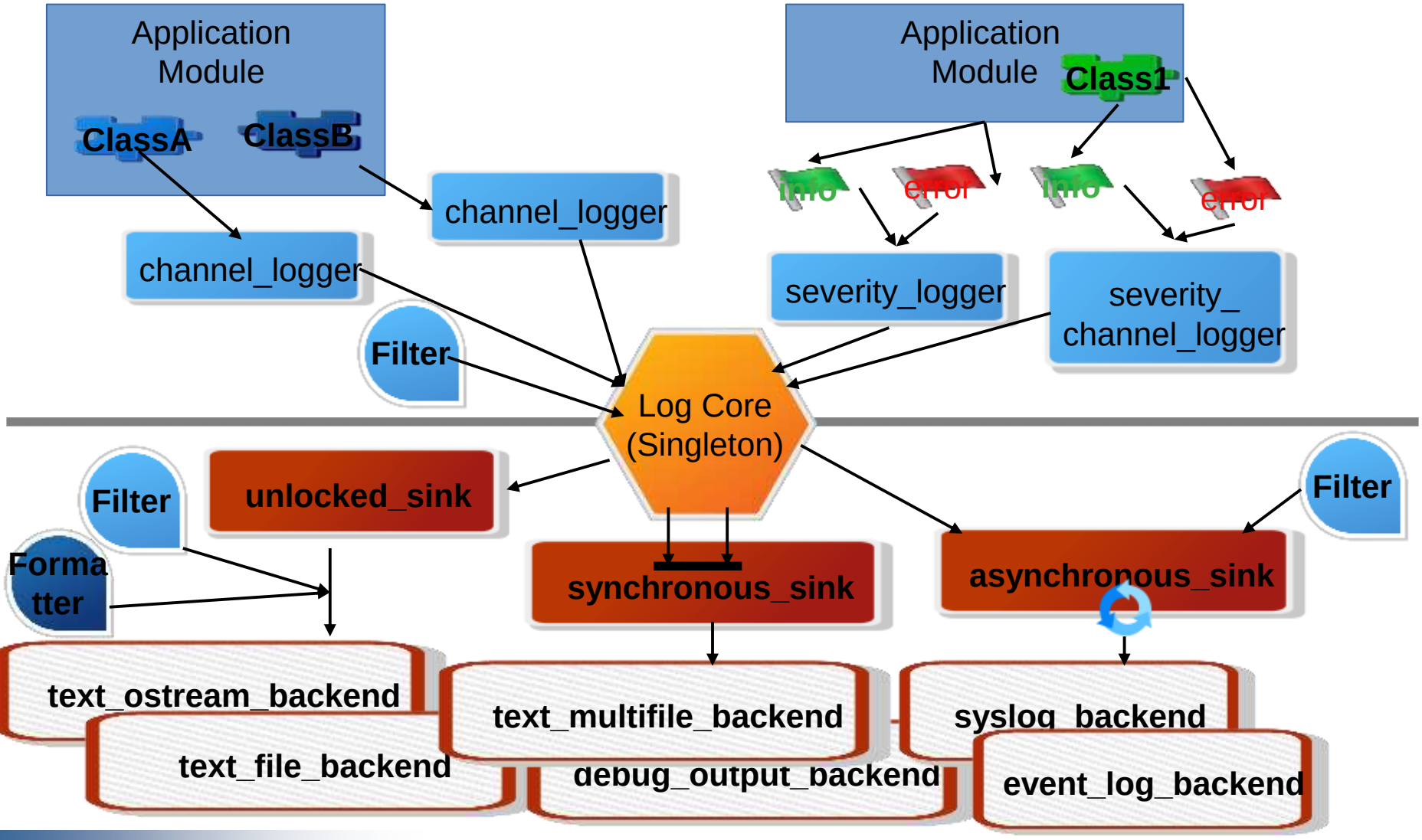
GHF automation GmbH

Boost.Log

Introduction to the Boost Library for Logging

Architecture

GHF automation GmbH



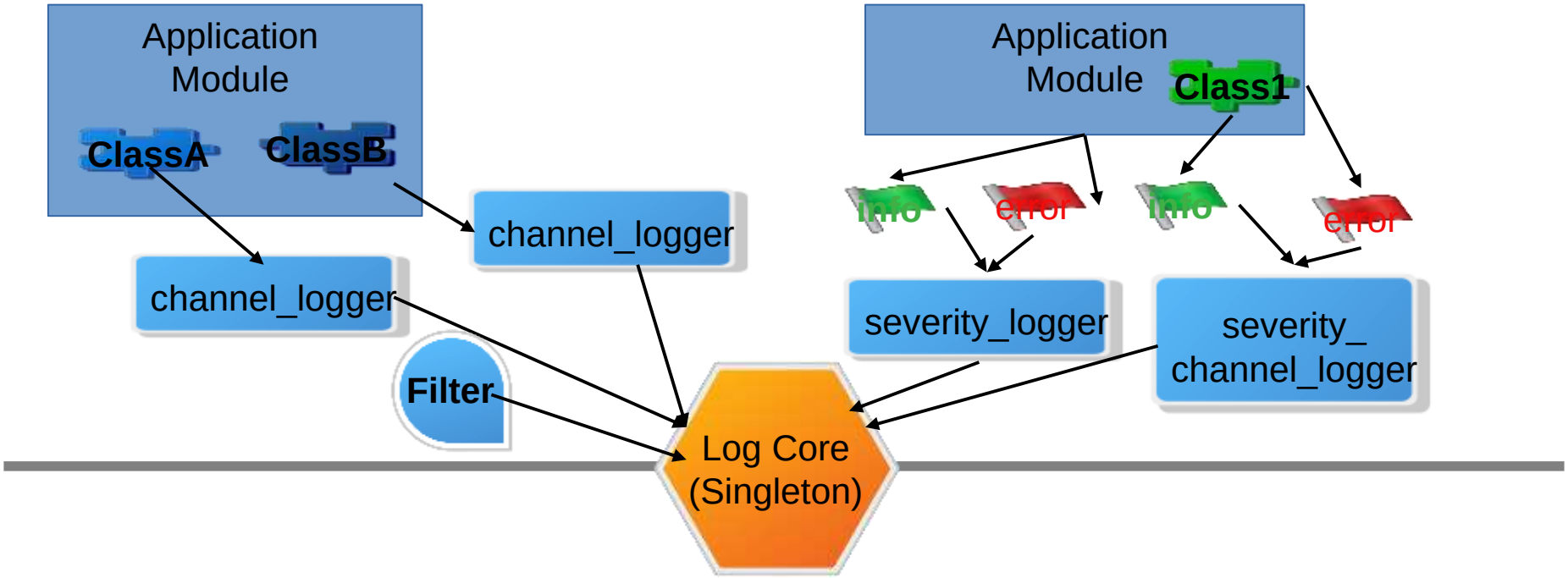
Main Features

GHF automation GmbH

- Generating log messages in code is independent of where the log messages are output
- Log messages can be output to text files, streams, event logs, debug output
- Filters can be applied to redirect messages to different output destinations (e.g. info messages to DebugView, error messages to a file)
- Singleton architecture requires use of shared version of Boost Log library
- Presentation covers log v2

Architecture - Logger/Sources

GHF automation GmbH



Log Sources (‘Loggers’)

GHF automation GmbH

- Loggers provide the interface for the application to output log messages
- Predefined logger classes add information to the log messages, which can be accessed by filters

Class	Function
logger	basic logger
severity_logger / severity_logger_mt	adds 'Severity' attribute, specifying importance of log messages
channel_logger / channel_logger_mt	adds 'Channel' attribute, specifying origin of the log messages
severity_channel_logger / severity_channel_logger_mt	adds both 'Severity' and 'Channel' attribute

How to Create a Log Record

GHF automation GmbH

- Macros make the loggers easier accessible and avoid clogging the source code with log statements
- Macros depend on logger type
- Logger instance is passed as parameter

channel_logger	`...`
	`sources::channel_logger<>`
	`lg{keywords::channel = "MyModule"};`
	`BOOST_LOG(lg) << "my message";`

severity_logger	`...`
	`sources::severity_logger<trivial::severity_`
	`level > sev_lg;`
	`BOOST_LOG_SEV(sev_lg,`
	`trivial::severity_level::info)`
	`  << "my message";`

macros for global severity_logger	… `LOG_INFO << "my message";`

Log Records and Attributes

GHF automation GmbH

- a log record consists of a set of attributes, which contain the information of the log entry
- attributes can be global or scoped
- attributes are used for filtering and formatting
- the function 'add_common_attributes() adds the following attributes to global scope

