Chapter 62. Boost.Log

Boost.Log is the logging library in Boost. It supports numerous back-ends to log data in various formats. Back-ends are accessed through front-ends that bundle services and forward log entries in different ways. For example, there is a front-end that uses a thread to forward log entries asynchronously. Front-ends can have filters to ignore certain log entries. And they define how log entries are formatted as strings. All these functions are extensible, which makes Boost.Log a powerful library.

Example 62.1. Back-end, front-end, core, and logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  sources::logger lg;

  BOOST_LOG(lg) << "note";
  sink->flush();
}
```

Example 62.1 introduces the essential components of Boost.Log. Boost.Log gives you access to back-ends, front-ends, the core, and loggers:

- Back-ends decide where data is written. `boost::log::sinks::text_ostream_backend` is initialized with a stream of type `std::ostream` and writes log entries to it.

- Front-ends are the connection between the core and a back-end. They implement various functions that don't need to be implemented by each individual back-end. For example, filters can be added to a front-end to choose which log entries get forwarded to the back-end and which don't.

  Example 62.1 uses the front-end `boost::log::sinks::asynchronous_sink`. You must use a front-end even if you don't use filters. `boost::log::sinks::asynchronous_sink` uses a thread that forwards log entries to a back-end asynchronously. This can improve the performance but defers write operations.

- The core is the central component that all log entries are routed through. It is implemented as a singleton. To get a pointer to the core, call `boost::log::core::get()`.

Front-ends must be added to the core to receive log entries. Whether log entries are forwarded to front-ends depends on the filter in the core. Filters can be registered either in front-ends or in the core. Filters registered in the core are global, and filters registered in front-ends are local. If a log entry is filtered out by the core, it isn't forwarded to any front-end. If it is filtered by a front-end, it can still be processed by other front-ends and forwarded to their back-ends.

- The logger is the component in Boost.Log you will use most often. While you access back-ends, front-ends, and the core only when you initialize the logging library, you use a logger every time you write a log entry. The logger forwards the entry to the core.

  The logger in Example 62.1 is of the type `boost::log::sources::logger`. This is the simplest logger. When you want to write a log entry, use the macro `BOOST_LOG` and pass the logger as a parameter. The log entry is created by writing data into the macro as if it is a stream of type `std::ostream`.

Back-end, front-end, core, and logger work together. `boost::log::sinks::asynchronous_sink`, a front-end, is a template that receives the back-end `boost::log::sinks::text_ostream_backend` as a parameter. Afterwards, the front-end is instantiated with `boost::shared_ptr`. The smart pointer is required to register the front-end in the core: the call to `boost::log::core::add_sink()` expects a `boost::shared_ptr`.

Because the back-end is a template parameter of the front-end, it can only be configured after the front-end has been instantiated. The back-end determines how this is done. The member function `add_stream()` is provided by the back-end `boost::log::sinks::text_ostream_backend` to add streams. You can add more than one stream to `boost::log::sinks::text_ostream_backend`. Other back-ends provide different member functions for configuration. Consult the documentation for details.

To get access to a back-end, all front-ends provide the member function `locked_backend()`. This member function is called `locked_backend()` because it returns a pointer that provides synchronized access to the back-end as long as the pointer exists. You can access a back-end through pointers returned by `locked_backend()` from multiple threads without having to synchronize access yourself.

You can instantiate a logger like `boost::log::sources::logger` with the default constructor. The logger automatically calls `boost::log::core::get()` to forward log entries to the core.

You can access loggers without macros. Loggers are objects with member functions you can call. However, macros like `BOOST_LOG` make it easier to write log entries. Without macros it wouldn't be possible to write a log entry in one line of code.

Example 62.1 calls `boost::log::sinks::asynchronous_sink::flush()` at the end of `main()`. This call is required because the front-end is asynchronous and uses a thread to forward log entries. The call makes sure that all buffered log entries are passed to the back-end and are written. Without the call to `flush()`, the example could terminate without displaying `note`.

Example 62.2. `boost::sources::severity_logger` with a filter

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

bool only_warnings(const attribute_value_set &set)
{
  return set["Severity"].extract<int>() > 0;
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(&only_warnings);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG(lg) << "note";
  BOOST_LOG_SEV(lg, 0) << "another note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Example 62.2 is based on Example 62.1, but it replaces `boost::sources::logger` with the logger `boost::sources::severity_logger`. This logger adds an attribute for a log level to every log entry. You can use the macro `BOOST_LOG_SEV` to set the log level.

The type of the log level depends on a template parameter passed to `boost::sources::severity_logger`. Example 62.2 uses `int`. That's why numbers like 0 and 1 are passed to `BOOST_LOG_SEV`. If `BOOST_LOG` is used, the log level is set to 0.

