C/C++

Tweet Permalink

A Highly Configurable Logging Framework In C++

By Michael Schulze, June 18, 2010

Logging is both a crucial technique during development and a great tool for investigating problems which arise while running software on a client's system. One of the major benefits of logging is that users of software can provide helpful information to software maintainers by simply delivering the logged data — a job for which they don't need to know anything about programming languages, debuggers, and the like.

Of course, there are numerous implementations of logging, which begs the questions "So why do we need yet another logging utility?" One reason is that many logging tools suffer from drawbacks of one kind or another:

- Logging can consume a lot of resources even if it is fully disabled in release builds. Log messages are still part of the build and in case of special debug messages (e.g., for a proprietary algorithm) it enables reverse-engineering of your code. Moreover, the logging framework can be ever present, due to disabling logging only at runtime
- Preprocessor implemented logging often has side effects. For instance, if you want to call a function always and for debugging purpose you print its return value with the help of a logging macro like `LOGGING(ret=f())` the function is only called if logging is active. If the macro is empty defined to disable logging, the function will never be called. Such Heisenbugs are hard to find.
- Adding new features like different back-ends (sinks) or customized prefixes on logging messages requires careful study.

So what are the feature requirements that a basic logging framework should have?

1. Easy to use (maybe with the typical behavior of operator<<).
2. Easy to extend
3. Portable
4. Type-safe
5. Possible to enable/disable certain parts both at compile time and at runtime, depending on the configuration.
6. Possible to be disabled at compile time, leading to zero overhead at runtime. Thus, neither code nor memory is wasted
7. Efficient and economical in terms of resource so that it can be used in resource-constrained environments (like microcontrollers).
8. Able to avoid macro-related pitfalls in logging statements

In this article I present a flexible, highly configurable, and easily customizable logging framework that uses standard C++ in meeting these requirements The complete source code and related files are available here.

The following code snippet illustrates a typical use case:

```
log::emit() << "Hello World!" << log::endl;
```

This example outputs "Hello World! with linefeed and is easy to use (Requirement #1). Can you also see similarities to `std::cout`? I use `log` instead of `std` and `emit()` instead of `cout`. `emit()` is a function returning "something", not an object like `cout`. However, it is usable with `operator<<()` for outputting logging data. Furthermore, no macro is used at the statement level (Requirement #8).

A Highly Configurable Logging Framework In C++

By Michael Schulze, June 18, 2010

Behind the Scenes

The architecture of the framework has three parts:

- Front-end
- Core
- Back-end

You can configure and adapt each part, and the configuration determines what the framework does. My implementation uses C++ features like function overloading, free operator functions, templates, and template-specialization to name a few only, and all of the logging framework functionality is encapsulated in a separate "logging" namespace, avoiding name clashes.

The front-end is the top of the framework, presenting the API. The `log` structure that provides the `emit()` functions and two enumerations is shown in Listing One.

Listing One

```
struct log {

    template<typename Level>
    static inline
    typename Logger<Level>::return_type& emit () {
        return Logger<Level>::logging() << Level::level() << Level::desc();
    }

    static inline
    Logger<>::return_type& emit () {
        return Logger<>::logging() << Void::level();
    }

    enum Numerative {
        bin  =  2,   ///< switch to binary output
        oct  =  8,   ///< switch to octal output
        dec  = 10,   ///< switch to decimal output
        hex  = 16    ///< switch to hexadecimal output
    };

    enum Manipulator {
        tab  =  '\t',   ///< prints a tabulator to the output
        endl =  '\n'    ///< adds a line feed to the output
    };
};
```

All of the logging levels are not shown. However, I do present the definition of `Error` as an example that's typical of all logging levels:

```
struct Error {
    static Level::levels level () {
        return Level::error;
    }

    static const char * desc() {
        return "[ ERROR ]";
    }
};
```

Again, each log statement starts with the `emit()` function. `emit()` exists in two flavors — a general form and one with a template-parameter, giving the logging level (see Listing One). The template version of the `emit()` function is used this way:

```
log::emit< Error >()<<Logging an Error<< log::endl;
```

