested level for all realms.

For a list of all LogRealms supported by FeatStd, refer to *FeatStd/Diagnostics/LogRealm.h*:

```
FEATSTD_LOG_DECLARE_REALM(FeatStdSystem);
FEATSTD_LOG_DECLARE_REALM(FeatStdPlatform);
FEATSTD_LOG_DECLARE_REALM(FeatStdIO);
FEATSTD_LOG_DECLARE_REALM(FeatStdMemoryManagement);
FEATSTD_LOG_DECLARE_REALM(FeatStdMonitor);
FEATSTD_LOG_DECLARE_REALM(FeatStdAsync);
```

Use the macro FEATSTD_LOG_SET_REALM to activate a realm (see also chapter Custom Log Output), e.g. FeatStd class FileAppender will produce logging output under the realm "FeatStdSystem", because of the following declaration:

```
class FileAppender : public Appender
{
    typedef Appender Base;
```

```
    FEATSTD_LOG_SET_REALM(Diagnostics::LogRealm::FeatStdSystem);
```

For each realm different log levels can be set, see log levels defined in FeatStd/Diagnostics/Loglevel.h.

```
enum Enum
{
    All = 0,
    Debug = All,
    Info = 1,
    Warning = 2,
    Error = 3,
    Fatal = 4,
    Off = 5
};
```

Use FeatStd::Diagnostics::LogControl::SetLogLevel<LogRealm>() to set the desired log level.

```
    LogControl::SetAppender(&CgiAppLogAppender::GetCgiAppLogAppenderInstance());
    LogControl::SetLogLevel<CgiAppLog>(LogLevel::All);
```

> The default log level is *Warning*.

Output Channel - Log Appenders

Logging output is handled by instances of FeatStd::Diagnostics::Appender - for this purpose, all FeatStd::Diagnostics::Logger instances share a list of appenders.

FeatStd provides following appenders:

- FeatStd::Diagnostics::ConsoleAppender
- FeatStd::Diagnostics::FileAppender
- FeatStd::UnitTestHelper::MemorizeAppender

By default, a ConsoleAppender is configured, means once an application doesn't configure anything regarding Logging, log messages from Candera will be sent to the console.

The list of appenders can be modified by FeatStd::Diagnostics::LogControl::SetAppender, AppendAppender and RemoveAppender.

Implement a Custom Appender

A custom appender class has to be derived from FeatStd::Diagnostics::Appender and implement the abstract method FeatStd::Diagnostics::Appender.DoAppend().

A custom **CgiAppLogAppender** has been created for modifying the LogLevel::Info output.

```
void CgiAppLogAppender::DoAppend(const LogEvent& logEvent)
{
    // Info traces will be outputted as is, for all others keep ConsoleAppender implementation
    if (logEvent.mLogLevel <= LogLevel::Info) {
        FeatStd::Internal::Diagnostic::ConsoleOut("%s\n", logEvent.mMessage);
        FeatStd::Internal::Diagnostic::DebuggerOut("%s\n", logEvent.mMessage);
    } else {
        FeatStd::Diagnostics::ConsoleAppender::DoAppend(logEvent);
    }
}
```

This custom appender is set to LogControl and activated for all log levels.

```
    LogControl::SetAppender(&CgiAppLogAppender::GetCgiAppLogAppenderInstance());
    LogControl::SetLogLevel<CgiAppLog>(LogLevel::All);
```

Custom Log Output

Player Logging Support

This chapter explains how to create logging output from a custom application.

Include *Log.h* in your program.

Default Application Realm

FeatStd predefines a default application realm:

```
FEATSTD_LOG_DECLARE_REALM(User);
```

To activate this realm for an application, use the macro FEATSTD_LOG_SET_REALM() with the default application realm as parameter:

```
FEATSTD_LOG_SET_REALM(LogRealm::User);
```

Produce Logging Output

Output itself is done by calling level specific **FEATSTD_LOG_<level>** macros in your code:

- FEATSTD_LOG_DEBUG(...)
- FEATSTD_LOG_INFO(...)
- FEATSTD_LOG_WARN(...)
- FEATSTD_LOG_ERROR(...)
- FEATSTD_LOG_FATAL(...)

FEATSTD_LOG_<level> macros provide a **printf-like** interface, which means that a format string has to be given as first argument, followed by a variable number of arguments (referenced in the format string).

FEATSTD_LOG_INFO() example when loading a Scene.

```
bool CgiApp::LoadScene(const Char* sceneName) {
    FEATSTD_LOG_INFO(" -> Loading scene '%s'", sceneName);
```

FEATSTD_LOG_ERROR() example when loading a asset configuration fails.

```
    bool result =  LoadAssetConfig();
    if (!result) {
        FEATSTD_LOG_ERROR("\n%s\n", m_loadError);
        m_assetConfig.ClearRepositoryList();
    }
```

> The length of the assembled information **must be** less than 511 characters!

Define Custom Realms

For simple applications the usage of the predefined realm **LogRealm::User** may suffice but more sophisticated applications possibly want to structure the output by further realms.

To do so, first define a LogRealm class by using the FEATSTD_LOG_DECLARE_REALM(name_of_the_realm) macro.

```
#include <FeatStd/FeatStd.h>

namespace CgiApplication {

FEATSTD_LOG_DECLARE_REALM(CgiAppLog);
```

```
FEATSTD_LOG_DECLARE_REALM(TutorialWidgets);
```

See LightPlayer/CgiAppLogRealm.[h|cpp] for details.

> The new realm doesn't take part in any iteration functionality (e.g. FeatStd::Diagnostics::LogControl::SetLogLevel()) until the realm (Logger) is used a first time (instantiated).

So it is good practice to implement an initialization function (using FEATSTD_LOG_DEFINE_REALM(name_of_the_realm))

```
void CgiApp_LoggerInitialization()
{
    FEATSTD_LOG_DEFINE_REALM(CgiAppLog);
    FEATSTD_LOG_DEFINE_REALM(TutorialWidgets);
}
```

and to call it at an early point in program execution.

Finally, the new custom realms must be set for the application, which shall log on that realm:

```
FEATSTD_LOG_SET_REALM(CgiAppLog);
```

Use an appropriate **FEATSTD_LOG_<level>** macro to output the message.

Revision #7
Created 2023-02-27 07:23:54 UTC by Tsuyoshi.Kato
Updated 2025-01-24 06:17:06 UTC by Elisabeth Zauner