Example 62.2 also calls `set_filter()` to register a filter at the front-end. The filter function is called for every log entry. If the function returns `true`, the log entry is forwarded to the back-end. Example 62.2 defines the function `only_warnings()` with a return value of type `bool`.

`only_warnings()` expects a parameter of type `boost::log::attribute_value_set`. This type represents log entries while they are being passed around in the logging framework. `boost::log::record` is another type for log entries that is like a wrapper for `boost::log::attribute_value_set`. This type provides the member function `attribute_values()`, which retrieves a reference to the `boost::log::attribute_value_set`. Filter functions receive a `boost::log::attribute_value_set` directly and no `boost::log::record`. `boost::log::attribute_value_set` stores key/value pairs. Think of it as a `std::unordered_map`.

Log entries consist of attributes. Attributes have a name and a value. You can create attributes yourself. They can also be created automatically – for example by loggers. In fact, that's why Boost.Log provides multiple loggers. `boost::log::sources::severity_logger` adds an attribute called Severity to every log entry. This attribute stores the log level. That way a filter can check whether the log level of a log entry is greater than 0.

`boost::log::attribute_value_set` provides several member functions to access attributes. The member functions are similar to the ones provided by `std::unordered_map`. For example, `boost::log::attribute_value_set` overloads the operator `operator[]`. This operator returns the value of an attribute whose name is passed as a parameter. If the attribute doesn't exist, it is created.

The type of attribute names is `boost::log::attribute_name`. This class provides a constructor that accepts a string, so you can pass a string directly to `operator[]`, as in Example 62.2.

The type of attribute values is `boost::log::attribute_value`. This class provides member functions to receive the value in the attribute's original type. Because the log level is an `int` value, `int` is passed as a template parameter to `extract()`.

`boost::log::attribute_value` also defines the member functions `extract_or_default()` and `extract_or_throw()`. `extract()` returns a value created with the default constructor if a type conversion fails – for example 0 in case of an `int`. `extract_or_default()` returns a default value which is passed as another parameter to that member function. `extract_or_throw()` throws an exception of type `boost::log::runtime_error` in the event of an error.

For type-safe conversions, Boost.Log provides the visitor function `boost::log::visit()`, which you can use instead of `extract()`.

Example 62.2 displays `warning`. This log entry has a log level greater than 0 and thus isn't filtered.

Example 62.3. Changing the format of a log entry with `set_formatter()`

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

void severity_and_message(const record_view &view, formatting_ostream &os)
{
  os << view.attribute_values()["Severity"].extract<int>() << ": " <<
    view.attribute_values()["Message"].extract<std::string>();
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();
```

```
  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_formatter(&severity_and_message);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Example 62.3 is based on Example 62.2. This time the log level is displayed.

Front-ends provide the member function `set_formatter()`, which can be passed a format function. If a log entry isn't filtered by a front-end, it is forwarded to the format function. This function formats the log entry as a string that is then passed from the front-end to the back-end. If you don't call `set_formatter()`, by default the back-end only receives what is on the right side of a macro like `BOOST_LOG`.

Example 62.3 passes the function `severity_and_message()` to `set_formatter()`. `severity_and_message()` expects parameters of type `boost::log::record_view` and `boost::log::formatting_ostream`. `boost::log::record_view` is a view on a log entry. It's similar to `boost::log::record`. However, `boost::log::record_view` is an immutable log entry.

`boost::log::record_view` provides the member function `attribute_values()`, which returns a constant reference to `boost::log::attribute_value_set`. `boost::log::formatting_ostream` is the stream used to create the string that is passed to the back-end.

`severity_and_message()` accesses the attributes Severity and Message. `extract()` is called to get the attribute values, which are then written to the stream. Severity returns the log level as an `int` value. Message provides access to what is on the right side of a macro like `BOOST_LOG`. Consult the documentation for a complete list of available attribute names.

Example 62.3 uses no filter. The example writes two log entries: `0: note` and `1: warning`.

Example 62.4. Filtering log entries and formatting them with lambda functions

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();
```

```
  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::attr<int>("Severity") > 0);
  sink->set_formatter(expressions::stream <<
    expressions::attr<int>("Severity") << ": " << expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Example 62.4 uses both a filter and a format function. This time the functions are implemented as lambda functions – not as C++11 lambda functions but as Boost.Phoenix lambda functions.

Boost.Log provides helpers for lambda functions in the namespace `boost::log::expressions`. For example, **`boost::log::expressions::stream`** represents the stream. **`boost::log::expressions::smessage`** provides access to everything on the right side of a macro like `BOOST_LOG`. You can use `boost::log::expressions::attr()` to access any attribute. Instead of **`smessage`** Example 62.4 could use `attr<std::string>("Message")`.