Attribute Name	Type	Contents
line_id	counter	sequential index of log record
timestamp	local_clock	current local time
process_id	current_thread_id	ID of current thread (if multithreaded)
thread_id	current_process_id	ID of current process

More Attributes

GHF automation GmbH

- add code location to log entry with 'named_scope' type
- use macros 'BOOST_LOG_NAMED_SCOPE' or 'BOOST_LOG_FUNCTION' to add a 'named_scope' with method name, source file and line number to log
- with format_named_scope() one can control the output of the scope information in the log message, e.g. "%f(%l): %c" will output '<File>(<line_no>): <method with scope>'

```
void MyClass::MyFctC(int p_iValue)
{
  BOOST_LOG_FUNCTION();
  LOG_INFO << "Value is: '" << p_iValue << "'";
}
…
l_myClass.MyFctC(4711);
```

Line number of scope capture, not log message

Output

```
[PID: 0x00002d84][Thread: 0x00003edc][2019-01-10
16:53:31][info] [Scope: my_lib.cpp(35): MyClass::MyFctC]
Value is: '4711'
```

Custom Attributes

GHF automation GmbH

- define own attributes to pass data from logging source to backend
- or to replace 'local time' (part of common attributes) with 'UTC time'
- use a scoped attribute to make sure that the attribute was specifically added to that log record and you are not retrieving old values from the last update of a global attribute
- wrap attribute in a macro together with creating the log record

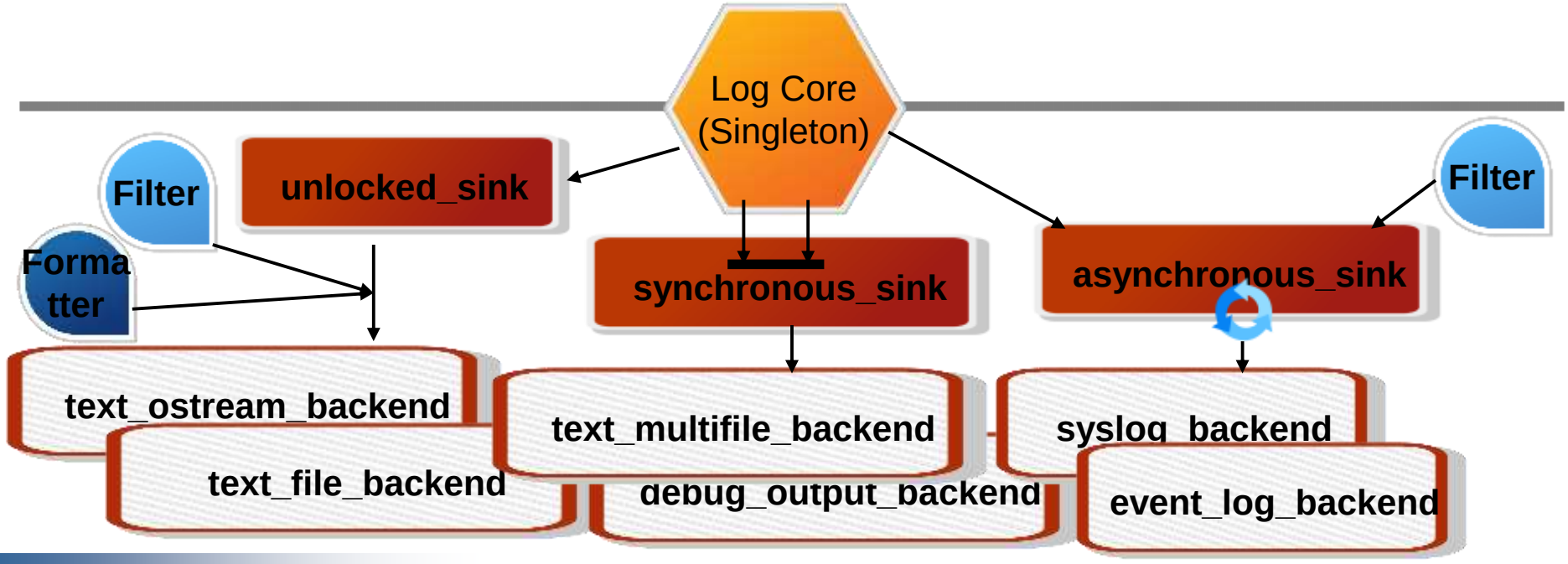
Example: add the line number of the log macro to the log record

```
#define LOG_LINE(sev, message) \
  do { \
  BOOST_LOG_SCOPED_LOGGER_ATTR(global_logger::get() \
        , "LineNo" \
        , boost::log::attributes::mutable_constant<int>(__LINE__) ); \
  BOOST_LOG_SEV(global_logger::get(),boost::log::trivial::sev) << message; \
  } while(false);
```

```
…
LOG_LINE(info, "LOG_LINE: Value is: '" << p_iValue << "'");
```