The code outputs "[ERROR] Logging an Error" and sets the logging level to Error for this statement. The return type of `emit()`, directly usable with `operator<<()` for outputting logging data, is determined by the `Logger` component (part of the core). The return type depends first on the logging level and second, on a default template parameter of the `Logger` component (not visible here; see Listing Two). The configuration of this type is key to meeting Requirements #2, #3, and #6 because, depending on how you deploy it, different behaviors are realized. Furthermore, configuring the return type can let you enhance the framework functionality (for example with coloration, as I'll show shortly). If you look carefully at Listing One, you see, that `endl`, `tab`, and so on are defined within an enumeration — not as manipulator functions. If I would use manipulator functions in the same way as in `std::cout`, then code generation always occurs because it implies taking the address of the function. Because of Requirement #6, I have avoided manipulator functions in the general design. However, users have the possibility to write manipulator functions, and the core is able to execute manipulators.

Most of the core is not interesting from the concepts point of view, because there are a lot of housekeeping code (e.g., logging level treatment) and code responsible for converting numbers to a given base, pointers, etc. to character streams. However, the `Logger` component is important because of its job for determining and handling the correct return type — the arcane template argument `loggingReturnType` (Listing Two).

Listing Two

```
struct loggingReturnType;

template<typename Level = ::logging::Void, typename R = loggingReturnType>
class Logger {
    public:
        typedef R return_type;

        static return_type& logging () {
            return Obj<typename return_type::output_base_type, int>::obj();
        }

    private:
        template <typename log_t, typename T>
        struct Obj {
            static return_type& obj () {
                typedef singleton<return_type> log_output;
                return log_output::instance();
            }
        };

        template <typename T>
        struct Obj<NullOutput, T> {
            static return_type& obj () {
                return *reinterpret_cast<return_type*>(0x0);
            }
        };
};
```

`loggingReturnType` is forward declared to allow including the framework and later defining the type by the user of the framework. Enabling this needs an indirection level via an additional template parameter R that is the `loggingReturnType` by default. Through that indirection, it is possible to defer the type definition to a later point, but allowing the use of this not-yet-defined type within the `Logger` component. Later at `log` statements, when the template code is evaluated, `loggingReturnType` has to be defined.

In general, the `Logger` implements a compile-time selector, using the C++ feature of partial template-specialisation. The selection mechanism uses an embedded `output_base_type` definition within the `loggingReturnType` (which is ensured through its declaration) for choosing the right implementation. If the `output_base_type` is not of type `NullOutput`, a singleton of type `loggingReturnType` is constructed and returned as reference. On the reference, methods for switching levels (printing numbers or characters) can be called. If the `output_base_type` is of type `NullOutput`, a reference to an object at address zero will be returned. "What is that?" you ask. "A reference to an object that does not really exist and calling methods on it? Is this allowed?" No, calling methods on it is not legal, or like the standard says, it leads to "undefined behavior". Therefore, I do not call methods on this reference. The `NullOutput` is defined as:

```
struct NullOutput {};
```

...and it is empty, thus it is also impossible to call methods on it. However, the type, in this case the returned reference, is usable to select an implementation. Now I use an overloaded version of a free operator function.

```
template<typename T>
static inline NullOutput& operator << (NullOutput& no, T) {
  return no;
}
```

The overloaded `operator<<()` has a template parameter matching on everything, meaning it catches every call having a `NullOutput` as reference, and also allows chaining of consecutive calls. Due to its definition as static inline and through its empty implementation, the compiler can fully inline the free operator function and omit the call as well as the generation of the function itself, allowing a full deactivation of the logging framework (Requirement #6). To meet Requirement #5 — deactivation of certain parts/levels at compile-time — I use the `NullOutput` again. A full template specialization of the `Logger` component enables the desired functionality. The following code snippet disables `Info` logging level output:

```
template<>
class Logger<Info, loggingReturnType> {
    public:
        typedef NullOutput return_type;
        static return_type& logging (){
            return *reinterpret_cast<return_type*>(0x0);
        }
}
```

Such code needs not to be written by hand, but instead it is generated by a user-friendly macro (LOGGING_DISABLE_LEVEL(level)). Yes, you read it right — a macro — but this is not critical because this kind of macro is not part of a log statement within user-code, instead, it is only part of the logging framework configuration.

