Boost.Log v2

Andrey Semashev

Table of Contents

Motivation

Today applications grow rapidly, becoming complicated and difficult to test and debug. Most of the time applications run on a remote site, leaving the developer little chance to monitor their execution and figure out the reasons for their failure, once it should happen. Moreover, even the local debugging may become problematic if the application behavior depends heavily on asynchronous side events, such as a device feedback or another process activity.

This is where logging can help. The application stores all essential information about its execution to a log, and when something goes wrong this information can be used to analyze the program behavior and make the necessary corrections. There are other very useful applications of logging, such as gathering statistical information and highlighting events (i.e. indicating that some situation has occurred or that the application is experiencing some problems). These tasks have proved to be vital for many real-world industrial applications.

This library aims to make logging significantly easier for the application developer. It provides a wide range of out-of-the-box tools along with public interfaces for extending the library. The main goals of the library are:

- Simplicity. A small example code snippet should be enough to get the feel of the library and be ready to use its basic features.
- Extensibility. A user should be able to extend functionality of the library for collecting and storing information into logs.
- Performance. The library should have as little performance impact on the user's application as possible.

How to read the documentation

The documentation is oriented to both new and experienced library users. However, users are expected to be familiar with commonly used Boost components, such as `shared_ptr`, `make_shared` (see Boost.SmartPtr), and `function` (Boost.Function). Some parts of the documentation will refer to other Boost libraries, as needed.

If this is your first experience with the library, it is recommended to read the Design overview section for a first glance at the library's capabilities and architecture. The Installation and Tutorial sections will help to get started experimenting with the library. The tutorial gives an overview of the library features with sample code snippets. Some tutorial steps are presented in two forms: simple and advanced. The simple form typically describes the most common and easy way to do the task and it is being recommended to be read by new users. The advanced form usually gives an expanded way to do the same thing but with an in-depth explanation and the ability to do some extra customization. This form may come in handy for more experienced users and should generally be read if the easy way does not satisfy your needs.

Besides the tutorial there is a Detailed features description chapter. This part describes other tools provided by the library that were not covered by the tutorial. This chapter is best read on a case by case basis.

Last, but not least, there is a reference which gives the formal description of library components.

To keep the code snippets in this documentation simple, the following namespace aliases are assumed to be defined:

```
namespace logging = boost::log;
namespace sinks = boost::log::sinks;
namespace src = boost::log::sources;
namespace expr = boost::log::expressions;
namespace attrs = boost::log::attributes;
namespace keywords = boost::log::keywords;
```

Installation and compatibility

Supported compilers and platforms

The library should build and work with a reasonably compliant compiler. The library was successfully built and tested on the following platforms:

- Windows XP, Windows Vista, Windows 7. MSVC 8.0 SP1, MSVC 9.0 and newer.
- Linux. GCC 4.5 and newer. Older versions may work too, but it was not tested.
- Linux. Intel C++ 13.1.0.146 Build 20130121.
- Linux. Clang 3.2.

The following compilers/platforms are not supported and will likely fail to compile the library:

- C++11 compilers with non-C++11 standard libraries (like Clang with libstdc++ from GCC 4.2). Please, use a C++11 standard library in C++11 mode.
- MSVC 8.0 (without SP1) and older.
- GCC 4.2 and older.
- Borland C++ 5.5.1 (free version). Newer versions might or might not work.
- Solaris Studio 12.3 and older.
- Windows 9x, ME, NT4 and older are not supported.

Boost.Log should be compatible with all hardware architectures supported by Boost. However, in case of 32 bit x86 architecture the library requires at least i586 class CPU to run.

Notes for GCC users

GCC versions since 4.5 support link time optimization (LTO), when most of optimizations and binary code generation happens at linking stage. This allows to perform more advanced optimizations and produce faster code. Unfortunately, it does not play well with projects containing source files that need to be compiled with different compiler options. Boost.Log is one of such projects, some parts of its sources contain optimizations for modern CPUs and will not run on older CPUs. Enabling LTO for Boost.Log will produce binaries incompatible with older CPUs (GCC bug), which will likely result in crashes in run time. For this reason GCC LTO is not supported in Boost.Log.

There is a GCC bug which may cause compilation failures when `-march=native` command line argument is used. It is recommended to avoid using `-march=native` argument (or `instruction-set=native` bjam property) and instead explicitly specify the target CPU (e.g. `instruction-set=sandy-bridge`).

Notes for MinGW, Cygwin and Visual Studio Express Edition users

In order to compile the library with these compilers special preparations are needed. First, in case of MinGW or Cygwin make sure you have installed the latest GCC version. The library will most likely fail to compile with GCC 3.x.

Second, at some point the library will require a Message Compiler tool (`mc.exe`), which is not available in MinGW, Cygwin and some versions of MSVC Express Edition. Typically the library build scripts will automatically detect if message compiler is present on the system and disable Event log related portion of the library if it's not. If Event log support is required and it is not found on the system, you have three options to settle the problem. The recommended solution is to obtain the original `mc.exe`. This tool is available in Windows SDK, which can be downloaded from the Microsoft site freely (for example, here). Also, this tool should be available in Visual Studio 2010 Express Edition. During the compilation, `mc.exe` should be accessible through one of the directories in your `PATH` environment variable.

Another way is to attempt to use the `windmc.exe` tool distributed with MinGW and Cygwin, which is the analogue of the original `mc.exe`. In order to do that you will have to patch Boost.Build files (in particular, the `tools/build/tools/mc.jam` file) as described in this ticket. After that you will be able to specify the `mc-compiler=windmc` option to bjam to build the library.

In case if message compiler detection fails for some reason, you can explicitly disable support for event log backend by defining the `BOOST_LOG_WITHOUT_EVENT_LOG` configuration macro when building the library. This will remove the need for the message compiler. See this section for more configuration options.

MinGW users on Windows XP may be affected by the bug in msvcrt.dll that is bundled with the operating system. The bug manifests itself as crashes while the library formats log records. This problem is not specific to Boost.Log and may also show in different contexts related to locale and IO-streams management.

Additional notes for Cygwin users

Cygwin support is very preliminary. The default GCC version available in Cygwin (4.5.3 as of this writing) is unable to compile the library because of compiler errors. You will have to build a newer GCC from sources. Even then some Boost.Log functionality is not available. In particular, the socket-based syslog backend is not supported, as it is based on Boost.ASIO, which doesn't compile on this platform. However, the native syslog support is still in place.

Configuring and building the library

The library has a separately compiled part which should be built as described in the Getting Started guide. One thing should be noted, though. If your application consists of more than one module (e.g. an exe and one or several dll's) that use Boost.Log, the library must be built as a shared object. If you have a single executable or a single module that works with Boost.Log, you may build the library as a static library.

The library supports a number of configuration macros:

Table 1. Configuration macros

Macro name	Effect
`BOOST_LOG_DYN_LINK`	If defined in user code, the library will assume the binary is built as a dynamically loaded library ("dll" or "so"). Otherwise it is assumed that the library is built in static mode. This macro must be either defined or not defined for all translation units of user application that uses logging. This macro can help with auto-linking on platforms that support it.
`BOOST_ALL_DYN_LINK`	Same as `BOOST_LOG_DYN_LINK` but also affects other Boost libraries the same way.
`BOOST_LOG_NO_THREADS`	If defined, disables multithreading support. Affects the compilation of both the library and users' code. The macro is automatically defined if no threading support is detected.
`BOOST_LOG_WITHOUT_CHAR`	If defined, disables support for narrow character logging. Affects the compilation of both the library and users' code.
`BOOST_LOG_WITHOUT_WCHAR_T`	If defined, disables support for wide character logging. Affects the compilation of both the library and users' code.
`BOOST_LOG_NO_QUERY_PERFORMANCE_COUNTER`	This macro is only useful on Windows. It affects the compilation of both the library and users' code. If defined, disables support for the `QueryPerformanceCounter` API in the `timer` attribute. This will result in significantly less accurate time readings. The macro is intended to solve possible problems with earlier revisions of AMD Athlon CPU, described here and here. There are also known chipset hardware failures that may prevent this API from functioning properly (see here).
`BOOST_LOG_USE_NATIVE_SYSLOG`	Affects only compilation of the library. If for some reason support for the native SysLog API is not detected automatically, define this macro to forcibly enable it
`BOOST_LOG_WITHOUT_DEFAULT_FACTORIES`	Affects only compilation of the library. If defined, the parsers for settings will be built without any default factories for filters and formatters. The user will have to register all attributes in the library before parsing any filters or formatters from strings. This can substantially reduce the binary size.
`BOOST_LOG_WITHOUT_SETTINGS_PARSERS`	Affects only compilation of the library. If defined, none of the facilities related to the parsers for settings will be built. This can substantially reduce the binary size.
`BOOST_LOG_WITHOUT_DEBUG_OUTPUT`	Affects only compilation of the library. If defined, the support for debugger output on Windows will not be built.
`BOOST_LOG_WITHOUT_EVENT_LOG`	Affects only compilation of the library. If defined, the support for Windows event log will not be built. Defining the macro also makes Message Compiler toolset unnecessary.
`BOOST_LOG_WITHOUT_SYSLOG`	Affects only compilation of the library. If defined, the support for syslog backend will not be built.

Macro name	Effect
`BOOST_LOG_NO_SHORTHAND_NAMES`	Affects only compilation of users' code. If defined, some deprecated shorthand macro names will not be available.
`BOOST_LOG_USE_WINNT6_API`	Affects compilation of both the library and users' code. This macro is Windows-specific. If defined, the library makes use of the Windows NT 6 (Vista, Server 2008) and later APIs to generate more efficient code. This macro will also enable some experimental features of the library. Note, however, that the resulting binary will not run on Windows prior to NT 6. In order to use this feature Platform SDK 6.0 or later is required.
`BOOST_LOG_USE_COMPILER_TLS`	Affects only compilation of the library. This macro enables support for compiler intrinsics for thread-local storage. Defining it may improve performance of Boost.Log if certain usage limitations are acceptable. See below for more comments.
`BOOST_LOG_USE_STD_REGEX`, `BOOST_LOG_USE_BOOST_REGEX` or `BOOST_LOG_USE_BOOST_XPRESSIVE`	Affects only compilation of the library. By defining one of these macros the user can instruct Boost.Log to use `std::regex`, Boost.Regex or Boost.Xpressive internally for string matching filters parsed from strings and settings. If none of these macros is defined then Boost.Log uses Boost.Regex by default. Using `std::regex` or Boost.Regex typically produces smaller executables, Boost.Regex usually also being the fastest in run time. Using Boost.Xpressive allows to eliminate the dependency on Boost.Regex compiled binary. Note that these macros do not affect filtering expressions created by users.

You can define configuration macros in the `bjam` command line, like this:

```
bjam --with-log variant=release define=BOOST_LOG_WITHOUT_EVENT_LOG define=BOOST_LOG_USE_WIN↵
NT6_API stage
```

However, it may be more convenient to define configuration macros in the "boost/config/user.hpp" file in order to automatically define them both for the library and user's projects. If none of the options are specified, the library will try to support the most comprehensive setup, including support for all character types and features available for the target platform.

The logging library uses several other Boost libraries that require building too. These are Boost.Filesystem, Boost.System, Boost.DateTime, Boost.Thread and in some configurations Boost.Regex. Refer to their documentation for detailed instructions on the building procedure.

One final thing should be added. The library requires run-time type information (RTTI) to be enabled for both the library compilation and user's code compilation. Normally, this won't need anything from you except to verify that RTTI support is not disabled in your project.

Notes about compiler-supplied intrinsics for TLS

Many widely used compilers support builtin intrinsics for managing thread-local storage, which is used in several parts of the library. This feature is also included in the C++11 standard. Generally, these intrinsics allow for a much more efficient access to the storage than any surrogate implementation, be that Boost.Thread or even native operating system API. However, this feature has several caveats:

- Some operating systems don't support the use of these intrinsics in case if the TLS is defined in a shared library that is dynamically loaded during the application run time. These systems include Linux and Windows prior to Vista. Windows Vista and later do not have this issue.

- The TLS may not be reliably accessed from global constructors and destructors. At least MSVC 8.0 on Windows is known to have this problem.

The library provides the `BOOST_LOG_USE_COMPILER_TLS` configuration macro that allows to enable the use of this feature, which will improve the library performance at the cost of these limitations:

- The application executable must be linked with the Boost.Log library. It should not be loaded dynamically during run time.
- The application must not use logging in global constructors or destructors.

Definitions

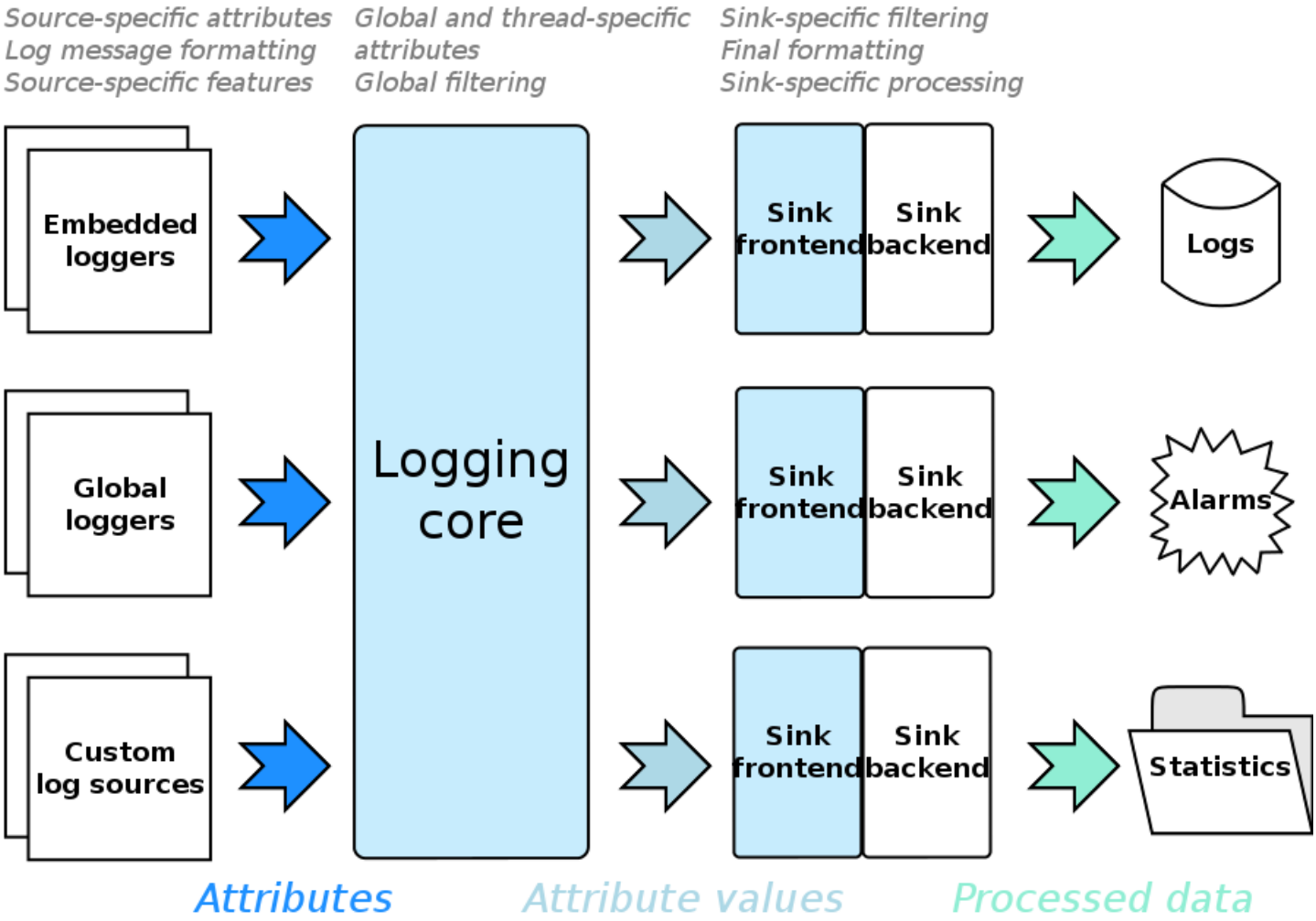
Here are definitions of some terms that will be used widely throughout the documentation:

Log record	A single bundle of information, collected from the user's application, that is a candidate to be put into the log. In a simple case the log record will be represented as a line of text in the log file after being processed by the logging library.
Attribute	An "attribute" is a piece of meta-information that can be used to specialize a log record. In Boost.Log attributes are represented by function objects with a specific interface, which return the actual attribute value when invoked.
Attribute value	Attribute values are the actual data acquired from attributes. This data is attached to the specific log record and processed by the library. Values can have different types (integers, strings and more complex, including user defined types). Some examples of attribute values: current time stamp value, file name, line number, current scope name, etc.. Attribute values are enveloped in a type erasing wrapper, so the actual type of the attribute is not visible in the interface. The actual (erased) type of the value is sometimes called the stored type.
(Attribute) value visitation	A way of processing the attribute value. This approach involves a function object (a visitor) which is applied to the attribute value. The visitor should know the stored type of the attribute value in order to process it.
(Attribute) value extraction	A way of processing the attribute value when the caller attempts to obtain a reference to the stored value. The caller should know the stored type of the attribute value in order to be able to extract it.
Log sink	A target, to which all log records are fed after being collected from the user's application. The sink defines where and how the log records are going to be stored or processed.
Log source	An entry point for the user's application to put log records to. In a simple case it is an object (logger) which maintains a set of attributes that will be used to form a log record upon the user's request. However, one can surely create a source that would emit log records on some side events (for example, by intercepting and parsing console output of another application).
Log filter	A predicate that takes a log record and tells whether this record should be passed through or discarded. The predicate typically forms its decision based on the attribute values attached to the record.
Log formatter	A function object that generates the final textual output from a log record. Some sinks, e.g. a binary logging sink, may not need it, although almost any text-based sink would use a formatter to compose its output.
Logging core	The global entity that maintains connections between sources and sinks and applies filters to records. It is mainly used when the logging library is initialized.
i18n	Internationalization. The ability to manipulate wide characters.
TLS	Thread-local storage. The concept of having a variable that has independent values for each thread that attempts to access it.
RTTI	Run-time type information. This is the C++ language support data structures required for `dynamic_cast` and `typeid` operators to function properly.

Design overview

Boost.Log was designed to be very modular and extensible. It supports both narrow-character and wide-character logging. Both narrow and wide-character loggers provide similar capabilities, so through most of the documentation only the narrow-character interface will be described.

The library consists of three main layers: the layer of log data collection, the layer of processing the collected data and the central hub that interconnects the former two layers. The design is presented on the figure below.



The arrows show the direction of logging information flow - from parts of your application, at the left, to the final storage, (if any) at the right. The storage is optional because the result of log processing may include some actions without actually storing the information anywhere. For example, if your application is in a critical state, it can emit a special log record that will be processed so that the user sees an error message as a tool-tip notification over the application icon in the system tray and hears an alarming sound. This is a very important library feature: it is orthogonal to collecting, processing logging data and, in fact, what data logging records consist of. This allows for use of the library not only for classic logging, but to indicate some important events to the application user and accumulate statistical data.

Logging sources

Getting back to the figure, in the left side your application emits log records with help of loggers - special objects that provide streams to format messages that will eventually be put to log. The library provides a number of different logger types and you can craft many more yourself, extending the existing ones. Loggers are designed as a mixture of distinct features that can be combined with each other in any combination. You can simply develop your own feature and add it to the soup. You will be able to use the constructed logger just like the others - embed it into your application classes or create and use a global instance of the logger. Either approach

provides its benefits. Embedding a logger into some class provides a way to differentiate logs from different instances of the class. On the other hand, in functional-style programming it is usually more convenient to have a single global logger somewhere and have a simple access to it.

Generally speaking, the library does not require the use of loggers to write logs. The more generic term "log source" designates the entity that initiates logging by constructing a log record. Other log sources might include captured console output of a child application or data received from network. However, loggers are the most common kind of log sources.

Attributes and attribute values

In order to initiate logging a log source must pass all data, associated with the log record, to the logging core. This data or, more precisely, the logic of the data acquisition is represented with a set of named attributes. Each attribute is, basically, a function, whose result is called "attribute value" and is actually processed on further stages. An example of an attribute is a function that returns the current time. Its result - the particular time point - is the attribute value.

There are three kinds of attribute sets:

- global
- thread-specific
- source-specific

You can see in the figure that the former two sets are maintained by the logging core and thus need not be passed by the log source in order to initiate logging. Attributes that participate in the global attribute set are attached to any log record ever made. Obviously, thread-specific attributes are attached only to the records made from the thread in which they were registered in the set. The source-specific attribute set is maintained by the source that initiates logging, these attributes are attached only to the records being made through that particular source.

When a source initiates logging, attribute values are acquired from attributes of all three attribute sets. These attribute values then form a single set of named attribute values, which is processed further. You can add more attribute values to the set; these values will only be attached to the particular log record and will not be associated with the logging source or logging core. As you may notice, it is possible for a same-named attribute to appear in several attribute sets. Such conflicts are solved on priority basis: global attributes have the least priority, source-specific attributes have the highest; the lower priority attributes are discarded from consideration in case of conflicts.

Logging core and filtering

When the set of attribute values is composed, the logging core decides if this log record is going to be processed in sinks. This is called filtering. There are two layers of filtering available: the global filtering is applied first within the logging core itself and allows quickly wiping away unneeded log records; the sink-specific filtering is applied second, for each sink separately. The sink-specific filtering allows directing log records to particular sinks. Note that at this point it is not significant which logging source emitted the record, the filtering relies solely on the set of attribute values attached to the record.

It must be mentioned that for a given log record filtering is performed only once. Obviously, only those attribute values attached to the record before filtering starts can participate in filtering. Some attribute values, like log record message, are typically attached to the record after the filtering is done; such values cannot be used in filters, they can only be used by formatters and sinks themselves.

Sinks and formatting

If a log record passes filtering for at least one sink the record is considered to be consumable. If the sink supports formatted output, this is the point when log message formatting takes place. The formatted message along with the composed set of attribute values is passed to the sink that accepted the record. Note that formatting is performed on the per-sink basis so that each sink can output log records in its own specific format.

As you may have noticed on the figure above, sinks consist of two parts: the frontend and the backend. This division is made in order to extract the common functionality of sinks, such as filtering, formatting and thread synchronization, into separate entities (frontends). Sink frontends are provided by the library, most likely users won't have to re-implement them. Backends, on the other hand, are one of the most likely places for extending the library. It is sink backends that do the actual processing of log records. There can be a

sink that stores log records into a file; there can be a sink that sends log records over the network to the remote log processing node; there can be the aforementioned sink that puts record messages into tool-tip notifications - you name it. The most commonly used sink backends are already provided by the library.

Along with the primary facilities described above, the library provides a wide variety of auxiliary tools, such as attributes, support for formatters and filters, represented as lambda expressions, and even basic helpers for the library initialization. You will find their description in the Detailed features description section. However, for new users it is recommended to start discovering the library from the Tutorial section.

Tutorial

In this section we shall walk through the essential steps to get started with the library. After reading it you should be able to initialize the library and add logging to your application. The code of this tutorial is also available in examples residing in the `libs/log/examples` directory. Feel free to play with them, compile and see the result.

Trivial logging

For those who don't want to read tons of clever manuals and just need a simple tool for logging, here you go:

```
#include <boost/log/trivial.hpp>

int main(int, char*[])
{
    BOOST_LOG_TRIVIAL(trace) << "A trace severity message";
    BOOST_LOG_TRIVIAL(debug) << "A debug severity message";
    BOOST_LOG_TRIVIAL(info) << "An informational severity message";
    BOOST_LOG_TRIVIAL(warning) << "A warning severity message";
    BOOST_LOG_TRIVIAL(error) << "An error severity message";
    BOOST_LOG_TRIVIAL(fatal) << "A fatal severity message";

    return 0;
}
```

See the complete code.

The `BOOST_LOG_TRIVIAL` macro accepts a severity level and results in a stream-like object that supports insertion operator. As a result of this code, the log messages will be printed on the console. As you can see, this library usage pattern is quite similar to what you would do with `std::cout`. However, the library offers a few advantages:

1. Besides the record message, each log record in the output contains a timestamp, the current thread identifier and severity level.

2. It is safe to write logs from different threads concurrently, log messages will not be corrupted.

3. As will be shown later, filtering can be applied.

It must be said that the macro, along with other similar macros provided by the library, is not the only interface the library offers. It is possible to issue log records without using any macros at all.

Trivial logging with filters

While severity levels can be used for informative purposes, you will normally want to apply filters to output only significant records and ignore the rest. It is easy to do so by setting a global filter in the library core, like this:

```
void init()
{
    logging::core::get()->set_filter
    (
        logging::trivial::severity >= logging::trivial::info
    );
}

int main(int, char*[])
{
    init();

    BOOST_LOG_TRIVIAL(trace) << "A trace severity message";
    BOOST_LOG_TRIVIAL(debug) << "A debug severity message";
    BOOST_LOG_TRIVIAL(info) << "An informational severity message";
    BOOST_LOG_TRIVIAL(warning) << "A warning severity message";
    BOOST_LOG_TRIVIAL(error) << "An error severity message";
    BOOST_LOG_TRIVIAL(fatal) << "A fatal severity message";

    return 0;
}
```

See the complete code.

Now, if we run this code sample, the first two log records will be ignored, while the remaining four will pass on to the console.

Important

Remember that the streaming expression is only executed if the record passed filtering. Don't specify business-critical calls in the streaming expression, as these calls may not get invoked if the record is filtered away.

A few words must be said about the filter setup expression. Since we're setting up a global filter, we have to acquire the logging core instance. This is what `logging::core::get()` does - it returns a pointer to the core singleton. The `set_filter` method of the logging core sets the global filtering function.

The filter in this example is built as a Boost.Phoenix lambda expression. In our case, this expression consists of a single logical predicate, whose left argument is a placeholder that describes the attribute to be checked, and the right argument is the value to be checked against. The `severity` keyword is a placeholder provided by the library. This placeholder identifies the severity attribute value in the template expressions; this value is expected to have name "Severity" and type `severity_level`. This attribute is automatically provided by the library in case of trivial logging; the user only has to supply its value in logging statements. The placeholder along with the ordering operator creates a function object that will be called by the logging core to filter log records. As a result, only log records with severity level not less than `info` will pass the filter and end up on the console.

It is possible to build more complex filters, combining logical predicates like this with each other, or even define your own function (including a C++11 lambda function) that would act as a filter. We will return to filtering in the following sections.

Setting up sinks

Sometimes trivial logging doesn't provide enough flexibility. For example, one may want a more sophisticated logic of log processing, rather than simply printing it on the console. In order to customize this, you have to construct logging sinks and register them with the logging core. This should normally be done only once somewhere in the startup code of your application.

Note

It must be mentioned that in the previous sections we did not initialize any sinks, and trivial logging worked somehow anyway. This is because the library contains a *default* sink that is used as a fallback when the user did not set up any sinks. This sink always prints log records to the console in a fixed format which we saw in our previous examples. The default sink is mostly provided to allow trivial logging to be used right away, without any library initialization whatsoever. Once you add any sinks to the logging core, the default sink will no longer be used. You will still be able to use trivial logging macros though.

File logging unleashed

As a starting point, here is how you would initialize logging to a file:

```
void init()
{
    logging::add_file_log("sample.log");

    logging::core::get()->set_filter
    (
        logging::trivial::severity >= logging::trivial::info
    );
}
```

The added piece is the call to the `add_file_log` function. As the name implies, the function initializes a logging sink that stores log records into a text file. The function also accepts a number of customization options, such as the file rotation interval and size limits. For instance:

```
void init()
{
    logging::add_file_log
    (
        keywords::file_name = "sample_%N.log",                                        ❶
        keywords::rotation_size = 10 * 1024 * 1024,                                   ❷
        keywords::time_based_rotation = sinks::file::rotation_at_time_point(0, 0, 0), ❸
        keywords::format = "[%TimeStamp%]: %Message%"                                 ❹
    );

    logging::core::get()->set_filter
    (
        logging::trivial::severity >= logging::trivial::info
    );
}
```

❶ file name pattern
❷ rotate files every 10 MiB...
❸ ...or at midnight
❹ log record format

See the complete code.

You can see that the options are passed to the function in the named form. This approach is also taken in many other places of the library. You'll get used to it. The meaning of the parameters is mostly self-explaining and is documented in this manual (see here for what regards the text file sink). This and other convenience initialization functions are described in this section.

Note

You can register more than one sink. Each sink will receive and process log records as you emit them independently from others.

Sinks in depth: More sinks

If you don't want to go into details, you can skip this section and continue reading from the next one. Otherwise, if you need more comprehensive control over sink configuration or want to use more sinks than those available through helper functions, you can register sinks manually.

In the simplest form, the call to the `add_file_log` function in the section above is nearly equivalent to this:

```
void init()
{
    // Construct the sink
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > text_sink;
    boost::shared_ptr< text_sink > sink = boost::make_shared< text_sink >();

    // Add a stream to write log to
    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("sample.log"));

    // Register the sink in the logging core
    logging::core::get()->add_sink(sink);
}
```

See the complete code.

Ok, the first thing you may have noticed about sinks is that they are composed of two classes: the frontend and the backend. The frontend (which is the `synchronous_sink` class template in the snippet above) is responsible for various common tasks for all sinks, such as thread synchronization model, filtering and, for text-based sinks, formatting. The backend (the `text_ostream_backend` class above) implements everything specific to the sink, such as writing to a file in this case. The library provides a number of frontends and backends that can be used with each other out of the box.

The `synchronous_sink` class template above indicates that the sink is synchronous, that is, it allows for several threads to log simultaneously and will block in case of contention. This means that the backend `text_ostream_backend` doesn't have to worry about multithreading at all. There are other sink frontends available, you can read more about them here.

The `text_ostream_backend` class writes formatted log records into STL-compatible streams. We have used a file stream above but we could have used any type of stream. For example, adding output to console could look as follows:

```
#include <boost/core/null_deleter.hpp>

// We have to provide an empty deleter to avoid destroying the global stream object
boost::shared_ptr< std::ostream > stream(&std::clog, boost::null_deleter());
sink->locked_backend()->add_stream(stream);
```

The `text_ostream_backend` supports adding several streams. In that case its output will be duplicated to all added streams. It can be useful to duplicate the output to console and file since all the filtering, formatting and other overhead of the library happen only once per record for the sink.

Note

Please note the difference between registering several distinct sinks and registering one sink with several target streams. While the former allows for independently customizing output to each sink, the latter would work considerably faster if such customization is not needed. This feature is specific to this particular backend.

The library provides a number of backends that provide different log processing logic. For instance, by specifying the syslog backend you can send log records over the network to the syslog server, or by setting up the Windows NT event log backend you can monitor your application run time with the standard Windows tools.

The last thing worth noting here is the `locked_backend` member function call to access the sink backend. It is used to get thread-safe access to the backend and is provided by all sink frontends. This function returns a smart-pointer to the backend and as long as it exists the backend is locked (which means even if another thread tries to log and the log record is passed to the sink, it will not be logged until you release the backend). The only exception is the `unlocked_sink` frontend which does not synchronize at all and simply returns an unlocked pointer to the backend.

Creating loggers and writing logs

Dedicated logger objects

Now that we have defined where and how the log is to be stored, it's time to go on and try logging. In order to do this one has to create a logging source. This would be a logger object in our case and it is as simple as that:

```
src::logger lg;
```

Note

A curious reader could have noticed that we did not create any loggers for trivial logging. In fact the logger is provided by the library and is used behind the scenes.

Unlike sinks, sources need not be registered anywhere since they interact directly with the logging core. Also note that there are two versions of loggers provided by the library: the thread-safe ones and the non-thread-safe ones. For the non-thread-safe loggers it is safe for different threads to write logs through different instances of loggers and thus there should be a separate logger for each thread that writes logs. The thread-safe counterparts can be accessed from different threads concurrently, but this will involve locking and may slow things down in case of intense logging. The thread-safe logger types have the `_mt` suffix in their name.

Regardless of the thread safety, all loggers provided by the library are default and copy-constructible and support swapping, so there should be no problem in making a logger a member of your class. As you will see later, such approach can give you additional benefits.

The library provides a number of loggers with different features, such as severity and channel support. These features can be combined with each other in order to construct more complex loggers. See here for more details.

Global logger objects

In case you cannot put a logger into your class (suppose you don't have one), the library provides a way of declaring global loggers like this:

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(my_logger, src::logger_mt)
```

Here `my_logger` is a user-defined tag name that will be used later to retrieve the logger instance and `logger_mt` is the logger type. Any logger type provided by the library or defined by the user can participate in such declaration. However, since the logger will have a single instance, you will normally want to use thread-safe loggers in a multithreaded application as global ones.

Tip

There are other macros for more sophisticated cases available. The detailed description is in this section.

Later on you can acquire the logger like this:

```
src::logger_mt& lg = my_logger::get();
```

The `lg` will refer to the one and only instance of the logger throughout the application, even if the application consists of multiple modules. The `get` function itself is thread-safe, so there is no need in additional synchronization around it.

Writing logs

No matter what kind of logger you use (class member or global, thread-safe or not), to write a log record into a logger you can write something like this:

```
logging::record rec = lg.open_record();
if (rec)
{
    logging::record_ostream strm(rec);
    strm << "Hello, World!";
    strm.flush();
    lg.push_record(boost::move(rec));
}
```

Here the `open_record` function call determines if the record to be constructed is going to be consumed by at least one sink. Filtering is applied at this stage. If the record is to be consumed, the function returns a valid record object, and one can fill in the record message string. After that the record processing can be completed with the call to `push_record`.

Of course, the above syntax can easily be wrapped in a macro and, in fact, users are encouraged to write their own macros instead of using the C++ logger interface directly. The log record above can be written like this:

```
BOOST_LOG(lg) << "Hello, World!";
```

Looks a bit shorter, doesn't it? The `BOOST_LOG` macro, along with other similar ones, is defined by the library. It automatically provides an STL-like stream in order to format the message with ordinary insertion expressions. Having all that code written, compiled and executed you should be able to see the "Hello, World!" record in the "sample.log" file. You will find the full code of this section here.

Adding more information to log: Attributes

In previous sections we mentioned attributes and attribute values several times. Here we will discover how attributes can be used to add more data to log records.

Each log record can have a number of named attribute values attached. Attributes can represent any essential information about the conditions in which the log record occurred, such as position in the code, executable module name, current date and time, or any piece of data relevant to your particular application and execution environment. An attribute may behave as a value generator, in which case it would return a different value for each log record it's involved in. As soon as the attribute generates the value, the latter becomes independent from the creator and can be used by filters, formatters and sinks. But in order to use the attribute value one has to know its name and type, or at least a set of types it may have. There are a number of commonly used attributes implemented in the library, you can find the types of their values in the documentation.

Aside from that, as described in the Design overview section, there are three possible scopes of attributes: source-specific, thread-specific and global. When a log record is made, attribute values from these three sets are joined into a single set and passed to sinks. This implies that the origin of the attribute makes no difference for sinks. Any attribute can be registered in any scope. When registered, an attribute is given a unique name in order to make it possible to search for it. If it happens that the same named attribute is found in several scopes, the attribute from the most specific scope is taken into consideration in any further processing, including filtering and formatting. Such behavior makes it possible to override global or thread-scoped attributes with the ones registered in your local logger, thus reducing thread interference.

Below is the description of the attribute registration process.

Commonly used attributes

There are attributes that are likely to be used in nearly any application. Log record counter and a time stamp are good candidates. They can be added with a single function call:

```
logging::add_common_attributes();
```

With this call attributes "LineID", "TimeStamp", "ProcessID" and "ThreadID" are registered globally. The "LineID" attribute is a counter that increments for each record being made, the first record gets identifier 1. The "TimeStamp" attribute always yields the current time (i.e. the time when the log record is created, not the time it was written to a sink). The last two attributes identify the process and the thread in which every log record is emitted.

Note

In single-threaded builds the "ThreadID" attribute is not registered.

Tip

By default, when application starts, no attributes are registered in the library. The application has to register all the necessary attributes in the library before it starts writing logs. This can be done as a part of the library initialization. A curious reader could have wondered how trivial logging works then. The answer is that the default sink doesn't really use any attribute values, except for the severity level, to compose its output. This is done to avoid the need for any initialization for trivial logging. Once you use filters or formatters and non-default sinks you will have to register the attributes you need.

The `add_common_attributes` function is one of the several convenience helpers described here.

Some attributes are registered automatically on loggers construction. For example, `severity_logger` registers a source-specific attribute "Severity" which can be used to add a level of emphasis for different log records. For example:

```
// We define our own severity levels
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

void logging_function()
{
    // The logger implicitly adds a source-specific attribute 'Severity'
    // of type 'severity_level' on construction
    src::severity_logger< severity_level > slg;

    BOOST_LOG_SEV(slg, normal) << "A regular message";
    BOOST_LOG_SEV(slg, warning) << "Something bad is going on but I can handle it";
    BOOST_LOG_SEV(slg, critical) << "Everything crumbles, shoot me now!";
}
```

Tip

You can define your own formatting rules for the severity level by defining `operator <<` for this type. It will be automatically used by the library formatters. See this section for more details.

The `BOOST_LOG_SEV` macro acts pretty much like `BOOST_LOG` except that it takes an additional argument for the `open_record` method of the logger. The `BOOST_LOG_SEV` macro can be replaced with this equivalent:

```
void manual_logging()
{
    src::severity_logger< severity_level > slg;

    logging::record rec = slg.open_record(keywords::severity = normal);
    if (rec)
    {
        logging::record_ostream strm(rec);
        strm << "A regular message";
        strm.flush();
        slg.push_record(boost::move(rec));
    }
}
```

You can see here that the `open_record` can take named arguments. Some logger types provided by the library have support for such additional parameters and this approach can certainly be used by users when writing their own loggers.

More attributes

Let's see what's under the hood of that `add_common_attributes` function we used in the simple form section. It might look something like this:

```
void add_common_attributes()
{
    boost::shared_ptr< logging::core > core = logging::core::get();
    core->add_global_attribute("LineID", attrs::counter< unsigned int >(1));
    core->add_global_attribute("TimeStamp", attrs::local_clock());

    // other attributes skipped for brevity
}
```

Here the `counter` and `local_clock` components are attribute classes, they derive from the common attribute interface `attribute`. The library provides a number of other attribute classes, including the `function` attribute that calls some function object on value acquisition. For example, we can in a similar way register a `named_scope` attribute:

```
core->add_global_attribute("Scope", attrs::named_scope());
```

This will give us the ability to store scope names in log for every log record the application makes. Here is how it's used:

```
void named_scope_logging()
{
    BOOST_LOG_NAMED_SCOPE("named_scope_logging");

    src::severity_logger< severity_level > slg;

    BOOST_LOG_SEV(slg, normal) << "Hello from the function named_scope_logging!";
}
```

Logger-specific attributes are no less useful than global ones. Severity levels and channel names are the most obvious candidates to be implemented on the source level. Nothing prevents you from adding more attributes to your loggers, like this:

```
void tagged_logging()
{
    src::severity_logger< severity_level > slg;
    slg.add_attribute("Tag", attrs::constant< std::string >("My tag value"));

    BOOST_LOG_SEV(slg, normal) << "Here goes the tagged record";
}
```

Now all log records made through this logger will be tagged with the specific attribute. This attribute value may be used later in filtering and formatting.

Another good use of attributes is the ability to mark log records made by different parts of application in order to highlight activity related to a single process. One can even implement a rough profiling tool to detect performance bottlenecks. For example:

```
void timed_logging()
{
    BOOST_LOG_SCOPED_THREAD_ATTR("Timeline", attrs::timer());

    src::severity_logger< severity_level > slg;
    BOOST_LOG_SEV(slg, normal) << "Starting to time nested functions";

    logging_function();

    BOOST_LOG_SEV(slg, normal) << "Stopping to time nested functions";
}
```

Now every log record made from the `logging_function` function, or any other function it calls, will contain the "Timeline" attribute with a high precision time duration passed since the attribute was registered. Based on these readings, one will be able to detect which parts of the code require more or less time to execute. The "Timeline" attribute will be unregistered upon leaving the scope of function `timed_logging`.

See the Attributes section for detailed description of attributes provided by the library. The complete code for this section is available here.

Defining attribute placeholders

As we will see in the coming sections, it is useful to define a keyword describing a particular attribute the application uses. This keyword will be able to participate in filtering and formatting expressions, like the `severity` placeholder we have used in previous sections. For example, to define placeholders for some of the attributes we used in the previous examples we can write this:

```
BOOST_LOG_ATTRIBUTE_KEYWORD(line_id, "LineID", unsigned int)
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)
BOOST_LOG_ATTRIBUTE_KEYWORD(tag_attr, "Tag", std::string)
BOOST_LOG_ATTRIBUTE_KEYWORD(scope, "Scope", attrs::named_scope::value_type)
BOOST_LOG_ATTRIBUTE_KEYWORD(timeline, "Timeline", attrs::timer::value_type)
```

Each macro defines a keyword. The first argument is the placeholder name, the second is the attribute name and the last parameter is the attribute value type. Once defined, the placeholder can be used in template expressions and some other contexts of the library. More details on defining attribute keywords are available here.

Log record formatting

If you tried running examples from the previous sections, you may have noticed that only log record messages are written to the files. This is the default behavior of the library when no formatter is set. Even if you added attributes to the logging core or a logger, the attribute values will not reach the output unless you specify a formatter that will use these values.

Returning to one of the examples in previous tutorial sections:

```
void init()
{
    logging::add_file_log
    (
        keywords::file_name = "sample_%N.log",
        keywords::rotation_size = 10 * 1024 * 1024,
        keywords::time_based_rotation = sinks::file::rotation_at_time_point(0, 0, 0),
        keywords::format = "[%TimeStamp%]: %Message%"
    );

    logging::core::get()->set_filter
    (
        logging::trivial::severity >= logging::trivial::info
    );
}
```

In the case of the `add_file_log` function, the `format` parameter allows to specify format of the log records. If you prefer to set up sinks manually, sink frontends provide the `set_formatter` member function for this purpose.

The format can be specified in a number of ways, as described further.

Lambda-style formatters

You can create a formatter with a lambda-style expression like this:

```
void init()
{
    logging::add_file_log
    (
        keywords::file_name = "sample_%N.log",
        // This makes the sink to write log records that look like this:
        // 1: <normal> A normal severity message
        // 2: <error> An error severity message
        keywords::format =
        (
            expr::stream
                << expr::attr< unsigned int >("LineID")
                << ": <" << logging::trivial::severity
                << "> " << expr::smessage
        )
    );
}
```

See the complete code.

Here the `stream` is a placeholder for the stream to format the record in. Other insertion arguments, such as `attr` and `message`, are manipulators that define what should be stored in the stream. We have already seen the `severity` placeholder in filtering expressions, and here it is used in a formatter. This is a nice unification: you can use the same placeholders in both filters and formatters. The `attr` placeholder is similar to the `severity` placeholder as it represents the attribute value, too. The difference is that the `severity` placeholder represents the particular attribute with the name "Severity" and type `trivial::severity_level` and `attr` can be used to represent any attribute. Otherwise the two placeholders are equivalent. For instance, it is possible to replace `severity` with the following:

```
expr::attr< logging::trivial::severity_level >("Severity")
```

Tip

As shown in the previous section, it is possible to define placeholders like `severity` for user's attributes. As an additional benefit to the simpler syntax in the template expressions such placeholders allow to concentrate all the information about the attribute (the name and the value type) in the placeholder definition. This makes coding less error-prone (you won't misspell the attribute name or specify incorrect value type) and therefore is the recommended way of defining new attributes and using them in template expressions.

There are other formatter manipulators that provide advanced support for date, time and other types. Some manipulators accept additional arguments that customize their behavior. Most of these arguments are named and can be passed in Boost.Parameter style.

For a change, let's see how it's done when manually initializing sinks:

```
void init()
{
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > text_sink;
    boost::shared_ptr< text_sink > sink = boost::make_shared< text_sink >();

    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("sample.log"));

    sink->set_formatter
    (
        expr::stream
               // line id will be written in hex, 8-digits, zero-filled
            << std::hex << std::setw(8) << std::setfill('0') << expr::attr< un↵
signed int >("LineID")
            << ": <" << logging::trivial::severity
            << "> " << expr::smessage
    );

    logging::core::get()->add_sink(sink);
}
```

See the complete code.

You can see that it is possible to bind format changing manipulators in the expression; these manipulators will affect the subsequent attribute value format when log record is formatted, just like with streams. More manipulators are described in the Detailed features description section.

Boost.Format-style formatters

As an alternative, you can define formatters with a syntax similar to Boost.Format. The same formatter as described above can be written as follows:

```
void init()
{
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > text_sink;
    boost::shared_ptr< text_sink > sink = boost::make_shared< text_sink >();

    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("sample.log"));

    // This makes the sink to write log records that look like this:
    // 1: <normal> A normal severity message
    // 2: <error> An error severity message
    sink->set_formatter
    (
        expr::format("%1%: <%2%> %3%")
            % expr::attr< unsigned int >("LineID")
            % logging::trivial::severity
            % expr::smessage
    );

    logging::core::get()->add_sink(sink);
}
```

See the complete code.

The `format` placeholder accepts the format string with positional specification of all arguments being formatted. Note that only positional format is currently supported. The same format specification can be used with the `add_file_log` and similar functions.

Specialized formatters

The library provides specialized formatters for a number of types, such as date, time and named scope. These formatters provide extended control over the formatted values. For example, it is possible to describe date and time format with a format string compatible with Boost.DateTime:

```
void init()
{
    logging::add_file_log
    (
        keywords::file_name = "sample_%N.log",
        // This makes the sink to write log records that look like this:
        // YYYY-MM-DD HH:MI:SS: <normal> A normal severity message
        // YYYY-MM-DD HH:MI:SS: <error> An error severity message
        keywords::format =
        (
            expr::stream
                << expr::format_date_time< boost::posix_time::ptime >("TimeStamp", "%Y-%m-%d ↵
%H:%M:%S")
                << ": <" << logging::trivial::severity
                << "> " << expr::smessage
        )
    );
}
```

See the complete code.

The same formatter can also be used in the context of a Boost.Format-style formatter.

String templates as formatters

In some contexts textual templates are accepted as formatters. In this case library initialization support code is invoked in order to parse the template and reconstruct the appropriate formatter. There are a number of caveats to keep in mind when using this approach, but here it will suffice to just briefly describe the template format.

```
void init()
{
    logging::add_file_log
    (
        keywords::file_name = "sample_%N.log",
        keywords::format = "[%TimeStamp%]: %Message%"
    );
}
```

See the complete code.

Here, the `format` parameter accepts such a format template. The template may contain a number of placeholders enclosed with percent signs (%). Each placeholder must contain an attribute value name to insert instead of the placeholder. The `%Message%` placeholder will be replaced with the logging record message.

Note

Textual format templates are not accepted by sink backends in the `set_formatter` method. In order to parse textual template into a formatter function one has to call `parse_formatter` function. See here for more details.

Custom formatting functions

You can add a custom formatter to a sink backend that supports formatting. The formatter is actually a function object that supports the following signature:

```
void (logging::record_view const& rec, logging::basic_formatting_ostream< CharT >& strm);
```

Here `CharT` is the target character type. The formatter will be invoked whenever a log record view `rec` passes filtering and is to be stored in log.

Tip

Record views are very similar to records. The notable distinction is that the view is immutable and implements shallow copy. Formatters and sinks only operate on record views, which prevents them from modifying the record while it can be still in use by other sinks in other threads.

The formatted record should be composed by insertion into STL-compatible output stream `strm`. Here's an example of a custom formatter function usage:

```
void my_formatter(logging::record_view const& rec, logging::formatting_ostream& strm)
{
    // Get the LineID attribute value and put it into the stream
    strm << logging::extract< unsigned int >("LineID", rec) << ": ";

    // The same for the severity level.
    // The simplified syntax is possible if attribute keywords are used.
    strm << "<" << rec[logging::trivial::severity] << "> ";

    // Finally, put the record message to the stream
    strm << rec[expr::smessage];
}

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

    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("sample.log"));

    sink->set_formatter(&my_formatter);

    logging::core::get()->add_sink(sink);
}
```

See the complete code.

Filtering revisited

We've already touched filtering in the previous sections but we barely scratched the surface. Now that we are able to add attributes to log records and set up sinks, we can build however complex filtering we need. Let's consider this example:

```
BOOST_LOG_ATTRIBUTE_KEYWORD(line_id, "LineID", unsigned int)
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)
BOOST_LOG_ATTRIBUTE_KEYWORD(tag_attr, "Tag", std::string)

void init()
{
    // Setup the common formatter for all sinks
    logging::formatter fmt = expr::stream
        << std::setw(6) << std::setfill('0') << line_id << std::setfill(' ')
        << ": <" << severity << ">\t"
        << expr::if_(expr::has_attr(tag_attr))
           [
               expr::stream << "[" << tag_attr << "] "
           ]
        << expr::smessage;

    // Initialize sinks
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > text_sink;
    boost::shared_ptr< text_sink > sink = boost::make_shared< text_sink >();

    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("full.log"));

    sink->set_formatter(fmt);

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

    sink = boost::make_shared< text_sink >();

    sink->locked_backend()->add_stream(
        boost::make_shared< std::ofstream >("important.log"));

    sink->set_formatter(fmt);

    sink->set_filter(severity >= warning || (expr::has_attr(tag_attr) && tag_attr == "IMPORT↵
ANT_MESSAGE"));

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

    // Add attributes
    logging::add_common_attributes();
}
```

See the complete code.

In this sample we initialize two sinks - one for the complete log file and the other for important messages only. Both sinks will be writing to text files with the same log record format, which we initialize first and save to the `fmt` variable. The `formatter` type is a type-erased function object with the formatter calling signature; in many respects it can be viewed similar to `boost::function` or `std::function` except that it is never empty. There is also `a similar function object` for filters.

Notably, the formatter itself contains a filter here. As you can see, the format contains a conditional part that is only present when log records contain the "Tag" attribute. The `has_attr` predicate checks whether the record contains the "Tag" attribute value and controls whether it is put into the file or not. We used the attribute keyword to specify the name and type of the attribute for the predicate, but it is also possible to specify them in the `has_attr` call site. Conditional formatters are explained in more details here.

Further goes the initialization of the two sinks. The first sink does not have any filter, which means it will save every log record to the file. We call `set_filter` on the second sink to only save log records with severity no less than `warning` or having a "Tag" attribute with value "IMPORTANT_MESSAGE". As you can see, the filter syntax resembles usual C++ very much, especially when attribute keywords are used.

Like with formatters, it is also possible to use custom functions as filters. Boost.Phoenix can be very helpful in this case as its `bind` implementation is compatible with attribute placeholders. The previous example can be modified in the following way:

```
bool my_filter(logging::value_ref< severity_level, tag::severity > const& level,
               logging::value_ref< std::string, tag::tag_attr > const& tag)
{
    return level >= warning || tag == "IMPORTANT_MESSAGE";
}

void init()
{
    // ...

    namespace phoenix = boost::phoenix;
    sink->set_filter(phoenix::bind(&my_filter, severity.or_none(), tag_attr.or_none()));

    // ...
}
```

As you can see, the custom formatter receives attribute values wrapped into the `value_ref` template. This wrapper contains an optional reference to the attribute value of the specified type; the reference is valid if the log record contains the attribute value of the required type. The relational operators used in `my_filter` can be applied unconditionally because they will automatically return `false` if the reference is not valid. The rest is done with the `bind` expression which will recognize the `severity` and `tag_attr` keywords and extract the corresponding values before passing them to `my_filter`.

Note

Because of limitations related to the integration with Boost.Phoenix (see #7996), it is required to explicitly specify the fallback policy in case if the attribute value is missing, when attribute keywords are used with `phoenix::bind` or `phoenix::function`. In the example above, this is done by calling `or_none`, which results in an empty `value_ref` if the value is not found. In other contexts this policy is the default. There are other policies that can be used instead.

You can try how this works by compiling and running the test.

Wide character logging

The library supports logging strings containing national characters. There are basically two ways of doing this. On UNIX-like systems typically some multibyte character encoding (e.g. UTF-8) is used to represent national characters. In this case the library can be used just the way it is used for plain ASCII logging, no additional setup is required.

On Windows the common practice is to use wide strings to represent national characters. Also, most of the system API is wide character oriented, which requires Windows-specific sinks to also support wide strings. On the other hand, generic sinks, like the text file sink, are byte-oriented (because, well, you store bytes in files, not characters). This forces the library to perform character code conversion when needed by the sink. To set up the library for this one has to imbue the sink with a locale with the appropriate `codecvt` facet. Boost.Locale can be used to generate such a locale. Let's see an example:

```
// Declare attribute keywords
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "TimeStamp", boost::posix_time::ptime)

void init_logging()
{
    boost::shared_ptr< sinks::synchronous_sink< sinks::text_file_backend > > sink = log↵
ging::add_file_log
    (
        "sample.log",
        keywords::format = expr::stream
            << expr::format_date_time(timestamp, "%Y-%m-%d, %H:%M:%S.%f")
            << " <" << severity.or_default(normal)
            << "> " << expr::message
    );

    // The sink will perform character code conversion as needed, according to the locale set ↵
with imbue()
    std::locale loc = boost::locale::generator()("en_US.UTF-8");
    sink->imbue(loc);

    // Let's add some commonly used attributes, like timestamp and record counter.
    logging::add_common_attributes();
}
```

First let's take a look at the formatter we pass in the `format` parameter. We initialize the sink with a narrow-character formatter because the text file sink processes bytes. It is possible to use wide strings in the formatter, but not in format strings, like the one we used with the `format_date_time` function. Also note that we used `message` keyword to denote the log record messages. This placeholder supports both narrow and wide character messages, so the formatter will work with both. As part of the formatting process, the library will convert wide character messages to multibyte encoding using the imbued locale, which we set to UTF-8.

Tip

Attribute values can also contain wide strings. Like log record messages, these strings will be converted with the imbued locale to the target character encoding.

One thing missing here is our `severity_level` type definition. The type is just an enumeration, but if we want to support its formatting for both narrow and wide character sinks, its streaming operator has to be a template. This may be useful if we create multiple sinks with different character types.

```
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

template< typename CharT, typename TraitsT >
inline std::basic_ostream< CharT, TraitsT >& operator<< (
    std::basic_ostream< CharT, TraitsT >& strm, severity_level lvl)
{
    static const char* const str[] =
    {
        "normal",
        "notification",
        "warning",
        "error",
        "critical"
    };
    if (static_cast< std::size_t >(lvl) < (sizeof(str) / sizeof(*str)))
        strm << str[lvl];
    else
        strm << static_cast< int >(lvl);
    return strm;
}
```

Now we can emit log records. We can use loggers with `w` prefix in their names to compose wide character messages.

```
void test_narrow_char_logging()
{
    // Narrow character logging still works
    src::logger lg;
    BOOST_LOG(lg) << "Hello, World! This is a narrow character message.";
}

void test_wide_char_logging()
{
    src::wlogger lg;
    BOOST_LOG(lg) << L"Hello, World! This is a wide character message.";

    // National characters are also supported
   const wchar_t national_chars[] = { 0x041f, 0x0440, 0x0438, 0x0432, 0x0435, 0x0442, L',', L' ↵
', 0x043c, 0x0438, 0x0440, L'!', 0 };
    BOOST_LOG(lg) << national_chars;

    // Now, let's try logging with severity
    src::wseverity_logger< severity_level > slg;
    BOOST_LOG_SEV(slg, normal) << L"A normal severity message, will not pass to the file";
    BOOST_LOG_SEV(slg, warning) << L"A warning severity message, will pass to the file";
    BOOST_LOG_SEV(slg, error) << L"An error severity message, will pass to the file";
}
```

As you can see, wide character message composition is similar to narrow logging. Note that you can use both narrow and wide character logging at the same time; all records will be processed by our file sink. The complete code of this example can be found here.

It must be noted that some sinks (mostly, Windows-specific ones) allow to specify the target character type. When national characters are expected in log records, one should always use `wchar_t` as the target character type in these cases because the sink will use

wide character OS API to process log records. In this case all narrow character strings will be widened using the locale imbued into the sink when formatting is performed.

Detailed features description

Core facilities

Logging records

```
#include <boost/log/core/record.hpp>
```

All the information that the logging library processes is packed into a single object of type `record`. All attached data, including the message text, is represented as named attribute values that can be fetched and processed by filters, formatters and sinks. Particular attribute values can be accessed in different ways, here are a few quick examples:

- Through value visitation and extraction.

```
enum severity_level { ... };
std::ostream& operator<< (std::ostream& strm, severity_level level);

struct print_visitor
{
    typedef void result_type;
    result_type operator() (severity_level level) const
    {
        std::cout << level << std::endl;
    };
};

// Prints severity level through visitation API
void print_severity_visitation(logging::record const& rec)
{
    logging::visit< severity_level >("Severity", rec, print_visitor());
}

// Prints severity level through extraction API
void print_severity_extraction(logging::record const& rec)
{
    logging::value_ref< severity_level > level = logging::extract< severity_level >("Sever↵
ity", rec);
    std::cout << level << std::endl;
}
```

- By searching the set of attribute values accessible with the `attribute_values` method of the record.

```
// Prints severity level by searching the attribute values
void print_severity_lookup(logging::record const& rec)
{
    logging::attribute_value_set const& values = rec.attribute_values();
    logging::attribute_value_set::const_iterator it = values.find("Severity");
    if (it != values.end())
    {
        logging::attribute_value const& value = it->second;

        // A single attribute value can also be visited or extracted
        std::cout << value.extract< severity_level >() << std::endl;
    }
}
```

- By applying the subscript operator with the attribute keyword. This is actually a convenience wrapper around the value extraction API.

```
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)

// Prints severity level by using the subscript operator
void print_severity_subscript(logging::record const& rec)
{
    // Use the attribute keyword to communicate the name and type of the value
    logging::value_ref< severity_level, tag::severity > level = rec[severity];
    std::cout << level << std::endl;
}
```

Log records cannot be copied, only moved. A record can be default-constructed in which case it is in an empty state; such records are mostly unusable and should not be passed to the library for processing. Non-empty log records can only be created by the logging core as a result of successful filtering. The non-empty record contains attribute values acquired from attributes. More attribute values can be added to the non-empty record after filtering. The added values will not affect filtering results but can still be used by formatters and sinks.

In multithreaded environments, after being constructed a non-empty log record is considered to be tied to the current thread as it may refer to some thread-specific resources. For example, the record may contain an attribute value which refers to the named scope list which is stored on the stack. For this reason log records must not be passed between different threads.

Record views

```
#include <boost/log/core/record_view.hpp>
```

While records are used for filling the information, the library uses another type to actually process it. Record views provide a similar interface to records with a few notable distinctions:

- Record views are immutable. This prevents formatters and sinks from modifying the record while it is being processed.
- Record views are copyable. Since its contents are constant, the copy operation is shallow and therefore cheap.

The library will automatically create a record view from the record by calling the `lock` method. The call will also make sure the resulting view is not attached to the current thread if a sink is asynchronous. The `lock` call is a one time operation; the record is left in the empty state afterwards. All APIs for interacting with attribute values described for log records are also applicable to record views and can be used in custom formatters and sinks.

Logging core

```
#include <boost/log/core/core.hpp>
```

The logging core is a central hub that provides the following facilities:

- Maintains global and thread-specific attribute sets.
- Performs global filtering of log records.
- Dispatches log records between sinks by applying sink-specific filters.
- Provides a global hook for exception handlers.
- Provides an entry point for log sources to put log records to.
- Provides the `flush` method that can be used to enforce the synchronized state for all log sinks.

The logging core is an application-wide singleton, thus every logging source has access to it. The core instance is accessible with the static method `get`.

```
void foo()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // ...
}
```

Attribute sets

In order to add or remove global or thread-specific attributes to the core there are corresponding methods: `add_global_attribute`, `remove_global_attribute`, `add_thread_attribute` and `remove_thread_attribute`. Attribute sets provide interface similar to `std::map`, so the `add_*` methods accept an attribute name string (key) and a pointer to the attribute (mapped value) and return a pair of iterator and boolean value, like `std::map< ... >::insert` does. The `remove_*` methods accept an iterator to a previously added attribute.

```
void foo()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Add a global attribute
    std::pair< logging::attribute_set::iterator, bool > res =
        core->add_global_attribute("LineID", attrs::counter< unsigned int >());

    // ...

    // Remove the added attribute
    core->remove_global_attribute(res.first);
}
```

Tip

It must be said that all methods of logging core are thread-safe in multithreaded environments. However, that may not be true for other components, such as iterators or attribute sets.

It is possible to acquire a copy of the whole attribute set (global or thread-specific) or install it into the core. Methods `get_global_attributes`, `set_global_attributes`, `get_thread_attributes` and `set_thread_attributes` serve this purpose.

Warning

After installing a whole attribute set into the core, all iterators that were previously returned by the corresponding `add_*` methods are invalidated. In particular, it affects scoped attributes, so the user must be careful when to switch attribute sets.

Global filtering

Global filtering is handled by the filter function object, which can be provided with the `set_filter` method. More on creating filters appears in this section. Here it will suffice to say that the filter accepts a set of attribute values and returns a boolean value that tells whether a log record with these attribute values passed filtering or not. The global filter is applied to every log record made throughout the application, so it can be used to wipe out excessive log records quickly.

The global filter can be removed by the `reset_filter` method. When there is no filter set in the core it is assumed that no records are filtered away. This is the default after initial construction of the logging core.

```
enum severity_level
{
    normal,
    warning,
    error,
    critical
};

void foo()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Set a global filter so that only error messages are logged
    core->set_filter(expr::attr< severity_level >("Severity") >= error);

    // ...
}
```

The core also provides another way to disable logging. By calling the `set_logging_enabled` with a boolean argument one may completely disable or re-enable logging, including applying filtering. Disabling logging with this method may be more beneficial in terms of application performance than setting a global filter that always fails.

Sink management

After global filtering is applied, log sinks step into action. In order to add and remove sinks the core provides `add_sink` and `remove_sink` methods. Both these methods accept a pointer to the sink. The `add_sink` will add the sink to the core if it's not added already. The `remove_sink` method will seek for the provided sink in an internal list of previously added sinks and remove the sink if it finds it. The order in which the core processes sinks internally is unspecified.

```
void foo()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Set a sink that will write log records to the console
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));

    typedef sinks::unlocked_sink< sinks::text_ostream_backend > sink_t;
    boost::shared_ptr< sink_t > sink = boost::make_shared< sink_t >(backend);
    core->add_sink(sink);

    // ...

    // Remove the sink
    core->remove_sink(sink);
}
```

You can read more on the design of sinks in the following sections: Sink Frontends and Sink Backends.

Exception handling

The core provides a way to set up centralized exception handling. If an exception takes place during filtering or processing in one of the added sinks, the core will invoke an exception handler if one was installed with the `set_exception_handler` method. An exception handler is a nullary function object that is invoked from within a `catch` clause. The library provides tools to simplify exception handlers construction.

Tip

The exception handler in the logging core is global and thus is intended to perform some common actions on errors. Logging sinks and sources also provide exception handling facilities (see here and here), which can be used to do a finer grained error processing.

```
struct my_handler
{
    typedef void result_type;

    void operator() (std::runtime_error const& e) const
    {
        std::cout << "std::runtime_error: " << e.what() << std::endl;
    }
    void operator() (std::logic_error const& e) const
    {
        std::cout << "std::logic_error: " << e.what() << std::endl;
        throw;
    }
};

void init_exception_handler()
{
    // Setup a global exception handler that will call my_handler::operator()
    // for the specified exception types
    logging::core::get()->set_exception_handler(logging::make_exception_handler<
        std::runtime_error,
        std::logic_error
    >(my_handler()));
}
```

Feeding log records

One of the most important functions of the logging core is providing an entry point for all logging sources to feed log records into. This is done with the `open_record` and `push_record` methods.

The first method is used to initiate the record logging process. It accepts the source-specific set of attributes. The method constructs a common set of attribute values of the three sets of attributes (global, thread-specific and source-specific) and applies filtering. If the filtering succeeded, i.e. at least one sink accepts a record with these attribute values, the method returns a non-empty record object, which can be used to fill in the log record message. If the filtering failed, an empty record object is returned.

When the log source is ready to complete the logging procedure, it has to call the `push_record` method with the record returned by the `open_record` method. Note that one should not call `push_record` with an empty record. The record should be passed as rvalue reference. During the call the record view will be constructed from the record. The view will then be passed on to the sinks that accepted it during filtering. This may involve record formatting and further processing, like storing it into a file or sending it over the network. After that the record object can be destroyed.

```
void logging_function(logging::attribute_set const& attrs)
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Attempt to open a log record
    logging::record rec = core->open_record(attrs);
    if (rec)
    {
        // Ok, the record is accepted. Compose the message now.
        logging::record_ostream strm(rec);
        strm << "Hello, World!";
        strm.flush();

        // Deliver the record to the sinks.
        core->push_record(boost::move(rec));
    }
}
```

All this logic is usually hidden in the loggers and macros provided by the library. However, this may be useful for those developing new log sources.

Logging sources

Basic loggers

```
#include <boost/log/sources/basic_logger.hpp>
```

The simplest logging sources provided by the library are loggers `logger` and its thread-safe version, `logger_mt` (`wlogger` and `wlogger_mt` for wide-character logging, accordingly). These loggers only provide the ability to store source-specific attributes within themselves and, of course, to form log records. This type of logger should probably be used when there is no need for advanced features like severity level checks. It may well be used as a tool to collect application statistics and register application events, such as notifications and alarms. In such cases the logger is normally used in conjunction with scoped attributes to attach the needed data to the notification event. Below is an example of usage:

```
class network_connection
{
    src::logger m_logger;
    logging::attribute_set::iterator m_remote_addr;

public:
    void on_connected(std::string const& remote_addr)
    {
        // Put the remote address into the logger to automatically attach it
        // to every log record written through the logger
        m_remote_addr = m_logger.add_attribute("RemoteAddress",
            attrs::constant< std::string >(remote_addr)).first;

        // The straightforward way of logging
        if (logging::record rec = m_logger.open_record())
        {
            rec.attribute_values().insert("Message",
                attrs::make_attribute_value(std::string("Connection established")));
            m_logger.push_record(boost::move(rec));
        }
    }
    void on_disconnected()
    {
        // The simpler way of logging: the above "if" condition is wrapped into a neat macro
        BOOST_LOG(m_logger) << "Connection shut down";

        // Remove the attribute with the remote address
        m_logger.remove_attribute(m_remote_addr);
    }
    void on_data_received(std::size_t size)
    {
        // Put the size as an additional attribute
        // so it can be collected and accumulated later if needed.
        // The attribute will be attached only to this log record.
       BOOST_LOG(m_logger) << logging::add_value("ReceivedSize", size) << "Some data received";
    }
    void on_data_sent(std::size_t size)
    {
        BOOST_LOG(m_logger) << logging::add_value("SentSize", size) << "Some data sent";
    }
};
```

The class `network_connection` in the code snippet above represents an approach to implementing simple logging and statistical information gathering in a network-related application. Each of the presented methods of the class effectively marks a corresponding event that can be tracked and collected on the sinks level. Furthermore, other methods of the class, that are not shown here for simplicity, are able to write logs too. Note that every log record ever made in the connected state of the `network_connection` object will be implicitly marked up with the address of the remote site.

Loggers with severity level support

```
#include <boost/log/sources/severity_feature.hpp>
#include <boost/log/sources/severity_logger.hpp>
```

The ability to distinguish some log records from others based on some kind of level of severity or importance is one of the most frequently requested features. The class templates `severity_logger` and `severity_logger_mt` (along with their `wseverity_logger` and `wseverity_logger_mt` wide-character counterparts) provide this functionality.

The loggers automatically register a special source-specific attribute "Severity", which can be set for every record in a compact and efficient manner, with a named argument `severity` that can be passed to the constructor and/or the `open_record` method. If passed to the logger constructor, the `severity` argument sets the default value of the severity level that will be used if none is provided in

the `open_record` arguments. The `severity` argument passed to the `open_record` method sets the level of the particular log record being made. The type of the severity level can be provided as a template argument for the logger class template. The default type is `int`.

The actual values of this attribute and their meaning are entirely user-defined. However, it is recommended to use the level of value equivalent to zero as a base point for other values. This is because the default-constructed logger object sets its default severity level to zero. It is also recommended to define the same levels of severity for the entire application in order to avoid confusion in the written logs later. The following code snippet shows the usage of `severity_logger`.

```
// We define our own severity levels
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

void logging_function()
{
    // The logger implicitly adds a source-specific attribute 'Severity'
    // of type 'severity_level' on construction
    src::severity_logger< severity_level > slg;

    BOOST_LOG_SEV(slg, normal) << "A regular message";
    BOOST_LOG_SEV(slg, warning) << "Something bad is going on but I can handle it";
    BOOST_LOG_SEV(slg, critical) << "Everything crumbles, shoot me now!";
}
```

```
void default_severity()
{
    // The default severity can be specified in constructor.
    src::severity_logger< severity_level > error_lg(keywords::severity = error);

    BOOST_LOG(error_lg) << "An error level log record (by default)";

    // The explicitly specified level overrides the default
    BOOST_LOG_SEV(error_lg, warning) << "A warning level log record (overrode the default)";
}
```

Or, if you prefer logging without macros:

```
void manual_logging()
{
    src::severity_logger< severity_level > slg;

    logging::record rec = slg.open_record(keywords::severity = normal);
    if (rec)
    {
        logging::record_ostream strm(rec);
        strm << "A regular message";
        strm.flush();
        slg.push_record(boost::move(rec));
    }
}
```

And, of course, severity loggers also provide the same functionality the basic loggers do.

Loggers with channel support

```
#include <boost/log/sources/channel_feature.hpp>
#include <boost/log/sources/channel_logger.hpp>
```

Sometimes it is important to associate log records with some application component, such as the module or class name, the relation of the logged information to some specific domain of application functionality (e.g. network or file system related messages) or some arbitrary tag that can be used later to route these records to a specific sink. This feature is fulfilled with loggers `channel_logger`, `channel_logger_mt` and their wide-char counterparts `wchannel_logger`, `wchannel_logger_mt`. These loggers automatically register an attribute named "Channel". The default channel name can be set in the logger constructor with a named argument `channel`. The type of the channel attribute value can be specified as a template argument for the logger, with `std::string` (`std::wstring` in case of wide character loggers) as a default. Aside from that, the usage is similar to the basic loggers:

```
class network_connection
{
    src::channel_logger< > m_net, m_stat;
    logging::attribute_set::iterator m_net_remote_addr, m_stat_remote_addr;

public:
    network_connection() :
        // We can dump network-related messages through this logger
        // and be able to filter them later
        m_net(keywords::channel = "net"),
        // We also can separate statistic records in a different channel
        // in order to route them to a different sink
        m_stat(keywords::channel = "stat")
    {
    }

    void on_connected(std::string const& remote_addr)
    {
        // Add the remote address to both channels
        attrs::constant< std::string > addr(remote_addr);
        m_net_remote_addr = m_net.add_attribute("RemoteAddress", addr).first;
        m_stat_remote_addr = m_stat.add_attribute("RemoteAddress", addr).first;

        // Put message to the "net" channel
        BOOST_LOG(m_net) << "Connection established";
    }

    void on_disconnected()
    {
        // Put message to the "net" channel
        BOOST_LOG(m_net) << "Connection shut down";

        // Remove the attribute with the remote address
        m_net.remove_attribute(m_net_remote_addr);
        m_stat.remove_attribute(m_stat_remote_addr);
    }

    void on_data_received(std::size_t size)
    {
        BOOST_LOG(m_stat) << logging::add_value("ReceivedSize", size) << "Some data received";
    }

    void on_data_sent(std::size_t size)
    {
        BOOST_LOG(m_stat) << logging::add_value("SentSize", size) << "Some data sent";
    }
};
```

It is also possible to set the channel name of individual log records. This can be useful when a global logger is used instead of an object-specific one. The channel name can be set by calling the `channel` modifier on the logger or by using a special macro for logging. For example:

```
// Define a global logger
BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS(my_logger, src::channel_logger_mt< >, (keywords::chan↵
nel = "general"))

class network_connection
{
    std::string m_remote_addr;

public:
    void on_connected(std::string const& remote_addr)
    {
        m_remote_addr = remote_addr;

        // Put message to the "net" channel
        BOOST_LOG_CHANNEL(my_logger::get(), "net")
            << logging::add_value("RemoteAddress", m_remote_addr)
            << "Connection established";
    }

    void on_disconnected()
    {
        // Put message to the "net" channel
        BOOST_LOG_CHANNEL(my_logger::get(), "net")
            << logging::add_value("RemoteAddress", m_remote_addr)
            << "Connection shut down";

        m_remote_addr.clear();
    }

    void on_data_received(std::size_t size)
    {
        BOOST_LOG_CHANNEL(my_logger::get(), "stat")
            << logging::add_value("RemoteAddress", m_remote_addr)
            << logging::add_value("ReceivedSize", size)
            << "Some data received";
    }

    void on_data_sent(std::size_t size)
    {
        BOOST_LOG_CHANNEL(my_logger::get(), "stat")
            << logging::add_value("RemoteAddress", m_remote_addr)
            << logging::add_value("SentSize", size)
            << "Some data sent";
    }
};
```

Note that changing the channel name is persistent, so unless the channel name is reset, the subsequent records will also belong to the new channel.

Tip

For performance reasons it is advised to avoid dynamically setting the channel name individually for every log record, when possible. Changing the channel name involves dynamic memory allocation. Using distinct loggers for different channels allows to avoid this overhead.

Loggers with exception handling support

```
#include <boost/log/sources/exception_handler_feature.hpp>
```

The library provides a logger feature that enables the user to handle and/or suppress exceptions at the logger level. The `exception_handler` feature adds a `set_exception_handler` method to the logger that allows setting a function object to be called if an exception is thrown from the logging core during the filtering or processing of log records. One can use the library-provided adapters to simplify implementing exception handlers. Usage example is as follows:

```
enum severity_level
{
    normal,
    warning,
    error
};

// A logger class that allows to intercept exceptions and supports severity level
class my_logger_mt :
    public src::basic_composite_logger<
        char,
        my_logger_mt,
        src::multi_thread_model< boost::shared_mutex >,
        src::features<
            src::severity< severity_level >,
            src::exception_handler
        >
    >
{
    BOOST_LOG_FORWARD_LOGGER_MEMBERS(my_logger_mt)
};

BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(my_logger, my_logger_mt)
{
    my_logger_mt lg;

    // Set up exception handler: all exceptions that occur while
    // logging through this logger, will be suppressed
    lg.set_exception_handler(logging::make_exception_suppressor());

    return lg;
}

void logging_function()
{
    // This will not throw
    BOOST_LOG_SEV(my_logger::get(), normal) << "Hello, world";
}
```

Tip

Logging core and sink frontends also support installing exception handlers.

Loggers with mixed features

```
#include <boost/log/sources/severity_channel_logger.hpp>
```

If you wonder whether you can use a mixed set of several logger features in one logger, then yes, you certainly can. The library provides `severity_channel_logger` and `severity_channel_logger_mt` (with their wide-char analogues `wseverity_channel_logger` and `wseverity_channel_logger_mt`) which combine features of the described loggers with severity level and channels support. The composite loggers are templates, too, which allows you to specify severity level and channel types. You can also design your own logger features and combine them with the ones provided by the library, as described in the Extending the library section.

The usage of the loggers with several features does not conceptually differ from the usage of the single-featured loggers. For instance, here is how a `severity_channel_logger_mt` could be used:

```
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

typedef src::severity_channel_logger_mt<
    severity_level,     // the type of the severity level
    std::string         // the type of the channel name
> my_logger_mt;


BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(my_logger, my_logger_mt)
{
    // Specify the channel name on construction, similarly as with the channel_logger
    return my_logger_mt(keywords::channel = "my_logger");
}

void logging_function()
{
    // Do logging with the severity level. The record will have both
    // the severity level and the channel name attached.
    BOOST_LOG_SEV(my_logger::get(), normal) << "Hello, world!";
}
```

Global storage for loggers

```
#include <boost/log/sources/global_logger_storage.hpp>
```

Sometimes it is inconvenient to have a logger object to be able to write logs. This issue is often present in functional-style code with no obvious places where a logger could be stored. Another domain where the problem persists is generic libraries that would benefit from logging. In such cases it would be more convenient to have one or several global loggers in order to easily access them in every place when needed. In this regard `std::cout` is a good example of such a logger.

The library provides a way to declare global loggers that can be accessed pretty much like `std::cout`. In fact, this feature can be used with any logger, including user-defined ones. Having declared a global logger, one can be sure to have a thread-safe access to this logger instance from any place of the application code. The library also guarantees that a global logger instance will be unique even across module boundaries. This allows employing logging even in header-only components that may get compiled into different modules.

One may wonder why there is a need for something special in order to create global loggers. Why not just declare a logger variable at namespace scope and use it wherever you need? While technically this is possible, declaring and using global logger variables is complicated for the following reasons:

- Order of initialization of namespace scope variables is not specified by the C++ Standard. This means that generally you cannot use the logger during this stage of initialization (i.e. before `main`).

- Initialization of namespace scope variables is not thread-safe. You may end up initializing the same logger twice or using an uninitialized logger.
- Using namespace scope variables in a header-only library is quite complicated. One either has to declare a variable with external linkage and define it only in a single translation unit (that is, in a separate .cpp file, which defeats the "header-only" thesis), or define a variable with internal linkage, or as a special case in an anonymous namespace (this will most likely break ODR and give unexpected results when the header is used in different translation units). There are other compiler-specific and standard tricks to tackle the problem, but they are not quite trivial and portable.
- On most platforms namespace scope variables are local to the module where they were compiled in. That is, if variable `a` has external linkage and was compiled into modules X and Y, each of these modules has its own copy of variable `a`. To make things worse, on other platforms this variable can be shared between the modules.

Global logger storage is intended to eliminate all these problems.

The easiest way to declare a global logger is to use the following macro:

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(my_logger, src::severity_logger_mt< >)
```

The `my_logger` argument gives the logger a name that may be used to acquire the logger instance. This name acts as a tag of the declared logger. The second parameter denotes the logger type. In multithreaded applications, when the logger can be accessed from different threads, users will normally want to use the thread-safe versions of loggers.