Example 62.4 displays `1: warning` and `2: error`.

Example 62.5. Defining keywords for attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << severity << ": " <<
    expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Boost.Log supports user-defined keywords. You can use the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords to access attributes without having to repeatedly pass attribute names as strings to `boost::log::expressions::attr()`.

Example 62.5 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define a keyword **`severity`**. The macro expects three parameters: the name of the keyword, the attribute name as a string, and the type of the attribute. The new keyword can be used in filter and format lambda functions. This means you are not restricted to using keywords, such as **`boost::log::expressions::smessage`**, that are provided by Boost.Log – you can also define new keywords.

In all of the examples so far, the attributes used are the ones defined in Boost.Log. Example 62.6 shows how to create user-defined attributes.

Example 62.6. Defining attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << counter << " - " << severity <<
    ": " << expressions::smessage << " (" << timestamp << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

You create a global attribute by calling `add_global_attribute()` on the core. The attribute is global because it is added to every log entry automatically.

`add_global_attribute()` expects two parameters: the name and the type of the new attribute. The name is passed as a string. For the type you use a class from the namespace `boost::log::attributes`, which provides classes to define different attributes. Example 62.6 uses `boost::log::attributes::counter` to define the attribute LineCounter, which adds a line number to every log entry. This attribute will number log entries starting at 1.

`add_global_attribute()` is not a function template. `boost::log::attributes::counter` isn't passed as a template parameter. The attribute type must be instantiated and passed as an object.

Example 62.6 uses a second attribute called Timestamp. This is a scoped attribute that is created with `BOOST_LOG_SCOPED_LOGGER_ATTR`. This macro adds an attribute to a logger. The first parameter is the logger, the second is the attribute name, and the third is the attribute object. The type of the attribute object is `boost::log::attribute::local_clock`. The attribute is set to the current time for each log entry.

The attribute Timestamp is added to the log entry "error" only. Timestamp exists only in the scope where `BOOST_LOG_SCOPED_LOGGER_ATTR` is used. When the scope ends, the attribute is removed. `BOOST_LOG_SCOPED_LOGGER_ATTR` is similar to a call to `add_attribute()` and `remove_attribute()`.

As in Example 62.5, Example 62.6 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords for the new attributes. The format function accesses the keywords to write the line number and current time. The value of **`timestamp`** will be an empty string for those log entries where the attribute Timestamp is undefined.

Example 62.7. Helper functions for filters and formats

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <iomanip>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
```

```
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::is_in_range(severity, 1, 3));
  sink->set_formatter(expressions::stream << std::setw(5) << counter <<
    " - " << severity << ": " << expressions::smessage << " (" <<
    expressions::format_date_time(timestamp, "%H:%M:%S") << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

Boost.Log provides numerous helper functions for filters and formats. Example 62.7 calls the helper `boost::log::expressions::is_in_range()` to filter log entries whose log level is outside a range. `boost::log::expressions::is_in_range()` expects the attribute as its first parameter and lower and upper bounds as its second and third parameters. As with iterators, the upper bound is exclusive and doesn't belong to the range.

`boost::log::expressions::format_date_time()` is called in the format function. It is used to format a timepoint. Example 62.7 uses `boost::log::expressions::format_date_time()` to write the time without a date. You can also use manipulators from the standard library in format functions. Example 62.7 uses `std::setw()` to set the width for the counter.

Example 62.8. Several loggers, front-ends, and back-ends

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/channel_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/utility/string_literal.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <string>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(channel, "Channel", std::string)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend>
    ostream_sink;
  boost::shared_ptr<ostream_sink> ostream =
    boost::make_shared<ostream_sink>();
  boost::shared_ptr<std::ostream> clog{&std::clog,
    boost::empty_deleter{}};
  ostream->locked_backend()->add_stream(clog);
  core::get()->add_sink(ostream);
```

```
  typedef sinks::synchronous_sink<sinks::text_multifile_backend>
    multifile_sink;
  boost::shared_ptr<multifile_sink> multifile =
    boost::make_shared<multifile_sink>();
  multifile->locked_backend()->set_file_name_composer(
    sinks::file::as_file_name_composer(expressions::stream <<
    channel.or_default<std::string>("None") << "-" <<
    severity.or_default(0) << ".log"));
  core::get()->add_sink(multifile);

  sources::severity_logger<int> severity_lg;
  sources::channel_logger<> channel_lg{keywords::channel = "Main"};

  BOOST_LOG_SEV(severity_lg, 1) << "severity message";
  BOOST_LOG(channel_lg) << "channel message";
  ostream->flush();
}
```