The used `NullOutput` is from the architecture point of view a special back-end used for deactivation, however, one will not always want to deactivate logging. Usually, you want to output something. Hence, the consideration of the lowest part, the back-end is still missing. In general, a back-end or sink defines an output device for logged data. It could be a file, a console, a serial port, or something that you imagine. For that, sinks have to provide a simple interface to be a possible back-end.

```
struct Sink {
    Sink& operator<<(const char c) {
        // implement the sink specific behavior for printing c
        return *this;
    }
};
```

A sink takes a single character and delivers a reference to itself, allowing the chaining of consecutive output actions. However, the performed action depends on the back-ends type.

Having all parts of the architecture together, one question remains: How do you configure and setup the framework to a working state? In the first instance, you have to choose what back-end you want. As an example, I show a configuration with `StdOutput` as back-end parameterized with `std::clog` stream as output medium. Furthermore, for the example, I do not want to switch logging levels on or off at runtime, so I configure `OutputLevelSwitchDisabled`. The contained `OutputStream` is responsible for handling numbers, pointers, conversions, and so on. I provide the configuration as a `typedef`.

```
typedef OutputLevelSwitchDisabled <
            OutputStream <
                StdOutput< ::std::clog>
            >
        > StdLogType;
```

The architecture is implemented as mixin-layers, thus it is easy to extend, allowing adding new functionality. (See "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs" by Y. Smaragdakis and D. Batory, ACM Transactions on Software Engineering and Methodology, vol. 11, 2002.) The defined `StdLogType` is directly usable. However, if I would do so, Requirements #5 and #6 are not able to be met because this type is not bound to the frameworks front-end; therefore, `log::emit()` in its flavors is not configured yet. To get the conjunction, I need the following macro:

```
#define LOGGING_DEFINE_OUTPUT(BASE)                                     \
    namespace logging {                                                 \
        struct loggingReturnType : public BASE {                        \
                // The provided typedef is used for compile-time        \
                // selection of different implementations of the        \
                // logging framework. Thus, it is necessary             \
                // that any output type supports this type              \
                // definition, why it is defined here.                  \
                typedef BASE    output_base_type;                       \
            };                                                          \
    }
```

that is used this way:

```
LOGGING_DEFINE_OUTPUT( StdLogType)
```

After that, the logging framework is ready for action.

Previous 1 **2** 3 4 Next

Related Reading

News
Commentary
Slideshow
Video
Most Popular

News

Tools To Build Payment-Enabled Mobile Apps

Parallels Supports Docker Apps

Did Barcode Reading Just Get Interesting?

XMind 6 Public Beta Now Available

More News»

Commentary

AppGyver AppArchitect 2.0 Appears

Abstractions For Binary Search, Part 10: Putting It All Together

Devart dbForge Studio For MySQL With Phrase Completion

Jelastic Docker Integration For Orchestrated Delivery

More Commentary»

Slideshow

Jolt Awards: The Best Testing Tools

Developer Reading List

Jolt Awards 2013: The Best Programmer Libraries

2012 Jolt Awards: Mobile Tools

More Slideshows»

Video

The Purpose of Hackathons

What Is the Future of Mobile?

Teen Computer Scientist Wins Big at ISEF

Open Source for Private Clouds

More Videos»

Most Popular

Developer Reading List: The Must-Have Books for JavaScript

A Lightweight Logger for C++

Jolt Awards 2015: Coding Tools

Finding the Median of Two Sorted Arrays Efficiently

More Popular»

