**logging: A C++ simple logging library**

Author: Yongtian Zhang ([yongtianzhang@gmail.com](mailto:yongtianzhang@gmail.com))

Date: 2011.10.15

目录

[Framework 2](#_Toc306407230)

[Description 2](#_Toc306407231)

[Flow Chart 3](#_Toc306407232)

[Examples 4](#_Toc306407233)

[C++ stream style usage 4](#_Toc306407234)

[Convenient logging functions 4](#_Toc306407235)

[Trace message helper 5](#_Toc306407236)

[Indent control helper 5](#_Toc306407237)

[Attach management helper 6](#_Toc306407238)

[Reference 7](#_Toc306407239)

[Namespace logging 7](#_Toc306407240)

[Enumeration Level 7](#_Toc306407241)

[Class Logger 8](#_Toc306407242)

[Constructors 8](#_Toc306407243)

[C++ stream style API 9](#_Toc306407244)

[Convenient log functions 10](#_Toc306407245)

[Sink manipulators 11](#_Toc306407246)

[Indent manipulators 11](#_Toc306407247)

[Class Trace 12](#_Toc306407248)

[Constructor 12](#_Toc306407249)

[Destructor 12](#_Toc306407250)

[Class Indent 13](#_Toc306407251)

[Constructor 13](#_Toc306407252)

[Destructor 13](#_Toc306407253)

[Class Attach 13](#_Toc306407254)

[Constructor 13](#_Toc306407255)

[Destructor 14](#_Toc306407256)

[Global standard logger object 14](#_Toc306407257)

[Global convenient functions 14](#_Toc306407258)

[Global utility functions 15](#_Toc306407259)

# Framework

## Description

User calls log API of class Logger, to send message and severity level to Logger object. Logger object format this message with timestamp and severity level, and then dispatch it to sinks bound to the Logger object.

Sink is C++ std::ostream object. So the C++ style stream output is performed. Every sink has a severity level, only message that has equal or higher severity level will be accepted by sink.

The library also provides a global standard Logger object, initialized with std::clog. The library provides several convenient free functions with the same name as the corresponding member functions of class Logger, just to perform the corresponding operations on the standard Logger object.

## Flow Chart

The frame work is shown as follows:

Assume the message is “warning message”, severity level is WARNING\_.

User code

Logger object

“warn”

WARNING\_

Info sink

warning sink

error sink

WARNING\_

INFO\_

ERROR\_

WARNING\_

“2011-08-26 02:19:00 WARNING warn\n”

# Examples

## C++ stream style usage

**Code:**

using namespace logging;

log(INFO\_) << “executing information message” << endl;

int ret = -1;

log(ERROR\_) << “error return code ” << ret << endl;

**Output:**

2011-08-27 22:55:05 INFO executing information message

2011-08-27 22:55:05 ERROR error return code -1

## Convenient logging functions

**Code:**

logging::info(“executing information message”);

logging::error(“return code error”);

**Output:**

2011-08-27 22:55:05 INFO executing information message

2011-08-27 22:55:05 ERROR return code error

## Trace message helper

**Code:**

using namespace logging;

void test\_trace() {

Trace trace(INFO\_, “test\_trace()”);

log(INFO\_) << “do something” << std::endl;

}

int main() {

test\_trace();

return 0;

}

**Output:**

2011-08-27 22:55:05 INFO enter test\_trace()

2011-08-27 22:55:05 INFO do something

2011-08-27 22:55:05 INFO leave test\_trace()

## Indent control helper

**Code:**

using namespace logging;

void test\_indent() {

Trace trace(INFO\_, “test\_indent()”);

Indent indent(“\*\*\*\*”);

log(INFO\_) << “some message” << std::endl;

}

Int main() {

test\_indent();

return 0;

}

**Output:**

2011-08-27 22:55:05 INFO enter test\_indent()

2011-08-27 22:55:05 INFO \*\*\*\*some message

2011-08-27 22:55:05 INFO leave test\_indent()