Architecture - Sinks and Backends

GHF automation GmbH



Sink Frontends

GHF automation GmbH

- get the log messages from the core and pass them on the assigned sink backend
- available sink frontends are

Class	Function
unlocked_sink	for single-threaded use
synchronous_sink	synchronizes access to the sink backend
asynchronous_sink	sink backend is accessed from a separate thread

- use **asynchronous_sink**, when output device is slower. e.g. a database
- with **asynchronous_sink** call '`flush()`' to make sure that all messages are stored in the backend

Filters and Formatters

GHF automation GmbH

- a filter condition selets the log records, which are passed on to the respective backend, by evaluating the attributes of the record
- filters are assigned to the sink frontend, e.g. by calling synchronous_sink::set_filter()
- a formatter outputs a log record to a text-based backend
- formatters are assigned to the sink frontend, e.g. by calling synchronous_sink::set_formatter()
- there are specific formatters for certain types, e.g. for 'local_time' or 'named_scope'
- use conditional formatters, if attributes may not exist in each log record (see 'expressions::if_' and 'expressions::has_attr')

Sink Backends

GHF automation GmbH

- define, where log entries are stored
- below are predefined types, but custom backends can be added, e.g. to store log records in a database

Class	Function
text_ostream_backend	text stream
text_file_backend	text file Note: file name can be constructed at runtime; file rotation by date or reaching a max size
text_multifile_backend	multiple files with file names selected from attributes, e.g. one file for each thread
text_ipc_message_queue_backend	to pass log entries between processes
syslog_backend	via BSD syslog protocol
debug_output_backend	for Windows OutputDebugString() API
event_log_backend	Windows Event log

Take Care of…

GHF automation GmbH

- when using Boost.Log from multiple binaries link to shared version ('**BOOST_LOG_DYN_LINK**')

  Note: this will required the shared version of Boost DateTime, Filesystem and Thread

- on Windows Boost Log linkage depends on setting of **BOOST_USE_WINAPI_VERSION** (e.g. 0x0601 for Windows 7 and newer)
- when specifying your own type for the **severity level** of 'severity_logger_mt', make sure to use that type throughout sink and formatters too
- careful with **global logger** and **program termination**; consider calling core::get()->remove_all_sinks() to ensure that sinks are destroyed before the logger core

Caveats

GHF automation GmbH

- when using file sinks careful with crash at process termination (see '**enable_final_rotation**')
- call flush() and stop() on asynchronous sinks to ensure that all sinks have been written through to the backend
- use the same identifier for an attribute throughout the code
- the value of a global attribute may not have been updated together with the log record (e.g. see line number set with 'LOG_FUNCTION')
- use 'boost::log::keywords' where necessary, e.g.
  wrong: channel_logger<> ch_lg{"myModule"};
  correct: channel_logger<> ch_lg{*keywords::channel* = "myModule"};

Start with a Global Logger

GHF automation GmbH

- definition of a global logger is the easiest way to start
- based on 'severity_logger' (a single global 'channel_logger' does not really make sense)
- easier with macros 'BOOST_LOG_GLOBAL_LOGGER() /_INIT()'
- link the logger implementation as static library to all shared modules, that use the logger
- access logger via static 'get' method e.g. **`global_logger::get()`**
- initializer contains definition of attributes, formatters, filters and sinks

```
BOOST_LOG_GLOBAL_LOGGER_INIT(global_logger, global_logger_t)
{
  1. define attributes
  2. define formatters and filters
  3. create and register sinks and backends
}
```