More Insights

White Papers

Simplify IT With Cloud-Based Wireless Management

Challenging Some of the Myths About Static Code Analysis

More >>

Reports

Return of the Silos

Research: Federal Government Cloud Computing Survey

More >>

Webcasts

New Technologies to Optimize Mobile Financial Services

How to Create an End-to-End Enterprise Payments Hub

More >>

A Highly Configurable Logging Framework In C++

By Michael Schulze, June 18, 2010

Using the Logger With an 8-Bit Microcontroller

Enabling deployment of my logging framework in resource constrained environments (Requirement #7) needs not as much, because most of the code is portable and not system specific, meaning that only the lowest part, the back-end, has to be adapted to the envisioned environment. I show the applicability in a resource-constrained environment on an 8-Bit AVR at90can128 microcontroller from Atmel using AVR-Libc as the low-level library for register access. The struct `UART0at90can128` presents a new back-end, that does the initialization of UART0 in the constructor, and providing the usual `operator<<()`interface of a sink:

```
template<uint32_t baudrate, uint32_t CPU_FREQ>
struct UART0at90can128 {
    UART0at90can128 () {
        uint16_t value=((CPU_FREQ/8/baudrate)-1)/2;
        UBRR0H = (unsigned char) (value>>8);
        UBRR0L = (unsigned char) value;

        /* Enable UART transmitter */
        UCSR0B = ( 1 << TXEN0 );

        /* Set frame format: 8N1 */
        UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);

    }
    UART0at90can128 & operator<<(const char c) {
         /* Wait for empty transmit buffer */
         while ( !(UCSR0A & (1<<UDRE0)) );
         /* Start transmittion */
         UDR0 = c;
         return *this;
    }
};
```

Defining this is all, enabling the usage of the logging framework in this environment.

Related Reading

News
Commentary
Slideshow
Video
Most Popular

News

Java Plumbr Unlocks Threads
Did Barcode Reading Just Get Interesting?
Devart dbForge Studio For MySQL With Phrase Completion
Parasoft DevTest Shifts To

Slideshow

Jolt Awards: Mobile Development Tools
Developer Reading List
Developer Reading List
2012 Jolt Awards: Mobile Tools

Most Popular

Hadoop: Writing and Running Your First Project
Read/Write Properties Files in Java
Unit Testing with Python
Logging In C++

More Popular»

C/C++

Tweet Permalink

A Highly Configurable Logging Framework In C++

By Michael Schulze, June 18, 2010

Extending the Logging Framework

Due to its high configurability, it is possible to extend the framework (Requirement #2) and I've already shown one extension for the back-end side. Now, I will explain customizing the front-end — first, using customized prefixes and second, beautifying the output with colors.

Let's start with prefixed log statements, and where such prefixes are worthwhile. If you are a developer of a special algorithm termed "XYZ" and you would like to debug it with easy recognizable messages, an option you have is prefixing each debug message with a special string. To do that with my framework, you have to write a simple `struct`, holding the logging level interface:

```
struct LogAlgorithmXYZ {
    static Level::levels level () {
        return Level::debug;
    }

    static const char * desc() {
        return "[ AlgorithmXYZ ]";
    }
};
```

By providing a lot of such structs with different levels (debug, warning, info, etc) and descriptions, you can distinguish between different phases of the algorithm. If you finish debugging and want keep the statements there for later debugging sessions and/or documentation purposes, the logging level interface lets you selectively disable, thus in release builds none of the debugging messages are contained.

Enhancing the framework by additional logging levels is more like parameterization the framework than a real extension. The secondly promised extension — coloration — is from another category. Enabling beautified output requires an enhanced `loggingReturnType`, which is the topic of the following paragraphs.

Here, I describe the coloration for POSIX-conforming terminals, using ANSI escape sequences. First, you have to tell the framework that you want to extend the output type by your own, disabling the default definition of the `loggingReturnType`:

```
#define LOGGING_DEFINE_EXTENDED_OUTPUT_TYPE
#include "logging/logging.h"
```

Then I define a `struct ConsoleColors` containing an enumeration of ANSI color values. This struct lets me to have a catchable type for a special overloaded `operator<<()`that does the hard work.

```
struct ConsoleColors {
    // define some ANSI console values
    enum Colors {
        blue = 34,
        magenta = 35,
        reset = 0
    };
};
```

After defining the color type, I build a component that is capable to interpret colors and produce the correct escape sequences. The defined `Colorizer` component is derived from a type given by the template parameter.