Example 62.8 uses several loggers, front-ends, and back-ends. In addition to using the classes `boost::log::sinks::asynchronous_sink`, `boost::log::sinks::text_ostream_backend` and `boost::log::sources::severity_logger`, the example also uses the front-end `boost::log::sinks::synchronous_sink`, the back-end `boost::log::sinks::text_multifile_backend`, and the logger `boost::log::sources::channel_logger`.

The front-end `boost::log::sinks::synchronous_sink` provides synchronous access to a back-end, which lets you use a back-end in a multithreaded application even if the back-end isn't thread safe.

The difference between the two front-ends `boost::log::sinks::asynchronous_sink` and `boost::log::sinks::synchronous_sink` is that the latter isn't based on a thread. Log entries are passed to the back-end in the same thread.

Example 62.8 uses the front-end `boost::log::sinks::synchronous_sink` with the back-end `boost::log::sinks::text_multifile_backend`. This back-end writes log entries to one or more files. File names are created according to a rule passed by `set_file_name_composer()` to the back-end. If you use the free-standing function `boost::log::sinks::file::as_file_name_composer()`, as in the example, the rule can be created as a lambda function with the same building blocks used for format functions. However, the attributes aren't used to create the string that is written to a back-end. Instead, the string will be the name of the file that log entries will be written to.

Example 62.8 uses the keywords **`channel`** and **`severity`**, which are defined with the macro `BOOST_LOG_ATTRIBUTE_KEYWORD`. They refer to the attributes Channel and Severity. The member function `or_default()` is called on the keywords to pass a default value if an attribute isn't set. If a log entry is written and Channel and Severity are not set, the entry is written to the file `None-0.log`. If a log entry is written with the log level 1, it is stored in the file `None-1.log`. If the log level is 1 and the channel is called Main, the log entry is saved in the file `Main-1.log`.

The attribute Channel is defined by the logger `boost::log::sources::channel_logger`. The constructor expects a channel name. The name can't be passed directly as a string. Instead, it must be passed as a named parameter. That's why the example uses `keywords::channel =`

"Main" even though boost::log::sources::channel_logger doesn't accept any other parameters.

Please note that the named parameter boost::log::keywords::channel has nothing to do with the keywords you create with the macro BOOST_LOG_ATTRIBUTE_KEYWORD.

boost::log::sources::channel_logger identifies log entries from different components of a program. Components can use their own objects of type boost::log::sources::channel_logger, giving them unique names. If components only access their own loggers, it's clear which component a particular log entry came from.

Example 62.9. Handling exceptions centrally

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/log/utility/exception_handler.hpp>
#include <boost/log/exceptions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

struct handler
{
  void operator()(const runtime_error &ex) const
  {
    std::cerr << "boost::log::runtime_error: " << ex.what() << '\n';
  }

  void operator()(const std::exception &ex) const
  {
    std::cerr << "std::exception: " << ex.what() << '\n';
  }
};

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);
  core::get()->set_exception_handler(
    make_exception_handler<runtime_error, std::exception>(handler{}));

  sources::logger lg;

  BOOST_LOG(lg) << "note";
}
```

Boost.Log provides the option to handle exceptions in the logging framework centrally. This means you don't need to wrap every BOOST_LOG in a try block to handle exceptions in catch.

Example 62.9 calls the member function set_exception_handler(). The core provides this member function to register a handler. All exceptions in the logging framework will be passed to

that handler. The handler is implemented as a function object. It has to overload `operator()` for every exception type expected. An instance of that function object is passed to `set_exception_handler()` through the function template `boost::log::make_exception_handler()`. All exception types you want to handle must be passed as template parameters to `boost::log::make_exception_handler()`.

The function `boost::log::make_exception_suppressor()` let's you discard all exceptions in the logging framework. You call this function instead of `boost::log::make_exception_handler()`.

Example 62.10. A macro to define a global logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(lg, sources::wlogger_mt)

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  BOOST_LOG(lg::get()) << L"note";
}
```

All of the examples in this chapter use local loggers. If you want to define a global logger, use the macro `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT` as in Example 62.10. You pass the name of the logger as the first parameter and the type as the second. You don't access the logger through its name. Instead, you call `get()`, which returns a pointer to a singleton.

Boost.Log provides additional macros such as `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS`. They let you initialize global loggers. `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` lets you pass parameters to the constructor of a global logger. All of these macros guarantee that global loggers will be correctly initialized.

Boost.Log provides many more functions that are worth a look. For example, you can configure the logging framework through a container with key/value pairs as strings. Then, you don't need to instantiate classes and call member functions. For example, a key Destination can be set to Console, which will automatically make the logging framework use the back-end `boost::log::sinks::text_ostream_backend`. The back-end can be configured through additional key/value pairs. Because the container can also be serialized in an INI-file, it is possible to store the configuration in a text file and initialize the logging framework with that file.