If passing arguments to the logger constructor is needed, there is another macro:

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS(
    my_logger,
    src::severity_channel_logger< >,
    (keywords::severity = error)(keywords::channel = "my_channel"))
```

The last macro argument is a Boost.Preprocessor sequence of arguments passed to the logger constructor. Be careful, however, when using non-constant expressions and references to objects as constructor arguments, since the arguments are evaluated only once and it is often difficult to tell the exact moment when it is done. The logger is constructed on the first request from whichever part of the application that has the knowledge of the logger declaration. It is up to user to make sure that all arguments have valid states at that point.

The third macro of this section provides maximum initialization flexibility, allowing the user to actually define the logic of creating the logger.

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(my_logger, src::severity_logger_mt)
{
    // Do something that needs to be done on logger initialization,
    // e.g. add a stop watch attribute.
    src::severity_logger_mt< > lg;
    lg.add_attribute("StopWatch", boost::make_shared< attrs::timer >());
    // The initializing routine must return the logger instance
    return lg;
}
```

Like the `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` macro, the initialization code is called only once, on the first request of the logger.

Important

Beware of One Definition Rule (ODR) issues. Regardless of the way of logger declaration you choose, you should ensure that the logger is declared in exactly the same way at all occurrences and all symbol names involved in the declaration resolve to the same entities. The latter includes the names used within the initialization routine of the `BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT` macro, such as references to external variables, functions and types. The library tries to protect itself from ODR violations to a certain degree, but in general the behavior is undefined if the rule is violated.

In order to alleviate ODR problems, it is possible to separate the logger declaration and its initialization routine. The library provides the following macros to achieve this:

- `BOOST_LOG_GLOBAL_LOGGER` provides the logger declaration. It can be used in a header, similarly to the `BOOST_LOG_INLINE_GLOBAL_LOGGER*` macros described above.
- `BOOST_LOG_GLOBAL_LOGGER_INIT`, `BOOST_LOG_GLOBAL_LOGGER_DEFAULT` and `BOOST_LOG_GLOBAL_LOGGER_CTOR_ARGS` define the logger initialization routine. Their semantics and usage is similar to the corresponding `BOOST_LOG_INLINE_GLOBAL_LOGGER*` macros, for one exception: these macros should be used in a single .cpp file.

For example:

```
// my_logger.h
// ===========

BOOST_LOG_GLOBAL_LOGGER(my_logger, src::severity_logger_mt)


// my_logger.cpp
// ===========

#include "my_logger.h"

BOOST_LOG_GLOBAL_LOGGER_INIT(my_logger, src::severity_logger_mt)
{
    src::severity_logger_mt< > lg;
    lg.add_attribute("StopWatch", boost::make_shared< attrs::timer >());
    return lg;
}
```

Regardless of the macro you used to declare the logger, you can acquire the logger instance with the static `get` function of the logger tag:

```
src::severity_logger_mt< >& lg = my_logger::get();
```

Further usage of the logger is the same as if it was a regular logger object of the corresponding type.

Warning

It should be noted that it is not advised to use global loggers during the deinitialization stage of the application. Like any other global object in your application, the global logger may get destroyed before you try to use it. In such cases it's better to have a dedicated logger object that is guaranteed to be available as long as needed.

Sink frontends

Sink frontends are the part of sinks provided by the library, that implements the common functionality shared between all sinks. This includes support for filtering, exception handling and thread synchronization. Also, since formatting is typical for text-based

sinks, it is implemented by frontends as well. Every sink frontend receives log records from the logging core and then passes them along to the associated sink backend. The frontend does not define how to process records but rather in what way the core should interact with the backend. It is the backend that defines the processing rules of the log records. You probably won't have to write your own frontend when you need to create a new type of sink, because the library provides a number of frontends that cover most use cases.

Sink frontends derive from the `sink` class template, which is used by the logging core to supply log records. Technically speaking, one can derive his class from the `sink` template and have his new-found sink, but using sink frontends saves from quite an amount of routine work. As every sink frontend is associated with a backend, the corresponding backend will also be constructed by the frontend upon its construction (unless the user provides the backend instance himself), making the sink complete. Therefore, when the frontend is constructed it can be registered in the logging core to begin processing records. See the Sink Backends section for more details on interactions between frontends and backends.

Below is a more detailed overview of the services provided by sink frontends.

Basic sink frontend services

There are a number of basic functionalities that all sink frontends provide.

Filtering

All sink frontends support filtering. The user can specify a custom filtering function object or a filter constructed with the library-provided tools. The filter can be set with the `set_filter` method or cleared with the `reset_filter` method. The filter is invoked during the call to the `will_consume` method that is issued by the logging core. If the filter is not set, it is assumed that the sink will accept any log record.

Note

Like the logging core, all sink frontends assume it is safe to call filters from multiple threads concurrently. This is fine with the library-provided filters.

Formatting

For text-based sink backends, frontends implement record formatting. Like with filters, lambda expressions can be used to construct formatters. The formatter can be set for a text-based sink by calling the `set_formatter` method or cleared by calling `reset_formatter`.

Exception handling

All sink frontends allow setting up exception handlers in order to customize error processing on a per-sink basis. One can install an exception handling function with the `set_exception_handler` method, this function will be called with no arguments from a `catch` block if an exception occurs during record processing in the backend or during the sink-specific filtering. The exception handler is free to rethrow an exception or to suppress it. In the former case the exception is propagated to the core, where another layer of exception handling can come into action.

Tip

Logging core and loggers also support installing exception handlers.

The library provides a convenient tool for dispatching exceptions into a unary polymorphic function object.

Note

An exception handler is not allowed to return a value. This means you are not able to alter the filtering result once an exception occurs, and thus filtering will always fail.

Note

All sink frontends assume it is safe to call exception handlers from multiple threads concurrently. This is fine with the library-provided exception dispatchers.

Unlocked sink frontend

```
#include <boost/log/sinks/unlocked_frontend.hpp>
```

The unlocked sink frontend is implemented with the `unlocked_sink` class template. This frontend provides the most basic service for the backend. The `unlocked_sink` frontend performs no thread synchronization when accessing the backend, assuming that synchronization either is not needed or is implemented by the backend. Nevertheless, setting up a filter is still thread-safe (that is, one can safely change the filter in the `unlocked_sink` frontend while other threads are writing logs through this sink). This is the only sink frontend available in a single threaded environment. The example of use is as follows:

```
enum severity_level
{
    normal,
    warning,
    error
};

// A trivial sink backend that requires no thread synchronization
class my_backend :
    public sinks::basic_sink_backend< sinks::concurrent_feeding >
{
public:
    // The function is called for every log record to be written to log
    void consume(logging::record_view const& rec)
    {
        // We skip the actual synchronization code for brevity
        std::cout << rec[expr::smessage] << std::endl;
    }
};

// Complete sink type
typedef sinks::unlocked_sink< my_backend > sink_t;

void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // The simplest way, the backend is default-constructed
    boost::shared_ptr< sink_t > sink1(new sink_t());
    core->add_sink(sink1);

    // One can construct backend separately and pass it to the frontend
    boost::shared_ptr< my_backend > backend(new my_backend());
    boost::shared_ptr< sink_t > sink2(new sink_t(backend));
    core->add_sink(sink2);

    // You can manage filtering through the sink interface
    sink1->set_filter(expr::attr< severity_level >("Severity") >= warning);
    sink2->set_filter(expr::attr< std::string >("Channel") == "net");
}
```

See the complete code.

All sink backends provided by the library require thread synchronization on the frontend part. If we tried to instantiate the frontend on the backend that requires more strict threading guarantees than what the frontend provides, the code wouldn't have compiled. Therefore this frontend is mostly useful in single-threaded environments and with custom backends.

Synchronous sink frontend

```
#include <boost/log/sinks/sync_frontend.hpp>
```

The synchronous sink frontend is implemented with the `synchronous_sink` class template. It is similar to the `unlocked_sink` but additionally provides thread synchronization with a mutex before passing log records to the backend. All sink backends that support formatting currently require thread synchronization in the frontend.

The synchronous sink also introduces the ability to acquire a pointer to the locked backend. As long as the pointer exists, the backend is guaranteed not to be accessed from other threads, unless the access is done through another frontend or a direct reference to the backend. This feature can be useful if there is a need to perform some updates on the sink backend while other threads may be writing logs. Beware, though, that while the backend is locked any other thread that tries to write a log record to the sink gets blocked until the backend is released.

The usage is similar to the `unlocked_sink`.

```
enum severity_level
{
    normal,
    warning,
    error
};

// Complete sink type
typedef sinks::synchronous_sink< sinks::text_ostream_backend > sink_t;

void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and initialize it with a stream
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));

    // Wrap it into the frontend and register in the core
    boost::shared_ptr< sink_t > sink(new sink_t(backend));
    core->add_sink(sink);

    // You can manage filtering and formatting through the sink interface
    sink->set_filter(expr::attr< severity_level >("Severity") >= warning);
    sink->set_formatter
    (
        expr::stream
            << "Level: " << expr::attr< severity_level >("Severity")
            << " Message: " << expr::smessage
    );

    // You can also manage backend in a thread-safe manner
    {
        sink_t::locked_backend_ptr p = sink->locked_backend();
        p->add_stream(boost::make_shared< std::ofstream >("sample.log"));
    } // the backend gets released here
}
```

See the complete code.

Asynchronous sink frontend

```
#include <boost/log/sinks/async_frontend.hpp>

// Related headers
#include <boost/log/sinks/unbounded_fifo_queue.hpp>
#include <boost/log/sinks/unbounded_ordering_queue.hpp>
#include <boost/log/sinks/bounded_fifo_queue.hpp>
#include <boost/log/sinks/bounded_ordering_queue.hpp>
#include <boost/log/sinks/drop_on_overflow.hpp>
#include <boost/log/sinks/block_on_overflow.hpp>
```

The frontend is implemented in the `asynchronous_sink` class template. Like the synchronous one, asynchronous sink frontend provides a way of synchronizing access to the backend. All log records are passed to the backend in a dedicated thread, which makes it suitable for backends that may block for a considerable amount of time (network and other hardware device-related sinks, for example). The internal thread of the frontend is spawned on the frontend constructor and joined on its destructor (which implies that the frontend destruction may block).

Note

The current implementation of the asynchronous sink frontend use record queueing. This introduces a certain latency between the fact of record emission and its actual processing (such as writing into a file). This behavior may be inadequate in some contexts, such as debugging an application that is prone to crashes.

```
enum severity_level
{
    normal,
    warning,
    error
};

// Complete sink type
typedef sinks::asynchronous_sink< sinks::text_ostream_backend > sink_t;

boost::shared_ptr< sink_t > init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and initialize it with a stream
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));

    // Wrap it into the frontend and register in the core
    boost::shared_ptr< sink_t > sink(new sink_t(backend));
    core->add_sink(sink);

    // You can manage filtering and formatting through the sink interface
    sink->set_filter(expr::attr< severity_level >("Severity") >= warning);
    sink->set_formatter
    (
        expr::stream
            << "Level: " << expr::attr< severity_level >("Severity")
            << " Message: " << expr::message
    );

    // You can also manage backend in a thread-safe manner
    {
        sink_t::locked_backend_ptr p = sink->locked_backend();
        p->add_stream(boost::make_shared< std::ofstream >("sample.log"));
    } // the backend gets released here

    return sink;
}
```

Important

If asynchronous logging is used in a multi-module application, one should decide carefully when to unload dynamically loaded modules that write logs. The library has many places where it may end up using resources that reside in the dynamically loaded module. Examples of such resources are virtual tables, string literals and functions. If any of these resources are still used by the library when the module in which they reside gets unloaded, the application will most likely crash. Strictly speaking, this problem exists with any sink type (and is not limited to sinks in the first place), but asynchronous sinks introduce an additional problem. One cannot tell which resources are used by the asynchronous sink because it works in a dedicated thread and uses buffered log records. There is no general solution for this issue. Users are advised to either avoid dynamic module unloading during the application's work time, or to avoid asynchronous logging. As an additional way to cope with the problem, one may try to shutdown all asynchronous sinks before unloading any modules, and after unloading re-create them. However, avoiding dynamic unloading is the only way to solve the problem completely.

In order to stop the dedicated thread feeding log records to the backend one can call the `stop` method of the frontend. This method will be called automatically in the frontend destructor. The `stop` method, unlike thread interruption, will only terminate the feeding loop when a log record that is being fed is processed by the backend (i.e. it will not interrupt the record processing that has already started). However, it may happen that some records are still left in the queue after returning from the `stop` method. In order to flush them to the backend an additional call to the `feed_records` method is required. This is useful in the application termination stage.

```
void stop_logging(boost::shared_ptr< sink_t >& sink)
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Remove the sink from the core, so that no records are passed to it
    core->remove_sink(sink);

    // Break the feeding loop
    sink->stop();

    // Flush all log records that may have left buffered
    sink->flush();

    sink.reset();
}
```

See the complete code.

Spawning the dedicated thread for log record feeding can be suppressed with the optional boolean `start_thread` named parameter of the frontend. In this case the user can select either way of processing records:

- Call the `run` method of the frontend. This call will block in the feeding loop. This loop can be interrupted with the call to `stop`.
- Periodically call `feed_records`. This method will process all the log records that were in the frontend queue when the call was issued and then return.

Note

Users should take care not to mix these two approaches concurrently. Also, none of these methods should be called if the dedicated feeding thread is running (i.e., the `start_thread` was not specified in the construction or had the value of `true`.

Customizing record queueing strategy

The `asynchronous_sink` class template can be customized with the record queueing strategy. Several strategies are provided by the library:

- `unbounded_fifo_queue`. This strategy is the default. As the name implies, the queue is not limited in depth and does not order log records.
- `unbounded_ordering_queue`. Like `unbounded_fifo_queue`, the queue has unlimited depth but it applies an order on the queued records. We will return to ordering queues in a moment.
- `bounded_fifo_queue`. The queue has limited depth specified in a template parameter as well as the overflow handling strategy. No record ordering is applied.
- `bounded_ordering_queue`. Like `bounded_fifo_queue` but also applies log record ordering.

Warning

Be careful with unbounded queueing strategies. Since the queue has unlimited depth, if log records are continuously generated faster than being processed by the backend the queue grows uncontrollably which manifests itself as a memory leak.

Bounded queues support the following overflow strategies:

- `drop_on_overflow`. When the queue is full, silently drop excessive log records.
- `block_on_overflow`. When the queue is full, block the logging thread until the backend feeding thread manages to process some of the queued records.

For example, this is how we could modify the previous example to limit the record queue to 100 elements:

```
// Complete sink type
typedef sinks::asynchronous_sink<
    sinks::text_ostream_backend,
    sinks::bounded_fifo_queue<                                  ❶
        100,                                                    ❷
        sinks::drop_on_overflow                                 ❸
    >
> sink_t;

boost::shared_ptr< sink_t > init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and initialize it with a stream
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));

    // Wrap it into the frontend and register in the core
    boost::shared_ptr< sink_t > sink(new sink_t(backend));
    core->add_sink(sink);

    // ...

    return sink;
}
```

❶ log record queueing strategy
❷ record queue capacity
❸ overflow handling policy

See the complete code.

Also see the `bounded_async_log` example in the library distribution.

Ordering log records

Record ordering can be useful to alleviate the weak record ordering issue present in multithreaded applications.

Ordering queueing strategies introduce a small latency to the record processing. The latency duration and the ordering predicate can be specified on the frontend construction. It may be useful to employ the log record ordering tools to implement ordering predicates.

```
// Complete sink type
typedef sinks::asynchronous_sink<
    sinks::text_ostream_backend,
    sinks::unbounded_ordering_queue<                                    ❶
        logging::attribute_value_ordering<                              ❷
            unsigned int,                                               ❸
            std::less< unsigned int >                                   ❹
        >
    >
> sink_t;

boost::shared_ptr< sink_t > init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and initialize it with a stream
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));

    // Wrap it into the frontend and register in the core
    boost::shared_ptr< sink_t > sink(new sink_t(
        backend,                                                        ❺
        keywords::order =
            logging::make_attr_ordering("LineID", std::less< unsigned int >()),  ❻
        keywords::ordering_window = boost::posix_time::seconds(1)       ❼
    ));
    core->add_sink(sink);

    // You can manage filtering and formatting through the sink interface
    sink->set_filter(expr::attr< severity_level >("Severity") >= warning);
    sink->set_formatter
    (
        expr::stream
            << "Level: " << expr::attr< severity_level >("Severity")
            << " Message: " << expr::smessage
    );

    // You can also manage backend in a thread-safe manner
    {
        sink_t::locked_backend_ptr p = sink->locked_backend();
        p->add_stream(boost::make_shared< std::ofstream >("sample.log"));
    } // the backend gets released here

    return sink;
}
```

❶ log record queueing strategy
❷ log record ordering predicate type
❸ attribute value type
❹ optional, attribute value comparison predicate; `std::less` equivalent is used by default

❺ pointer to the pre-initialized backend
❻ log record ordering predicate
❼ latency of log record processing

In the code sample above the sink frontend will keep log records in the internal queue for up to one second and apply ordering based on the log record counter of type `unsigned int`. The `ordering_window` parameter is optional and will default to some reasonably small system-specific value that will suffice to maintain chronological flow of log records to the backend.

The ordering window is maintained by the frontend even upon stopping the internal feeding loop, so that it would be possible to reenter the loop without breaking the record ordering. On the other hand, in order to ensure that all log records are flushed to the backend one has to call the `flush` method at the end of the application.

```
void stop_logging(boost::shared_ptr< sink_t >& sink)
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Remove the sink from the core, so that no records are passed to it
    core->remove_sink(sink);

    // Break the feeding loop
    sink->stop();

    // Flush all log records that may have left buffered
    sink->flush();

    sink.reset();
}
```

This technique is also demonstrated in the `async_log` example in the library distribution.

Sink backends

Text stream backend

```
#include <boost/log/sinks/text_ostream_backend.hpp>
```

The text output stream sink backend is the most generic backend provided by the library out of the box. The backend is implemented in the `basic_text_ostream_backend` class template (`text_ostream_backend` and `wtext_ostream_backend` convenience typedefs provided for narrow and wide character support). It supports formatting log records into strings and putting into one or several streams. Each attached stream gets the same result of formatting, so if you need to format log records differently for different streams, you will need to create several sinks - each with its own formatter.

The backend also provides a feature that may come useful when debugging your application. With the `auto_flush` method one can tell the sink to automatically flush the buffers of all attached streams after each log record is written. This will, of course, degrade logging performance, but in case of an application crash there is a good chance that last log records will not be lost.

```
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and attach a couple of streams to it
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(&std::clog, boost::null_deleter()));
    backend->add_stream(
        boost::shared_ptr< std::ostream >(new std::ofstream("sample.log")));

    // Enable auto-flushing after each log record written
    backend->auto_flush(true);

    // Wrap it into the frontend and register in the core.
    // The backend requires synchronization in the frontend.
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > sink_t;
    boost::shared_ptr< sink_t > sink(new sink_t(backend));
    core->add_sink(sink);
}
```

Text file backend

```
#include <boost/log/sinks/text_file_backend.hpp>
```

Although it is possible to write logs into files with the text stream backend the library also offers a special sink backend with an extended set of features suitable for file-based logging. The features include:

- Log file rotation based on file size and/or time
- Flexible log file naming
- Placing the rotated files into a special location in the file system
- Deleting the oldest files in order to free more space on the file system

The backend is called `text_file_backend`.

Warning

This sink uses Boost.Filesystem internally, which may cause problems on process termination. See here for more details.

File rotation

File rotation is implemented by the sink backend itself. The file name pattern and rotation thresholds can be specified when the `text_file_backend` backend is constructed.

```
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    boost::shared_ptr< sinks::text_file_backend > backend =
        boost::make_shared< sinks::text_file_backend >(
            keywords::file_name = "file_%5N.log",                                    ❶
            keywords::rotation_size = 5 * 1024 * 1024,                               ❷
            keywords::time_based_rotation = sinks::file::rotation_at_time_point(12, 0, 0)  ❸
        );

    // Wrap it into the frontend and register in the core.
    // The backend requires synchronization in the frontend.
    typedef sinks::synchronous_sink< sinks::text_file_backend > sink_t;
    boost::shared_ptr< sink_t > sink(new sink_t(backend));

    core->add_sink(sink);
}
```

❶ file name pattern
❷ rotate the file upon reaching 5 MiB size...
❸ ...or every day, at noon, whichever comes first

Note

The file size at rotation can be imprecise. The implementation counts the number of characters written to the file, but the underlying API can introduce additional auxiliary data, which would increase the log file's actual size on disk. For instance, it is well known that Windows and DOS operating systems have a special treatment with regard to new-line characters. Each new-line character is written as a two byte sequence 0x0D 0x0A instead of a single 0x0A. Other platform-specific character translations are also known.

The time-based rotation is not limited by only time points. There are following options available out of the box:

1. Time point rotations: `rotation_at_time_point` class. This kind of rotation takes place whenever the specified time point is reached. The following variants are available:

   - Every day rotation, at the specified time. This is what was presented in the code snippet above:

     ```
     sinks::file::rotation_at_time_point(12, 0, 0)
     ```

   - Rotation on the specified day of every week, at the specified time. For instance, this will make file rotation to happen every Tuesday, at midnight:

     ```
     sinks::file::rotation_at_time_point(date_time::Tuesday, 0, 0, 0)
     ```

     in case of midnight, the time can be omitted:

     ```
     sinks::file::rotation_at_time_point(date_time::Tuesday)
     ```

   - Rotation on the specified day of each month, at the specified time. For example, this is how to rotate files on the 1-st of every month:

     ```
     sinks::file::rotation_at_time_point(gregorian::greg_day(1), 0, 0, 0)
     ```

like with weekdays, midnight is implied:

```
sinks::file::rotation_at_time_point(gregorian::greg_day(1))
```

2. Time interval rotations: `rotation_at_time_interval` class. With this predicate the rotation is not bound to any time points and happens as soon as the specified time interval since the previous rotation elapses. This is how to make rotations every hour:

```
sinks::file::rotation_at_time_interval(posix_time::hours(1))
```

If none of the above applies, one can specify his own predicate for time-based rotation. The predicate should take no arguments and return `bool` (the `true` value indicates that the rotation should take place). The predicate will be called for every log record being written to the file.

```
bool is_it_time_to_rotate();

void init_logging()
{
    // ...

    boost::shared_ptr< sinks::text_file_backend > backend =
        boost::make_shared< sinks::text_file_backend >(
            keywords::file_name = "file_%5N.log",
            keywords::time_based_rotation = &is_it_time_to_rotate
        );

    // ...
}
```

Note

The log file rotation takes place on an attempt to write a new log record to the file. Thus the time-based rotation is not a strict threshold, either. The rotation will take place as soon as the library detects that the rotation should have happened.

The file name pattern may contain a number of wildcards, like the one you can see in the example above. Supported placeholders are:

- Current date and time components. The placeholders conform to the ones specified by Boost.DateTime library.
- File counter (`%N`) with an optional width specification in the `printf`-like format. The file counter will always be decimal, zero filled to the specified width.
- A percent sign (`%%`).

A few quick examples:

Template	Expands to
file_%N.log	file_1.log, file_2.log...
file_%3N.log	file_001.log, file_002.log...
file_%Y%m%d.log	file_20080705.log, file_20080706.log...
file_%Y-%m-%d_%H-%M-%S.%N.log	file_2008-07-05_13-44-23.1.log, file_2008-07-06_16-00-10.2.log...

Important

Although all Boost.DateTime format specifiers will work, there are restrictions on some of them, if you intend to scan for old log files. This functionality is discussed in the next section.

The sink backend allows hooking into the file rotation process in order to perform pre- and post-rotation actions. This can be useful to maintain log file validity by writing headers and footers. For example, this is how we could modify the `init_logging` function in order to write logs into XML files:

```
// Complete file sink type
typedef sinks::synchronous_sink< sinks::text_file_backend > file_sink;

void write_header(sinks::text_file_backend::stream_type& file)
{
    file << "<?xml version=\"1.0\"?>\n<log>\n";
}

void write_footer(sinks::text_file_backend::stream_type& file)
{
    file << "</log>\n";
}

void init_logging()
{
    // Create a text file sink
    boost::shared_ptr< file_sink > sink(new file_sink(
        keywords::file_name = "%Y%m%d_%H%M%S_%5N.xml",  ❶
        keywords::rotation_size = 16384                 ❷
    ));

    sink->set_formatter
    (
        expr::format("\t<record id=\"%1%\" timestamp=\"%2%\">%3%</record>")
            % expr::attr< unsigned int >("RecordID")
            % expr::attr< boost::posix_time::ptime >("TimeStamp")
            % expr::xml_decor[ expr::stream << expr::smessage ]            ❸
    );

    // Set header and footer writing functors
    sink->locked_backend()->set_open_handler(&write_header);
    sink->locked_backend()->set_close_handler(&write_footer);

    // Add the sink to the core
    logging::core::get()->add_sink(sink);
}
```

❶ the resulting file name pattern

❷ rotation size, in characters
❸ the log message has to be decorated, if it contains special characters

See the complete code.

Finally, the sink backend also supports the auto-flush feature, like the text stream backend does.

Managing rotated files

After being closed, the rotated files can be collected. In order to do so one has to set up a file collector by specifying the target directory where to collect the rotated files and, optionally, size thresholds. For example, we can modify the `init_logging` function to place rotated files into a distinct directory and limit total size of the files. Let's assume the following function is called by `init_logging` with the constructed sink:

```
void init_file_collecting(boost::shared_ptr< file_sink > sink)
{
    sink->locked_backend()->set_file_collector(sinks::file::make_collector(
        keywords::target = "logs",                      ❶
        keywords::max_size = 16 * 1024 * 1024,          ❷
        keywords::min_free_space = 100 * 1024 * 1024    ❸
    ));
}
```

❶ the target directory
❷ maximum total size of the stored files, in bytes
❸ minimum free space on the drive, in bytes

The `max_size` and `min_free_space` parameters are optional, the corresponding threshold will not be taken into account if the parameter is not specified.

One can create multiple file sink backends that collect files into the same target directory. In this case the most strict thresholds are combined for this target directory. The files from this directory will be erased without regard for which sink backend wrote it, i.e. in the strict chronological order.

Warning

The collector does not resolve log file name clashes between different sink backends, so if the clash occurs the behavior is undefined, in general. Depending on the circumstances, the files may overwrite each other or the operation may fail entirely.

The file collector provides another useful feature. Suppose you ran your application 5 times and you have 5 log files in the "logs" directory. The file sink backend and file collector provide a `scan_for_files` method that searches the target directory for these files and takes them into account. So, if it comes to deleting files, these files are not forgotten. What's more, if the file name pattern in the backend involves a file counter, scanning for older files allows updating the counter to the most recent value. Here is the final version of our `init_logging` function:

```
void init_logging()
{
    // Create a text file sink
    boost::shared_ptr< file_sink > sink(new file_sink(
        keywords::file_name = "%Y%m%d_%H%M%S_%5N.xml",
        keywords::rotation_size = 16384
    ));

    // Set up where the rotated files will be stored
    init_file_collecting(sink);

    // Upon restart, scan the directory for files matching the file_name pattern
    sink->locked_backend()->scan_for_files();

    sink->set_formatter
    (
        expr::format("\t<record id=\"%1%\" timestamp=\"%2%\">%3%</record>")
            % expr::attr< unsigned int >("RecordID")
            % expr::attr< boost::posix_time::ptime >("TimeStamp")
            % expr::xml_decor[ expr::stream << expr::smessage ]
    );

    // Set header and footer writing functors
    namespace bll = boost::lambda;

    sink->locked_backend()->set_open_handler
    (
        bll::_1 << "<?xml version=\"1.0\"?>\n<log>\n"
    );
    sink->locked_backend()->set_close_handler
    (
        bll::_1 << "</log>\n"
    );

    // Add the sink to the core
    logging::core::get()->add_sink(sink);
}
```

There are two methods of file scanning: the scan that involves file name matching with the file name pattern (the default) and the scan that assumes that all files in the target directory are log files. The former applies certain restrictions on the placeholders that can be used within the file name pattern, in particular only file counter placeholder and these placeholders of Boost.DateTime are supported: `%y`, `%Y`, `%m`, `%d`, `%H`, `%M`, `%S`, `%f`. The latter scanning method, in its turn, has its own drawback: it does not allow updating the file counter in the backend. It is also considered to be more dangerous as it may result in unintended file deletion, so be cautious. The all-files scanning method can be enabled by passing it as an additional parameter to the `scan_for_files` call:

```
// Look for all files in the target directory
backend->scan_for_files(sinks::file::scan_all);
```

Text multi-file backend

```
#include <boost/log/sinks/text_multifile_backend.hpp>
```

While the text stream and file backends are aimed to store all log records into a single file/stream, this backend serves a different purpose. Assume we have a banking request processing application and we want logs related to every single request to be placed into a separate file. If we can associate some attribute with the request identity then the `text_multifile_backend` backend is the way to go.

```
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    boost::shared_ptr< sinks::text_multifile_backend > backend =
        boost::make_shared< sinks::text_multifile_backend >();

    // Set up the file naming pattern
    backend->set_file_name_composer
    (
       sinks::file::as_file_name_composer(expr::stream << "logs/" << expr::attr< std::string >("Re↵
questID") << ".log")
    );

    // Wrap it into the frontend and register in the core.
    // The backend requires synchronization in the frontend.
    typedef sinks::synchronous_sink< sinks::text_multifile_backend > sink_t;
    boost::shared_ptr< sink_t > sink(new sink_t(backend));

    // Set the formatter
    sink->set_formatter
    (
        expr::stream
            << "[RequestID: " << expr::attr< std::string >("RequestID")
            << "] " << expr::smessage
    );

    core->add_sink(sink);
}
```

You can see we used a regular formatter in order to specify file naming pattern. Now, every log record with a distinct value of the "RequestID" attribute will be stored in a separate file, no matter how many different requests are being processed by the application concurrently. You can also find the `multiple_files` example in the library distribution, which shows a similar technique to separate logs generated by different threads of the application.

If using formatters is not appropriate for some reason, you can provide your own file name composer. The composer is a mere function object that accepts a log record as a single argument and returns a value of the `text_multifile_backend::path_type` type.

Note

The multi-file backend has no knowledge of whether a particular file is going to be used or not. That is, if a log record has been written into file A, the library cannot tell whether there will be more records that fit into the file A or not. This makes it impossible to implement file rotation and removing unused files to free space on the file system. The user will have to implement such functionality himself.

Syslog backend

```
#include <boost/log/sinks/syslog_backend.hpp>
```

The syslog backend, as comes from its name, provides support for the syslog API that is available on virtually any UNIX-like platform. On Windows there exists at least one public implementation of the syslog client API. However, in order to provide maximum flexibility and better portability the library offers built-in support for the syslog protocol described in RFC 3164. Thus on Windows only the built-in implementation is supported, while on UNIX-like systems both built-in and system API based implementations are supported.

The backend is implemented in the `syslog_backend` class. The backend supports formatting log records, and therefore requires thread synchronization in the frontend. The backend also supports severity level translation from the application-specific values to

the syslog-defined values. This is achieved with an additional function object, level mapper, that receives a set of attribute values of each log record and returns the appropriate syslog level value. This value is used by the backend to construct the final priority value of the syslog record. The other component of the syslog priority value, the facility, is constant for each backend object and can be specified in the backend constructor arguments.

Level mappers can be written by library users to translate the application log levels to the syslog levels in the best way. However, the library provides two mappers that would fit this need in obvious cases. The `direct_severity_mapping` class template provides a way to directly map values of some integral attribute to syslog levels, without any value conversion. The `custom_severity_mapping` class template adds some flexibility and allows to map arbitrary values of some attribute to syslog levels.

Anyway, one example is better than a thousand words.

```
// Complete sink type
typedef sinks::synchronous_sink< sinks::syslog_backend > sink_t;

void init_native_syslog()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend
    boost::shared_ptr< sinks::syslog_backend > backend(new sinks::syslog_backend(
        keywords::facility = sinks::syslog::user,               ❶
        keywords::use_impl = sinks::syslog::native              ❷
    ));

    // Set the straightforward level translator for the "Severity" attribute of type int
    backend->set_severity_mapper(sinks::syslog::direct_severity_mapping< int >("Severity"));

    // Wrap it into the frontend and register in the core.
    // The backend requires synchronization in the frontend.
    core->add_sink(boost::make_shared< sink_t >(backend));
}

void init_builtin_syslog()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a new backend
    boost::shared_ptr< sinks::syslog_backend > backend(new sinks::syslog_backend(
        keywords::facility = sinks::syslog::local0,             ❸
        keywords::use_impl = sinks::syslog::udp_socket_based    ❹
    ));

    // Setup the target address and port to send syslog messages to
    backend->set_target_address("192.164.1.10", 514);

    // Create and fill in another level translator for "MyLevel" attribute of type string
    sinks::syslog::custom_severity_mapping< std::string > mapping("MyLevel");
    mapping["debug"] = sinks::syslog::debug;
    mapping["normal"] = sinks::syslog::info;
    mapping["warning"] = sinks::syslog::warning;
    mapping["failure"] = sinks::syslog::critical;
    backend->set_severity_mapper(mapping);

    // Wrap it into the frontend and register in the core.
    core->add_sink(boost::make_shared< sink_t >(backend));
}
```

❶ the logging facility
❷ the native syslog API should be used
❸ the logging facility

❹ the built-in socket-based implementation should be used

Please note that all syslog constants, as well as level extractors, are declared within a nested namespace `syslog`. The library will not accept (and does not declare in the backend interface) native syslog constants, which are macros, actually.

Also note that the backend will default to the built-in implementation and `user` logging facility, if the corresponding constructor parameters are not specified.

Tip

The `set_target_address` method will also accept DNS names, which it will resolve to the actual IP address. This feature, however, is not available in single threaded builds.

Windows debugger output backend

```
#include <boost/log/sinks/debug_output_backend.hpp>
```

Windows API has an interesting feature: a process, being run under a debugger, is able to emit messages that will be intercepted and displayed in the debugger window. For example, if an application is run under the Visual Studio IDE it is able to write debug messages to the IDE window. The `basic_debug_output_backend` backend provides a simple way of emitting such messages. Additionally, in order to optimize application performance, a special filter is available that checks whether the application is being run under a debugger. Like many other sink backends, this backend also supports setting a formatter in order to compose the message text.

The usage is quite simple and straightforward:

```
// Complete sink type
typedef sinks::synchronous_sink< sinks::debug_output_backend > sink_t;

void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create the sink. The backend requires synchronization in the frontend.
    boost::shared_ptr< sink_t > sink(new sink_t());

    // Set the special filter to the frontend
    // in order to skip the sink when no debugger is available
    sink->set_filter(expr::is_debugger_present());

    core->add_sink(sink);
}
```

Note that the sink backend is templated on the character type. This type defines the Windows API version that is used to emit messages. Also, `debug_output_backend` and `wdebug_output_backend` convenience typedefs are provided.

Windows event log backends

```
#include <[boost/log/sinks/event_log_backend.hpp]>
```

Windows operating system provides a special API for publishing events related to application execution. A wide range of applications, including Windows components, use this facility to provide the user with all essential information about computer health in a single place - an event log. There can be more than one event log. However, typically all user-space applications use the common Application log. Records from different applications or their parts can be selected from the log by a record source name. Event logs can be read with a standard utility, an Event Viewer, that comes with Windows.

Although it looks very tempting, the API is quite complicated and intrusive, which makes it difficult to support. The application is required to provide a dynamic library with special resources that describe all events the application supports. This library must be registered in the Windows registry, which pins its location in the file system. The Event Viewer uses this registration to find the resources and compose and display messages. The positive feature of this approach is that since event resources can describe events differently for different languages, it allows the application to support event internationalization in a quite transparent manner: the application simply provides event identifiers and non-localizable event parameters to the API, and it does the rest of the work.

In order to support both the simplistic approach "it just works" and the more elaborate event composition, including internationalization support, the library provides two sink backends that work with event log API.

Simple event log backend

The `basic_simple_event_log_backend` backend is intended to encapsulate as much of the event log API as possible, leaving interface and usage model very similar to other sink backends. It contains all resources that are needed for the Event Viewer to function properly, and registers the Boost.Log library in the Windows registry in order to populate itself as the container of these resources.

Important

The library must be built as a dynamic library in order to use this backend flawlessly. Otherwise event description resources are not linked into the executable, and the Event Viewer is not able to display events properly.

The only thing user has to do to add Windows event log support to his application is to provide event source and log names (which are optional and can be automatically suggested by the library), set up an appropriate filter, formatter and event severity mapping.

```
// Complete sink type
typedef sinks::synchronous_sink< sinks::simple_event_log_backend > sink_t;

// Define application-specific severity levels
enum severity_level
{
    normal,
    warning,
    error
};

void init_logging()
{
    // Create an event log sink
    boost::shared_ptr< sink_t > sink(new sink_t());

    sink->set_formatter
    (
        expr::format("%1%: [%2%] - %3%")
            % expr::attr< unsigned int >("LineID")
            % expr::attr< boost::posix_time::ptime >("TimeStamp")
            % expr::smessage
    );

    // We'll have to map our custom levels to the event log event types
    sinks::event_log::custom_event_type_mapping< severity_level > mapping("Severity");
    mapping[normal] = sinks::event_log::info;
    mapping[warning] = sinks::event_log::warning;
    mapping[error] = sinks::event_log::error;

    sink->locked_backend()->set_event_type_mapper(mapping);

    // Add the sink to the core
    logging::core::get()->add_sink(sink);
}
```

Having done that, all logging records that pass to the sink will be formatted the same way they are in the other sinks. The formatted message will be displayed in the Event Viewer as the event description.

Advanced event log backend

The `basic_event_log_backend` allows more detailed control over the logging API, but requires considerably more scaffolding during initialization and usage.

First, the user has to build his own library with the event resources (the process is described in MSDN). As a part of this process one has to create a message file that describes all events. For the sake of example, let's assume the following contents were used as the message file:

```
; /* ------------------------------------------------------
; HEADER SECTION
; */
SeverityNames=(Debug=0x0:MY_SEVERITY_DEBUG
            Info=0x1:MY_SEVERITY_INFO
            Warning=0x2:MY_SEVERITY_WARNING
            Error=0x3:MY_SEVERITY_ERROR
            )

; /* ------------------------------------------------------
; MESSAGE DEFINITION SECTION
; */

MessageIdTypedef=WORD

MessageId=0x1
SymbolicName=MY_CATEGORY_1
Language=English
Category 1
.

MessageId=0x2
SymbolicName=MY_CATEGORY_2
Language=English
Category 2
.

MessageId=0x3
SymbolicName=MY_CATEGORY_3
Language=English
Category 3
.

MessageIdTypedef=DWORD

MessageId=0x100
Severity=Warning
Facility=Application
SymbolicName=LOW_DISK_SPACE_MSG
Language=English
The drive %1 has low free disk space. At least %2 Mb of free space is recommended.
.

MessageId=0x101
Severity=Error
Facility=Application
SymbolicName=DEVICE_INACCESSIBLE_MSG
Language=English
The drive %1 is not accessible.
.
```

```
MessageId=0x102
Severity=Info
Facility=Application
SymbolicName=SUCCEEDED_MSG
Language=English
Operation finished successfully in %1 seconds.
.
```

After compiling the resource library, the path to this library must be provided to the sink backend constructor, among other parameters used with the simple backend. The path may contain placeholders that will be expanded with the appropriate environment variables.

```
// Create an event log sink
boost::shared_ptr< sinks::event_log_backend > backend(
    new sinks::event_log_backend((
        keywords::message_file = "%SystemDir%\\event_log_messages.dll",
        keywords::log_name = "My Application",
        keywords::log_source = "My Source"
    ))
);
```

Like the simple backend, `basic_event_log_backend` will register itself in the Windows registry, which will enable the Event Viewer to display the emitted events.