As template argument, a configured output type like `StdLogType` is applicable, allowing the use of all framework core functionality, which is under the hood. The `Colorizer` provides an `operator<<()` that catches the color type. The `operator<<()`first saves the current set converting base, then produces the right escape sequence according to the given color, and finally restores the converting base. Saving and restoring the converting base is needed, due to the switch to decimal output used within the operator method itself. In addition to `operator<<()`, an additional `operator<<()`with template is used to forward all untreated types to the base class, that should be able to process them.

```
template < typename Base>
struct Colorizer : public Base {
    // catch color type and produce correct escape sequence
    Colorizer& operator << (const ConsoleColors::Colors l) {
        unsigned char tmp=Base::getBase();
        *this << log::dec
              << "\033["
              << static_cast<unsigned short>(l)
              << 'm';
        Base::setBase(tmp);
        return *this;
    }

    // forward unknown types to base for further processing
    template<typename T>
    Colorizer& operator << (const T &t) {
        Base::operator<<(t);
        return *this;
    }
};
```

Having the color type and the processing component ready, one step remains — setting up `loggingReturnType`. We again use the known setup macro:

```
LOGGING_DEFINE_OUTPUT( Colorizer<StdLogType > )
```

As template argument for our new `Colorizer` extension, I use the former defined `StdLogType`, which outputs all logging content to `std::clog`. After this, I can use the extension to color the output:

```
log::emit() <<ConsoleColors::blue
            <<"Hello World in blue!"
            <<ConsoleColors::reset
            << "and back to normal color mode"
            <<log::endl;
```

The presented extension is for POSIX-compliant systems, only. If you want to beautify Windows terminals as well, you will need some small adaptations. However, the general strategy of enhancing the framework is still the same. The provided sources contain the presented `Colorizer` and also a `Colorizer` example for Windows systems.

Evaluation and Resource Usage

Until now I simply presented my framework and its features, asking you to take my word that it is possible to disable it in parts and use resources efficiently, or disabling it completely, leading to no resource consumption whatsoever. Now, I would like to prove prove that this is possible. To do so, I present two examples comparing resource usage in terms of needed memory.

Example 1: Usage scenario, which uses both flavors of emit()

```
#include "logging.h"
using namespace ::logging;

int main(int, char**) {
    log::emit() << "Hello World! with the logging framework" <<  log::endl;

    log::emit< Error>()   <<  "Logging an Error"  <<  log::endl;
    log::emit< Trace>()   <<  "Logging a Trace"   <<  log::endl;
    log::emit< Warning>() <<  "Logging a Warning" <<  log::endl;
    log::emit< Info>()    <<  "Logging an Info"   <<  log::endl;
    return 0;
}
```

In the first usage scenario, Example 1 outputs all messages. For the second scenario, I disable the error level, followed by disabling the trace level in the third scenario until all levels are disabled in the fifth one. After switching all levels off, only the first log statement outputs its message. All tests are compiled by g++ version 4.4.1 for x86 platform with optimization option -Os, leading to size optimized programs. In general, the resource consumption should shrink, which is indeed the case and the results of the size program are depicted in Table 1.

Table 1: Program section size in bytes of the different scenarios

		program section size in bytes			
scenario	disabled levels	text	data	bss	sum
1	0	2455	304	172	2931
2	1	2363	304	172	2839
3	2	2288	304	172	2764
4	3	2223	304	172	2699
5	4	2092	304	172	2568

The only remaining thing is to deactivate the framework as a whole, which is feasible with the setting of an additional command-line parameter (-DLOGGING_DISABLE) for the compiler run. For showing that no more resources are needed by the framework itself, I compare the program size of Example 1 with a deactivated logging framework against an empty main implementation; see Example 2.

Example 2: Minimal program used for comparing resource consumption

```
int main(int,char**) {
    return 0;
}
```

The results speak for themselves in Table 2. Both the empty main and the program with deactivated logging framework result in the same resource consumption, which proves my statements and the fulfillment of the requirements.

Table 2: Resource consumption of the disabled framework is equivalent to a minimal example program

	program section size in bytes			
scenario	text	data	bss	sum
disabled framework	1167	276	8	1451
empty main	1167	276	8	1451

Conclusion

In this article I presented a logging framework, that is easy to use, portable, extensible, type-safe and enables selective enabling/disabling of levels as well as deactivation as a whole, with the premise of being as resource efficient as possible. One of the great benefits is, that all these properties are realized with standard C++ features, only, thus each standard compliant compiler can be used. I tested the framework with different g++ versions and with Microsoft Visual C++ 9.0.

Michael Schulze is a Ph.D. student at Otto-von-Guericke University, working on adaptable event-based middleware for resource-constraint embedded systems. He can be contacted at mschulze@ivs.cs.ovgu.de.