## Attach management helper

**Code:**

using namespace logging;

void test\_attach() {

Trace trace(INFO\_, “test\_attach()”);

std::ofstream flog(“log.txt”);

{

Attach attach(INFO\_, flog);

log(INFO\_) << “you can see this line in file” << std::endl;

}

log(INFO\_) << “but you can’t see this line in file” << std::endl;

}

**Output:**

Screen:

2011-08-27 22:55:05 INFO enter test\_attach()

2011-08-27 22:55:05 INFO you can see this line in file

2011-08-27 22:55:05 INFO but you can’t see this line in file

2011-08-27 22:55:05 INFO leave test\_attach()

File ‘log.txt’:

2011-08-27 22:55:05 INFO enter test\_attach()

2011-08-27 22:55:05 INFO you can see this line in file

2011-08-27 22:55:05 INFO leave test\_attach()

# Reference

## Namespace logging

This is the only namespace of this library. All following stuffs are defined in this scope.

## Enumeration Level

1. **Named values of Level enumeration logging::Level**
2. logging::DEBUG\_: diagnosing information
3. logging::INFO\_: normal executing status information
4. logging::WARNING\_: warning information
5. logging::ERROR\_: error information
6. logging::CRITICAL\_: critical error information
7. **std::string logging::to\_string(logging::Level level, size\_t width=0);**

This is a convenient function to convert Level enumeration /level/ into string.

/width/ is optional. If /level/’s string form length is less than /width/, the string will append some space until string.size()==width. If /level/’s string form length is equal or greater than /width/, string form is unchanged.

1. **std::ostream& logging::operator << (std::ostream& os, logging::Level level);**

This function insert /level/’s string form into ostream /os/. The two calls have the same effect:

1. cout << logging::INFO\_ << endl;
2. cout << to\_string(logging::INFO\_) << endl;

## Class Logger

### Constructors

1. **logging::Logger::Logger();**

Default constructor of Logger, with no sink bound. Logging operations will take no effect before bound sinks.

1. **logging::Logger::Logger(std::ostream& sink,**

**logging::Level level=logging::INFO\_);**

Construct a Logger object with one sink /sink/, and severity level of /sink/ is set to /level/. default level is logging::INFO\_, to show at least executing status messages.

### C++ stream style API

1. **template <typename T>**

**logging::Logger& logging::Logger::operator << (const T& value);**

This function overloads the “<<” operator, to provide a C++ stream like API to log message.

1. **logging::Logger& logging::Logger::operator () (logging::Level level);**

This function overloads the “()” operator, to set severity level of the coming logging message, and generate logging item prefix by current timestamp and severity level string.

1. **Note: typical usage of these API is:**

**Code:**

#include “logger.h”

#include <iostream>

using namespace logging;

int main() {

Logger logger(clog, WARNING\_);

logger(WARNING\_) << “warning message” << std::endl;

logger(ERROR\_) << “error message” << std::endl;

}

**Output:**

2011-08-26 03:10:03 WARNING warning message

2011-08-26 03:10:03 ERROR error message

### Convenient log functions

1. **void logging::Logger::log(logging::Level level,**

**const std::string& message);**

This is a convenient function to log string message. The two call do the same thing:

1. logger(INFO\_) << “message” << endl;
2. logger.log(INFO\_, “message”);
3. **void logging::Logger::debug(const std::string& message);**
4. **void logging::Logger::info(const std::string& message);**
5. **void logging::Logger::warning(const std::string& message);**
6. **void logging::Logger::error(const std::string& message);**
7. **void logging::Logger::critical(const std::string& message);**

The 5 functions are all convenient functions that call logging::Logger::log(level, message) with the corresponding severity level.

### Sink manipulators

1. **logging::Level& logging::Logger::level\_of\_sink(std::ostream& sink);**

Get reference to the level of specified /sink/. If /sink/ is not in Logger’s sink set, throw std::runtime\_error.