Next, the user will have to provide the mapping between the application logging attributes and event identifiers. These identifiers were provided in the message compiler output as a result of compiling the message file. One can use `basic_event_composer` and one of the event ID mappings, like in the following example:

```
// Create an event composer. It is initialized with the event identifier mapping.
sinks::event_log::event_composer composer(
    sinks::event_log::direct_event_id_mapping< int >("EventID"));

// For each event described in the message file, set up the insertion string formatters
composer[LOW_DISK_SPACE_MSG]
    // the first placeholder in the message
    // will be replaced with contents of the "Drive" attribute
    % expr::attr< std::string >("Drive")
    // the second placeholder in the message
    // will be replaced with contents of the "Size" attribute
    % expr::attr< boost::uintmax_t >("Size");

composer[DEVICE_INACCESSIBLE_MSG]
    % expr::attr< std::string >("Drive");

composer[SUCCEEDED_MSG]
    % expr::attr< unsigned int >("Duration");

// Then put the composer to the backend
backend->set_event_composer(composer);
```

As you can see, one can use regular formatters to specify which attributes will be inserted instead of placeholders in the final event message. Aside from that, one can specify mappings of attribute values to event types and categories. Suppose our application has the following severity levels:

```
// Define application-specific severity levels
enum severity_level
{
    normal,
    warning,
    error
};
```

Then these levels can be mapped onto the values in the message description file:

```
// We'll have to map our custom levels to the event log event types
sinks::event_log::custom_event_type_mapping< severity_level > type_mapping("Severity");
type_mapping[normal] = sinks::event_log::make_event_type(MY_SEVERITY_INFO);
type_mapping[warning] = sinks::event_log::make_event_type(MY_SEVERITY_WARNING);
type_mapping[error] = sinks::event_log::make_event_type(MY_SEVERITY_ERROR);

backend->set_event_type_mapper(type_mapping);

// Same for event categories.
// Usually event categories can be restored by the event identifier.
sinks::event_log::custom_event_category_mapping< int > cat_mapping("EventID");
cat_mapping[LOW_DISK_SPACE_MSG] = sinks::event_log::make_event_category(MY_CATEGORY_1);
cat_mapping[DEVICE_INACCESSIBLE_MSG] = sinks::event_log::make_event_category(MY_CATEGORY_2);
cat_mapping[SUCCEEDED_MSG] = sinks::event_log::make_event_category(MY_CATEGORY_3);

backend->set_event_category_mapper(cat_mapping);
```

Tip

As of Windows NT 6 (Vista, Server 2008) it is not needed to specify event type mappings. This information is available in the message definition resources and need not be duplicated in the API call.

Now that initialization is done, the sink can be registered into the core.

```
// Create the frontend for the sink
boost::shared_ptr< sinks::synchronous_sink< sinks::event_log_backend > > sink(
    new sinks::synchronous_sink< sinks::event_log_backend >(backend));

// Set up filter to pass only records that have the necessary attribute
sink->set_filter(expr::has_attr< int >("EventID"));

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

In order to emit events it is convenient to create a set of functions that will accept all needed parameters for the corresponding events and announce that the event has occurred.

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(event_logger, src::severity_logger_mt< severity_level >)

// The function raises an event of the disk space depletion
void announce_low_disk_space(std::string const& drive, boost::uintmax_t size)
{
    BOOST_LOG_SCOPED_THREAD_TAG("EventID", (int)LOW_DISK_SPACE_MSG);
    BOOST_LOG_SCOPED_THREAD_TAG("Drive", drive);
    BOOST_LOG_SCOPED_THREAD_TAG("Size", size);
    // Since this record may get accepted by other sinks,
    // this message is not completely useless
    BOOST_LOG_SEV(event_logger::get(), warning) << "Low disk " << drive
        << " space, " << size << " Mb is recommended";
}

// The function raises an event of inaccessible disk drive
void announce_device_inaccessible(std::string const& drive)
{
    BOOST_LOG_SCOPED_THREAD_TAG("EventID", (int)DEVICE_INACCESSIBLE_MSG);
    BOOST_LOG_SCOPED_THREAD_TAG("Drive", drive);
    BOOST_LOG_SEV(event_logger::get(), error) << "Cannot access drive " << drive;
}

// The structure is an activity guard that will emit an event upon the activity completion
struct activity_guard
{
    activity_guard()
    {
        // Add a stop watch attribute to measure the activity duration
        m_it = event_logger::get().add_attribute("Duration", attrs::timer()).first;
    }
    ~activity_guard()
    {
        BOOST_LOG_SCOPED_THREAD_TAG("EventID", (int)SUCCEEDED_MSG);
        BOOST_LOG_SEV(event_logger::get(), normal) << "Activity ended";
        event_logger::get().remove_attribute(m_it);
    }

private:
    logging::attribute_set::iterator m_it;
};
```

Now you are able to call these helper functions to emit events. The complete code from this section is available in the `event_log` example in the library distribution.

Lambda expressions

As it was pointed out in tutorial, filters and formatters can be specified as Lambda expressions with placeholders for attribute values. This section will describe the placeholders that can be used to build more complex Lambda expressions.

There is also a way to specify the filter in the form of a string template. This can be useful for initialization from the application settings. This part of the library is described here.

Generic attribute placeholder

```
#include <boost/log/expressions/attr_fwd.hpp>
#include <boost/log/expressions/attr.hpp>
```

The `attr` placeholder represents an attribute value in template expressions. Given the record view or a set of attribute values, the placeholder will attempt to extract the specified attribute value from the argument upon invocation. This can be roughly described with the following pseudo-code:

```
logging::value_ref< T, TagT > val = expr::attr< T, TagT >(name)(rec);
```

where `val` is the reference to the extracted value, `name` and `T` are the attribute value name and type, `TagT` is an optional tag (we'll return to it in a moment) and `rec` is the log record view or attribute value set. `T` can be a Boost.MPL type sequence with possible expected types of the value; the extraction will succeed if the type of the value matches one of the types in the sequence.

The `attr` placeholder can be used in Boost.Phoenix expressions, including the `bind` expression.

```
bool my_filter(logging::value_ref< severity_level, tag::severity > const& level,
               logging::value_ref< std::string, tag::tag_attr > const& tag)
{
    return level >= warning || tag == "IMPORTANT_MESSAGE";
}

void init()
{
    // ...

    namespace phoenix = boost::phoenix;
    sink->set_filter(phoenix::bind(&my_filter, severity.or_none(), tag_attr.or_none()));

    // ...
}
```

The placeholder can be used both in filters and formatters:

```
sink->set_filter
(
    expr::attr< int >("Severity") >= 5 &&
    expr::attr< std::string >("Channel") == "net"
);

sink->set_formatter
(
    expr::stream
        << expr::attr< int >("Severity")
        << " [" << expr::attr< std::string >("Channel") << "] "
        << expr::smessage
);
```

The call to `set_filter` registers a composite filter that consists of two elementary subfilters: the first one checks the severity level, and the second checks the channel name. The call to `set_formatter` installs a formatter that composes a string containing the severity level and the channel name along with the message text.

Customizing fallback policy

By default, when the requested attribute value is not found in the record, `attr` will return an empty reference. In case of filters, this will result in `false` in any ordering expressions, and in case of formatters the output from the placeholder will be empty. This behavior can be changed:

- To throw an exception (`missing_value` or `invalid_type`, depending on the reason of the failure). Add the `or_throw` modifier:

```
sink->set_filter
(
    expr::attr< int >("Severity").or_throw() >= 5 &&
    expr::attr< std::string >("Channel").or_throw() == "net"
);
```

- To use a default value instead. Add the `or_default` modifier with the desired default value:

```
sink->set_filter
(
    expr::attr< int >("Severity").or_default(0) >= 5 &&
    expr::attr< std::string >("Channel").or_default(std::string("general")) == "net"
);
```

Tip

You can also use the `has_attr` predicate to implement filters and formatters conditional on the attribute value presence.

The default behavior is also accessible through the `or_none` modifier. The modified placeholders can be used in filters and formatters just the same way as the unmodified ones.

In `bind` expressions, the bound function object will still receive the `value_ref`-wrapped values in place of the modified `attr` placeholder. Even though both `or_throw` and `or_default` modifiers guarantee that the bound function will receive a filled reference, `value_ref` is still needed if the value type is specified as a type sequence. Also, the reference wrapper may contain a tag type which may be useful for formatting customization.

Attribute tags and custom formatting operators

The `TagT` type in the abstract description of `attr` above is optional and by default is `void`. This is an attribute tag which can be used to customize the output formatters produce for different attributes. This tag is forwarded to the `to_log` manipulator when the extracted attribute value is put to a stream (this behavior is warranted by `value_ref` implementation). Here's a quick example:

```
// We define our own severity levels
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

// The operator is used for regular stream formatting
std::ostream& operator<< (std::ostream& strm, severity_level level)
{
    static const char* strings[] =
    {
        "normal",
        "notification",
        "warning",
        "error",
        "critical"
    };

    if (static_cast< std::size_t >(level) < sizeof(strings) / sizeof(*strings))
        strm << strings[level];
    else
        strm << static_cast< int >(level);

    return strm;
}

// Attribute value tag type
struct severity_tag;

// The operator is used when putting the severity level to log
logging::formatting_ostream& operator<<
(
    logging::formatting_ostream& strm,
    logging::to_log_manip< severity_level, severity_tag > const& manip
)
{
    static const char* strings[] =
    {
        "NORM",
        "NTFY",
        "WARN",
        "ERRR",
        "CRIT"
    };

    severity_level level = manip.get();
    if (static_cast< std::size_t >(level) < sizeof(strings) / sizeof(*strings))
        strm << strings[level];
    else
        strm << static_cast< int >(level);

    return strm;
}

void init()
{
    logging::add_console_log
    (
        std::clog,
```

```
        // This makes the sink to write log records that look like this:
        // 1: <NORM> A normal severity message
        // 2: <ERRR> An error severity message
        keywords::format =
        (
            expr::stream
                << expr::attr< unsigned int >("LineID")
                << ": <" << expr::attr< severity_level, severity_tag >("Severity")
                << "> " << expr::smessage
        )
    );
}
```

See the complete code.

Here we specify a different formatting operator for the severity level wrapped in the `to_log_manip` manipulator marked with the tag `severity_tag`. This operator will be called when log records are formatted while the regular `operator<<` will be used in other contexts.

Defining attribute keywords

```
#include <boost/log/expressions/keyword_fwd.hpp>
#include <boost/log/expressions/keyword.hpp>
```

Attribute keywords can be used as replacements for the `attr` placeholders in filters and formatters while providing a more concise and less error prone syntax. An attribute keyword can be declared with the `BOOST_LOG_ATTRIBUTE_KEYWORD` macro:

```
BOOST_LOG_ATTRIBUTE_KEYWORD(keyword, "Keyword", type)
```

Here the macro declares a keyword `keyword` for an attribute named "Keyword" with the value type of `type`. Additionally, the macro defines an attribute tag type `keyword` within the `tag` namespace. We can rewrite the previous example in the following way:

```
// We define our own severity levels
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

// Define the attribute keywords
BOOST_LOG_ATTRIBUTE_KEYWORD(line_id, "LineID", unsigned int)
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)

// The operator is used for regular stream formatting
std::ostream& operator<< (std::ostream& strm, severity_level level)
{
    static const char* strings[] =
    {
        "normal",
        "notification",
        "warning",
        "error",
        "critical"
    };

    if (static_cast< std::size_t >(level) < sizeof(strings) / sizeof(*strings))
        strm << strings[level];
    else
        strm << static_cast< int >(level);

    return strm;
}

// The operator is used when putting the severity level to log
logging::formatting_ostream& operator<<
(
    logging::formatting_ostream& strm,
    logging::to_log_manip< severity_level, tag::severity > const& manip
)
{
    static const char* strings[] =
    {
        "NORM",
        "NTFY",
        "WARN",
        "ERRR",
        "CRIT"
    };

    severity_level level = manip.get();
    if (static_cast< std::size_t >(level) < sizeof(strings) / sizeof(*strings))
        strm << strings[level];
    else
        strm << static_cast< int >(level);

    return strm;
}

void init()
{
    logging::add_console_log
    (
```

```
        std::clog,
        // This makes the sink to write log records that look like this:
        // 1: <NORM> A normal severity message
        // 2: <ERRR> An error severity message
        keywords::format =
        (
            expr::stream
                << line_id
                << ": <" << severity
                << "> " << expr::smessage
        )
    );
}
```

Attribute keywords behave the same way as the `attr` placeholders and can be used both in filters and formatters. The `or_throw` and `or_default` modifiers are also supported.

Keywords can also be used in attribute value lookup expressions in log records and attribute value sets:

```
void print_severity(logging::record_view const& rec)
{
    logging::value_ref< severity_level, tag::severity > level = rec[severity];
    std::cout << level << std::endl;
}
```

Record placeholder

```
#include <boost/log/expressions/record.hpp>
```

The `record` placeholder can be used in `bind` expressions to pass the whole log record view to the bound function object.

```
void my_formatter(logging::formatting_ostream& strm, logging::record_view const& rec)
{
    // ...
}

namespace phoenix = boost::phoenix;
sink->set_formatter(phoenix::bind(&my_formatter, expr::stream, expr::record));
```

Note

In case of filters, the placeholder will correspond to the set of attribute values rather than the log record itself. This is because the record is not constructed yet at the point of filtering, and filters only operate on the set of attribute values.

Message text placeholders

```
#include <boost/log/expressions/message.hpp>
```

Log records typically contain a special attribute "Message" with the value of one of the string types (more specifically, an `std::basic_string` specialization). This attribute contains the text of the log message that is constructed at the point of the record creation. This attribute is only constructed after filtering, so filters cannot use it. There are several keywords to access this attribute value:

- `smessage` - the attribute value is expected to be an `std::string`

- `wmessage` - the attribute value is expected to be an `std::wstring`
- `message` - the attribute value is expected to be an `std::string` or `std::wstring`

The `message` keyword has to dispatch between different string types, so it is slightly less efficient than the other two keywords. If the application is able to guarantee the fixed character type of log messages, it is advised to use the corresponding keyword for better performance.

```
// Sets up a formatter that will ignore all attributes and only print log record text
sink->set_formatter(expr::stream << expr::message);
```

Predicate expressions

This section describes several expressions that can be used as predicates in the filtering expressions.

Attribute presence filter

```
#include <boost/log/expressions/predicates/has_attr.hpp>
```

The filter `has_attr` checks if an attribute value with the specified name and, optionally, type is attached to a log record. If no type specified to the filter, the filter returns `true` if any value with the specified name is found. If an MPL-compatible type sequence in specified as a value type, the filter returns `true` if a value with the specified name and one of the specified types is found.

This filter is usually used in conjunction with conditional formatters, but it also can be used as a quick filter based on the log record structure. For example, one can use this filter to extract statistic records and route them to a specific sink.

```
// Declare attribute keywords
BOOST_LOG_ATTRIBUTE_KEYWORD(stat_stream, "StatisticStream", std::string)
BOOST_LOG_ATTRIBUTE_KEYWORD(change, "Change", int)

// A simple sink backend to accumulate statistic information
class my_stat_accumulator :
    public sinks::basic_sink_backend< sinks::synchronized_feeding >
{
    // A map of accumulated statistic values,
    // ordered by the statistic information stream name
    typedef std::map< std::string, int > stat_info_map;
    stat_info_map m_stat_info;

public:
    // Destructor
    ~my_stat_accumulator()
    {
        // Display the accumulated data
        stat_info_map::const_iterator it = m_stat_info.begin(), end = m_stat_info.end();
        for (; it != end; ++it)
        {
            std::cout << "Statistic stream: " << it->first
                << ", accumulated value: " << it->second << "\n";
        }
        std::cout.flush();
    }

    // The method is called for every log record being put into the sink backend
    void consume(logging::record_view const& rec)
    {
        // First, acquire statistic information stream name
        logging::value_ref< std::string, tag::stat_stream > name = rec[stat_stream];
        if (name)
        {
            // Next, get the statistic value change
            logging::value_ref< int, tag::change > change_amount = rec[change];
            if (change_amount)
            {
                // Accumulate the statistic data
                m_stat_info[name.get()] += change_amount.get();
            }
        }
    }
};

// The function registers two sinks - one for statistic information,
// and another one for other records
void init()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create a backend and attach a stream to it
    boost::shared_ptr< sinks::text_ostream_backend > backend =
        boost::make_shared< sinks::text_ostream_backend >();
    backend->add_stream(
        boost::shared_ptr< std::ostream >(new std::ofstream("test.log")));

    // Create a frontend and setup filtering
    typedef sinks::synchronous_sink< sinks::text_ostream_backend > log_sink_type;
    boost::shared_ptr< log_sink_type > log_sink(new log_sink_type(backend));
    // All records that don't have a "StatisticStream" attribute attached
    // will go to the "test.log" file
    log_sink->set_filter(!expr::has_attr(stat_stream));
```

```
    core->add_sink(log_sink);

    // Create another sink that will receive all statistic data
    typedef sinks::synchronous_sink< my_stat_accumulator > stat_sink_type;
    boost::shared_ptr< stat_sink_type > stat_sink(new stat_sink_type());
    // All records with a "StatisticStream" string attribute attached
    // will go to the my_stat_accumulator sink
    stat_sink->set_filter(expr::has_attr(stat_stream));

    core->add_sink(stat_sink);
}

// This simple macro will simplify putting statistic data into a logger
#define PUT_STAT(lg, stat_stream_name, change)\
    if (true) {\
        BOOST_LOG_SCOPED_LOGGER_TAG(lg, "StatisticStream", stat_stream_name);\
        BOOST_LOG(lg) << logging::add_value("Change", (int)(change));\
    } else ((void)0)

void logging_function()
{
    src::logger lg;

    // Put a regular log record, it will go to the "test.log" file
    BOOST_LOG(lg) << "A regular log record";

    // Put some statistic data
    PUT_STAT(lg, "StreamOne", 10);
    PUT_STAT(lg, "StreamTwo", 20);
    PUT_STAT(lg, "StreamOne", -5);
}
```

See the complete code.

In this example, log records emitted with the `PUT_STAT` macro will be directed to the `my_stat_accumulator` sink backend, which will accumulate the changes passed in the "Change" attribute values. All other records (even those made through the same logger) will be passed to the filter sink. This is achieved with the mutually exclusive filters set for the two sinks.

Please note that in the example above we extended the library in two ways: we defined a new sink backend `my_stat_accumulator` and a new macro `PUT_STAT`. Also note that `has_attr` can accept attribute keywords to identify the attribute to check.

Range checking filter

```
#include <boost/log/expressions/predicates/is_in_range.hpp>
```

The `is_in_range` predicate checks that the attribute value fits in the half-open range (i.e. it returns `true` if the attribute value `x` satisfies the following condition: `left <= x < right`). For example:

```
sink->set_filter
(
    // drops all records that have level below 3 or greater than 4
    expr::is_in_range(expr::attr< int >("Severity"), 3, 5)
);
```

The attribute can also be identified by an attribute keyword or name and type:

```
sink->set_filter
(
    expr::is_in_range(severity, 3, 5)
);

sink->set_filter
(
    expr::is_in_range< int >("Severity", 3, 5)
);
```

Simple string matching filters

```
#include <boost/log/expressions/predicates/begins_with.hpp>
#include <boost/log/expressions/predicates/ends_with.hpp>
#include <boost/log/expressions/predicates/contains.hpp>
```

Predicates `begins_with`, `ends_with` and `contains` provide an easy way of matching string attribute values. As follows from their names, the functions construct filters that return `true` if an attribute value begins with, ends with or contains the specified substring, respectively. The string comparison is case sensitive.

```
sink->set_filter
(
    // selects only records that are related to Russian web domains
    expr::ends_with(expr::attr< std::string >("Domain"), ".ru")
);
```

The attribute can also be identified by an attribute keyword or name and type.

Advanced string matching filter

```
#include <boost/log/expressions/predicates/matches.hpp>

// Supporting headers
#include <boost/log/support/regex.hpp>
#include <boost/log/support/std_regex.hpp>
#include <boost/log/support/xpressive.hpp>
#include <boost/log/support/spirit_qi.hpp>
#include <boost/log/support/spirit_classic.hpp>
```

The `matches` function creates a filter that apples a regular expression or a parser to a string attribute value. The regular expression can be provided by Boost.Regex or Boost.Xpressive. Parsers from Boost.Spirit and Boost.Spirit2 are also supported. The filter returns `true` if the regular expression matches or the parser successfully parses the attribute value.

Note

In order to use this predicate, a corresponding supporting header should also be included.

```
sink->set_filter
(
    expr::matches(expr::attr< std::string >("Domain"), boost::regex("www\\..*\\.ru"))
);
```

The attribute can also be identified by an attribute keyword or name and type.

Severity threshold per channel filter

```
#include <boost/log/expressions/predicates/channel_severity_filter.hpp>
```

This filter is aimed for a specific but commonly encountered use case. The `channel_severity_filter` function creates a predicate that will check log record severity levels against a threshold. The predicate allows setting different thresholds for different channels. The mapping between channel names and severity thresholds can be filled in `std::map` style by using the subscript operator or by calling `add` method on the filter itself (the `channel_severity_filter_actor` instance). Let's see an example:

```
// We define our own severity levels
enum severity_level
{
    normal,
    notification,
    warning,
    error,
    critical
};

// Define the attribute keywords
BOOST_LOG_ATTRIBUTE_KEYWORD(line_id, "LineID", unsigned int)
BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)
BOOST_LOG_ATTRIBUTE_KEYWORD(channel, "Channel", std::string)


void init()
{
    // Create a minimal severity table filter
    typedef expr::channel_severity_filter_actor< std::string, severity_level > min_severity_filter;
    min_severity_filter min_severity = expr::channel_severity_filter(channel, severity);

    // Set up the minimum severity levels for different channels
    min_severity["general"] = notification;
    min_severity["network"] = warning;
    min_severity["gui"] = error;

    logging::add_console_log
    (
        std::clog,
        keywords::filter = min_severity || severity >= critical,
        keywords::format =
        (
            expr::stream
                << line_id
                << ": <" << severity
                << "> [" << channel << "] "
                << expr::smessage
        )
    );
}

// Define our logger type
typedef src::severity_channel_logger< severity_level, std::string > logger_type;

void test_logging(logger_type& lg, std::string const& channel_name)
{
    BOOST_LOG_CHANNEL_SEV(lg, channel_name, normal) << "A normal severity level message";
```

```
    BOOST_LOG_CHANNEL_SEV(lg, channel_name, notification) << "A notification severity level mes↵
sage";
    BOOST_LOG_CHANNEL_SEV(lg, channel_name, warning) << "A warning severity level message";
    BOOST_LOG_CHANNEL_SEV(lg, channel_name, error) << "An error severity level message";
    BOOST_LOG_CHANNEL_SEV(lg, channel_name, critical) << "A critical severity level message";
}
```

See the complete code.

The filter for the console sink is composed from the `channel_severity_filter_actor` filter and a general severity level check. This general check will be used when log records do not have a channel attribute or the channel name is not one of those specified in `channel_severity_filter_actor` initialization. It should be noted that it is possible to set the default result of the threshold filter that will be used in this case; the default result can be set by the `set_default` method. The `channel_severity_filter_actor` filter is set up to limit record severity levels for channels "general", "network" and "gui" - all records in these channels with levels below the specified thresholds will not pass the filter and will be ignored.

The threshold filter is implemented as an equivalent to `std::map` over the channels, which means that the channel value type must support partial ordering. Obviously, the severity level type must also support ordering to be able to be compared against thresholds. By default the predicate will use `std::less` equivalent for channel name ordering and `std::greater_equal` equivalent to compare severity levels. It is possible to customize the ordering predicates. Consult the reference of the `channel_severity_filter_actor` class and `channel_severity_filter` generator to see the relevant template parameters.

Debugger presence filter

```
#include <boost/log/expressions/predicates/is_debugger_present.hpp>
```

This filter is implemented for Windows only. The `is_debugger_present` filter returns `true` if the application is run under a debugger and `false` otherwise. It does not use any attribute values from the log record. This predicate is typically used with the debugger output sink.

```
// Complete sink type
typedef sinks::synchronous_sink< sinks::debug_output_backend > sink_t;

void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Create the sink. The backend requires synchronization in the frontend.
    boost::shared_ptr< sink_t > sink(new sink_t());

    // Set the special filter to the frontend
    // in order to skip the sink when no debugger is available
    sink->set_filter(expr::is_debugger_present());

    core->add_sink(sink);
}
```

See the complete code.

Formatting expressions

As was noted in the tutorial, the library provides several ways of expressing formatters, most notable being with a stream-style syntax and Boost.Format-style expression. Which of the two formats is chosen is determined by the appropriate anchor expression. To use stream-style syntax one should begin the formatter definition with the `stream` keyword, like that:

```
#include <boost/log/expressions/formatters/stream.hpp>

sink->set_formatter(expr::stream << expr1 << expr2 << ... << exprN);
```

Here expressions `expr1` through `exprN` may be either manipulators, described in this section, or other expressions resulting in an object that supports putting into an STL-stream.

To use Boost.Format-style syntax one should use `format` construct:

```
#include <boost/log/expressions/formatters/format.hpp>

sink->set_formatter(expr::format("format string") % expr1 % expr2 % ... % exprN);
```

The format string passed to the `format` keyword should contain positional placeholders for the appropriate expressions. In the case of wide-character logging the format string should be wide. Expressions `expr1` through `exprN` have the same meaning as in stream-like variant. It should be noted though that using stream-like syntax usually results in a faster formatter than the one constructed with the `format` keyword.

Another useful way of expressing formatters is by using string templates. This part of the library is described in this section and is mostly intended to support initialization from the application settings.

Date and time formatter

```
#include <boost/log/expressions/formatters/date_time.hpp>

// Supporting headers
#include <boost/log/support/date_time.hpp>
```

The library provides the `format_date_time` formatter dedicated to date and time-related attribute value types. The function accepts the attribute value name and the format string compatible with Boost.DateTime.

```
sink->set_formatter
(
    expr::stream << expr::format_date_time< boost::posix_time::ptime >("TimeStamp", "%Y-%m-%d ↵
%H:%M:%S")
);
```

The attribute value can alternatively be identified with the `attr` placeholder or the attribute keyword.

The following placeholders are supported in the format string:

Table 2. Date format placeholders

Placeholder	Meaning	Example
%a	Abbreviated weekday name	"Mon" => Monday
%A	Long weekday name	"Monday"
%b	Abbreviated month name	"Feb" => February
%B	Long month name	"February"
%d	Numeric day of month with leading zero	"01"
%e	Numeric day of month with leading space	" 1"
%m	Numeric month, 01-12	"01"
%w	Numeric day of week, 1-7	"1"
%y	Short year	"12" => 2012
%Y	Long year	"2012"

Table 3. Time format placeholders

Placeholder	Meaning	Example
%f	Fractional seconds with leading zeros	"000231"
%H, %O	Hours in 24 hour clock or hours in time duration types with leading zero if less than 10	"07"
%I	Hours in 12 hour clock with leading zero if less than 10	"07"
%k	Hours in 24 hour clock or hours in time duration types with leading space if less than 10	" 7"
%l	Hours in 12 hour clock with leading space if less than 10	" 7"
%M	Minutes	"32"
%p	AM/PM mark, uppercase	"AM"
%P	AM/PM mark, lowercase	"am"
%q	ISO time zone	"-0700" => Mountain Standard Time
%Q	Extended ISO time zone	"-05:00" => Eastern Standard Time
%S	Seconds	"26"

Table 4. Miscellaneous placeholders

Placeholder	Meaning	Example
%-	Negative sign in case of time duration, if the duration is less than zero	"-"
%+	Sign of time duration, even if positive	"+"
%%	An escaped percent sign	"%"
%T	Extended ISO time, equivalent to "%H:%M:%S"	"07:32:26"

Note that in order to use this formatter you will also have to include a supporting header. When `boost/log/support/date_time.hpp` is included, the formatter supports the following types of Boost.DateTime:

- Date and time types: `boost::posix_time::ptime` and `boost::local_time::local_date_time`.
- Gregorian date type: `boost::gregorian::date`.
- Time duration types: `boost::posix_time::time_duration` as well as all the specialized time units such as `boost::posix_time::seconds`, including subsecond units.
- Date duration types: `boost::gregorian::date_duration`.

Tip

Boost.DateTime already provides formatting functionality implemented as a number of locale facets. This functionality can be used instead of this formatter, although the formatter is expected to provide better performance.

Named scope formatter

```
#include <boost/log/expressions/formatters/named_scope.hpp>
```

The formatter `format_named_scope` is intended to add support for flexible formatting of the named scope attribute values. The basic usage is quite straightforward and its result is similar to what `attr` provides:

```
// Puts the scope stack from outer ones towards inner ones: outer scope -> inner scope
sink->set_formatter(expr::stream << expr::format_named_scope("Scopes", "%n"));
```

The first argument names the attribute and the second is the format string. The string can contain the following placeholders:

Table 5. Named scope format placeholders

Placeholder	Meaning	Example
%n	Scope name	"void bar::foo()"
%c	Function name, if the scope is denoted with `BOOST_LOG_FUNCTION`, otherwise the full scope name. See the note below.	"bar::foo"
%C	Function name, without the function scope, if the scope is denoted with `BOOST_LOG_FUNCTION`, otherwise the full scope name. See the note below.	"foo"
%f	Source file name of the scope	"/home/user/project/foo.cpp"
%F	Source file name of the scope, without the path	"foo.cpp"
%l	Line number in the source file	"45"

Note

As described in the named scope attribute description, it is possible to use `BOOST_LOG_FUNCTION` macro to automatically generate scope names from the enclosing function name. Unfortunately, the actual format of the generated strings is compiler-dependent and in many cases it includes the complete signature of the function. When "%c" or "%C" format flag is specified, the library attempts to parse the generated string to extract the function name. Since C++ syntax is very context dependent and complex, it is not possible to parse function signature correctly in all cases, so the library is basically guessing. Depending on the string format, this may fail or produce incorrect results. In particular, type conversion operators can pose problems for the parser. In case if the parser fails to recognize the function signature the library falls back to using the whole string (i.e. behave equivalent to the "%n" flag). To alleviate the problem the user can replace the problematic `BOOST_LOG_FUNCTION` usage with the `BOOST_LOG_NAMED_SCOPE` macro and explicitly write the desired scope name. Scope names denoted with `BOOST_LOG_NAMED_SCOPE` will not be interpreted by the library and will be output as is. In general, for portability and runtime performance reasons it is preferable to always use `BOOST_LOG_NAMED_SCOPE` and "%n" format flag.

While the format string describes the presentation of each named scope in the list, the following named arguments allow to customize the list traversal and formatting:

- `format`. The named scope format string, as described above. This parameter is used to specify the format when other named parameters are used.
- `iteration`. The argument describes the direction of iteration through scopes. Can have values `forward` (default) or `reverse`.
- `delimiter`. The argument can be used to specify the delimiters between scopes. The default delimiter depends on the `iteration` argument. If `iteration == forward` the default `delimiter` will be "->", otherwise it will be "<-".
- `depth`. The argument can be used to limit the number of scopes to put to log. The formatter will print `depth` innermost scopes and, if there are more scopes left, append an ellipsis to the written sequence. By default the formatter will write all scope names.
- `incomplete_marker`. The argument can be used to specify the string that is used to indicate that the list has been limited by the `depth` argument. By default the "..." string is used as the marker.
- `empty_marker`. The argument can be used to specify the string to output in case if the scope list is empty. By default nothing is output in this case.

Here are a few usage examples:

```
// Puts the scope stack in reverse order:
// inner scope (file:line) <- outer scope (file:line)
sink->set_formatter
(
    expr::stream
        << expr::format_named_scope(
            "Scopes",
            keywords::format = "%n (%f:%l)",
            keywords::iteration = expr::reverse)
);

// Puts the scope stack in reverse order with a custom delimiter:
// inner scope | outer scope
sink->set_formatter
(
    expr::stream
        << expr::format_named_scope(
            "Scopes",
            keywords::format = "%n",
            keywords::iteration = expr::reverse,
            keywords::delimiter = " | ")
);

// Puts the scope stack in forward order, no more than 2 inner scopes:
// ... outer scope -> inner scope
sink->set_formatter
(
    expr::stream
        << expr::format_named_scope(
            "Scopes",
            keywords::format = "%n",
            keywords::iteration = expr::forward,
            keywords::depth = 2)
);

// Puts the scope stack in reverse order, no more than 2 inner scopes:
// inner scope <- outer scope <<and more>>...
sink->set_formatter
(
    expr::stream
        << expr::format_named_scope(
            "Scopes",
            keywords::format = "%n",
            keywords::iteration = expr::reverse,
            keywords::incomplete_marker = " <<and more>>..."
            keywords::depth = 2)
);
```

Tip

An empty string can be specified as the `incomplete_marker` parameter, in which case there will be no indication that the list was truncated.

Conditional formatters

```
#include <boost/log/expressions/formatters/if.hpp>
```

There are cases when one would want to check some condition about the log record and format it depending on that condition. One example of such a need is formatting an attribute value depending on its runtime type. The general syntax of the conditional formatter is as follows:

```
expr::if_ (filter)
[
    true_formatter
]
.else_
[
    false_formatter
]
```

Those familiar with Boost.Phoenix lambda expressions will find this syntax quite familiar. The `filter` argument is a filter that is applied to the record being formatted. If it returns `true`, the `true_formatter` is executed, otherwise `false_formatter` is executed. The `else_` section with `false_formatter` is optional. If it is omitted and `filter` yields `false`, no formatter is executed. Here is an example:

```
sink->set_formatter
(
    expr::stream
        // First, put the current time
        << expr::format_date_time("TimeStamp", "%Y-%m-%d %H:%M:%S.%f") << " "
        << expr::if_ (expr::has_attr< int >("ID"))
           [
               // if "ID" is present then put it to the record
               expr::stream << expr::attr< int >("ID")
           ]
           .else_
           [
               // otherwise put a missing marker
               expr::stream << "--"
           ]
        // and after that goes the log record text
        << " " << expr::message
);
```

Character decorators

There are times when one would like to additionally post-process the composed string before passing it to the sink backend. For example, in order to store log into an XML file the formatted log record should be checked for special characters that have a special meaning in XML documents. This is where decorators step in.

Note

Unlike most other formatters, decorators are dependent on the character type of the formatted output and this type cannot be deduced from the decorated formatter. By default, the character type is assumed to be `char`. If the formatter is used to compose a wide-character string, prepend the decorator name with the `w` letter (e.g. use `wxml_decor` instead of `xml_decor`). Also, for each decorator there is a generator function that accepts the character type as a template parameter; the function is named similarly to the decorator prepended with the `make_` prefix (e.g. `make_xml_decor`).

XML character decorator

```
#include <boost/log/expressions/formatters/xml_decorator.hpp>
```

This decorator replaces XML special characters (&, <, > and ') with the corresponding tokens (`&`, `<`, `>` and `'`, correspondingly). The usage is as follows:

```
xml_sink->set_formatter
(
    // Apply the decoration to the whole formatted record
    expr::xml_decor
    [
        expr::stream << expr::message
    ]
);
```

Since character decorators are yet another kind of formatters, it's fine to use them in other contexts where formatters are appropriate. For example, this is also a valid example:

```
xml_sink->set_formatter
(
    expr::format("<message>%1%: %2%</message>")
        % expr::attr< unsigned int >("LineID")
        % expr::xml_decor[ expr::stream << expr::message ]; // Only decorate the message text
);
```

There is an example of the library set up for logging into an XML file, see here.

CSV character decorator

```
#include <boost/log/expressions/formatters/csv_decorator.hpp>
```

This decorator allows to ensure that the resulting string conforms to the CSV format requirements. In particular, it duplicates the quote characters in the formatted string.

```
csv_sink->set_formatter
(
    expr::stream
        << expr::attr< unsigned int >("LineID") << ","
        << expr::csv_decor[ expr::stream << expr::attr< std::string >("Tag") ] << ","
        << expr::csv_decor[ expr::stream << expr::message ]
);
```

C-style character decorators

```
#include <boost/log/expressions/formatters/c_decorator.hpp>
```

The header defines two character decorators: `c_decor` and `c_ascii_decor`. The first one replaces the following characters with their escaped counterparts: \ (backslash, 0x5c), \a (bell character, 0x07), \b (backspace, 0x08), \f (formfeed, 0x0c), \n (newline, 0x0a), \r (carriage return, 0x0d), \t (horizontal tabulation, 0x09), \v (vertical tabulation, 0x0b), ' (apostroph, 0x27), " (quote, 0x22), ? (question mark, 0x3f). The `c_ascii_decor` decorator does the same but also replaces all other non-printable and non-ASCII characters with escaped hexadecimal character codes in C notation (e.g. "\x8c"). The usage is similar to other character decorators:

```
sink->set_formatter
(
    expr::stream
        << expr::attr< unsigned int >("LineID") << ": ["
        << expr::c_decor[ expr::stream << expr::attr< std::string >("Tag") ] << "] "
        << expr::c_ascii_decor[ expr::stream << expr::message ]
);
```

General character decorator

```
#include <boost/log/expressions/formatters/char_decorator.hpp>
```

This decorator allows the user to define his own character replacement mapping in one of the two forms. The first form is a range of `std::pairs` of strings (which can be C-style strings or ranges of characters, including `std::strings`). The strings in the `first` elements of pairs will be replaced with the `second` elements of the corresponding pair.

```
std::array< std::pair< const char*, const char* >, 3 > shell_escapes =
{
    { "\"", "\\\"" },
    { "'",  "\\'" },
    { "$",  "\\$" }
};

sink->set_formatter
(
    expr::char_decor(shell_escapes)
    [
        expr::stream << expr::message
    ]
);
```

The second form is two same-sized sequences of strings; the first containing the search patterns and the second - the corresponding replacements.

```
std::array< const char*, 3 > shell_patterns =
{
    "\"", "'", "$"
};
std::array< const char*, 3 > shell_replacements =
{
    "\\\"", "\\'", "\\$"
};

sink->set_formatter
(
    expr::char_decor(shell_patterns, shell_replacements)
    [
        expr::stream << expr::message
    ]
);
```

In both cases the patterns are not interpreted and are sought in the formatted characters in the original form.

Attributes

```
#include <boost/log/attributes/attribute.hpp>
#include <boost/log/attributes/attribute_cast.hpp>
#include <boost/log/attributes/attribute_value.hpp>
```

All attributes in the library are implemented using the pimpl idiom, or more specifically - shared pimpl idiom. Every attribute provides an interface class which derives from the `attribute` class and an implementation class that derives from `impl`. The interface class only holds a reference counted pointer to the actual implementation of the attribute; this pointer is a member of the `attribute` class, so derived interface classes don't have any data members. When the interface class is default constructed, it creates the corresponding implementation object and initializes the `attribute` base class with a pointer to the implementation. Therefore the pimpl nature of attributes is transparent for users in a typical workflow.

The shared pimpl design comes significant in a few cases though. One such case is copying the attribute. The copy operation is shallow, so multiple interface objects may refer to a single implementation object. There is no way to deep copy an attribute. Another case is default construction of `attribute` which creates an empty object that does not refer to an implementation. Attributes in such empty state should not be passed to the library but can be useful in some cases, e.g. when a delayed variable initialization is needed.

It is possible to upcast the attribute interface from `attribute` to the actual interface class. To do this one has to apply `attribute_cast`:

```
logging::attribute attr = ...;
attrs::constant< int > const_attr = logging::attribute_cast< attrs::constant< int > >(attr);
```

In this example, the cast will succeed (i.e. the `const_attr` will be non-empty) if the attribute `attr` was originally created as `attrs::constant< int >`. Since all data is stored in the implementation object, no data is lost in the casting process.