Global Logger - Header with Macros

GHF automation GmbH

```
typedef
boost::log::sources::severity_logger_mt<boost::log::trivial::severity_level>
global_logger_t;

BOOST_LOG_GLOBAL_LOGGER(global_logger, global_logger_t)

#define LOG(sev) BOOST_LOG_SEV(global_logger::get(),boost::log::trivial::sev)

static const char* g_szAttribName_LineNo = "LineNo"; // attrib identifier
#define LOG_LINE(sev, message) \
  do { \
  BOOST_LOG_SCOPED_LOGGER_ATTR(global_logger::get(), g_szAttribName_LineNo,
boost::log::attributes::mutable_constant<int>(__LINE__) ); \
  BOOST_LOG_SEV(global_logger::get(),boost::log::trivial::sev) << message; \
  } while(false);

// macros for logging to the global severity_logger
#define LOG_TRACE   LOG(trace)
#define LOG_DEBUG   LOG(debug)
#define LOG_INFO    LOG(info)
#define LOG_WARNING LOG(warning)
#define LOG_ERROR   LOG(error)
#define LOG_FATAL   LOG(fatal)
```

Global Logger - Define Attributes

GHF automation GmbH

```
// add common attributes (log index, local_clock,
//   current_process_id, current_thread_id)
blog::add_common_attributes();

// timestamp in UTC
boost::log::core::get()->add_global_attribute("timeUTC"
        , boost::log::attributes::utc_clock());

// add specific attribute for named_scope
blog::core::get()->add_global_attribute("Scope",
blog::attributes::named_scope());
```

- see other predefined attributes, e.g. **'attributes::timer'** to obtain expired time of the process

Global Logger - Define Formatter

GHF automation GmbH

```
boost::log::formatter l_formatterText = boost::log::expressions::stream
  << "[PID: " << processid << "]"
  << "[Thread: " << threadid << "]"
  << "["
  << boost::log::expressions::format_date_time(timestampUTC, "%Y-%m-%d
%H:%M:%S")
  << " (UTC)]"
  << "[" << severity << "] "
  << "[Scope: " << boost::log::expressions::format_named_scope( scope
        , boost::log::keywords::format = "%F(%l): %c"
        , boost::log::keywords::iteration = boost::log::expressions::reverse
        , boost::log::keywords::depth = 2) << "] "
  << boost::log::expressions::if_(boost::log::expressions::has_attr(lineNo))
        [
          boost::log::expressions::stream << "[Line: " << lineNo << "] "
        ]
  << boost::log::expressions::smessage;
```

- if attribute may be missing in record (e.g. a scoped attribute), check with 'has_attr', whether it exists (use if_/else_ in conditional formatter)

Global Logger - Create File Sink

GHF automation GmbH

```
auto l_fileBackend = boost::make_shared<
boost::log::sinks::text_file_backend >(
    boost::log::keywords::file_name = "file_%5N.log",
    boost::log::keywords::rotation_size = 256, // start new file, when
size reaches 256 bytes
    boost::log::keywords::time_based_rotation
      = boost::log::sinks::file::rotation_at_time_point(23, 58, 0),
    boost::log::keywords::enable_final_rotation = false );

// add a sink for the file backend
auto l_sinkFile = boost::make_shared<
boost::log::sinks::synchronous_sink<
boost::log::sinks::text_file_backend> >(l_fileBackend);

l_sinkFile->set_formatter(formatter);
l_sinkFile->set_filter(severity >= boost::log::trivial::error);
// "register" our sink
boost::log::core::get()->add_sink(l_sinkFile);
```

- check specific formatting placeholders for file name

Global Logger - Debug/Console Sink

GHF automation GmbH

```
// add sink for DbgView
auto l_sinkDbg = boost::make_shared<
boost::log::sinks::synchronous_sink<boost::log::sinks::debug_output_ba
ckend> >();

l_sinkDbg->set_formatter(formatter);
// only messages with severity >= debug are written
l_sinkDbg->set_filter(severity >= boost::log::trivial::debug);
// "register" our sink
boost::log::core::get()->add_sink(l_sinkDbg);

// add a sink for the console
auto l_sinkConsole = boost::log::add_console_log(std::cout);
l_sinkConsole->set_formatter(formatter);
l_sinkConsole->set_filter(severity >= boost::log::trivial::debug);
```

- debug output is Windows specific (tool 'DebugView' for capture/view)
- support for Windows Event log with message IDs and related texts; messages are stored language independent and texts are constructed when viewing the log