1. **bool logging::Logger::verify\_sink(std::ostream& sink);**

Check whether /sink/ is in Logger’s sink set.

1. **void logging::Logger::attach\_sink(std::ostream& sink,**

**logging::Level level=logging::INFO\_);**

Insert /sink/ to Logger’s sink set, with severity level /level/. If /sink/ is already in sink set, this function takes no effect. Sinks in sink set are not in certain order.

1. **void logging::Logger::detach\_sink(std::ostream& sink);**

remove /sink/ from Logger’s sink set. If /sink/ is not in Logger’s sink set, this function takes no effect.

### Indent manipulators

1. **void logging::Logger::indent(const std::string& info);**

Add a new level indent to the logger object, so that each message logging after this function call will have a new indent added.

1. **void logging::Logger::unindent();**

Remove an indent level of the logger object, so that each message logging after this function call will have the newest added indent removed.

## Class Trace

It is a convenient class to log scope enter and leaving messages. It logs enter message while constructing and leaving message while destructing.

### Constructor

1. **logging::Trace::Trace(logging::Level level, const std::string& scope,**

**logging::Logger& logger=logging::standard\_logger());**

Construct a /Trace/ object with a severity level, scope information, and a logger object. Constructor will perform as:

logger(level) << “enter ” << scope << std::endl;

### Destructor

1. **logging::Trace::~Trace();**

As the corresponding constructor, it will perform as:

logger(level) << “leave ” << scope << std::endl;

## Class Indent

It’s a convenient class to control indent of message logging. It add a new level of indent to logger object when constructing and remove that indent when destructing. So that every message will have a new level indent in the life scope of an /Indent/ instance.

### Constructor

1. **logging::Indent::Indent(const std::string& info=” ”,**

**logging::Logger& logger=logging::standard\_logger());**

Construct a /Indent/ object with an indent string and a logger object. Constructor will perform as:

logger.indent(info);

### Destructor

1. **logging::Indent::~Indent();**

Destruct the /Indent/ object, perform as:

logger.unindent();

## Class Attach

### Constructor

1. **logging::Attach:: Attach(logging::Level level, std::ostream& sink,**

**logging::Logger& logger=logging::standard\_logger());**

Construct a /Attach/ object with a severity level, std::ostream object, and a logger object. Constructor will perform as:

logger.attach\_sink(sink);

### Destructor

1. **logging::Attach::~Attach();**

As the corresponding constructor, it will perform as:

logger.detach\_sink(sink);

## Global standard logger object

1. **logging::Logger& standard\_logger();**

This function constructs a static Logger object with std::clog and severity level logging::INFO\_. This object will be initialized only once even if multiple calls to this function.

This function returns this static constructed Logger instance, as the standard logger object.

## Global convenient functions

In namespace logging, there are several convenient functions, to perform convenient log operations on the standard logger object.

1. **logging::Logger& logging::log(logging::Level level);**

Convenient function to logging::standard\_logger().operator () (level).

The two calls are equivalent:

1. logging::standard\_logger()(logging::INFO\_) << “message” << std::endl;
2. logging::log(logging::INFO\_) << “message” << std::endl;

It’s obviously that the latter is simpler.

1. **void logging::log(logging::Level level, const std::string& message);**

Equivalent to logging::standard\_logger().log(level, message);

1. **void logging::debug(const std::string& message);**

Equivalent to logging::standard\_logger().debug(message);

1. **void logging::info(const std::string& message);**

Equivalent to logging::standard\_logger().info(message);

1. **void logging::warning(const std::string& message);**

Equivalent to logging::standard\_logger().warning(message);

1. **void logging::error(const std::string& message);**

Equivalent to logging::standard\_logger().error(message);

1. **void logging::critical(const std::string& message);**

Equivalent to logging::standard\_logger().critical(message);

## Global utility functions

1. **std::string logging::current\_timestamp();**

Get the current timestamp string, with format “YYYY-MM-DD HH:MM:SS”.