The main purpose of attributes is to generate attribute values. Values are semantically distinct from the attributes. Such separation allows implementing attributes that can return different values at different time points (like clock-related attributes, for example) and, on the other hand, allows using different values of the same attribute independently. The `attribute` interface has a method named `get_value` that returns the actual attribute value. Attribute values are also implemented using the shared pimpl approach, the interface class is `attribute_value` and implementation classes derive from `impl`.

The attribute value object is mostly intended to store the actual attribute value and implement type dispatching in order to be able to extract the stored value. One should not confuse the attribute value object type and the stored value type. The former is in most cases not needed by users and provides type erasure, but the latter is needed to be able to extract the value. For brevity we call the stored attribute value type simply the attribute value type in this documentation.

Constants

```
#include <boost/log/attributes/constant.hpp>
```

The most simple and frequently used attribute type is a constant value of some type. This kind of attribute is implemented with the `constant` class template. The template is parametrized with the attribute value type. The constant value should be passed to the attribute constructor. Here is an example:

```
void foo()
{
    src::logger lg;

    // Register a constant attribute that always yields value -5
    lg.add_attribute("MyInteger", attrs::constant< int >(-5));

    // Register another constant attribute. Make it a string this time.
    lg.add_attribute("MyString", attrs::constant< std::string >("Hello world!"));

    // There is also a convenience generator function. "MyInteger2" is constant< int > here.
    lg.add_attribute("MyInteger2", attrs::make_constant(10));
}
```

That's it, there's nothing much you can do with a constant attribute. Constants are very useful when one wants to highlight some log records or just pass some data to a sink backend (e.g. pass statistical parameters to the collector).

Mutable constants

```
#include <boost/log/attributes/mutable_constant.hpp>
```

This kind of attribute is an extension for the constant attribute. In addition to being able to store some value, the `mutable_constant` class template has two distinctions:

- it allows modification of the stored value without re-registering the attribute
- it allows synchronization of the stores and reads of the stored value

In order to change the stored value of the attribute, one must call the `set` method:

```
void foo()
{
    src::logger lg;

    // Register a mutable constant attribute that always yields value -5
    attrs::mutable_constant< int > attr(-5);
    lg.add_attribute("MyInteger", attr);
    BOOST_LOG(lg) << "This record has MyInteger == -5";

    // Change the attribute value
    attr.set(100);
    BOOST_LOG(lg) << "This record has MyInteger == 100";
}
```

In multithreaded applications the `set` method calls must be serialized with the `get_value` calls (which, generally speaking, happen on every log record being made). By default `mutable_constant` does not serialize calls in any way, assuming that the user will do so externally. However, the `mutable_constant` template provides three additional template arguments: synchronization primitive type, scoped exclusive lock type and scoped shareable lock type. If a synchronization primitive type is specified, the scoped exclusive lock type is a mandatory parameter. If the scoped shareable lock type is not specified, the attribute will fall back to the exclusive lock instead of shared locks. For example:

```
// This mutable constant will always lock exclusively
// either for reading or storing the value
typedef attrs::mutable_constant<
    int,                                        // attribute value type
    boost::mutex,                               // synchronization primitive
    boost::lock_guard< boost::mutex >           // exclusive lock type
> exclusive_mc;
exclusive_mc my_int1(10);

// This mutable constant will use shared clocking for reading the value
// and exclusive locking for storing
typedef attrs::mutable_constant<
    int,                                        // attribute value type
    boost::shared_mutex,                        // synchronization primitive
    boost::unique_lock< boost::shared_mutex >,  // exclusive lock type
    boost::shared_lock< boost::shared_mutex >   // shared lock type
> shared_mc;
shared_mc my_int2(20);

BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(my_logger, src::logger_mt)
{
    src::logger_mt lg;
    lg.add_attribute("MyInteger1", my_int1);
    lg.add_attribute("MyInteger2", my_int2);

    return lg;
}

void foo()
{
    src::logger_mt& lg = get_my_logger();

    // This is safe, even if executed in multiple threads
    my_int1.set(200);
    BOOST_LOG(lg) << "This record has MyInteger1 == 200";

    my_int2.set(300);
    BOOST_LOG(lg) << "This record has MyInteger2 == 300";
}
```

Mutable constants are often used as auxiliary attributes inside loggers to store attributes that may change on some events. As opposed to regular constants, which would require re-registering in case of value modification, mutable constants allow modifying the value in-place.

Counters

```
#include <boost/log/attributes/counter.hpp>
```

Counters are one of the simplest attributes that generate a new value each time requested. Counters are often used to identify log records or to count some events, e.g. accepted network connections. The class template `counter` provides such functionality. This template is parametrized with the counter value type, which should support arithmetic operations, such as `operator +` and `operator -`. The counter attribute allows specification of the initial value and step (which can be negative) on construction.

```
BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(my_logger, src::logger_mt)
{
    src::logger_mt lg;

    // This counter will count lines, starting from 0
    lg.add_attribute("LineCounter", attrs::counter< unsigned int >());

    // This counter will count backwards, starting from 100 with step -5
    lg.add_attribute("CountDown", attrs::counter< int >(100, -5));

    return lg;
}

void foo()
{
    src::logger_mt& lg = get_my_logger();
    BOOST_LOG(lg) << "This record has LineCounter == 0, CountDown == 100";
    BOOST_LOG(lg) << "This record has LineCounter == 1, CountDown == 95";
    BOOST_LOG(lg) << "This record has LineCounter == 2, CountDown == 90";
}
```

Note

Don't expect that the log records with the `counter` attribute will always have ascending or descending counter values in the resulting log. In multithreaded applications counter values acquired by different threads may come to a sink in any order. See Rationale for a more detailed explanation on why it can happen. For this reason it is more accurate to say that the `counter` attribute generates an identifier in an ascending or descending order rather than that it counts log records in either order.

Wall clock

```
#include <boost/log/attributes/clock.hpp>
```

One of the "must-have" features of any logging library is support for attaching a time stamp to every log record. The library provides two attributes for this purpose: `utc_clock` and `local_clock`. The former returns the current UTC time and the latter returns the current local time. In either case the returned time stamp is acquired with the maximum precision for the target platform. The attribute value is `boost::posix_time::ptime` (see Boost.DateTime). The usage is quite straightforward:

```
BOOST_LOG_DECLARE_GLOBAL_LOGGER(my_logger, src::logger_mt)

void foo()
{
    logging::core::get()->add_global_attribute(
        "TimeStamp",
        attrs::local_clock());

    // Now every log record ever made will have a time stamp attached
    src::logger_mt& lg = get_my_logger();
    BOOST_LOG(lg) << "This record has a time stamp";
}
```

Stop watch (timer)

```
#include <boost/log/attributes/timer.hpp>
```

The `timer` attribute is very useful when there is a need to estimate the duration of some prolonged process. The attribute returns the time elapsed since the attribute construction. The attribute value type is `boost::posix_time::ptime::time_duration_type` (see Boost.DateTime).

```
// The class represents a single peer-to-peer connection
class network_connection
{
    src::logger m_logger;

public:
    network_connection()
    {
        m_logger.add_attribute("Duration", attrs::timer());
        BOOST_LOG(m_logger) << "Connection established";
    }
    ~network_connection()
    {
        // This log record will show the whole life time duration of the connection
        BOOST_LOG(m_logger) << "Connection closed";
    }
};
```

The attribute provides high resolution of the time estimation and can even be used as a simple in-place performance profiling tool.

Tip

The `timer` attribute can even be used to profile the code in different modules without recompiling them. The trick is to wrap an expensive call to a foreign module with the thread-specific `timer` scoped attribute, which will markup all log records made from within the module with time readings.

Named scopes

```
#include <boost/log/attributes/named_scope.hpp>

// Supporting headers
#include <boost/log/support/exception.hpp>
```

The logging library supports maintaining scope stack tracking during the application's execution. This stack may either be written to log or be used for other needs (for example, to save the exact call sequence that led to an exception when throwing one). Each stack element contains the following information (see the `named_scope_entry` structure template definition):

- Scope name. It can be defined by the user or generated by the compiler, but in any case it must be a constant string literal (see Rationale).
- Source file name, where the scope begins. It is usually a result of the standard `__FILE__` macro expansion. Like the scope name, the file name must be a constant string literal.
- Line number in the source file. Usually it is a result of the standard `__LINE__` macro expansion.

The scope stack is implemented as a thread-specific global storage internally. There is the `named_scope` attribute that allows hooking this stack into the logging pipeline. This attribute generates value of the nested type `named_scope::scope_stack` which is the instance of the scope stack. The attribute can be registered in the following way:

```
logging::core::get()->add_global_attribute("Scope", attrs::named_scope());
```

Note that it is perfectly valid to register the attribute globally because the scope stack is thread-local anyway. This will also implicitly add scope tracking to all threads of the application, which is often exactly what is needed.

Now we can mark execution scopes with the macros `BOOST_LOG_FUNCTION` and `BOOST_LOG_NAMED_SCOPE` (the latter accepts the scope name as its argument). These macros automatically add source position information to each scope entry. An example follows:

```
void foo(int n)
{
    // Mark the scope of the function foo
    BOOST_LOG_FUNCTION();

    switch (n)
    {
    case 0:
        {
            // Mark the current scope
            BOOST_LOG_NAMED_SCOPE("case 0");
            BOOST_LOG(lg) << "Some log record";
            bar(); // call some function
        }
        break;

    case 1:
        {
            // Mark the current scope
            BOOST_LOG_NAMED_SCOPE("case 1");
            BOOST_LOG(lg) << "Some log record";
            bar(); // call some function
        }
        break;

    default:
        {
            // Mark the current scope
            BOOST_LOG_NAMED_SCOPE("default");
            BOOST_LOG(lg) << "Some log record";
            bar(); // call some function
        }
        break;
    }
}
```

After executing `foo` we will be able to see in the log that the `bar` function was called from `foo` and, more precisely, from the case statement that corresponds to the value of `n`. This may be very useful when tracking down subtle bugs that show up only when `bar` is called from a specific location (e.g. if `bar` is being passed invalid arguments in that particular location).

Note

The `BOOST_LOG_FUNCTION` macro uses compiler-specific extensions to generate the scope name from the enclosing function. C++11 defines a standard macro `__func__` for this purpose, but it is not universally supported. Additionally, format of the string is not standardized and may vary from one compiler to another. For this reason it is generally advised to use `BOOST_LOG_NAMED_SCOPE` instead of `BOOST_LOG_FUNCTION` to ensure consistent and portable behavior.

Another good use case is attaching the scope stack information to an exception. With the help of Boost.Exception, this is possible:

```
void bar(int x)
{
    BOOST_LOG_FUNCTION();

    if (x < 0)
    {
        // Attach a copy of the current scope stack to the exception
        throw boost::enable_error_info(std::range_error("x must not be negative"))
            << logging::current_scope();
    }
}

void foo()
{
    BOOST_LOG_FUNCTION();

    try
    {
        bar(-1);
    }
    catch (std::range_error& e)
    {
        // Acquire the scope stack from the exception object
        BOOST_LOG(lg) << "bar call failed: " << e.what() << ", scopes stack:\n"
            << *boost::get_error_info< logging::current_scope_info >(e);
    }
}
```

Note

In order this code to compile, the Boost.Exception support header has to be included.

Note

We do not inject the `named_scope` attribute into the exception. Since scope stacks are maintained globally, throwing an exception will cause stack unwinding and, as a result, will truncate the global stack. Instead we create a copy of the scope stack by calling `current_scope` at the throw site. This copy will be kept intact even if the global stack instance changes during the stack unwinding.

Current process identifier

```
#include <boost/log/attributes/current_process_id.hpp>
```

It is often useful to know the process identifier that produces the log, especially if the log can eventually combine the output of different processes. The `current_process_id` attribute is a constant that formats into the current process identifier. The value type of the attribute can be determined by the `current_process_id::value_type` typedef.

```
void foo()
{
    logging::core::get()->add_global_attribute(
        "ProcessID",
        attrs::current_process_id());
}
```

Current process name

```
#include <boost/log/attributes/current_process_name.hpp>
```

The `current_process_name` produces `std::string` values with the executable name of the current process.

Note

This attribute is not universally portable, although Windows, Linux and OS X are supported. The attribute may work on other POSIX systems as well, but it was not tested. If the process name cannot be obtained the attribute will generate a string with the process id.

```
void foo()
{
    logging::core::get()->add_global_attribute(
        "Process",
        attrs::current_process_name());
}
```

Current thread identifier

```
#include <boost/log/attributes/current_thread_id.hpp>
```

Multithreaded builds of the library also support the `current_thread_id` attribute with value type `current_thread_id::value_type`. The attribute will generate values specific to the calling thread. The usage is similar to the process id.

```
void foo()
{
    logging::core::get()->add_global_attribute(
        "ThreadID",
        attrs::current_thread_id());
}
```

Tip

You may have noticed that the attribute is registered globally. This will not result in all threads having the same ThreadID in log records as the attribute will always return a thread-specific value. The additional benefit is that you don't have to do a thing in the thread initialization routines to have the thread-specific attribute value in log records.

Function objects as attributes

```
#include <boost/log/attributes/function.hpp>
```

This attribute is a simple wrapper around a user-defined function object. Each attempt to acquire the attribute value results in the function object call. The result of the call is returned as the attribute value (this implies that the function must not return `void`). The function object attribute can be constructed with the `make_function` helper function, like this:

```
void foo()
{
    logging::core::get()->add_global_attribute("MyRandomAttr", attrs::make_function(&std::rand));
}
```

Auto-generated function objects, like the ones defined in Boost.Bind or STL, are also supported.

Note

Some deficient compilers may not support `result_of` construct properly. This metafunction is used in the `make_function` function to automatically detect the return type of the function object. If `result_of` breaks or detects incorrect type, one can try to explicitly specify the return type of the function object as a template argument to the `make_function` function.

Other attribute-related components

Attribute names

```
#include <boost/log/attributes/attribute_name.hpp>
```

Attribute names are represented with `attribute_name` objects which are used as keys in associative containers of attributes used by the library. The `attribute_name` object can be created from a string, so most of the time its use is transparent.

The name is not stored as a string within the `attribute_name` object. Instead, a process-wide unique identifier is generated and associated with the particular name. This association is preserved until the process termination, so every time the `attribute_name` object is created for the same name it obtains the same identifier. The association is not stable across the different runs of the application though.

Warning

Since the association between string names and identifiers involves some state allocation, it is not advised to use externally provided or known to be changing strings for attribute names. Even if the name is not used in any log records, the association is preserved anyway. Continuously constructing `attribute_name` objects with unique string names may manifest itself as a memory leak.

Working with identifiers is much more efficient than with strings. For example, copying does not involve dynamic memory allocation and comparison operators are very lightweight. On the other hand, it is easy to get a human-readable attribute name for presentation, if needed.

The `attribute_name` class supports an empty (uninitialized) state when default constructed. In this state the name object is not equal to any other initialized name object. Uninitialized attribute names should not be passed to the library but can be useful in some contexts (e.g. when a delayed initialization is desired).

Attribute set

```
#include <boost/log/attributes/attribute_set.hpp>
```

Attribute set is an unordered associative container that maps attribute names to attributes. It is used in loggers and the logging core to store source-specific, thread-specific and global attributes. The interface is very similar to STL associative containers and is described in the `attribute_set` class reference.

Attribute value set

```
#include <boost/log/attributes/attribute_value_set.hpp>
```

Attribute value set is an unordered associative container that maps attribute names to attribute values. This container is used in log records to represent attribute values. Unlike conventional containers, `attribute_value_set` does not support removing or modifying elements after being inserted. This warrants that the attribute values that participated filtering will not disappear from the log record in the middle of the processing.

Additionally, the set can be constructed from three attribute sets, which are interpreted as the sets of source-specific, thread-specific and global attributes. The constructor adopts attribute values from the three attribute sets into a single set of attribute values. After construction, `attribute_value_set` is considered to be in an unfrozen state. This means that the container may keep references to the elements of the attribute sets used as the source for the value set construction. While in this state, neither the attribute sets nor the value set must not be modified in any way as this may make the value set corrupted. The value set can be used for reading in this state, its lookup operations will perform as usual. The value set can be frozen by calling the `freeze` method; the set will no longer be attached to the original attribute sets and will be available for further insertions after this call. The library will ensure that the value set is always frozen when a log record is returned from the logging core; the set is not frozen during filtering though.

Tip

In the unfrozen state the value set may not have all attribute values acquired from the attributes. It will only acquire the values as requested by filters. After freezing the container has all attribute values. This transition allows to optimize the library so that attribute values are only acquired when needed.

For further details on the container interface please consult the `attribute_value_set` reference.

Attribute value extraction and visitation

Since attribute values do not expose the stored value in the interface, an API is needed to acquire the stored value. The library provides two APIs for this purpose: value visitation and extraction.

Value visitation

```
#include <boost/log/attributes/value_visitation_fwd.hpp>
#include <boost/log/attributes/value_visitation.hpp>
```

Attribute value visitation implements the visitor design pattern, hence the naming. The user has to provide a unary function object (a visitor) which will be invoked on the stored attribute value. The caller also has to provide the expected type or set of possible types of the stored value. Obviously, the visitor must be capable of receiving an argument of the expected type. Visitation will only succeed if the stored type matches the expectation.

In order to apply the visitor, one should call the `visit` function on the attribute value. Let's see an example:

```
// Our attribute value visitor
struct print_visitor
{
    typedef void result_type;

    result_type operator() (int val) const
    {
        std::cout << "Visited value is int: " << val << std::endl;
    }

    result_type operator() (std::string const& val) const
    {
        std::cout << "Visited value is string: " << val << std::endl;
    }
};

void print_value(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Apply our visitor
    logging::visitation_result result = logging::visit< types >(attr, print_visitor());

    // Check the result
    if (result)
        std::cout << "Visitation succeeded" << std::endl;
    else
        std::cout << "Visitation failed" << std::endl;
}
```

See the complete code.

In this example we print the stored attribute value in our `print_visitor`. We expect the attribute value to have either `int` or `std::string` stored type; only in this case the visitor will be invoked and the visitation result will be positive. In case of failure the `visitation_result` class provides additional information on the failure reason. The class has the method named `code` which returns visitation error code. The following error codes are possible:

- `ok` - visitation succeeded, the visitor has been invoked; visitation result is positive when this code is used
- `value_not_found` - visitation failed because the requested value was not found; this code is used when visitation is applied to a log record or a set of attribute values rather than a single value
- `value_has_invalid_type` - visitation failed because the value has type differing from any of the expected types

By default the visitor function result is ignored but it is possible to obtain it. To do this one should use a special `save_result` wrapper for the visitor; the wrapper will save the visitor resulting value into an external variable captured by reference. The visitor result is initialized when the returned `visitation_result` is positive. See the following example where we compute the hash value on the stored value.

```
struct hash_visitor
{
    typedef std::size_t result_type;

    result_type operator() (int val) const
    {
        std::size_t h = val;
        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }

    result_type operator() (std::string const& val) const
    {
        std::size_t h = 0;
        for (std::string::const_iterator it = val.begin(), end = val.end(); it != end; ++it)
            h += *it;

        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }
};

void hash_value(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Apply our visitor
    std::size_t h = 0;
   logging::visitation_result result = logging::visit< types >(attr, logging::save_result(hash_vis↵
itor(), h));

    // Check the result
    if (result)
        std::cout << "Visitation succeeded, hash value: " << h << std::endl;
    else
        std::cout << "Visitation failed" << std::endl;
}
```

See the complete code.

Tip

When there is no default state for the visitor result it is convenient to use Boost.Optional to wrap the returned value. The `optional` will be initialized with the visitor result if visitation succeeded. In case if visitor is polymorphic (i.e. it has different result types depending on its argument type) Boost.Variant can be used to receive the resulting value. It is also worthwhile to use an empty type, such as `boost::blank`, to indicate the uninitialized state of the `variant`.

As it has been mentioned, visitation can also be applied to log records and attribute value sets. The syntax is the same, except that the attribute name also has to be specified. The `visit` algorithm will try to find the attribute value by name and then apply the visitor to the found element.

```
void hash_value(logging::record_view const& rec, logging::attribute_name name)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Apply our visitor
    std::size_t h = 0;
    logging::visitation_result result = logging::visit< types >(name, rec, logging::save_res↵
ult(hash_visitor(), h));

    // Check the result
    if (result)
        std::cout << "Visitation succeeded, hash value: " << h << std::endl;
    else
        std::cout << "Visitation failed" << std::endl;
}
```

Also, for convenience `attribute_value` has the method named `visit` with the same meaning as the free function applied to the attribute value.

Value extraction

```
#include <boost/log/attributes/value_extraction_fwd.hpp>
#include <boost/log/attributes/value_extraction.hpp>
```

Attribute value extraction API allows to acquire a reference to the stored value. It does not require a visitor function object, but the user still has to provide the expected type or a set of types the stored value may have.

```
void print_value(logging::attribute_value const& attr)
{
    // Extract a reference to the stored value
    logging::value_ref< int > val = logging::extract< int >(attr);

    // Check the result
    if (val)
        std::cout << "Extraction succeeded: " << val.get() << std::endl;
    else
        std::cout << "Extraction failed" << std::endl;
}
```

See the complete code.

In this example we expect the attribute value to have the stored type `int`. The `extract` function attempts to extract a reference to the stored value and returns the filled `value_ref` object if succeeded.

Value extraction can also be used with a set of expected stored types. The following code snippet demonstrates this:

```
void print_value_multiple_types(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Extract a reference to the stored value
    logging::value_ref< types > val = logging::extract< types >(attr);

    // Check the result
    if (val)
    {
        std::cout << "Extraction succeeded" << std::endl;
        switch (val.which())
        {
        case 0:
            std::cout << "int: " << val.get< int >() << std::endl;
            break;

        case 1:
            std::cout << "string: " << val.get< std::string >() << std::endl;
            break;
        }
    }
    else
        std::cout << "Extraction failed" << std::endl;
}
```

Notice that we used `which` method of the returned reference to dispatch between possible types. The method returns the index of the type in the `types` sequence. Also note that the `get` method now accepts an explicit template parameter to select the reference type to acquire; naturally, this type must correspond to the actual referred type, which is warranted by the switch/case statement in our case.

Value visitation is also supported by the `value_ref` object. Here is how we compute a hash value from the extracted value:

```
struct hash_visitor
{
    typedef std::size_t result_type;

    result_type operator() (int val) const
    {
        std::size_t h = val;
        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }

    result_type operator() (std::string const& val) const
    {
        std::size_t h = 0;
        for (std::string::const_iterator it = val.begin(), end = val.end(); it != end; ++it)
            h += *it;

        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }
};

void hash_value(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Extract the stored value
    logging::value_ref< types > val = logging::extract< types >(attr);

    // Check the result
    if (val)
        std::cout << "Extraction succeeded, hash value: " << val.apply_visitor(hash_visit↵
or()) << std::endl;
    else
        std::cout << "Extraction failed" << std::endl;
}
```

Lastly, like with value visitation, value extraction can also be applied to log records and attribute value sets.

```
void hash_value(logging::record_view const& rec, logging::attribute_name name)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Extract the stored value
    logging::value_ref< types > val = logging::extract< types >(name, rec);

    // Check the result
    if (val)
        std::cout << "Extraction succeeded, hash value: " << val.apply_visitor(hash_visit↵
or()) << std::endl;
    else
        std::cout << "Extraction failed" << std::endl;
}
```

In addition the library provides two special variants of the `extract` function: `extract_or_throw` and `extract_or_default`. As the naming implies, the functions provide different behavior in case if the attribute value cannot be extracted. The former one throws an exception if the value cannot be extracted and the latter one returns the default value.

Warning

Care must be taken with the `extract_or_default` function. The function accepts the default value is accepted by constant reference, and this reference can eventually be returned from `extract_or_default`. If a temporary object as used for the default value, user must ensure that the result of `extract_or_default` is saved by value and not by reference. Otherwise the saved reference may become dangling when the temporary is destroyed.

Similarly to `visit`, the `attribute_value` class has methods named `extract`, `extract_or_throw` and `extract_or_default` with the same meaning as the corresponding free functions applied to the attribute value.

Scoped attributes

```
#include <boost/log/attributes/scoped_attribute.hpp>
```

Scoped attributes are a powerful mechanism of tagging log records that can be used for different purposes. As the naming implies, scoped attributes are registered in the beginning of a scope and unregistered on the end of the scope. The mechanism includes the following macros:

```
BOOST_LOG_SCOPED_LOGGER_ATTR(logger, attr_name, attr);
BOOST_LOG_SCOPED_THREAD_ATTR(attr_name, attr);
```

The first macro registers a source-specific attribute in the `logger` logger object. The attribute name and the attribute itself are given in the `attr_name` and `attr` arguments. The second macro does exactly the same but the attribute is registered for the current thread in the logging core (which does not require a logger).

Note

If an attribute with the same name is already registered in the logger/logging core, the macros won't override the existing attribute and will eventually have no effect. See Rationale for a more detailed explanation of the reasons for such behavior.

Usage example follows:

```
BOOST_LOG_DECLARE_GLOBAL_LOGGER(my_logger, src::logger_mt)

void foo()
{
    // This log record will also be marked with the "Tag" attribute,
    // whenever it is called from the A::bar function.
    // It will not be marked when called from other places.
    BOOST_LOG(get_my_logger()) << "A log message from foo";
}

struct A
{
    src::logger m_Logger;

    void bar()
    {
        // Set a thread-wide markup tag.
        // Note the additional parentheses to form a Boost.PP sequence.
        BOOST_LOG_SCOPED_THREAD_ATTR("Tag",
            attrs::constant< std::string >("Called from A::bar"));

        // This log record will be marked
        BOOST_LOG(m_Logger) << "A log message from A::bar";

        foo();
    }
};

int main(int, char*[])
{
    src::logger lg;

    // Let's measure our application run time
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "RunTime", attrs::timer());

    // Mark application start.
    // The "RunTime" attribute should be nearly 0 at this point.
    BOOST_LOG(lg) << "Application started";

    // Note that no other log records are affected by the "RunTime" attribute.
    foo();

    A a;
    a.bar();

    // Mark application ending.
    // The "RunTime" attribute will show the execution time elapsed.
    BOOST_LOG(lg) << "Application ended";

    return 0;
}
```

It is quite often convenient to mark a group of log records with a constant value in order to be able to filter the records later. The library provides two convenience macros just for this purpose:

```
BOOST_LOG_SCOPED_LOGGER_TAG(logger, tag_name, tag_value);
BOOST_LOG_SCOPED_THREAD_TAG(tag_name, tag_value);
```

The macros are effectively wrappers around `BOOST_LOG_SCOPED_LOGGER_ATTR` and `BOOST_LOG_SCOPED_THREAD_ATTR`, respectively. For example, the "Tag" scoped attribute from the example above can be registered like this:

```
BOOST_LOG_SCOPED_THREAD_TAG("Tag", "Called from A::bar");
```

Warning

When using scoped attributes, make sure that the scoped attribute is not altered in the attribute set in which it was registered. For example, one should not clear or reinstall the attribute set of the logger if there are logger-specific scoped attributes registered in it. Otherwise the program will likely crash. This issue is especially critical in multi-threaded application, when one thread may not know whether there are scoped attributes in the logger or there are not. Future releases may solve this limitation but currently the scoped attribute must remain intact until unregistered on leaving the scope.

Although the described macros are intended to be the primary interface for the functionality, there is also a C++ interface available. It may be useful if the user decides to develop his own macros that cannot be based on the existing ones.

Any scoped attribute is attached to a generic sentry object of type `scoped_attribute`. As long as the sentry exists, the attribute is registered. There are several functions that create sentries for source or thread-specific attributes:

```
// Source-specific scoped attribute registration
template< typename LoggerT >
[unspecified] add_scoped_logger_attribute(
    LoggerT& l,
    attribute_name const& name,
    attribute const& attr);

// Thread-specific scoped attribute registration
template< typename CharT >
[unspecified] add_scoped_thread_attribute(
    attribute_name const& name,
    attribute const& attr);
```

An object of the `scoped_attribute` type is able to attach results of each of these functions on its construction. For example, `BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "RunTime", attrs::timer())` can roughly be expanded to this:

```
attrs::scoped_attribute sentry =
    attrs::add_scoped_logger_attribute(lg, "RunTime", attrs::timer());
```

Utilities

String literals

```
#include <boost/log/utility/string_literal.hpp>
```

String literals are used in several places throughout the library. However, this component can be successfully used outside of the library in users' code. It is header-only and does not require linking with the library binary. String literals can improve performance significantly if there is no need to modify stored strings. What is also important, since string literals do not dynamically allocate memory, it is easier to maintain exception safety when using string literals instead of regular strings.

The functionality is implemented in the `basic_string_literal` class template, which is parametrized with the character and character traits, similar to `std::basic_string`. There are also two convenience typedefs provided: `string_literal` and `wstring_literal`, for narrow and wide character types, respectively. In order to ease string literal construction in generic code there is also a `str_literal` function template that accepts a string literal and returns a `basic_string_literal` instance for the appropriate character type.

String literals support interface similar to STL strings, except for string modification functions. However, it is possible to assign to or clear string literals, as long as only string literals involved. Relational and stream output operators are also supported.

Type information wrapper

```
#include <boost/log/utility/type_info_wrapper.hpp>
```

The language support for run time type information is essential for the library. But partially because of limitations that the C++ Standard imposes on this feature, partially because of differences of implementation of different compilers, there was a need for a lightweight wrapper around the `std::type_info` class to fill the gaps. The table below briefly shows the differences between the `std::type_info` and `type_info_wrapper` classes.

Table 6. Type information classes comparison

Feature	**`std::type_info`**	**`type_info_wrapper`**
Is default-constructable	No	Yes. The default-constructed wrapper is in an empty state.
Is copy-constructable	No	Yes
Is assignable	No	Yes
Is swappable	No	Yes
Life time duration	Static, until the application terminates	Dynamic
Has an empty state	No	Yes. In empty state the type info wrapper is never equal to other non-empty type info wrappers. It is equal to other empty type info wrappers and can be ordered with them.
Supports equality comparison	Yes	Yes
Supports ordering	Yes, partial ordering with the `before` method.	Yes, partial ordering with the complete set of comparison operators. The semantics of ordering is similar to the `std::type_info::before` method.
Supports printable representation of type	Yes, with the `name` function.	Yes, with `pretty_name` function. The function does its best to print the type name in a human-readable form. On some platforms it does better than `std::type_info::name`.

Given the distinctions above, using type information objects becomes much easier. For example, the ability to copy and order with regular operators allows using `type_info_wrapper` with containers. The ability to default-construct and assign allows using type information as a regular object and not to resort to pointers which may be unsafe.

Type dispatchers

```
#include <boost/log/utility/type_dispatch/type_dispatcher.hpp>
```

Type dispatchers are used throughout the library in order to work with attribute values. Dispatchers allow acquiring the stored attribute value using the Visitor concept. The most notable places where the functionality is used are filters and formatters. However, this mechanism is orthogonal to attributes and can be used for other purposes as well. Most of the time users won't need to dig into the details of type dispatchers, but this information may be useful for those who intend to extend the library and wants to understand what's under the hood.

Every type dispatcher supports the `type_dispatcher` interface. When an attribute value needs to be extracted, this interface is passed to the attribute value object, which then tries to acquire the callback for the actual type of the value. All callbacks are objects of the [class_type_dispatcher_callback] class template, instantiated on the actual type of the value. If the dispatcher is able to consume the value of the requested type, it must return a non-empty callback object. When (and if) the corresponding callback is acquired, the attribute value object only has to pass the contained value to its `operator ()`.

Happily, there is no need to write type dispatchers from scratch. The library provides two kinds of type dispatchers that implement the `type_dispatcher` and [class_type_dispatcher_callback] interfaces and encapsulate the callback lookup.

Static type dispatcher

```
#include <boost/log/utility/type_dispatch/static_type_dispatcher.hpp>
```

Static type dispatchers are used when the set of types that needs to be supported for extraction is known at compile time. The `static_type_dispatcher` class template is parametrized with an MPL type sequence of types that need to be supported. The dispatcher inherits from the `type_dispatcher` interface which provides the `get_callback` method for acquiring the function object to invoke on the stored value. All you need to do is provide a visitor function object to the dispatcher at construction point and invoke the callback when dispatching the stored value:

```
// Base interface for the custom opaque value
struct my_value_base
{
    virtual ~my_value_base() {}
    virtual bool dispatch(logging::type_dispatcher& dispatcher) const = 0;
};

// A simple attribute value
template< typename T >
struct my_value :
    public my_value_base
{
    T m_value;

    explicit my_value(T const& value) : m_value(value) {}

    // The function passes the contained type into the dispatcher
    bool dispatch(logging::type_dispatcher& dispatcher) const
    {
        logging::type_dispatcher::callback< T > cb = dispatcher.get_callback< T >();
        if (cb)
        {
            cb(m_value);
            return true;
        }
        else
            return false;
    }
};

// Value visitor for the supported types
struct print_visitor
{
    typedef void result_type;

    // Implement visitation logic for all supported types
    void operator() (int const& value) const
    {
        std::cout << "Received int value = " << value << std::endl;
    }
    void operator() (double const& value) const
    {
```

```
        std::cout << "Received double value = " << value << std::endl;
    }
    void operator() (std::string const& value) const
    {
        std::cout << "Received string value = " << value << std::endl;
    }
};

// Prints the supplied value
bool print(my_value_base const& val)
{
    typedef boost::mpl::vector< int, double, std::string > types;

    print_visitor visitor;
    logging::static_type_dispatcher< types > disp(visitor);

    return val.dispatch(disp);
}
```

See the complete code.

Dynamic type dispatcher

```
#include <boost/log/utility/type_dispatch/dynamic_type_dispatcher.hpp>
```

If the set of types that have to be supported is not available at compile time, the `dynamic_type_dispatcher` class is there to help. One can use its `register_type` method to add support for a particular type. The user has to pass a function object along with the type, this functor will be called when a visitor for the specified type is invoked. Considering the `my_value` from the code sample for static type dispatcher is intact, the code can be rewritten as follows:

```
// Visitor functions for the supported types
void on_int(int const& value)
{
    std::cout << "Received int value = " << value << std::endl;
}

void on_double(double const& value)
{
    std::cout << "Received double value = " << value << std::endl;
}

void on_string(std::string const& value)
{
    std::cout << "Received string value = " << value << std::endl;
}

logging::dynamic_type_dispatcher disp;

// The function initializes the dispatcher object
void init_disp()
{
    // Register type visitors
    disp.register_type< int >(&on_int);
    disp.register_type< double >(&on_double);
    disp.register_type< std::string >(&on_string);
}

// Prints the supplied value
bool print(my_value_base const& val)
{
    return val.dispatch(disp);
}
```

See the complete code.

Of course, complex function objects, like those provided by Boost.Bind, are also supported.

Predefined type sequences

```
#include <boost/log/utility/type_dispatch/standard_types.hpp>
#include <boost/log/utility/type_dispatch/date_time_types.hpp>
```

One may notice that when using type dispatchers and defining filters and formatters it may be convenient to have some predefined type sequences to designate frequently used sets of types. The library provides several such sets.

Table 7. Standard types (standard_types.hpp)

Type sequence	Meaning
`integral_types`	All integral types, including `bool`, character and 64 bit integral types, if available
`floating_point_types`	Floating point types
`numeric_types`	Includes `integral_types` and `floating_point_types`
`string_types`	Narrow and wide string types. Currently only includes STL string types and string literals.

There are also a number of time-related type sequences available:

Table 8. Time-related types (date_time_types.hpp)

Type sequence	Meaning
`native_date_time_types`	All types defined in C/C++ standard that have both date and time portions
`boost_date_time_types`	All types defined in Boost.DateTime that have both date and time portions
`date_time_types`	Includes `native_date_time_types` and `boost_date_time_types`
`native_date_types`	All types defined in C/C++ standard that have date portion. Currently equivalent to `native_date_time_types`.
`boost_date_types`	All types defined in Boost.DateTime that have date portion
`date_types`	Includes `native_date_types` and `boost_date_types`
`native_time_types`	All types defined in C/C++ standard that have time portion. Currently equivalent to `native_date_time_types`.
`boost_time_types`	All types defined in Boost.DateTime that have time portion. Currently equivalent to `boost_date_time_types`.
`time_types`	Includes `native_time_types` and `boost_time_types`
`native_time_duration_types`	All types defined in C/C++ standard that are used to represent time duration. Currently only includes `double`, as the result type of the `difftime` standard function.
`boost_time_duration_types`	All time duration types defined in Boost.DateTime
`time_duration_types`	Includes `native_time_duration_types` and `boost_time_duration_types`
`boost_time_period_types`	All time period types defined in Boost.DateTime
`time_period_types`	Currently equivalent to `boost_time_period_types`

Value reference wrapper

```
#include <boost/log/utility/value_ref.hpp>
```

The `value_ref` class template is an optional reference wrapper which is used by the library to refer to the stored attribute values. To a certain degree it shares features of Boost.Optional and Boost.Variant components.

The template has two type parameters. The first is the referred type. It can also be specified as a Boost.MPL type sequence, in which case the `value_ref` wrapper may refer to either type in the sequence. In this case, the `which` method will return the index of the referred type within the sequence. The second template parameter is an optional tag type which can be used to customize formatting behavior. This tag is forwarded to the `to_log` manipulator when the wrapper is put to a `basic_formatting_ostream` stream, which is used by the library for record formatting. For an example see how attribute value extraction is implemented:

```
void print_value_multiple_types(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Extract a reference to the stored value
    logging::value_ref< types > val = logging::extract< types >(attr);

    // Check the result
    if (val)
    {
        std::cout << "Extraction succeeded" << std::endl;
        switch (val.which())
        {
        case 0:
            std::cout << "int: " << val.get< int >() << std::endl;
            break;

        case 1:
            std::cout << "string: " << val.get< std::string >() << std::endl;
            break;
        }
    }
    else
        std::cout << "Extraction failed" << std::endl;
}
```

See the complete code.

The `value_ref` wrapper also supports applying a visitor function object to the referred object. This can be done by calling one of the following methods:

- `apply_visitor`. This method should only be used on a valid (non-empty) reference. The method returns the visitor result.
- `apply_visitor_optional`. The method checks if the reference is valid and applies the visitor to the referred value if it is. The method returns the visitor result wrapped into `boost::optional` which will be filled only if the reference is valid.
- `apply_visitor_or_default`. If the reference is valid, the method applies the visitor on the referred value and returns its result. Otherwise the method returns a default value passed as the second argument.

Note

Regardless of the method used, the visitor function object must define the `result_type` typedef. Polymorphic visitors are not supported as this would complicate the `value_ref` interface too much. This requirement also precludes free functions and C++11 lambda functions from being used as visitors. Please, use Boost.Bind or similar wrappers in such cases.

Here is an example of applying a visitor:

```
struct hash_visitor
{
    typedef std::size_t result_type;

    result_type operator() (int val) const
    {
        std::size_t h = val;
        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }

    result_type operator() (std::string const& val) const
    {
        std::size_t h = 0;
        for (std::string::const_iterator it = val.begin(), end = val.end(); it != end; ++it)
            h += *it;

        h = (h << 15) + h;
        h ^= (h >> 6) + (h << 7);
        return h;
    }
};

void hash_value(logging::attribute_value const& attr)
{
    // Define the set of expected types of the stored value
    typedef boost::mpl::vector< int, std::string > types;

    // Extract the stored value
    logging::value_ref< types > val = logging::extract< types >(attr);

    // Check the result
    if (val)
        std::cout << "Extraction succeeded, hash value: " << val.apply_visitor(hash_visit↵
or()) << std::endl;
    else
        std::cout << "Extraction failed" << std::endl;
}
```

Log record ordering

```
#include <boost/log/utility/record_ordering.hpp>
```

There are cases when log records need to be ordered. One possible use case is storing records in a container or a priority queue. The library provides two types of record ordering predicates out of the box:

Abstract record ordering

The `abstract_ordering` class allows application of a quick opaque ordering. The result of this ordering is not stable between different runs of the application and in general cannot be predicted before the predicate is applied, however it provides the best performance. The `abstract_ordering` class is a template that is specialized with an optional predicate function that will be able to compare `const void*` pointers. By default an `std::less` equivalent is used.

```
// A set of unique records
std::set< logging::record_view, logging::abstract_ordering< > > m_Records;
```

This kind of ordering can be useful if the particular order of log records is not important but nevertheless some order is required.

Attribute value based ordering

This kind of ordering is implemented with the `attribute_value_ordering` class and is based on the attribute values attached to the record. The predicate will seek for an attribute value with the specified name in both records being ordered and attempt to compare the attribute values.

```
// Ordering type definition
typedef logging::attribute_value_ordering<
    int     // attribute value type
> ordering;

// Records organized into a queue based on the "Severity" attribute value
std::priority_queue<
    logging::record_view,
    std::vector< logging::record_view >,
    ordering
> m_Records(ordering("Severity"));
```

Like the `abstract_ordering`, `attribute_value_ordering` also accepts the second optional template parameter, which should be the predicate to compare attribute values (`int`s in the example above). By default, an `std::less` equivalent is used.

You can also use the `make_attr_ordering` generator function to automatically generate the `attribute_value_ordering` instance based on the attribute value name and the ordering function. This might be useful if the ordering function has a non-trivial type, like the ones Boost.Bind provides.

Exception handlers

```
#include <boost/log/utility/exception_handler.hpp>
```

The library provides exception handling hooks in different places. Tools, defined in this header, provide an easy way of implementing function objects suitable for such hooks.

An exception handler is a function object that accepts no arguments. The result of the exception handler is ignored and thus should generally be `void`. Exception handlers are called from within `catch` sections by the library, therefore in order to reacquire the exception object it has to rethrow it. The header defines an `exception_handler` template functor that does just that and then forwards the exception object to a unary user-defined functional object. The `make_exception_handler` function can be used to simplify the handler construction. All expected exception types should be specified explicitly in the call, in the order they would appear in the `catch` sections (i.e. from most specific ones to the most general ones).

```
struct my_handler
{
    typedef void result_type;

    void operator() (std::runtime_error const& e) const
    {
        std::cout << "std::runtime_error: " << e.what() << std::endl;
    }
    void operator() (std::logic_error const& e) const
    {
        std::cout << "std::logic_error: " << e.what() << std::endl;
        throw;
    }
};

void init_exception_handler()
{
    // Setup a global exception handler that will call my_handler::operator()
    // for the specified exception types
    logging::core::get()->set_exception_handler(logging::make_exception_handler<
        std::runtime_error,
        std::logic_error
    >(my_handler()));
}
```

As you can see, you can either suppress the exception by returning normally from `operator()` in the user-defined handler functor, or rethrow the exception, in which case it will propagate further. If it appears that the exception handler is invoked for an exception type that cannot be caught by any of the specified types, the exception will be propagated without any processing. In order to catch such situations, there exists the `nothrow_exception_handler` class. It invokes the user-defined functor with no arguments if it cannot determine the exception type.

```
struct my_handler_nothrow
{
    typedef void result_type;

    void operator() (std::runtime_error const& e) const
    {
        std::cout << "std::runtime_error: " << e.what() << std::endl;
    }
    void operator() (std::logic_error const& e) const
    {
        std::cout << "std::logic_error: " << e.what() << std::endl;
        throw;
    }
    void operator() () const
    {
        std::cout << "unknown exception" << std::endl;
    }
};

void init_exception_handler_nothrow()
{
    // Setup a global exception handler that will call my_handler::operator()
    // for the specified exception types. Note the std::nothrow argument that
    // specifies that all other exceptions should also be passed to the functor.
    logging::core::get()->set_exception_handler(logging::make_exception_handler<
        std::runtime_error,
        std::logic_error
    >(my_handler_nothrow(), std::nothrow));
}
```

It is sometimes convenient to completely suppress all exceptions at a certain library level. The `make_exception_suppressor` function creates an exception handler that simply does nothing upon exception being caught. For example, this way we can disable all exceptions from the logging library:

```
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // Disable all exceptions
    core->set_exception_handler(logging::make_exception_suppressor());
}
```

Output manipulators

The library provides a number of stream manipulators that may be useful in some contexts.

Customized logging manipulator

```
#include <boost/log/utility/manipulators/to_log.hpp>
```

The `to_log` function creates a stream manipulator that simply outputs the adopted value to the stream. By default its behavior is equivalent to simply putting the value to the stream. However, the user is able to overload the `operator<<` for the adopted value to override formatting behavior when values are formatted for logging purposes. This is typically desired when the regular `operator<<` is employed for other tasks (such as serialization) and its behavior is neither suitable for logging nor can be easily changed. For example:

```
std::ostream& operator<<
(
    std::ostream& strm,
    logging::to_log_manip< int > const& manip
)
{
    strm << std::setw(4) << std::setfill('0') << std::hex << manip.get() << std::dec;
    return strm;
}

void test_manip()
{
    std::cout << "Regular output: " << 1010 << std::endl;
    std::cout << "Log output: " << logging::to_log(1010) << std::endl;
}
```

The second streaming statement in the `test_manip` function will invoke our custom stream insertion operator which defines special formatting rules.

It is also possible to define different formatting rules for different value contexts as well. The library uses this feature to allow different formatting ruled for different attribute values, even if the stored value type is the same. To do so one has to specify an explicit template argument for `to_log`, a tag type, which will be embedded into the manipulator type and thus will allow to define different insertion operators:

```
struct tag_A;
struct tag_B;

std::ostream& operator<<
(
    std::ostream& strm,
    logging::to_log_manip< int, tag_A > const& manip
)
{
    strm << "A[" << manip.get() << "]";
    return strm;
}

std::ostream& operator<<
(
    std::ostream& strm,
    logging::to_log_manip< int, tag_B > const& manip
)
{
    strm << "B[" << manip.get() << "]";
    return strm;
}

void test_manip_with_tag()
{
    std::cout << "Regular output: " << 1010 << std::endl;
    std::cout << "Log output A: " << logging::to_log< tag_A >(1010) << std::endl;
    std::cout << "Log output B: " << logging::to_log< tag_B >(1010) << std::endl;
}
```

See the complete code.

Note

The library uses `basic_formatting_ostream` stream type for record formatting, so when customizing attribute value formatting rules the `operator<<` must use `basic_formatting_ostream` instead of `std::ostream`.

Attribute value attaching manipulator

```
#include <boost/log/utility/manipulators/add_value.hpp>
```

The `add_value` function creates a manipulator that attaches an attribute value to a log record. This manipulator can only be used in streaming expressions with the `basic_record_ostream` stream type (which is the case when log record message is formatted). Since the message text is only formatted after filtering, attribute values attached with this manipulator do not affect filtering and can only be used in formatters and sinks themselves.

In addition to the value itself, the manipulator also requires the attribute name to be provided. For example:

```
// Creates a log record with attribute value "MyAttr" of type int attached
BOOST_LOG(lg) << logging::add_value("MyAttr", 10) << "Hello world!";
```

Binary dump manipulator

```
#include <boost/log/utility/manipulators/dump.hpp>
```

The `dump` function creates a manipulator that outputs binary contents of a contiguous memory region. This can be useful for logging some low level binary data, such as encoded network packets or entries of a binary file. The use is quite straightforward:

```
void on_receive(std::vector< unsigned char > const& packet)
{
    // Outputs something like "Packet received: 00 01 02 0a 0b 0c"
    BOOST_LOG(lg) << "Packet received: " << logging::dump(packet.data(), packet.size());
}
```

The manipulator also allows to limit the amount of data to be output, in case if the input data can be too large. Just specify the maximum number of bytes of input to dump as the last argument:

```
void on_receive(std::vector< unsigned char > const& packet)
{
    // Outputs something like "Packet received: 00 01 02 03 04 05 06 07 and 67 bytes more"
    BOOST_LOG(lg) << "Packet received: " << logging::dump(packet.data(), packet.size(), 8);
}
```

There is another manipulator called `dump_elements` for printing binary representation of non-byte array elements. The special manipulator for this case is necessary because the units of the size argument of `dump` can be confusing (is it in bytes or in elements?). Therefore `dump` will not compile when used for non-byte input data. `dump_elements` accepts the same arguments, and its size-related arguments always designate the number of elements to process.

```
void process(std::vector< double > const& matrix)
{
   // Note that dump_elements accepts the number of elements in the matrix, not its size in bytes
    BOOST_LOG(lg) << "Matrix dump: " << logging::dump_elements(matrix.data(), matrix.size());
}
```

Tip

Both these manipulators can also be used with regular output streams, not necessarily loggers.

Simplified library initialization tools

This part of the library is provided in order to simplify logging initialization and provide basic tools to develop user-specific initialization mechanisms. It is known that setup capabilities and preferences may vary widely from application to application, therefore the library does not attempt to provide a universal solution for this task. The provided tools are mostly intended to serve as a quick drop-in support for logging setup and a set of instruments to implement something more elaborate and more fitting users' needs.

Some of the features described in this section will require the separate library binary, with name based on "boost_log_setup" substring. This binary depends on the main library.

Convenience functions

```
#include <boost/log/utility/setup/console.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
```

The library provides a number of functions that simplify some common initialization procedures, like sink and commonly used attributes registration. This is not much functionality. However, it saves a couple of minutes of learning the library for a newcomer.

Logging to the application console is the simplest way to see the logging library in action. To achieve this, one can initialize the library with a single function call, like this:

```
int main(int, char*[])
{
    // Initialize logging to std::clog
    logging::add_console_log();

    // Here we go, we can write logs right away
    src::logger lg;
    BOOST_LOG(lg) << "Hello world!";

    return 0;
}
```

Pretty easy, isn't it? There is also the `wadd_console_log` function for wide-character console. If you want to put logs to some other standard stream, you can pass the stream to the `add_console_log` function as an argument. E.g. enabling logging to `std::cout` instead of `std::clog` would look like this:

```
logging::add_console_log(std::cout);
```

What's important, is that you can further manage the console sink if you save the `shared_ptr` to the sink that this function returns. This allows you to set up things like filter, formatter and auto-flush flag.

```
int main(int, char*[])
{
    // Initialize logging to std::clog
    boost::shared_ptr<
        sinks::synchronous_sink< sinks::text_ostream_backend >
    > sink = logging::add_console_log();

    sink->set_filter(expr::attr< int >("Severity") >= 3);
    sink->locked_backend()->auto_flush(true);

    // Here we go, we can write logs right away
    src::logger lg;
    BOOST_LOG(lg) << "Hello world!";

    return 0;
}
```

Similarly to console, one can use a single function call to enable logging to a file. All you have to do is to provide the file name:

```
int main(int, char*[])
{
    // Initialize logging to the "test.log" file
    logging::add_file_log("test.log");

    // Here we go, we can write logs right away
    src::logger lg;
    BOOST_LOG(lg) << "Hello world!";

    return 0;
}
```

The `add_console_log` and `add_file_log` functions do not conflict and may be combined freely, so it is possible to set up logging to the console and a couple of files, including filtering and formatting, in about 10 lines of code.

Lastly, there is an `add_common_attributes` function that registers two frequently used attributes: "LineID" and "TimeStamp". The former counts log record being made and has attribute value `unsigned int`. The latter, as its name implies, provides the current time for each log record, in the form of `boost::posix_time::ptime` (see Boost.DateTime). These two attributes are registered

globally, so they will remain available in all threads and loggers. This makes the final version of our code sample look something like this:

```
int main(int, char*[])
{
    // Initialize sinks
    logging::add_console_log()->set_filter(expr::attr< int >("Severity") >= 4);

    logging::formatter formatter =
        expr::stream
            << expr::attr< unsigned int >("LineID") << ": "
            << expr::format_date_time< boost::posix_time::ptime >("TimeStamp", "%Y-%m-%d ↵
%H:%M:%S") << " *"
            << expr::attr< int >("Severity") << "* "
            << expr::message;

    logging::add_file_log("complete.log")->set_formatter(formatter);

    boost::shared_ptr<
        sinks::synchronous_sink< sinks::text_ostream_backend >
    > sink = logging::add_file_log("essential.log");
    sink->set_formatter(formatter);
    sink->set_filter(expr::attr< int >("Severity") >= 1);

    // Register common attributes
    logging::add_common_attributes();

    // Here we go, we can write logs
    src::logger lg;
    BOOST_LOG(lg) << "Hello world!";

    return 0;
}
```

Filter and formatter parsers

```
#include <boost/log/utility/setup/filter_parser.hpp>
#include <boost/log/utility/setup/formatter_parser.hpp>
```

Filter and formatter parsers allow constructing filters and formatters from a descriptive string. The function `parse_filter` is responsible for recognizing filters and `parse_formatter` - for recognizing formatters.

In the case of filters the string is formed of a sequence of condition expressions, interconnected with boolean operations. There are two operations supported: conjunction (designated as "&" or "and") and disjunction ("|" or "or"). Each condition itself may be either a single condition or a sub-filter, taken in round brackets. Each condition can be negated with the "!" sign or "not" keyword. The condition, if it's not a sub-filter, usually consists of an attribute name enclosed in percent characters ("%"), a relation keyword and an operand. The relation and operand may be omitted, in which case the condition is assumed to be the requirement of the attribute presence (with any type).

```
filter:
    condition { op condition }

op:
    &
    and
    |
    or

condition:
    !condition
    not condition
    (filter)
    %attribute_name%
    %attribute_name% relation operand

relation:
    >
    <
    =
    !=
    >=
    <=
    begins_with
    ends_with
    contains
    matches
```

Below are some examples of filters:

Table 9. Examples of filters

Filter string	Description
`%Severity%`	The filter returns `true` if an attribute value with name "Severity" is found in a log record.
`%Severity% > 3`	The filter returns `true` if an attribute value with name "Severity" is found and it is greater than 3. The attribute value must be of one of the integral types.
`!(%Ratio% > 0.0 & %Ratio% <= 0.5)`	The filter returns `true` if an attribute value with name "Ratio" of one of the floating point types is not found or it is not between 0 and 0.5.
`%Tag% contains "net" or %Tag% contains "io" and not %StatFlow%`	The filter returns `true` if an attribute value with name "Tag" is found and contains words "net" or "io" and if an attribute value "StatFlow" is not found. The "Tag" attribute value must be of one of the string types, the "StatFlow" attribute value type is not considered.

The formatter string syntax is even simpler and pretty much resembles Boost.Format format string syntax. The string must contain attribute names enclosed in percent signs ("%"), the corresponding attribute value will replace these placeholders. The placeholder "%Message%" will be replaced with the log record text. For instance, `[%TimeStamp%] *%Severity%* %Message%` formatter string will make log records look like this: `[2008-07-05 13:44:23] *0* Hello world`.

Note

Previous releases of the library also supported the "%_%" placeholder for the message text. This placeholder is deprecated now, although it still works for backward compatibility. Its support will be removed in future releases.

It must be noted, though, that by default the library only supports those attribute value types which are known at the library build time. User-defined types will not work properly in parsed filters and formatters until registered in the library. More on this is available in the Extending the library section.

Note

The parsed formatters and filters are generally less optimal than the equivalent ones written in code. This is because of two reasons: (*) the programmer usually knows more about types of the attribute values that may be involved in formatting or filtering and (*) the compiler has a better chance to optimize the formatter or filter if it is known in compile time. Therefore, if the performance matters, it is advised to avoid parsed filters and formatters.

Library initialization from a settings container

```
#include <boost/log/utility/setup/settings.hpp>
#include <boost/log/utility/setup/from_settings.hpp>
```

The headers define components for library initialization from a settings container. The settings container is basically a set of named parameters divided into sections. The container is implemented with the `basic_settings` class template. There are several constraints on how parameters are stored in the container:

- Every parameter must reside in a section. There can be no parameters that do not belong to a section.
- Parameters must have names unique within the section they belong to. Parameters from different sections may have the same name.
- Sections can nest. When read from a file or accessed from the code, section names can express arbitrary hierarchy by separating the parent and child section names with '.' (e.g. "[Parent.Child.ChildChild]").
- Sections must have names unique within the enclosing section (or global scope, if the section is top level).

So basically, settings container is a layered associative container, with string keys and values. In some respect it is similar to Boost.PropertyTree, and in fact it supports construction from `boost::ptree`. The supported parameters are described below.

Tip

In the tables below, the `CharT` type denotes the character type that is used with the settings container.

Table 10. Section "Core". Logging core settings.

Parameter	Format	Description
Filter	Filter string as described here	Global filter to be installed to the core. If not specified, the global filter is not set.
DisableLogging	"true" or "false"	If `true`, results in calling `set_logging_enabled(false)` on the core. By default, value `false` is assumed.

Sink settings are divided into separate subsections within the common top-level section "Sinks" - one subsection for each sink. The subsection names should denote a user-defined sink name. For example, "MyFile".

Note

Previous versions of the library also supported top-level sections starting with the "Sink:" prefix to describe sink parameters. This syntax is deprecated now, although it still works when parsing a settings file for backward compatibility. The parser will automatically put these sections under the "Sinks" top-level section in the resulting settings container. Support for this syntax will be removed in future releases.

Table 11. Sections under the "Sinks" section. Common sink settings.

Parameter	Format	Description
Destination	Sink target, see description	Sink backend type. Mandatory parameter. May have one of these values: Console, TextFile, Syslog. On Windows the following values are additionally supported: SimpleEventLog, Debugger. Also, user-defined sink names may also be supported if registered by calling `register_sink_factory`.
Filter	Filter string as described here	Sink-specific filter. If not specified, the filter is not set.
Asynchronous	"true" or "false"	If `true`, the asynchronous sink frontend will be used. Otherwise the synchronous sink frontend will be used. By default, value `false` is assumed. In single-threaded builds this parameter is not used, as unlocked sink frontend is always used.

Besides the common settings that all sinks support, some sink backends also accept a number of specific parameters. These parameters should be specified in the same section.

Table 12. "Console" sink settings

Parameter	Format	Description
Format	Format string as described here	Log record formatter to be used by the sink. If not specified, the default formatter is used.
AutoFlush	"true" or "false"	Enables or disables the auto-flush feature of the backend. If not specified, the default value `false` is assumed.

Table 13. "TextFile" sink settings

Parameter	Format	Description
FileName	File name pattern	The file name pattern for the sink backend. This parameter is mandatory.
Format	Format string as described here	Log record formatter to be used by the sink. If not specified, the default formatter is used.
AutoFlush	"true" or "false"	Enables or disables the auto-flush feature of the backend. If not specified, the default value `false` is assumed.
RotationSize	Unsigned integer	File size, in bytes, upon which file rotation will be performed. If not specified, no size-based rotation will be made.
RotationInterval	Unsigned integer	Time interval, in seconds, upon which file rotation will be performed. See also the RotationTimePoint parameter and the note below.
RotationTimePoint	Time point format string, see below	Time point or a predicate that detects at what moment of time to perform log file rotation. See also the RotationInterval parameter and the note below.
Target	File system path to a directory	Target directory name, in which the rotated files will be stored. If this parameter is specified, rotated file collection is enabled. Otherwise the feature is not enabled, and all corresponding parameters are ignored.
MaxSize	Unsigned integer	Total size of files in the target directory, in bytes, upon which the oldest file will be deleted. If not specified, no size-based file cleanup will be performed.
MinFreeSpace	Unsigned integer	Minimum free space in the target directory, in bytes, upon which the oldest file will be deleted. If not specified, no space-based file cleanup will be performed.
ScanForFiles	"All" or "Matching"	Mode of scanning for old files in the target directory, see `scan_method`. If not specified, no scanning will be performed.

Warning

The text file sink uses Boost.Filesystem internally, which may cause problems on process termination. See here for more details.

The time-based rotation can be set up with one of the two parameters: RotationInterval or RotationTimePoint. Not more than one of these parameters should be specified for a given sink. If none is specified, no time-based rotation will be performed.

The RotationTimePoint parameter should have one of the following formats, according to the Boost.DateTime format notation:

- "%H:%M:%S". In this case, file rotation will be performed on a daily basis, at the specified time. For example, "12:00:00".
- "%a %H:%M:%S" or "%A %H:%M:%S". File rotation takes place every week, on the weekday specified in the long or short form, at the specified time. For example, "Saturday 09:00:00".
- "%d %H:%M:%S". File rotation takes place every month, on the specified day of month, at the specified time. For example, "01 23:30:00".

Table 14. "Syslog" sink settings

Parameter	Format	Description
Format	Format string as described here	Log record formatter to be used by the sink. If not specified, the default formatter is used.
LocalAddress	An IP address	Local address to initiate connection to the syslog server. If not specified, the default local address will be used.
TargetAddress	An IP address	Remote address of the syslog server. If not specified, the local address will be used.

Table 15. "SimpleEventLog" sink settings

Parameter	Format	Description
Format	Format string as described here	Log record formatter to be used by the sink. If not specified, the default formatter is used.
LogName	A string	Log name to write events into. If not specified, the default log name will be used.
LogSource	A string	Log source to write events from. If not specified, the default source will be used.
Registration	"Never", "OnDemand" or "Forced"	Mode of log source registration in Windows registry, see `registration_mode`. If not specified, on-demand registration will be performed.

The user is free to fill the settings container from whatever settings source he needs. The usage example is below:

```
void init_logging()
{
    logging::settings setts;

    setts["Core"]["Filter"] = "%Severity% >= warning";
    setts["Core"]["DisableLogging"] = false;

    // Subsections can be referred to with a single path
    setts["Sinks.Console"]["Destination"] = "Console";
    setts["Sinks.Console"]["Filter"] = "%Severity% >= fatal";
    setts["Sinks.Console"]["AutoFlush"] = true;

    // ...as well as the individual parameters
    setts["Sinks.File.Destination"] = "TextFile";
    setts["Sinks.File.FileName"] = "MyApp_%3N.log";
    setts["Sinks.File.AutoFlush"] = true;
    setts["Sinks.File.RotationSize"] = 10 * 1024 * 1024; // 10 MiB

    logging::init_from_settings(setts);
}
```

The settings reader also allows to be extended to support custom sink types. See the Extending the library section for more information.

Library initialization from a settings file

```
#include <boost/log/utility/setup/from_stream.hpp>
```

Support for configuration files is a frequently requested feature of the library. And despite the fact there is no ultimately convenient and flexible format of the library settings, the library provides preliminary support for this feature. The functionality is implemented with a simple function `init_from_stream`, which accepts an STL input stream and reads the library settings from it. The function then passes on the read settings to the `init_from_settings` function, described above. Therefore the parameter names and their meaning is the same as for the `init_from_settings` function.

The settings format is quite simple and widely used. Below is the description of syntax and parameters.

```
# Comments are allowed. Comment line begins with the '#' character
# and spans until the end of the line.

# Logging core settings section. May be omitted if no parameters specified within it.
[Core]
DisableLogging=false
Filter="%Severity% > 3"

# Sink settings sections
[Sinks.MySink1]

# Sink destination type
Destination=Console

# Sink-specific filter. Optional, by default no filter is applied.
Filter="%Target% contains \"MySink1\""

# Formatter string. Optional, by default only log record message text is written.
Format="<%TimeStamp%> - %Message%"

# The flag shows whether the sink should be asynchronous
Asynchronous=false

# Enables automatic stream flush after each log record.
AutoFlush=true
```

Here's the usage example:

```
int main(int, char*[])
{
    // Read logging settings from a file
    std::ifstream file("settings.ini");
    logging::init_from_stream(file);

    return 0;
}
```

Extending the library

Writing your own sinks

```
#include <boost/log/sinks/basic_sink_backend.hpp>
```

As was described in the Design overview section, sinks consist of two parts: frontend and backend. Frontends are provided by the library and usually do not need to be reimplemented. Thanks to frontends, implementing backends is much easier than it could be: all filtering, formatting and thread synchronization is done there.

In order to develop a sink backend, you derive your class from either `basic_sink_backend` or `basic_formatted_sink_backend`, depending on whether your backend requires formatted log records or not. Both base classes define a set of types that are required to interface with sink frontends. One of these types is `frontend_requirements`.

Frontend requirements

```
#include <boost/log/sinks/frontend_requirements.hpp>
```

In order to work with sink backends, frontends use the `frontend_requirements` type defined by all backends. The type combines one or several requirement tags:

- `synchronized_feeding`. If the backend has this requirement, it expects log records to be passed from frontend in synchronized manner (i.e. only one thread should be feeding a record at a time). Note that different threads may be feeding different records, the requirement merely states that there will be no concurrent feeds.
- `concurrent_feeding`. This requirement extends `synchronized_feeding` by allowing different threads to feed records concurrently. The backend implements all necessary thread synchronization in this case.
- `formatted_records`. The backend expects formatted log records. The frontend implements formatting to a string with character type defined by the `char_type` typedef within the backend. The formatted string will be passed along with the log record to the backend. The `basic_formatted_sink_backend` base class automatically adds this requirement to the `frontend_requirements` type.
- `flushing`. The backend supports flushing its internal buffers. If the backend indicates this requirement it has to implement the `flush` method taking no arguments; this method will be called by the frontend when flushed.

Tip

By choosing either of the thread synchronization requirements you effectively allow or prohibit certain sink frontends from being used with your backend.

Multiple requirements can be combined into `frontend_requirements` type with the `combine_requirements` metafunction:

```
typedef sinks::combine_requirements<
    sinks::synchronized_feeding,
    sinks::formatted_records,
    sinks::flushing
>::type frontend_requirements;
```

It must be noted that `synchronized_feeding` and `concurrent_feeding` should not be combined together as it would make the synchronization requirement ambiguous. The `synchronized_feeding` is a more strict requirement than `concurrent_feeding`, so whenever the backend requires concurrent feeding it is also capable of synchronized feeding.

The `has_requirement` metafunction can be used to test for a specific requirement in the `frontend_requirements` typedef.

Minimalistic sink backend

As an example of the `basic_sink_backend` class usage, let's implement a simple statistical information collector backend. Assume we have a network server and we want to monitor how many incoming connections are active and how much data was sent or received. The collected information should be written to a CSV-file every minute. The backend definition could look something like this:

```
// The backend collects statistical information about network activity of the application
class stat_collector :
    public sinks::basic_sink_backend<
        sinks::combine_requirements<
            sinks::synchronized_feeding,                                    ❶
            sinks::flushing                                                 ❷
        >::type
    >
{
private:
    // The file to write the collected information to
    std::ofstream m_csv_file;

    // Here goes the data collected so far:
    // Active connections
    unsigned int m_active_connections;
    // Sent bytes
    unsigned int m_sent_bytes;
    // Received bytes
    unsigned int m_received_bytes;

    // The number of collected records since the last write to the file
    unsigned int m_collected_count;
    // The time when the collected data has been written to the file last time
    boost::posix_time::ptime m_last_store_time;

public:
    // The constructor initializes the internal data
    explicit stat_collector(const char* file_name);

    // The function consumes the log records that come from the frontend
    void consume(logging::record_view const& rec);
    // The function flushes the file
    void flush();

private:
    // The function resets statistical accumulators to initial values
    void reset_accumulators();
    // The function writes the collected data to the file
    void write_data();
};
```

❶ we will have to store internal data, so let's require frontend to synchronize feeding calls to the backend
❷ also enable flushing support

As you can see, the public interface of the backend is quite simple. Only the `consume` and `flush` methods are called by frontends. The `consume` function is called every time a logging record passes filtering in the frontend. The record, as was stated before, contains a set of attribute values and the message string. Since we have no need for the record message, we will ignore it for now. But from the other attributes we can extract the statistical data to accumulate and write to the file. We can use attribute keywords and value visitation to accomplish this.

```
BOOST_LOG_ATTRIBUTE_KEYWORD(sent, "Sent", unsigned int)
BOOST_LOG_ATTRIBUTE_KEYWORD(received, "Received", unsigned int)

// The function consumes the log records that come from the frontend
void stat_collector::consume(logging::record_view const& rec)
{
    // Accumulate statistical readings
    if (rec.attribute_values().count("Connected"))
        ++m_active_connections;
    else if (rec.attribute_values().count("Disconnected"))
        --m_active_connections;
    else
    {
        namespace phoenix = boost::phoenix;
        logging::visit(sent, rec, phoenix::ref(m_sent_bytes) += phoenix::placeholders::_1);
       logging::visit(received, rec, phoenix::ref(m_received_bytes) += phoenix::placeholders::_1);
    }
    ++m_collected_count;

    // Check if it's time to write the accumulated data to the file
    boost::posix_time::ptime now = boost::posix_time::microsec_clock::universal_time();
    if (now - m_last_store_time >= boost::posix_time::minutes(1))
    {
        write_data();
        m_last_store_time = now;
    }
}

// The function writes the collected data to the file
void stat_collector::write_data()
{
    m_csv_file << m_active_connections
        << ',' << m_sent_bytes
        << ',' << m_received_bytes
        << std::endl;
    reset_accumulators();
}

// The function resets statistical accumulators to initial values
void stat_collector::reset_accumulators()
{
    m_sent_bytes = m_received_bytes = 0;
    m_collected_count = 0;
}
```

Note that we used Boost.Phoenix to automatically generate visitor function objects for attribute values.

The last bit of implementation is the `flush` method. It is used to flush all buffered data to the external storage, which is a file in our case. The method can be implemented in the following way:

```
// The function flushes the file
void stat_collector::flush()
{
    // Store any data that may have been collected since the list write to the file
    if (m_collected_count > 0)
    {
        write_data();
        m_last_store_time = boost::posix_time::microsec_clock::universal_time();
    }

    m_csv_file.flush();
}
```

You can find the complete code of this example here.

Formatting sink backend

As an example of a formatting sink backend, let's implement a sink that will display text notifications for every log record passed to it.

Tip

Real world applications would probably use some GUI toolkit API to display notifications but GUI programming is out of scope of this documentation. In order to display notifications we shall use an external program which does just that. In this example we shall employ `notify-send` program which is available on Linux (Ubuntu/Debian users can install it with the `libnotify-bin` package; other distros should also have it available in their package repositories). The program takes the notification parameters in the command line, displays the notification in the current desktop environment and then exits. Other platforms may also have similar tools.

The definition of the backend is very similar to what we have seen in the previous section:

```
// The backend starts an external application to display notifications
class app_launcher :
    public sinks::basic_formatted_sink_backend<
        char,                                                    ❶
        sinks::synchronized_feeding                              ❷
    >
{
public:
    // The function consumes the log records that come from the frontend
    void consume(logging::record_view const& rec, string_type const& command_line);
};
```

❶ target character type
❷ in order not to spawn too many application instances we require records to be processed serial

The first thing to notice is that the `app_launcher` backend derives from `basic_formatted_sink_backend` rather than `basic_sink_backend`. This base class accepts the character type in addition to the requirements. The specified character type defines the target string type the formatter will compose in the frontend and it typically corresponds to the underlying API the backend uses to process records. It must be mentioned that the character type the backend requires is not related to the character types of string attribute values, including the message text. The formatter will take care of character code conversion when needed.

The second notable difference from the previous examples is that `consume` method takes an additional string parameter besides the log record. This is the result of formatting. The `string_type` type is defined by the `basic_formatted_sink_backend` base class and it corresponds to the requested character type.

We don't need to flush any buffers in this example, so we didn't specify the `flushing` requirement and omitted the `flush` method in the backend. Although we don't need any synchronization in our backend, we specified `synchronized_feeding` requirement so that we don't spawn multiple instances of `notify-send` program and cause a "fork bomb".

Now, the `consume` implementation is trivial:

```
// The function consumes the log records that come from the frontend
void app_launcher::consume(logging::record_view const& rec, string_type const& command_line)
{
    std::system(command_line.c_str());
}
```

So the formatted string is expected to actually be a command line to start the application. The exact application name and arguments are to be determined by the formatter. This approach adds flexibility because the backend can be used for different purposes and updating the command line is as easy as updating the formatter.

The sink can be configured with the following code:

```
BOOST_LOG_ATTRIBUTE_KEYWORD(process_name, "ProcessName", std::string)
BOOST_LOG_ATTRIBUTE_KEYWORD(caption, "Caption", std::string)

// Custom severity level formatting function
std::string severity_level_as_urgency(
    logging::value_ref< logging::trivial::severity_level, logging::trivial::tag::sever↵
ity > const& level)
{
    if (!level || level.get() == logging::trivial::info)
        return "normal";
    logging::trivial::severity_level lvl = level.get();
    if (lvl < logging::trivial::info)
        return "low";
    else
        return "critical";
}

// The function initializes the logging library
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    typedef sinks::synchronous_sink< app_launcher > sink_t;
    boost::shared_ptr< sink_t > sink(new sink_t());

    const std::pair< const char*, const char* > shell_decorations[] =
    {
        std::pair< const char*, const char* >("\"", "\\\""),
        std::pair< const char*, const char* >("$", "\\$"),
        std::pair< const char*, const char* >("!", "\\!")
    };

    // Make the formatter generate the command line for notify-send
    sink->set_formatter
    (
        expr::stream << "notify-send -t 2000 -u "
            << boost::phoenix::bind(&severity_level_as_urgency, logging::trivial::sever↵
ity.or_none())
            << expr::if_(expr::has_attr(process_name))
               [
                    expr::stream << " -a '" << process_name << "'"
               ]
            << expr::if_(expr::has_attr(caption))
               [
                    expr::stream << " \"" << expr::char_decor(shell_decorations)[ ex↵
pr::stream << caption ] << "\""
               ]
            << " \"" << expr::char_decor(shell_decorations)[ expr::stream << expr::mes↵
sage ] << "\""
    );

    core->add_sink(sink);

    // Add attributes that we will use
    core->add_global_attribute("ProcessName", attrs::current_process_name());
}
```

The most interesting part is the sink setup. The `synchronous_sink` frontend (as well as any other frontend) will detect that the `app_launcher` backend requires formatting and enable the corresponding functionality. The `set_formatter` method becomes available and can be used to set the formatting expression that composes the command line to start the `notify-send` program. We used attribute keywords to identify particular attribute values in the formatter. Notice that string attribute values have to be preprocessed so that special characters interpreted by the shell are escaped in the command line. We achieve that with the `char_decor` decorator with our custom replacement map. After the sink is configured we also add the current process name attribute to the core so that we don't have to add it to every record.

After all this is done, we can finally display some notifications:

```
void test_notifications()
{
    BOOST_LOG_TRIVIAL(debug) << "Hello, it's a simple notification";
    BOOST_LOG_TRIVIAL(info) << logging::add_value(caption, "Caption text") << "And this notific↵
ation has caption as well";
}
```

The complete code of this example is available here.

Writing your own sources

```
#include <boost/log/sources/threading_models.hpp>
#include <boost/log/sources/basic_logger.hpp>
```

You can extend the library by developing your own sources and, for that matter, ways of collecting log data. Basically, you have two choices of how to start: you can either develop a new logger feature or design a whole new type of source. The first approach is good if all you need is to tweak the functionality of the existing loggers. The second approach is reasonable if the whole mechanism of collecting logs by the provided loggers is unsuitable for your needs.

Creating a new logger feature

Every logger provided by the library consists of a number of features that can be combined with each other. Each feature is responsible for a single and independent aspect of the logger functionality. For example, loggers that provide the ability to assign severity levels to logging records include the `severity` feature. You can implement your own feature and use it along with the ones provided by the library.

A logger feature should follow these basic requirements:

- A logging feature should be a class template. It should have at least one template parameter type (let's name it `BaseT`).
- The feature must publicly derive from the `BaseT` template parameter.
- The feature must be default-constructible and copy-constructible.
- The feature must be constructible with a single argument of a templated type. The feature may not use this argument itself, but it should pass this argument to the `BaseT` constructor.

These requirements allow composition of a logger from a number of features derived from each other. The root class of the features hierarchy will be the `basic_logger` class template instance. This class implements most of the basic functionality of loggers, like storing logger-specific attributes and providing the interface for log message formatting. The hierarchy composition is done by the `basic_composite_logger` class template, which is instantiated on a sequence of features (don't worry, this will be shown in an example in a few moments). The constructor with a templated argument allows initializing features with named parameters, using the Boost.Parameter library.

A logging feature may also contain internal data. In that case, to maintain thread safety for the logger, the feature should follow these additional guidelines:

1. Usually there is no need to introduce a mutex or another synchronization mechanism in each feature. Moreover, it is advised not to do so, because the same feature can be used in both thread-safe and not thread-safe loggers. Instead, features should use the threading model of the logger as a synchronization primitive, similar to how they would use a mutex. The threading model is accessible through the `get_threading_model` method, defined in the `basic_logger` class template.

2. If the feature has to override `*_unlocked` methods of the protected interface of the `basic_logger` class template (or the same part of the base feature interface), the following should be considered with regard to such methods:

   - The public methods that eventually call these methods are implemented by the `basic_composite_logger` class template. These implementations do the necessary locking and then pass control to the corresponding `_unlocked` method of the base features.

   - The thread safety requirements for these methods are expressed with lock types. These types are available as typedefs in each feature and the `basic_logger` class template. If the feature exposes a protected function `foo_unlocked`, it will also expose type `foo_lock`, which will express the locking requirements of `foo_unlocked`. The corresponding method `foo` in the `basic_composite_logger` class template will use this typedef in order to lock the threading model before calling `foo_unlocked`.

   - Feature constructors don't need locking, and thus there's no need for lock types for them.

3. The feature may implement a copy constructor. The argument of the constructor is already locked with a shared lock when the constructor is called. Naturally, the feature is expected to forward the copy constructor call to the `BaseT` class.

4. The feature need not implement an assignment operator. The assignment will be automatically provided by the `basic_composite_logger` class instance. However, the feature may provide a `swap_unlocked` method that will swap contents of this feature and the method argument, and call similar method in the `BaseT` class. The automatically generated assignment operator will use this method, along with copy constructor.

In order to illustrate all these lengthy recommendations, let's implement a simple logger feature. Suppose we want our logger to be able to tag individual log records. In other words, the logger has to temporarily add an attribute to its set of attributes, emit the logging record, and then automatically remove the attribute. Somewhat similar functionality can be achieved with scoped attributes, although the syntax may complicate wrapping it into a neat macro:

```
// We want something equivalent to this
{
    BOOST_LOG_SCOPED_LOGGER_TAG(logger, "Tag", "[GUI]");
    BOOST_LOG(logger) << "The user has confirmed his choice";
}
```

Let's declare our logger feature:

```
template< typename BaseT >
class record_tagger_feature :
    public BaseT                          ❶
{
public:
    // Let's import some types that we will need. These imports should be public,
    // in order to allow other features that may derive from record_tagger to do the same.
    typedef typename BaseT::char_type char_type;
    typedef typename BaseT::threading_model threading_model;

public:
    // Default constructor. Initializes m_Tag to an invalid value.
    record_tagger_feature();
    // Copy constructor. Initializes m_Tag to a value, equivalent to that.m_Tag.
    record_tagger_feature(record_tagger_feature const& that);
    // Forwarding constructor with named parameters
    template< typename ArgsT >
    record_tagger_feature(ArgsT const& args);

    // The method will require locking, so we have to define locking requirements for it.
    // We use the strictest_lock trait in order to choose the most restricting lock type.
    typedef typename logging::strictest_lock<
        boost::lock_guard< threading_model >,
        typename BaseT::open_record_lock,
        typename BaseT::add_attribute_lock,
        typename BaseT::remove_attribute_lock
    >::type open_record_lock;

protected:
    // Lock-less implementation of operations
    template< typename ArgsT >
    logging::record open_record_unlocked(ArgsT const& args);
};

// A convenience metafunction to specify the feature
// in the list of features of the final logger later
struct record_tagger :
    public boost::mpl::quote1< record_tagger_feature >
{
};
```

❶ the feature should derive from other features or the basic_logger class

You can see that we use the `strictest_lock` template in order to define lock types that would fulfill the base class thread safety requirements for methods that are to be called from the corresponding methods of `record_tagger_feature`. The `open_record_lock` definition shows that the `open_record_unlocked` implementation for the `record_tagger_feature` feature requires exclusive lock (which `lock_guard` is) for the logger, but it also takes into account locking requirements of the `open_record_unlocked`, `add_attribute_unlocked` and `remove_attribute_unlocked` methods of the base class, because it will have to call them. The generated `open_record` method of the final logger class will make use of this typedef in order to automatically acquire the corresponding lock type before forwarding to the `open_record_unlocked` methods.

Actually, in this particular example, there was no need to use the `strictest_lock` trait, because all our methods require exclusive locking, which is already the strictest one. However, this template may come in handy, should you use shared locking.

The implementation of the public interface becomes quite trivial:

```
template< typename BaseT >
record_tagger_feature< BaseT >::record_tagger_feature()
{
}

template< typename BaseT >
record_tagger_feature< BaseT >::record_tagger_feature(record_tagger_feature const& that) :
    BaseT(static_cast< BaseT const& >(that))
{
}

template< typename BaseT >
template< typename ArgsT >
record_tagger_feature< BaseT >::record_tagger_feature(ArgsT const& args) : BaseT(args)
{
}
```

Now, since all locking is extracted into the public interface, we have the most of our feature logic to be implemented in the protected part of the interface. In order to set up tag value in the logger, we will have to introduce a new Boost.Parameter keyword. Following recommendations from that library documentation, it's better to introduce the keyword in a special namespace:

```
namespace my_keywords {

    BOOST_PARAMETER_KEYWORD(tag_ns, tag)

}
```

Opening a new record can now look something like this:

```
template< typename BaseT >
template< typename ArgsT >
logging::record record_tagger_feature< BaseT >::open_record_unlocked(ArgsT const& args)
{
    // Extract the named argument from the parameters pack
    std::string tag_value = args[my_keywords::tag | std::string()];

    logging::attribute_set& attrs = BaseT::attributes();
    logging::attribute_set::iterator tag = attrs.end();
    if (!tag_value.empty())
    {
        // Add the tag as a new attribute
        std::pair<
            logging::attribute_set::iterator,
            bool
        > res = BaseT::add_attribute_unlocked("Tag",
            attrs::constant< std::string >(tag_value));
        if (res.second)
            tag = res.first;
    }

    // In any case, after opening a record remove the tag from the attributes
    BOOST_SCOPE_EXIT_TPL((&tag)(&attrs))
    {
        if (tag != attrs.end())
            attrs.erase(tag);
    }
    BOOST_SCOPE_EXIT_END

    // Forward the call to the base feature
    return BaseT::open_record_unlocked(args);
}
```

Here we add a new attribute with the tag value, if one is specified in call to `open_record`. When a log record is opened, all attribute values are acquired and locked after the record, so we remove the tag from the attribute set with the Boost.ScopeExit block.

Ok, we got our feature, and it's time to inject it into a logger. Assume we want to combine it with the standard severity level logging. No problems:

```
template< typename LevelT = int >
class my_logger :
    public src::basic_composite_logger<
        char,                           ❶
        my_logger< LevelT >,            ❷
        src::single_thread_model,       ❸
        src::features<                  ❹
            src::severity< LevelT >,
            record_tagger
        >
    >
{
    // The following line will automatically generate forwarding constructors that
    // will call to the corresponding constructors of the base class
    BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE(my_logger)
};
```

❶ character type for the logger
❷ final logger type
❸ the logger does not perform thread synchronization; use `multi_thread_model` to declare a thread-safe logger
❹ the list of features we want to combine

As you can see, creating a logger is a quite simple procedure. The `BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE` macro you see here is for mere convenience purpose: it unfolds into a default constructor, copy constructor, assignment operator and a number of constructors to support named arguments. For non-template loggers there is a similar `BOOST_LOG_FORWARD_LOGGER_MEMBERS` macro.

Assuming we have defined severity levels like this:

```
enum severity_level
{
    normal,
    warning,
    error
};
```

we can now use our logger as follows:

```
void manual_logging()
{
    my_logger< severity_level > logger;

    logging::record rec = logger.open_record((keywords::severity = nor↵
mal, my_keywords::tag = "GUI"));
    if (rec)
    {
        logging::record_ostream strm(rec);
        strm << "The user has confirmed his choice";
        strm.flush();
        logger.push_record(boost::move(rec));
    }
}
```

All this verbosity is usually not required. One can define a special macro to make the code more concise:

```
#define LOG_WITH_TAG(lg, sev, tg) \
    BOOST_LOG_WITH_PARAMS((lg), (keywords::severity = (sev))(my_keywords::tag = (tg)))

void logging_function()
{
    my_logger< severity_level > logger;

    LOG_WITH_TAG(logger, normal, "GUI") << "The user has confirmed his choice";
}
```

See the complete code.

Guidelines for designers of standalone logging sources

In general, you can implement new logging sources the way you like, the library does not mandate any design requirements on log sources. However, there are some notes regarding the way log sources should interact with logging core.

1. Whenever a logging source is ready to emit a log record, it should call the `open_record` in the core. The source-specific attributes should be passed into that call. During that call the core allocates resources for the record being made and performs filtering.

2. If the call to `open_record` returned a valid log record, then the record has passed the filtering and is considered to be opened. The record may later be either confirmed by the source by subsequently calling `push_record` or withdrawn by destroying it.

3. If the call to `open_record` returned an invalid (empty) log record, it means that the record has not been opened (most likely due to filtering rejection). In that case the logging core does not hold any resources associated with the record, and thus the source must not call `push_record` for that particular logging attempt.

4. The source may subsequently open more than one record. Opened log records exist independently from each other.

Writing your own attributes

```
#include <boost/log/attributes/attribute.hpp>
#include <boost/log/attributes/attribute_value.hpp>
#include <boost/log/attributes/attribute_value_impl.hpp>
```

Developing your own attributes is quite simple. Generally, you need to do the following:

1. Define what will be the attribute value. Most likely, it will be a piece of constant data that you want to participate in filtering and formatting. Envelop this data into a class that derives from the `impl` interface; this is the attribute value implementation class. This object will have to implement the `dispatch` method that will extract the stored data (or, in other words, the stored value) to a type dispatcher. In most cases the class `attribute_value_impl` provided by the library can be used for this.

2. Use the `attribute_value` class as the interface class that holds a reference to the attribute value implementation.

3. Define how attribute values are going to be produced. In a corner case the values do not need to be produced (like in the case of the `constant` attribute provided by the library), but often there is some logic that needs to be invoked to acquire the attribute value. This logic has to be concentrated in a class derived from the `impl` interface, more precisely - in the `get_value` method. This class is the attribute implementation class. You can think of it as an attribute value factory.

4. Define the attribute interface class that derives from `attribute`. By convention, the interface class should create the corresponding implementation on construction and pass the pointer to it to the `attribute` class constructor. The interface class may have interface methods but it typically should not contain any data members as the data will be lost when the attribute is added to the library. All relevant data should be placed in the implementation class instead.

While designing an attribute, one has to strive to make it as independent from the values it produces, as possible. The attribute can be called from different threads concurrently to produce a value. Once produced, the attribute value can be used several times by the library (maybe even concurrently), it can outlive the attribute object that created it, and several attribute values produced by the same attribute can exist simultaneously.

From the library perspective, each attribute value is considered independent from other attribute values or the attribute itself. That said, it is still possible to implement attributes that are also attribute values, which allows to optimize performance in some cases. This is possible if the following requirements are fulfilled:

- The attribute value never changes, so it's possible to store it in the attribute itself. The `constant` attribute is an example.

- The attribute stores its value in a global (external with regard to the attribute) storage, that can be accessed from any attribute value. The attribute values must guarantee, though, that their stored values do not change over time and are safely accessible concurrently from different threads.

As a special case for the second point, it is possible to store attribute values (or their parts) in a thread-specific storage. However, in that case the user has to implement the `detach_from_thread` method of the attribute value implementation properly. The result of this method - another attribute value - must be independent from the thread it is being called in, but its stored value should be equivalent to the original attribute value. This method will be called by the library when the attribute value passes to a thread that is different from the thread where it was created. As of this moment, this will only happen in the case of asynchronous logging sinks.

But in the vast majority of cases attribute values must be self-contained objects with no dependencies on other entities. In fact, this case is so common that the library provides a ready to use attribute value implementation class template `attribute_value_impl` and `make_attribute_value` generator function. The class template has to be instantiated on the stored value type, and the stored value has to be provided to the class constructor. For example, let's implement an attribute that returns system uptime in seconds. This is the attribute implementation class.

```
// The function returns the system uptime, in seconds
unsigned int get_uptime();

// Attribute implementation class
class system_uptime_impl :
    public logging::attribute::impl
{
public:
    // The method generates a new attribute value
    logging::attribute_value get_value()
    {
        return attrs::make_attribute_value(get_uptime());
    }
};
```

Since there is no need for special attribute value classes we can use the `make_attribute_value` function to create the value envelop.

Tip

For cases like this, when the attribute value can be obtained in a single function call, it is typically more convenient to use the `function` attribute.

The interface class of the attribute can be defined as follows:

```
// Attribute interface class
class system_uptime :
    public logging::attribute
{
public:
    system_uptime() : logging::attribute(new system_uptime_impl())
    {
    }
    // Attribute casting support
    explicit system_uptime(attrs::cast_source const& source) : logging::attribute(source.as< sys↵
tem_uptime_impl >())
    {
    }
};
```

As it was mentioned before, the default constructor creates the implementation instance so that the default constructed attribute can be used by the library.

The second constructor adds support for attribute casting. The constructor argument contains a reference to an attribute implementation object, and by calling `as` on it the constructor attempts to upcast it to the implementation object of our custom attribute. The `as` method will return `NULL` if the upcast fails, which will result in an empty attribute constructed in case of failure.

Having defined these two classes, the attribute can be used with the library as usual:

```
void init_logging()
{
    boost::shared_ptr< logging::core > core = logging::core::get();

    // ...

    // Add the uptime attribute to the core
    core->add_global_attribute("SystemUptime", system_uptime());
}
```

See the complete code.

Extending library settings support

If you write your own logging sinks or use your own types in attributes, you may want to add support for these components to the settings parser provided by the library. Without doing this, the library will not be aware of your types and thus will not be able to use them when parsing settings.

Adding support for user-defined types to the formatter parser

```
#include <boost/log/utility/setup/formatter_parser.hpp>
```

In order to add support for user-defined types to the formatter parser, one has to register a formatter factory. The factory is basically an object that derives from `formatter_factory` interface. The factory mainly implements the single `create_formatter` method which, when called, will construct a formatter for the particular attribute value.

When the user-defined type supports putting to a stream with `operator<<` and this operator behavior is suitable for logging, one can use a simple generic formatter factory provided by the library out of the box. For example, let's assume we have the following user-defined type that we want to use as an attribute value:

```
struct point
{
    float m_x, m_y;

    point() : m_x(0.0f), m_y(0.0f) {}
    point(float x, float y) : m_x(x), m_y(y) {}
};

template< typename CharT, typename TraitsT >
std::basic_ostream< CharT, TraitsT >& operator<< (std::basic_os↵
tream< CharT, TraitsT >& strm, point const& p)
{
    strm << "(" << p.m_x << ", " << p.m_y << ")";
    return strm;
}
```

Then, in order to register this type with the simple formatter factory, a single call to `register_simple_formatter_factory` will suffice:

```
void init_factories()
{
    logging::register_simple_formatter_factory< point, char >("Coordinates");
}
```

Note

The `operator<<` for the attribute value stored type must be visible from the point of this call.

The function takes the stored attribute value type (`point`, in our case) and the target character type used by formatters as template parameters. From the point of this call, whenever the formatter parser encounters a reference to the "Coordinates" attribute in the format string, it will invoke the formatter factory, which will construct the formatter that calls our `operator<<` for class `point`.

Tip

It is typically a good idea to register all formatter factories at an early stage of the application initialization, before any other library initialization, such as reading config files.

From the formatter parser description it is known that the parser supports passing additional parameters from the format string to the formatter factory. We can use these parameters to customize the output generated by the formatter.

For example, let's implement customizable formatting of our `point` objects, so that the following format string works as expected:

```
%TimeStamp% %Coordinates(format="{%0.3f; %0.3f}")% %Message%
```

The simple formatter factory ignores all additional parameters from the format string, so we have to implement our own factory instead. Custom factories are registered with the `register_formatter_factory` function, which is similar to `register_simple_formatter_factory` but accepts a pointer to the factory instead of the explicit template parameters.

```
// Custom point formatter
class point_formatter
{
public:
    typedef void result_type;

public:
    explicit point_formatter(std::string const& fmt) : m_format(fmt)
    {
    }

    void operator() (logging::formatting_ostream& strm, log↵
ging::value_ref< point > const& value) const
    {
        if (value)
        {
            point const& p = value.get();
            m_format % p.m_x % p.m_y;
            strm << m_format;
            m_format.clear();
        }
    }

private:
    mutable boost::format m_format;
};

// Custom point formatter factory
class point_formatter_factory :
    public logging::basic_formatter_factory< char, point >
{
public:
    formatter_type create_formatter(logging::attribute_name const& name, args_map const& args)
    {
        args_map::const_iterator it = args.find("format");
        if (it != args.end())
            return boost::phoenix::bind(point_formatter(it->second), expr::stream, expr::at↵
tr< point >(name));
        else
            return expr::stream << expr::attr< point >(name);
    }
};
```

```
void init_factories()
{
    logging::register_formatter_factory("Coordinates", boost::make_shared< point_formatter_fact↵
ory >());
}
```

Let's walk through this code sample. Our `point_formatter_factory` class derives from the `basic_formatter_factory` base class provided by the library. This class derives from the base `formatter_factory` interface and defines a few useful types, such as `formatter_type` and `args_map` that we use. The only thing left to do in our factory is to define the `create_formatter` method. The method analyzes the parameters from the format string which are passed as the `args` argument, which is basically `std::map` of string keys (parameter names) to string values (the parameter values). We seek for the `format` parameter and expect it to contain a Boost.Format-compatible format string for our `point` objects. If the parameter is found we create a formatter that invokes `point_formatter` for the attribute values. Otherwise we create a default formatter that simply uses the `operator<<`, like the simple formatter factory does. Note that we use the `name` argument of `create_formatter` to identify the attribute so that the same factory can be used for different attributes.

The `point_formatter` is our custom formatter based on Boost.Format. With help of Boost.Phoenix and expression placeholders we can construct a formatter that will extract the attribute value and pass it along with the target stream to the `point_formatter` function object. Note that the formatter accepts the attribute value wrapped into the `value_ref` wrapper which can be empty if the value is not present.

Lastly, the call to `register_formatter_factory` creates the factory and adds it to the library.

You can find the complete code of this example here.

Adding support for user-defined types to the filter parser

```
#include <boost/log/utility/setup/filter_parser.hpp>
```

You can extend filter parser in the similar way you can extend the formatter parser - by registering filter factories for your attribute values into the library. However, since it takes a considerably more complex syntax to describe filters, a filter factory typically implements several generator functions.

Like with formatter parser extension, you can avoid spelling out the filter factory and register a simple factory provided by the library:

```
void init_factories()
{
    logging::register_simple_filter_factory< point, char >("Coordinates");
}
```

In order this to work the user's type should fulfill these requirements:

1. Support reading from an input stream with `operator>>`.

2. Support the complete set of comparison and ordering operators.

Naturally, all these operators must be visible from the point of the `register_simple_filter_factory` call. Note that unlike the simple formatter factory, the filter factory requires the user's type to support reading from a stream. This is so because the filter factory would have to parse the argument of the filter relation from a string.

But we won't get away with a simple filter factory, because our `point` class doesn't have a sensible ordering semantics and thus we cannot define the complete set of operators. We'll have to implement our own filter factory instead. Filter factories derive from the `filter_factory` interface. This base class declares a number of virtual functions that will be called in order to create filters, according to the filter expression. If some functions are not overridden by the factory, the corresponding operations are considered to be not supported by the attribute value. But before we define the filter factory we have to improve our `point` class slightly:

```
struct point
{
    float m_x, m_y;

    point() : m_x(0.0f), m_y(0.0f) {}
    point(float x, float y) : m_x(x), m_y(y) {}
};

bool operator== (point const& left, point const& right);
bool operator!= (point const& left, point const& right);

template< typename CharT, typename TraitsT >
std::basic_ostream< CharT, TraitsT >& operator<< (std::basic_os↵
tream< CharT, TraitsT >& strm, point const& p);
template< typename CharT, typename TraitsT >
std::basic_istream< CharT, TraitsT >& operator>> (std::basic_is↵
tream< CharT, TraitsT >& strm, point& p);
```

We have added comparison and input operators for the `point` class. The output operator is still used by formatters and not required by the filter factory. Now we can define and register the filter factory:

```
// Custom point filter factory
class point_filter_factory :
    public logging::filter_factory< char >
{
public:
    logging::filter on_exists_test(logging::attribute_name const& name)
    {
        return expr::has_attr< point >(name);
    }

    logging::filter on_equality_relation(logging::attrib↵
ute_name const& name, string_type const& arg)
    {
        return expr::attr< point >(name) == boost::lexical_cast< point >(arg);
    }

    logging::filter on_inequality_relation(logging::attrib↵
ute_name const& name, string_type const& arg)
    {
        return expr::attr< point >(name) != boost::lexical_cast< point >(arg);
    }
};

void init_factories()
{
    logging::register_filter_factory("Coordinates", boost::make_shared< point_filter_factory >());
}
```

Having called the `register_filter_factory` function, whenever the filter parser encounters the "Coordinates" attribute mentioned in the filter, it will use the `point_filter_factory` object to construct the appropriate filter. For example, in the case of the following filter

```
%Coordinates% = "(10, 10)"
```

the `on_equality_relation` method will be called with `name` argument being "Coordinates" and `arg` being "(10, 10)".

Note

The quotes around the parenthesis are necessary because the filter parser should interpret the point coordinates as a single string. Also, round brackets are already used to group subexpressions of the filter expression. Whenever there is need to pass several parameters to the relation (like in this case - a number of components of the `point` class) the parameters should be encoded into a quoted string. The string may include C-style escape sequences that will be unfolded upon parsing.

The constructed filter will use the corresponding comparison operators for the `point` class. Ordering operations, like ">" or "<=", will not be supported for attributes named "Coordinates", and this is exactly the way we want it, because the `point` class does not support them either. The complete example is available here.

The library allows not only adding support for new types, but also associating new relations with them. For instance, we can create a new relation "is_in_rect" that will yield positive if the coordinates fit into a rectangle denoted with two points. The filter might look like this:

```
%Coordinates% is_in_rect "{(10, 10) - (20, 20)}"
```

First, let's define our rectangle class:

```
struct rectangle
{
    point m_top_left, m_bottom_right;
};

template< typename CharT, typename TraitsT >
std::basic_ostream< CharT, TraitsT >& operator<< (std::basic_ostream< CharT, TraitsT >& strm, rect↵
angle const& r);
template< typename CharT, typename TraitsT >
std::basic_istream< CharT, TraitsT >& operator>> (std::basic_istream< CharT, TraitsT >& strm, rect↵
angle& r);
```

As it was said, the rectangle is described by two points - the top left and the bottom right corners of the rectangle area. Now let's extend our filter factory with the `on_custom_relation` method:

```
// The function checks if the point is inside the rectangle
bool is_in_rect(logging::value_ref< point > const& p, rectangle const& r)
{
    if (p)
    {
        return p->m_x >= r.m_top_left.m_x && p->m_x <= r.m_bottom_right.m_x &&
               p->m_y >= r.m_top_left.m_y && p->m_y <= r.m_bottom_right.m_y;
    }
    return false;
}

// Custom point filter factory
class point_filter_factory :
    public logging::filter_factory< char >
{
public:
    logging::filter on_exists_test(logging::attribute_name const& name)
    {
        return expr::has_attr< point >(name);
    }

    logging::filter on_equality_relation(logging::attrib↵
ute_name const& name, string_type const& arg)
    {
        return expr::attr< point >(name) == boost::lexical_cast< point >(arg);
    }

    logging::filter on_inequality_relation(logging::attrib↵
ute_name const& name, string_type const& arg)
    {
        return expr::attr< point >(name) != boost::lexical_cast< point >(arg);
    }

    logging::filter on_custom_relation(logging::attrib↵
ute_name const& name, string_type const& rel, string_type const& arg)
    {
        if (rel == "is_in_rect")
        {
            return boost::phoenix::bind(&is_in_rect, expr::attr< point >(name), boost::lexic↵
al_cast< rectangle >(arg));
        }
        throw std::runtime_error("Unsupported filter relation: " + rel);
    }
};

void init_factories()
{
    logging::register_filter_factory("Coordinates", boost::make_shared< point_filter_factory >());
}
```

The `on_custom_relation` method is called with the relation name (the "is_in_rect" string in our case) and the right-hand argument for the relation (the rectangle description). All we have to do is to construct the filter, which is implemented by our `is_in_rect` function. We use `bind` from Boost.Phoenix to compose the filter from the function and the attribute placeholder. You can find the complete code of this example here.

Adding support for user-defined sinks

```
#include <boost/log/utility/setup/from_settings.hpp>
#include <boost/log/utility/setup/from_stream.hpp>
```

The library provides mechanism of extending support for sinks similar to the formatter and filter parsers. In order to be able to mention user-defined sinks in a settings file, the user has to register a sink factory, which essentially contains the `create_sink` method that receives a settings subsection and returns a pointer to the initialized sink. The factory is registered for a specific destination (see the settings file description), so whenever a sink with the specified destination is mentioned in the settings file, the factory gets called.

For example, let's register the `stat_collector` sink we described before in the library. First, let's remember the sink definition:

```
// The backend collects statistical information about network activity of the application
class stat_collector :
    public sinks::basic_sink_backend<
        sinks::combine_requirements<
            sinks::synchronized_feeding,
            sinks::flushing
        >::type
    >
{
private:
    // The file to write the collected information to
    std::ofstream m_csv_file;

    // Here goes the data collected so far:
    // Active connections
    unsigned int m_active_connections;
    // Sent bytes
    unsigned int m_sent_bytes;
    // Received bytes
    unsigned int m_received_bytes;

    // The number of collected records since the last write to the file
    unsigned int m_collected_count;
    // The time when the collected data has been written to the file last time
    boost::posix_time::ptime m_last_store_time;
    // The collected data writing interval
    boost::posix_time::time_duration m_write_interval;

public:
    // The constructor initializes the internal data
    stat_collector(const char* file_name, boost::posix_time::time_duration write_interval);

    // The function consumes the log records that come from the frontend
    void consume(logging::record_view const& rec);
    // The function flushes the file
    void flush();

private:
    // The function resets statistical accumulators to initial values
    void reset_accumulators();
    // The function writes the collected data to the file
    void write_data();
};
```

Compared to the earlier definition we added the `write_interval` constructor parameter so we can set the statistical information flush interval in the settings file. The implementation of the sink stays pretty much the same as before. Now we have to define the factory:

```
// Factory for the stat_collector sink
class stat_collector_factory :
    public logging::sink_factory< char >
{
public:
    // Creates the sink with the provided parameters
    boost::shared_ptr< sinks::sink > create_sink(settings_section const& settings)
    {
        // Read sink parameters
        std::string file_name;
        if (boost::optional< std::string > param = settings["FileName"])
            file_name = param.get();
        else
            throw std::runtime_error("No target file name specified in settings");

        boost::posix_time::time_duration write_interval = boost::posix_time::minutes(1);
        if (boost::optional< std::string > param = settings["WriteInterval"])
        {
            unsigned int sec = boost::lexical_cast< unsigned int >(param.get());
            write_interval = boost::posix_time::seconds(sec);
        }

        // Create the sink
        boost::shared_ptr< stat_collector > backend = boost::make_shared< stat_collect↵
or >(file_name.c_str(), write_interval);
        boost::shared_ptr< sinks::synchronous_sink< stat_collect↵
or > > sink = boost::make_shared< sinks::synchronous_sink< stat_collector > >(backend);

        if (boost::optional< std::string > param = settings["Filter"])
        {
            sink->set_filter(logging::parse_filter(param.get()));
        }

        return sink;
    }
};

void init_factories()
{
    logging::register_sink_factory("StatCollector", boost::make_shared< stat_collector_fact↵
ory >());
}
```

As you can see, we read parameters from settings and simply create our sink with them as a result of `create_sink` method. Generally, users are free to name parameters of their sinks the way they like, as long as settings file format is adhered. However, it is a good idea to follow the pattern established by the library and reuse parameter names with the same meaning. That is, it should be obvious that the parameter "Filter" means the same for both the library-provided "TextFile" sink and out custom "StatCollector" sink.

After defining the factory we only have to register it with the `register_sink_factory` call. The first argument is the new value of the "Destination" parameter in the settings. Whenever the library finds sink description with destination "StatCollector", our factory will be invoked to create the sink. It is also possible to override library-provided destination types with user-defined factories, however it is not possible to restore the default factories afterwards.

Note

As the "Destination" parameter is used to determine the sink factory, this parameter is reserved and cannot be used by sink factories for their own purposes.

Now that the factory is registered, we can use it when initializing from files or settings. For example, this is what the settings file could look like:

```
[Sinks.MyStat]

Destination=StatCollector
FileName=stat.csv
WriteInterval=30
```

The complete code of the example in this section can be found here.

Rationale and FAQ

Why string literals as scope names?

One may wonder why not allow arbitrary strings to be used as named scope names. The answer is simple: for performance and safety reasons. Named scope support functionality has one significant difference from other attribute-related features of the library. The scope stack is maintained even when no logging is done, so if a function `foo` has a `BOOST_LOG_FUNCTION()` statement in its body, it is always a slowdown. Allowing the scope name to be an arbitrary string would make the slowdown significantly greater because of the need to allocate memory and copy the string (not to mention that there would be a need to previously format it, which also takes its toll).

Dynamic memory allocation also introduces exception safety issues: the `BOOST_LOG_FUNCTION()` statement (and alikes) would become a potential source of exceptions. These issues would complicate user's code if he wants to solve memory allocation problems gracefully.

One possible alternative solution would be pooling pre-formatted and pre-allocated scope names somewhere but this would surely degrade performance even more and introduce the problem of detecting when to update or free pooled strings.

Therefore restricting to string literals seemed to be the optimal decision, which reduced dynamic memory usage and provided enough flexibility for common needs.

Why scoped attributes don't override existing attributes?

Initially scoped attributes were able to override other attributes with the same name if they were already registered by the time when a scoped attribute encountered. This allowed some interesting use cases like this:

```
BOOST_LOG_DECLARE_GLOBAL_LOGGER(my_logger, src::logger_mt)

void foo()
{
    // This scoped attribute would temporarily replace the existing tag
    BOOST_LOG_SCOPED_THREAD_TAG("Section", std::string, "In foo");

    // This log record will have a "Section" attribute with value "In foo"
    BOOST_LOG(get_my_logger()) << "We're in foo section";
}

int main(int, char*[])
{
    BOOST_LOG_SCOPED_THREAD_TAG("Section", std::string, "In main");

    // This log record will have a "Section" attribute with value "In main"
    BOOST_LOG(get_my_logger()) << "We're in main section";

    foo();

    // This log record will have a "Section" attribute with value "In main" again
    BOOST_LOG(get_my_logger()) << "We're in main section again";

    return 0;
}
```

However, this feature introduced a number of safety problems, including thread safety issues, that could be difficult to track down. For example, it was no longer safe to use logger-wide scoped attributes on the same logger from different threads, because the resulting attribute would be undefined:

```
BOOST_LOG_DECLARE_GLOBAL_LOGGER(my_logger, src::logger_mt)

void thread1()
{
    BOOST_LOG_SCOPED_LOGGER_TAG(get_my_logger(), "Tag", std::string, "thread1");
    BOOST_LOG(get_my_logger()) << "We're in thread1";
}

void thread2()
{
    BOOST_LOG_SCOPED_LOGGER_TAG(get_my_logger(), "Tag", int, 10);
    BOOST_LOG(get_my_logger()) << "We're in thread2";
}

int main(int, char*[])
{
    BOOST_LOG_SCOPED_LOGGER_TAG(get_my_logger(), "Tag", double, -2.2);

    BOOST_LOG(get_my_logger()) << "We're in main";

    boost::thread t1(&thread1);
    boost::thread t2(&thread2);

    t1.join();
    t2.join();

    // Which "Tag" is registered here?
    BOOST_LOG(get_my_logger()) << "We're in main again";

    return 0;
}
```

There were other issues, like having an attribute set iterator that points to one attribute object, then suddenly without seemingly modifying it it becomes pointing to a different attribute object (of, possibly, a different type). Such behavior could lead to tricky failures that would be difficult to investigate. Therefore this feature was eventually dropped, which simplified the scoped attributes implementation significantly.

Why log records are weakly ordered in a multithreaded application?

Although the library guarantees that log records made in a given thread are always delivered to sinks in the same order as they were made in, the library cannot provide such guarantee for different threads. For instance, it is possible that thread A emits a log record and gets preempted, then thread B emits its log record and manages to deliver it to a sink before being preempted. The resulting log will contain log record from thread B before the record made in thread A. However, attribute values attached to the records will always be actual with regard to the moment of emitting the record and not the moment of passing the record to the sink. This is the reason for a strange, at first glance, situation when a log record with an earlier time stamp follows a record with a later time stamp. The problem appears quite rarely, usually when thread contention on logging is very high.

There are few possible ways to cope with the problem:

- Enforce strict serialization of log record being made throughout the application. This solution implies a severe performance impact in multithreaded applications because log records that otherwise could be processed concurrently would have to go serial. Since this controverses one of the main library goals, it was rejected.
- Attempt to maintain log record ordering on the sink level. This solution is more or less viable. On the downside, it would introduce log record buffering, which in turn would compromise logs reliability. In the case of application crash all buffered records would be lost.

- Bear with the problem and let mis-ordered records appear in log files occasionally. Order log records upon reading the files, if needed.

The second solution was implemented as a special policy for the asynchronous sink frontend.

Why attributes set with stream manipulators do not participate in filtering?

One can add attributes to log records in the following way:

```
BOOST_LOG(logger) << logging::add_value("MyInt", 10) << logging::add_value("MyString", "string ↵
attribute value")
    << "Some log message";
```

However, filters will not be able to use MyInt and MyString attributes. The reason for this behavior is quite simple. The streaming expression is executed *after* the filtering takes place and only *if* the filter passed the log record. At this point these attributes have not been added to the record yet. The easiest way to pass attributes to the filter is to use scoped attributes or tags (see here).

Why not using lazy streaming?

One of the possible library implementations would be using lazy expressions to delay log record formatting. In essence, the expression:

```
logger << "Hello, world!";
```

would become a lambda-expression that is only invoked if the filtering is successful. Although this approach has advantages, it must be noted that lazy expression construction is not zero-cost in terms of performance, code size and compile times. The following expression:

```
logger << "Received packet from " << ip << " of " << packet.size() << " bytes";
```

would generate a considerable amount of code (proportional to the number of streaming operators) to be executed before filtering takes place. Another drawback is that the `packet.size()` is always called, whether or not the record is actually written to the log. In order to delay this call, yet more scaffolding is needed, possibly involving Boost.Bind, Boost.Lambda or Boost.Phoenix. This complication is not acceptable for such a basic use case, like this.

Although lazy streaming is not provided by the library out of the box, nothing prevents developing it in a separate hierarchy of loggers. See the Extending the library section for more information.

Why not using hierarchy of loggers, like in log4j? Why not Boost.Log4j? Etc.

There are enough log4j-like libraries available for C++ already (see here, here and here), so there is no point in implementing yet another one. Instead, this library was aimed to solve more complex tasks, including ones that do not directly fall under the common definition of "logging" term as a debugging tool. Additionally, as Boost.Log was to be a generic library, it had to provide more ways of extending itself, while keeping performance as high as possible. Log4j concept seemed too limiting and inappropriate for these tasks and therefore was rejected.

As for hierarchical loggers, there is no need for this feature in the current library design. One of the main benefits it provides in log4j is determining the appenders (sinks, in terms of this library) in which a log record will end up. This library achieves the same result by filtering. The other application of this feature in Boost.Log could be that the loggers in the hierarchy could combine their sets of attributes for each log record, but there was no demand in real world applications for such a feature. It can be added though, if it proves useful.

Does Boost.Log support process forking?

No, currently Boost.Log does not support process forking (i.e. `fork` call in UNIX systems). There are several issues with process forking, for instance:

- File sinks do not attempt to reopen log files or synchronize access to files between parent and child processes. The resulting output may be garbled.
- File collectors do not expect several processes attempting to collect log files to the same target directory. This may result in spurious failures at log file rotation.
- The current_process_id attribute value will not update in the child process.
- In multithreaded applications, one can generally not guarantee that a thread is not executing some Boost.Log code while an other thread forks. Some Boost.Log resources may be left irreversibly locked or broken in the forked process. This reservation is not specific to Boost.Log, other libraries and even the application itself are susceptible to this problem.

There may be other issues as well. It seems unlikely that support for forking will be added to Boost.Log any time soon.

Note

This does not preclude the `fork+exec` sequence from working. As long as the forked process doesn't try to use any of Boost.Log code, the process should be able to call `exec` or a similar function to load and start another executable.

Does Boost.Log support logging at process initialization and termination?

It should be fine to use logging during the application initialization (i.e. before `main()` starts). But there are a number of known problems with Boost.Log that prevent it from being used at process termination (i.e. after the `main()` function returns), so the official answer to the second part is no. It may work though, in some very restricted setups, if several rules are followed:

- Do not create any objects at process termination, including loggers, attributes or sinks. Try to create and cache the required objects as soon as the application starts (maybe even before `main()` starts).
- Do not use global loggers at process termination.
- Do not call `logging::core::get()` at process termination. Get that pointer as early as possible and keep it until the process terminates.
- Do not use named scopes in termination code.

These rules don't guarantee that the library will work in termination context but they may help to avoid problems. The library will get improved to support this use case better.

Why my application crashes on process termination when file sinks are used?

There are known problems with Boost.Filesystem (for example, #8642 and #9219), which affect Boost.Log file sink backends. When the file sink is destroyed, it attempts to perform a final log file rotation, which involves Boost.Filesystem for moving files. This typically happens when Boost.Log core is deinitialized, at the global deinitialization stage, after leaving `main()`. The crux of the problem is that Boost.Filesystem uses a global locale object internally to perform character code conversion for `path`s, and this locale may get destroyed before Boost.Log is deinitialized, which results in a crash.

There is no way for Boost.Log to influence the order of global deinitialization, but the problem can be worked around on the user's side. One solution is to make sure the locale is initialized *before* Boost.Log. This can be achieved by calling `boost::filesys-`

`tem::path::codecvt()` or `boost::filesystem::path::imbue()` early during the application startup, before performing any calls to Boost.Log. For example:

```
int main(int argc, char* argv[])
{
    boost::filesystem::path::imbue(std::locale("C"));
    initialize_log();

    // ...
}
```

Note that in this case you can't use Boost.Log in global constructors or you have to make sure that `boost::filesystem::path::imbue()` is still called first.

Another solution is to remove and destroy file sinks from the logging core before returning from `main()`. This way file rotation will happen before leaving `main()`, while the locale is still valid. The file sinks can be removed either individually or as a part of the `remove_all_sinks()` call:

```
int main(int argc, char* argv[])
{
    // ...

    logging::core::get()->remove_all_sinks();

    return 0;
}
```

Why my application fails to link with Boost.Log? What's the fuss about library namespaces?

The library declares the `boost::log` namespace which should be used in client code to access library components. However, internally the library uses another nested namespace for actual implementation. The namespace name is configuration and platform dependent, it can change between different releases of the library, so it should never be used in the user side code. This is done in order to make the library configuration synchronized with the application as much as possible and eliminate problems caused by configuration mismatch.

Most of the time users won't even notice the existence of this internal namespace, but it often appears in compiler and linker errors and in some cases it is useful to know how to decode its name. Currently, the namespace name is composed from the following elements:

```
<version><linkage>_<threading>_<system>
```

- The `<version>` component describes the library major version. It is currently `v2`.
- The `<linkage>` component tells whether the library is linked statically or dynamically. It is `s` if the library is linked statically and empty otherwise.
- The `<threading>` component is `st` for single-threaded builds and `mt` for multi-threaded ones.
- The `<system>` component describes the underlying OS API used by the library. Currently, it is only specified for multi-threaded builds. Depending on the target platform and configuration, it can be `posix`, `nt5` or `nt6`.

As a couple quick examples, `v2s_st` corresponds to v2 static single-threaded build of the library and `v2_mt_posix` - to v2 dynamic multi-threaded build for POSIX system API.

Namespace mangling may lead to linking errors if the application is misconfigured. One common mistake is to build dynamic version of the library and not define `BOOST_LOG_DYN_LINK` or `BOOST_ALL_DYN_LINK` when building the application, so that the library

assumes static linking by default. Whenever such linking errors appear, one can decode the namespace name in the missing symbols and the exported symbols of Boost.Log library and adjust library or application configuration accordingly.

Why MSVC 2010 fails to link the library with error LNK1123: failure during conversion to COFF: file invalid or corrupt?

If you have several versions of Visual Studio installed and trying to build the library with Visual Studio 2010, the compilation may fail with linker error LNK1123. This seems to be a known problem caused by some conflict between Visual Studio 2010 and .NET Framework 4.5, which is installed with Visual Studio 2012.

The suggested solution is to upgrade Visual Studio 2010 to Visual Studio 2010 SP1 or overwrite `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\cvtres.exe"` with the one of Visual Studio 2010 SP1 or 2012.

Reference

Top level headers

Header <boost/log/attributes.hpp>

Andrey Semashev

13.07.2009
This header includes other Boost.Log headers with all attributes.

Header <boost/log/common.hpp>

Andrey Semashev

14.03.2009
This header includes other Boost.Log headers that are commonly used in logging applications. Note that the header does not include any headers required to setup the library, as usually they aren't needed in more than one translation unit of the application.

Header <boost/log/core.hpp>

Andrey Semashev

19.04.2007
This header includes Boost.Log headers related to the logging core.

Header <boost/log/exceptions.hpp>

Andrey Semashev

31.10.2009
The header contains exception classes declarations.

```
namespace boost {
  namespace log {
    class conversion_error;
    class invalid_type;
    class invalid_value;
    class limitation_error;
    class logic_error;
    class missing_value;
    class odr_violation;
    class parse_error;
    class runtime_error;
    class setup_error;
    class system_error;
    class unexpected_call;
  }
}
```

Class conversion_error

boost::log::conversion_error — Exception class that is used to indicate conversion errors.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class conversion_error : public runtime_error {
public:
  // construct/copy/destruct
  conversion_error();
  explicit conversion_error(std::string const &);
  ~conversion_error();
};
```

Description

`conversion_error` public construct/copy/destruct

1. ```
   conversion_error();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit conversion_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~conversion_error();
   ```

   Destructor

Class invalid_type

boost::log::invalid_type — Exception class that is used to indicate errors of incorrect type of an object.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class invalid_type : public runtime_error {
public:
  // construct/copy/destruct
  invalid_type();
  explicit invalid_type(std::string const &);
  ~invalid_type();
};
```

Description

`invalid_type` public construct/copy/destruct

1. ```
   invalid_type();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit invalid_type(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~invalid_type();
   ```

   Destructor

Class invalid_value

boost::log::invalid_value — Exception class that is used to indicate errors of incorrect value of an object.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class invalid_value : public runtime_error {
public:
  // construct/copy/destruct
  invalid_value();
  explicit invalid_value(std::string const &);
  ~invalid_value();
};
```

Description

`invalid_value` public construct/copy/destruct

1. ```
   invalid_value();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit invalid_value(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~invalid_value();
   ```

   Destructor

Class limitation_error

boost::log::limitation_error — Exception class that is used to indicate library limitation.
   ```

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class limitation_error : public logic_error {
public:
  // construct/copy/destruct
  limitation_error();
  explicit limitation_error(std::string const &);
  ~limitation_error();
};
```

Description

`limitation_error` public construct/copy/destruct

1. ```
   limitation_error();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit limitation_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~limitation_error();
   ```

   Destructor

Class logic_error

boost::log::logic_error — Base class for logic exceptions from the logging library.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class logic_error : public logic_error {
public:
  // construct/copy/destruct
  explicit logic_error(std::string const &);
  ~logic_error();
};
```

Description

Exceptions derived from this class usually indicate errors on the user's side, such as incorrect library usage.

`logic_error` public construct/copy/destruct

1. ```
   explicit logic_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

2. ```
   ~logic_error();
   ```

   Destructor

Class missing_value

boost::log::missing_value — Exception class that is used to indicate errors of missing values.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class missing_value : public runtime_error {
public:
  // construct/copy/destruct
  missing_value();
  explicit missing_value(std::string const &);
  ~missing_value();
};
```

Description

`missing_value` public construct/copy/destruct

1. ```
   missing_value();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit missing_value(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~missing_value();
   ```

   Destructor

Class odr_violation

boost::log::odr_violation — Exception class that is used to indicate ODR violation.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class odr_violation : public logic_error {
public:
  // construct/copy/destruct
  odr_violation();
  explicit odr_violation(std::string const &);
  ~odr_violation();
};
```

Description

`odr_violation` public construct/copy/destruct

1. ```
   odr_violation();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit odr_violation(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~odr_violation();
   ```

   Destructor

Class parse_error

boost::log::parse_error — Exception class that is used to indicate parsing errors.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class parse_error : public runtime_error {
public:
  // construct/copy/destruct
  parse_error();
  explicit parse_error(std::string const &);
  ~parse_error();
};
```

Description

`parse_error` public construct/copy/destruct

1. ```
   parse_error();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit parse_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~parse_error();
   ```

   Destructor

Class runtime_error

boost::log::runtime_error — Base class for runtime exceptions from the logging library.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class runtime_error : public runtime_error {
public:
  // construct/copy/destruct
  explicit runtime_error(std::string const &);
  ~runtime_error();
};
```

Description

Exceptions derived from this class indicate a problem that may not directly be caused by the user's code that interacts with the library, such as errors caused by input data.

`runtime_error` public construct/copy/destruct

1. ```
   explicit runtime_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

2. ```
   ~runtime_error();
   ```

   Destructor

Class setup_error

boost::log::setup_error — Exception class that is used to indicate invalid library setup.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class setup_error : public logic_error {
public:
  // construct/copy/destruct
  setup_error();
  explicit setup_error(std::string const &);
  ~setup_error();
};
```

Description

`setup_error` public construct/copy/destruct

1. ```
   setup_error();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit setup_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~setup_error();
   ```

   Destructor

Class system_error

boost::log::system_error — Exception class that is used to indicate underlying OS API errors.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class system_error : public runtime_error {
public:
  // construct/copy/destruct
  system_error();
  explicit system_error(std::string const &);
  ~system_error();
};
```

Description

`system_error` public construct/copy/destruct

1. ```
   system_error();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit system_error(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~system_error();
   ```

   Destructor

Class unexpected_call

boost::log::unexpected_call — Exception class that is used to indicate invalid call sequence.

Synopsis

```
// In header: <boost/log/exceptions.hpp>


class unexpected_call : public logic_error {
public:
  // construct/copy/destruct
  unexpected_call();
  explicit unexpected_call(std::string const &);
  ~unexpected_call();
};
```

Description

`unexpected_call` public construct/copy/destruct

1. ```
   unexpected_call();
   ```

   Default constructor. Creates an exception with the default error message.

2. ```
   explicit unexpected_call(std::string const & descr);
   ```

   Initializing constructor. Creates an exception with the specified error message.

3. ```
   ~unexpected_call();
   ```

   Destructor

Header <boost/log/expressions.hpp>

Andrey Semashev

10.11.2012
This header includes other Boost.Log headers with all template expression tools.

Header <boost/log/sinks.hpp>

Andrey Semashev

13.07.2009
This header includes other Boost.Log headers with all sinks.

Header <boost/log/trivial.hpp>

Andrey Semashev

07.11.2009
This header defines tools for trivial logging support

```
BOOST_LOG_TRIVIAL(lvl)
```
```

```
namespace boost {
  namespace log {
    namespace trivial {
      struct logger;

      // Trivial severity levels.
      enum severity_level { trace, debug, info, warning, error, fatal };

     typedef sources::severity_logger_mt< severity_level > logger_type;  // Trivial logger type.
      const char * to_string(severity_level);
      template<typename CharT, typename TraitsT>
        std::basic_ostream< CharT, TraitsT > &
        operator<<(std::basic_ostream< CharT, TraitsT > &, severity_level);
      template<typename CharT, typename TraitsT>
        std::basic_istream< CharT, TraitsT > &
        operator>>(std::basic_istream< CharT, TraitsT > &, severity_level &);
    }
  }
}
```

Struct logger

boost::log::trivial::logger — Trivial logger tag.

Synopsis

```
// In header: <boost/log/trivial.hpp>


struct logger {
  // types
  typedef trivial::logger_type logger_type;  // Logger type.

  // public static functions
  static logger_type & get();
};
```

Description

This tag can be used to acquire the logger that is used with lrivial logging macros. This may be useful when the logger is used with other macros which require a logger.

`logger` public static functions

1. ```
   static logger_type & get();
   ```

   Returns a reference to the trivial logger instance

Function to_string

boost::log::trivial::to_string — Returns stringized enumeration value or `NULL`, if the value is not valid.
   ```

Synopsis

```
// In header: <boost/log/trivial.hpp>


const char * to_string(severity_level lvl);
```

Function template operator<<

boost::log::trivial::operator<< — Outputs stringized representation of the severity level to the stream.

Synopsis

```
// In header: <boost/log/trivial.hpp>


template<typename CharT, typename TraitsT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm, severity_level lvl);
```

Function template operator>>

boost::log::trivial::operator>> — Reads stringized representation of the severity level from the stream.

Synopsis

```
// In header: <boost/log/trivial.hpp>


template<typename CharT, typename TraitsT>
  std::basic_istream< CharT, TraitsT > &
  operator>>(std::basic_istream< CharT, TraitsT > & strm,
             severity_level & lvl);
```

Macro BOOST_LOG_TRIVIAL

BOOST_LOG_TRIVIAL

Synopsis

```
// In header: <boost/log/trivial.hpp>

BOOST_LOG_TRIVIAL(lvl)
```

Description

The macro is used to initiate logging. The `lvl` argument of the macro specifies one of the following severity levels: `trace`, `debug`, `info`, `warning`, `error` or `fatal` (see `severity_level` enum). Following the macro, there may be a streaming expression that composes the record message string. For example:

```
BOOST_LOG_TRIVIAL(info) << "Hello, world!";
```

Core components

Header <boost/log/core/core.hpp>

Andrey Semashev

19.04.2007
This header contains logging core class definition.

```
namespace boost {
  namespace log {
    class core;

    typedef shared_ptr< core > core_ptr;
  }
}
```

Class core

boost::log::core — Logging library core class.

Synopsis

```
// In header: <boost/log/core/core.hpp>


class core {
public:
  // types
  typedef unspecified exception_handler_type;  // Exception handler function type.

  // construct/copy/destruct
  core(core const &) = delete;
  core & operator=(core const &) = delete;
  ~core();

  // public member functions
  bool set_logging_enabled(bool = true);
  bool get_logging_enabled() const;
  void set_filter(filter const &);
  void reset_filter();
  void add_sink(shared_ptr< sinks::sink > const &);
  void remove_sink(shared_ptr< sinks::sink > const &);
  void remove_all_sinks();
  void flush();
  std::pair< attribute_set::iterator, bool >
  add_global_attribute(attribute_name const &, attribute const &);
  void remove_global_attribute(attribute_set::iterator);
  attribute_set get_global_attributes() const;
  void set_global_attributes(attribute_set const &);
  std::pair< attribute_set::iterator, bool >
  add_thread_attribute(attribute_name const &, attribute const &);
  void remove_thread_attribute(attribute_set::iterator);
  attribute_set get_thread_attributes() const;
  void set_thread_attributes(attribute_set const &);
  void set_exception_handler(exception_handler_type const &);
  record open_record(attribute_set const &);
  record open_record(attribute_value_set const &);
  record open_record(attribute_value_set &&);
  void push_record(record &&);

  // public static functions
  static core_ptr get();
};
```

Description

The logging core is used to interconnect log sources and sinks. It also provides a number of basic features, like global filtering and global and thread-specific attribute storage.

The logging core is a singleton. Users can acquire the core instance by calling the static method `get`.

`core` public construct/copy/destruct

1. ```
   core(core const &) = delete;
   ```

2. ```
   core & operator=(core const &) = delete;
   ```

3.
```
~core();
```

Destructor. Destroys the core, releases any sinks and attributes that were registered.

`core` public member functions

1.
```
bool set_logging_enabled(bool enabled = true);
```

The method enables or disables logging.

Setting this status to `false` allows you to completely wipe out any logging activity, including filtering and generation of attribute values. It is useful if you want to completely disable logging in a running application. The state of logging does not alter any other properties of the logging library, such as filters or sinks, so you can enable logging with the very same settings that you had when the logging was disabled. This feature may also be useful if you want to perform major changes to logging configuration and don't want your application to block on opening or pushing a log record.

By default logging is enabled.

Parameters: `enabled` The actual flag of logging activity.
Returns: The previous value of enabled/disabled logging flag

2.
```
bool get_logging_enabled() const;
```

The method allows to detect if logging is enabled. See the comment for `set_logging_enabled`.

3.
```
void set_filter(filter const & filter);
```

The method sets the global logging filter. The filter is applied to every log record that is processed.

Parameters: `filter` The filter function object to be installed.

4.
```
void reset_filter();
```

The method removes the global logging filter. All log records are passed to sinks without global filtering applied.

5.
```
void add_sink(shared_ptr< sinks::sink > const & s);
```

The method adds a new sink. The sink is included into logging process immediately after being added and until being removed. No sink can be added more than once at the same time. If the sink is already registered, the call is ignored.

Parameters: `s` The sink to be registered.

6.
```
void remove_sink(shared_ptr< sinks::sink > const & s);
```

The method removes the sink from the output. The sink will not receive any log records after removal. The call has no effect if the sink is not registered.

Parameters: `s` The sink to be unregistered.

7.
```
void remove_all_sinks();
```

The method removes all registered sinks from the output. The sinks will not receive any log records after removal.

8.
```
void flush();
```

The method performs flush on all registered sinks.

Note

This method may take long time to complete as it may block until all sinks manage to process all buffered log records. The call will also block all logging attempts until the operation completes.

9.
```
std::pair< attribute_set::iterator, bool >
add_global_attribute(attribute_name const & name, attribute const & attr);
```

The method adds an attribute to the global attribute set. The attribute will be implicitly added to every log record.

Parameters: `attr` The attribute factory.
`name` The attribute name.
Returns: A pair of values. If the second member is `true`, then the attribute is added and the first member points to the attribute. Otherwise the attribute was not added and the first member points to the attribute that prevents addition.

10.
```
void remove_global_attribute(attribute_set::iterator it);
```

The method removes an attribute from the global attribute set.

Parameters: `it` Iterator to the previously added attribute.
Requires: The attribute was added with the `add_global_attribute` call.
Postconditions: The attribute is no longer registered as a global attribute. The iterator is invalidated after removal.

11.
```
attribute_set get_global_attributes() const;
```

The method returns a copy of the complete set of currently registered global attributes.

12.
```
void set_global_attributes(attribute_set const & attrs);
```

The method replaces the complete set of currently registered global attributes with the provided set.

Note

The method invalidates all iterators and references that may have been returned from the `add_global_attribute` method.

Parameters: `attrs` The set of attributes to be installed.

13.
```
std::pair< attribute_set::iterator, bool >
add_thread_attribute(attribute_name const & name, attribute const & attr);
```

The method adds an attribute to the thread-specific attribute set. The attribute will be implicitly added to every log record made in the current thread.

Note

In single-threaded build the effect is the same as adding the attribute globally. This, however, does not imply that iterators to thread-specific and global attributes are interchangeable.

Parameters: `attr` The attribute factory.
`name` The attribute name.
Returns: A pair of values. If the second member is `true`, then the attribute is added and the first member points to the attribute. Otherwise the attribute was not added and the first member points to the attribute that prevents addition.

14.
```
void remove_thread_attribute(attribute_set::iterator it);
```

The method removes an attribute from the thread-specific attribute set.

Parameters: `it` Iterator to the previously added attribute.
Requires: The attribute was added with the `add_thread_attribute` call.
Postconditions: The attribute is no longer registered as a thread-specific attribute. The iterator is invalidated after removal.

15.
```
attribute_set get_thread_attributes() const;
```

The method returns a copy of the complete set of currently registered thread-specific attributes.

16.
```
void set_thread_attributes(attribute_set const & attrs);
```

The method replaces the complete set of currently registered thread-specific attributes with the provided set.

Note

The method invalidates all iterators and references that may have been returned from the `add_thread_attribute` method.

Parameters: `attrs` The set of attributes to be installed.

17.
```
void set_exception_handler(exception_handler_type const & handler);
```

The method sets exception handler function. The function will be called with no arguments in case if an exception occurs during either `open_record` or `push_record` method execution. Since exception handler is called from a `catch` statement, the exception can be rethrown in order to determine its type.

By default no handler is installed, thus any exception is propagated as usual.

See Also:

See also: `utility/exception_handler.hpp`

Note

The exception handler can be invoked in several threads concurrently. Thread interruptions are not affected by exception handlers.

Parameters: `handler` Exception handling function

18.
```
record open_record(attribute_set const & source_attributes);
```

The method attempts to open a new record to be written. While attempting to open a log record all filtering is applied. A successfully opened record can be pushed further to sinks by calling the `push_record` method or simply destroyed by destroying the returned object.

More than one open records are allowed, such records exist independently. All attribute values are acquired during opening the record and do not interact between records.

The returned records can be copied, however, they must not be passed between different threads.

Throws: If an exception handler is installed, only throws if the handler throws. Otherwise may throw if one of the sinks throws, or some system resource limitation is reached.
Parameters: `source_attributes` The set of source-specific attributes to be attached to the record to be opened.
Returns: A valid log record if the record is opened, an invalid record object if not (e.g. because it didn't pass filtering).

19.
```
record open_record(attribute_value_set const & source_attributes);
```

The method attempts to open a new record to be written. While attempting to open a log record all filtering is applied. A successfully opened record can be pushed further to sinks by calling the `push_record` method or simply destroyed by destroying the returned object.

More than one open records are allowed, such records exist independently. All attribute values are acquired during opening the record and do not interact between records.

The returned records can be copied, however, they must not be passed between different threads.

Throws: If an exception handler is installed, only throws if the handler throws. Otherwise may throw if one of the sinks throws, or some system resource limitation is reached.
Parameters: `source_attributes` The set of source-specific attribute values to be attached to the record to be opened.
Returns: A valid log record if the record is opened, an invalid record object if not (e.g. because it didn't pass filtering).

20.
```
record open_record(attribute_value_set && source_attributes);
```

The method attempts to open a new record to be written. While attempting to open a log record all filtering is applied. A successfully opened record can be pushed further to sinks by calling the `push_record` method or simply destroyed by destroying the returned object.

More than one open records are allowed, such records exist independently. All attribute values are acquired during opening the record and do not interact between records.

The returned records can be copied, however, they must not be passed between different threads.

Throws: If an exception handler is installed, only throws if the handler throws. Otherwise may throw if one of the sinks throws, or some system resource limitation is reached.
Parameters: `source_attributes` The set of source-specific attribute values to be attached to the record to be opened. The contents of this container are unspecified after this call.
Returns: A valid log record if the record is opened, an invalid record object if not (e.g. because it didn't pass filtering).

21.
```
void push_record(record && rec);
```

The method pushes the record to sinks. The record is moved from in the process.

Throws: If an exception handler is installed, only throws if the handler throws. Otherwise may throw if one of the sinks throws.
Parameters: `rec` A previously successfully opened log record.
Requires: `!!rec == true`
Postconditions: `!rec == true`

`core` public static functions

1.
```
static core_ptr get();
```

Returns: The method returns a pointer to the logging core singleton instance.

Header <boost/log/core/record.hpp>

Andrey Semashev

09.03.2009
This header contains a logging record class definition.

```
namespace boost {
  namespace log {
    class record;
    void swap(record &, record &);
  }
}
```

Class record

boost::log::record — Logging record class.

Synopsis

```
// In header: <boost/log/core/record.hpp>


class record {
public:
  // construct/copy/destruct
  record();
  record(record &&) noexcept;
  record & operator=(record &&) noexcept;
  ~record();

  // public member functions
  attribute_value_set & attribute_values() noexcept;
  attribute_value_set const & attribute_values() const noexcept;
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  void swap(record &) noexcept;
  void reset() noexcept;
  attribute_value_set::mapped_type
  operator[](attribute_value_set::key_type) const;
  template<typename DescriptorT, template< typename > class ActorT>
    result_of::extract< typename expressions::attrib↵
ute_keyword< DescriptorT, ActorT >::value_type, DescriptorT >::type
    operator[](expressions::attribute_keyword< DescriptorT, ActorT > const &) const;
  record_view lock();
};
```

Description

The logging record encapsulates all information related to a single logging statement, in particular, attribute values view and the log message string. The record can be updated before pushing for further processing to the logging core.

`record` public construct/copy/destruct

1. ```
   record();
   ```

   Default constructor. Creates an empty record that is equivalent to the invalid record handle.

   Postconditions: `!*this == true`

2. ```
   record(record && that) noexcept;
   ```

   Move constructor. Source record contents unspecified after the operation.

3. ```
   record & operator=(record && that) noexcept;
   ```

   Move assignment. Source record contents unspecified after the operation.

4. ```
   ~record();
   ```

   Destructor. Destroys the record, releases any sinks and attribute values that were involved in processing this record.

`record` public member functions

1. ```
   attribute_value_set & attribute_values() noexcept;
   ```

   Requires: `!!*this`
   Returns: A reference to the set of attribute values attached to this record

2. ```
   attribute_value_set const & attribute_values() const noexcept;
   ```

   Requires: `!!*this`
   Returns: A reference to the set of attribute values attached to this record

3. ```
   explicit operator bool() const noexcept;
   ```

   Conversion to an unspecified boolean type

   Returns: `true`, if the `*this` identifies a log record, `false`, if the `*this` is not valid

4. ```
   bool operator!() const noexcept;
   ```

   Inverted conversion to an unspecified boolean type

   Returns: `false`, if the `*this` identifies a log record, `true`, if the `*this` is not valid

5. ```
   void swap(record & that) noexcept;
   ```

   Swaps two handles

   Parameters: `that` Another record to swap with **Throws:** Nothing

6. ```
   void reset() noexcept;
   ```

Resets the log record handle. If there are no other handles left, the log record is closed and all resources referenced by the record are released.

Postconditions: `!*this == true`

7.
```
attribute_value_set::mapped_type
operator[](attribute_value_set::key_type name) const;
```

Attribute value lookup.

Parameters: `name` Attribute name.
Returns: An `attribute_value`, non-empty if it is found, empty otherwise.

8.
```
template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename expressions::attribute_keyword< DescriptorT, ActorT
>::value_type, DescriptorT >::type
  operator[](expressions::attribute_keyword< DescriptorT, ActorT > const & keyword) const;
```

Attribute value lookup.

Parameters: `keyword` Attribute keyword.
Returns: A `value_ref` with extracted attribute value if it is found, empty `value_ref` otherwise.

9.
```
record_view lock();
```

The function ensures that the log record does not depend on any thread-specific data. Then the record contents are used to construct a `record_view` which is returned from the function. The record is no longer valid after the call.

Requires: `!!*this`
Postconditions: `!*this`
Returns: The record view that contains all attribute values from the original record.

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/core/record.hpp>


void swap(record & left, record & right);
```

Description

A free-standing swap function overload for `record`

Header <boost/log/core/record_view.hpp>

Andrey Semashev

09.03.2009
This header contains a logging record view class definition.

```
namespace boost {
  namespace log {
    class record_view;
    void swap(record_view &, record_view &);
  }
}
```

Class record_view

boost::log::record_view — Logging record view class.

Synopsis

```
// In header: <boost/log/core/record_view.hpp>


class record_view {
public:
  // construct/copy/destruct
  record_view() = default;
  record_view(record_view const &) noexcept;
  record_view(record_view &&) noexcept;
  record_view & operator=(record_view const &) noexcept;
  record_view & operator=(record_view &&) noexcept;
  ~record_view();

  // public member functions
  attribute_value_set const & attribute_values() const noexcept;
  bool operator==(record_view const &) const noexcept;
  bool operator!=(record_view const &) const noexcept;
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  void swap(record_view &) noexcept;
  void reset() noexcept;
  attribute_value_set::mapped_type
  operator[](attribute_value_set::key_type) const;
  template<typename DescriptorT, template< typename > class ActorT>
    result_of::extract< typename expressions::attrib↵
ute_keyword< DescriptorT, ActorT >::value_type, DescriptorT >::type
    operator[](expressions::attribute_keyword< DescriptorT, ActorT > const &) const;
};
```

Description

The logging record encapsulates all information related to a single logging statement, in particular, attribute values view and the log message string. The view is immutable, it is implemented as a wrapper around a reference-counted implementation.

`record_view` public construct/copy/destruct

1. ```
   record_view() = default;
   ```

   Default constructor. Creates an empty record view that is equivalent to the invalid record handle.

   Postconditions: `!*this == true`

2. ```
   record_view(record_view const & that) noexcept;
   ```

   Copy constructor

3. ```
   record_view(record_view && that) noexcept;
   ```

   Move constructor. Source record contents unspecified after the operation.

4. ```
   record_view & operator=(record_view const & that) noexcept;
   ```

   Copy assignment

5. ```
   record_view & operator=(record_view && that) noexcept;
   ```

   Move assignment. Source record contents unspecified after the operation.

6. ```
   ~record_view();
   ```

   Destructor. Destroys the record, releases any sinks and attribute values that were involved in processing this record.

`record_view` public member functions

1. ```
   attribute_value_set const & attribute_values() const noexcept;
   ```

   Requires: `!!*this`
   Returns: A reference to the set of attribute values attached to this record

2. ```
   bool operator==(record_view const & that) const noexcept;
   ```

   Equality comparison

   Parameters: `that` Comparand
   Returns: `true` if both `*this` and *that* identify the same log record or both do not identify any record, `false` otherwise.

3. ```
   bool operator!=(record_view const & that) const noexcept;
   ```

   Inequality comparison

   Parameters: `that` Comparand
   Returns: `!(*this == that)`

4. ```
   explicit operator bool() const noexcept;
   ```

   Conversion to an unspecified boolean type

   Returns: `true`, if the `*this` identifies a log record, `false`, if the `*this` is not valid

5. ```
   bool operator!() const noexcept;
   ```

   Inverted conversion to an unspecified boolean type

   Returns: `false`, if the `*this` identifies a log record, `true`, if the `*this` is not valid

6. ```
   void swap(record_view & that) noexcept;
   ```

   Swaps two handles

Parameters: `that` Another record to swap with **Throws:** Nothing

7.
```
void reset() noexcept;
```

Resets the log record handle. If there are no other handles left, the log record is closed and all resources referenced by the record are released.

Postconditions: `!*this == true`

8.
```
attribute_value_set::mapped_type
operator[](attribute_value_set::key_type name) const;
```

Attribute value lookup.

Parameters: `name` Attribute name.
Returns: An `attribute_value`, non-empty if it is found, empty otherwise.

9.
```
template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename expressions::attribute_keyword< DescriptorT, ActorT
>::value_type, DescriptorT >::type
  operator[](expressions::attribute_keyword< DescriptorT, ActorT > const & keyword) const;
```

Attribute value lookup.

Parameters: `keyword` Attribute keyword.
Returns: A `value_ref` with extracted attribute value if it is found, empty `value_ref` otherwise.

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/core/record_view.hpp>


void swap(record_view & left, record_view & right);
```

Description

A free-standing swap function overload for `record_view`

Attributes

Header <boost/log/attributes/attribute.hpp>

Andrey Semashev

15.04.2007
The header contains attribute interface definition.

```
namespace boost {
  namespace log {
    class attribute;
    void swap(attribute &, attribute &);
  }
}
```

Class attribute

boost::log::attribute — A base class for an attribute value factory.

Synopsis

```
// In header: <boost/log/attributes/attribute.hpp>


class attribute {
public:
  // member classes/structs/unions

  // A base class for an attribute value factory.

  struct impl : public boost::intrusive_ref_counter< impl > {
    // construct/copy/destruct
    ~impl();

    // public member functions
    virtual attribute_value get_value() = 0;

    // public static functions
    static void * operator new(std::size_t);
    static void operator delete(void *, std::size_t) noexcept;
  };
  // construct/copy/destruct
  attribute() = default;
  attribute(attribute const &) noexcept;
  attribute(attribute &&) noexcept;
  explicit attribute(intrusive_ptr< impl >) noexcept;
  attribute & operator=(attribute const &) noexcept;
  attribute & operator=(attribute &&) noexcept;

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  attribute_value get_value() const;
  void swap(attribute &) noexcept;

  // protected member functions
  impl * get_impl() const noexcept;
  void set_impl(intrusive_ptr< impl >) noexcept;

  // friend functions
  template<typename T> friend T attribute_cast(attribute const &);
};
```

Description

Every attribute is represented with a factory that is basically an attribute value generator. The sole purpose of an attribute is to return an actual value when requested. A simplest attribute can always return the same value that it stores internally, but more complex ones can perform a considerable amount of work to return a value, and the returned values may differ each time requested.

A word about thread safety. An attribute should be prepared to be requested a value from multiple threads concurrently.

`attribute` public construct/copy/destruct

1.
```
attribute() = default;
```

Default constructor. Creates an empty attribute value factory, which is not usable until `set_impl` is called.

2.
```
attribute(attribute const & that) noexcept;
```

Copy constructor

3.
```
attribute(attribute && that) noexcept;
```

Move constructor

4.
```
explicit attribute(intrusive_ptr< impl > p) noexcept;
```

Initializing constructor

Parameters: `p` Pointer to the implementation. Must not be `NULL`.

5.
```
attribute & operator=(attribute const & that) noexcept;
```

Copy assignment

6.
```
attribute & operator=(attribute && that) noexcept;
```

Move assignment

`attribute` public member functions

1.
```
explicit operator bool() const noexcept;
```

Verifies that the factory is not in empty state

2.
```
bool operator!() const noexcept;
```

Verifies that the factory is in empty state

3.
```
attribute_value get_value() const;
```

Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

4.
```
void swap(attribute & that) noexcept;
```

The method swaps two factories (i.e. their implementations).

`attribute` protected member functions

1.
```
impl * get_impl() const noexcept;
```

Returns: The pointer to the implementation

2.
```
void set_impl(intrusive_ptr< impl > p) noexcept;
```

Sets the pointer to the factory implementation.

Parameters: p Pointer to the implementation. Must not be `NULL`.

`attribute` friend functions

1.
```
template<typename T> friend T attribute_cast(attribute const &);
```

The function casts one attribute factory to another

Struct impl

boost::log::attribute::impl — A base class for an attribute value factory.

Synopsis

```
// In header: <boost/log/attributes/attribute.hpp>


// A base class for an attribute value factory.

struct impl : public boost::intrusive_ref_counter< impl > {
  // construct/copy/destruct
  ~impl();

  // public member functions
  virtual attribute_value get_value() = 0;

  // public static functions
  static void * operator new(std::size_t);
  static void operator delete(void *, std::size_t) noexcept;
};
```

Description

All attributes must derive their implementation from this class.

`impl` public construct/copy/destruct

1.
```
~impl();
```

Virtual destructor.

`impl` public member functions

1.
```
virtual attribute_value get_value() = 0;
```

Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

`impl` public static functions

1.
```
static void * operator new(std::size_t size);
```

2.
```
static void operator delete(void * p, std::size_t size) noexcept;
```

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/attributes/attribute.hpp>


void swap(attribute & left, attribute & right);
```

Description

The function swaps two attribute value factories

Header <boost/log/attributes/attribute_cast.hpp>

Andrey Semashev

06.08.2010
The header contains utilities for casting between attribute factories.

```
namespace boost {
  namespace log {
    template<typename T> T attribute_cast(attribute const &);
    namespace attributes {
      class cast_source;
    }
  }
}
```

Class cast_source

boost::log::attributes::cast_source

Synopsis

```
// In header: <boost/log/attributes/attribute_cast.hpp>


class cast_source {
public:
  // construct/copy/destruct
  explicit cast_source(attribute::impl *);

  // public member functions
  template<typename T> T * as() const;
};
```

Description

The class holds a reference to the attribute factory implementation being casted

`cast_source` public construct/copy/destruct

1. ```
   explicit cast_source(attribute::impl * p);
   ```

   Initializing constructor. Creates a source that refers to the specified factory implementation.

`cast_source` public member functions

1. ```
   template<typename T> T * as() const;
   ```

   The function attempts to cast the aggregated pointer to the implementation to the specified type.

   Returns: The converted pointer or `NULL`, if the conversion fails.

Function template attribute_cast

boost::log::attribute_cast

Synopsis

```
// In header: <boost/log/attributes/attribute_cast.hpp>


template<typename T> T attribute_cast(attribute const & attr);
```

Description

The function casts one attribute factory to another

Header <boost/log/attributes/attribute_name.hpp>

Andrey Semashev

28.06.2010
The header contains attribute name interface definition.

```
namespace boost {
  namespace log {
    class attribute_name;
    template<typename CharT, typename TraitsT>
      std::basic_ostream< CharT, TraitsT > &
      operator<<(std::basic_ostream< CharT, TraitsT > &,
                 attribute_name const &);
  }
}
```

Class attribute_name

boost::log::attribute_name — The class represents an attribute name in containers used by the library.

Synopsis

```
// In header: <boost/log/attributes/attribute_name.hpp>


class attribute_name {
public:
  // types
  typedef std::string string_type;  // String type.
  typedef unspecified id_type;      // Associated identifier.

  // construct/copy/destruct
  attribute_name() noexcept;
  attribute_name(const char *);
  attribute_name(string_type const &);

  // public member functions
  bool operator==(attribute_name const &) const noexcept;
  bool operator!=(attribute_name const &) const noexcept;
  bool operator==(const char *) const;
  bool operator!=(const char *) const;
  bool operator==(string_type const &) const;
  bool operator!=(string_type const &) const;
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  id_type id() const noexcept;
  string_type const & string() const;
};
```

Description

The class mostly serves for optimization purposes. Each attribute name that is used with the library is automatically associated with a unique identifier, which is much lighter in terms of memory footprint and operations complexity. This is done transparently by this class, on object construction. Passing objects of this class to other library methods, such as attribute lookup functions, will not require this translation and/or string copying and thus will result in a more efficient code.

`attribute_name` public construct/copy/destruct

1. ```
   attribute_name() noexcept;
   ```

   Default constructor. Creates an object that does not refer to any attribute name.

2. ```
   attribute_name(const char * name);
   ```

Constructs an attribute name from the specified string

Parameters: `name` An attribute name
Requires: *name* is not NULL and points to a zero-terminated string

3.
```
attribute_name(string_type const & name);
```

Constructs an attribute name from the specified string

Parameters: `name` An attribute name

`attribute_name` public member functions

1.
```
bool operator==(attribute_name const & that) const noexcept;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to the same attribute name, and `false` otherwise.

2.
```
bool operator!=(attribute_name const & that) const noexcept;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to different attribute names, and `false` otherwise.

3.
```
bool operator==(const char * that) const;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to the same attribute name, and `false` otherwise.

4.
```
bool operator!=(const char * that) const;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to different attribute names, and `false` otherwise.

5.
```
bool operator==(string_type const & that) const;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to the same attribute name, and `false` otherwise.

6.
```
bool operator!=(string_type const & that) const;
```

Compares the attribute names

Returns: `true` if `*this` and `that` refer to different attribute names, and `false` otherwise.

7.
```
explicit operator bool() const noexcept;
```

Checks if the object was default-constructed

Returns: `true` if `*this` was constructed with an attribute name, `false` otherwise

8.
```
bool operator!() const noexcept;
```

Checks if the object was default-constructed

Returns: `true` if `*this` was default-constructed and does not refer to any attribute name, `false` otherwise

9.
```
id_type id() const noexcept;
```

Requires: `(!*this) == false`
Returns: The associated id value

10.
```
string_type const & string() const;
```

Requires: `(!*this) == false`
Returns: The attribute name string that was used during the object construction

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/attributes/attribute_name.hpp>


template<typename CharT, typename TraitsT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             attribute_name const & name);
```

Header <boost/log/attributes/attribute_set.hpp>

Andrey Semashev

08.03.2007
This header contains definition of the attribute set container.

```
namespace boost {
  namespace log {
    class attribute_set;
    void swap(attribute_set &, attribute_set &);
  }
}
```

Class attribute_set

boost::log::attribute_set — An attribute set class.

Synopsis

```
// In header: <boost/log/attributes/attribute_set.hpp>


class attribute_set {
public:
  // types
  typedef attribute_name                                key_type;         // Key type.
  typedef attribute                                     mapped_type;      // Mapped attribute type.
  typedef std::pair< const key_type, mapped_type >      value_type;       // Value type.
  typedef value_type &                                  reference;        // Reference type.
  typedef value_type const &                            const_reference;  // Const reference type.
  typedef value_type *                                  pointer;          // Pointer type.
  typedef value_type const *                            const_pointer;    // Const pointer type.
  typedef std::size_t                                   size_type;        // Size type.
  typedef std::ptrdiff_t                                difference_type;  // Difference type.
  typedef implementation_defined                        iterator;
  typedef implementation_defined                        const_iterator;

  // construct/copy/destruct
  attribute_set();
  attribute_set(attribute_set const &);
  attribute_set(attribute_set &&) noexcept;
  attribute_set & operator=(attribute_set) noexcept;
  ~attribute_set();

  // public member functions
  void swap(attribute_set &) noexcept;
  iterator begin() noexcept;
  iterator end() noexcept;
  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;
  size_type size() const noexcept;
  bool empty() const noexcept;
  iterator find(key_type) noexcept;
  const_iterator find(key_type) const noexcept;
  size_type count(key_type) const noexcept;
  unspecified operator[](key_type) noexcept;
  mapped_type operator[](key_type) const noexcept;
  std::pair< iterator, bool > insert(key_type, mapped_type const &);
  std::pair< iterator, bool > insert(const_reference);
  template<typename FwdIteratorT> void insert(FwdIteratorT, FwdIteratorT);
  template<typename FwdIteratorT, typename OutputIteratorT>
    void insert(FwdIteratorT, FwdIteratorT, OutputIteratorT);
  size_type erase(key_type) noexcept;
  void erase(iterator) noexcept;
  void erase(iterator, iterator) noexcept;
  void clear() noexcept;
};
```

Description

An attribute set is an associative container with attribute name as a key and pointer to the attribute as a mapped value. The container allows storing only one element for each distinct key value. In most regards attribute set container provides interface similar to `std::unordered_map`. However, there are differences in `operator[]` semantics and a number of optimizations with regard to iteration. Besides, attribute names are stored as a read-only `attribute_name`'s instead of `std::string`, which saves memory and CPU time.

`attribute_set` public types

1. typedef implementation_defined iterator;

Iterator type. The iterator complies to the bidirectional iterator requirements.

2. typedef implementation_defined const_iterator;

   Constant iterator type. The iterator complies to the bidirectional iterator requirements with read-only capabilities.

`attribute_set` public construct/copy/destruct

1. ```
   attribute_set();
   ```

   Default constructor.

   Postconditions: `empty() == true`

2. ```
   attribute_set(attribute_set const & that);
   ```

   Copy constructor.

   Postconditions: `size() == that.size() && std::equal(begin(), end(), that.begin()) == true`

3. ```
   attribute_set(attribute_set && that) noexcept;
   ```

   Move constructor

4. ```
   attribute_set & operator=(attribute_set that) noexcept;
   ```

   Copy assignment operator.

   Postconditions: `size() == that.size() && std::equal(begin(), end(), that.begin()) == true`

5. ```
   ~attribute_set();
   ```

   Destructor. All stored references to attributes are released.

`attribute_set` public member functions

1. ```
   void swap(attribute_set & that) noexcept;
   ```

   Swaps two instances of the container.

   Throws: Nothing.

2. ```
   iterator begin() noexcept;
   ```

   Returns: Iterator to the first element of the container.

3. ```
   iterator end() noexcept;
   ```

   Returns: Iterator to the after-the-last element of the container.

4. ```
   const_iterator begin() const noexcept;
   ```

   Returns: Constant iterator to the first element of the container.
   ```

5.
```
const_iterator end() const noexcept;
```

Returns: Constant iterator to the after-the-last element of the container.

6.
```
size_type size() const noexcept;
```

Returns: Number of elements in the container.

7.
```
bool empty() const noexcept;
```

Returns: true if there are no elements in the container, false otherwise.

8.
```
iterator find(key_type key) noexcept;
```

The method finds the attribute by name.

Parameters: `key` Attribute name.
Returns: Iterator to the found element or end() if the attribute with such name is not found.

9.
```
const_iterator find(key_type key) const noexcept;
```

The method finds the attribute by name.

Parameters: `key` Attribute name.
Returns: Iterator to the found element or `end()` if the attribute with such name is not found.

10.
```
size_type count(key_type key) const noexcept;
```

The method counts the number of the attribute occurrences in the container. Since there can be only one attribute with a particular key, the method always return 0 or 1.

Parameters: `key` Attribute name.
Returns: The number of times the attribute is found in the container.

11.
```
unspecified operator[](key_type key) noexcept;
```

Combined lookup/insertion operator. The operator semantics depends on the further usage of the returned reference.

- If the reference is used as an assignment target, the assignment expression is equivalent to element insertion, where the element is composed of the second argument of the `operator[]` as a key and the second argument of assignment as a mapped value.
- If the returned reference is used in context where a conversion to the mapped type is required, the result of the conversion is equivalent to the mapped value found with the second argument of the `operator[]` as a key, if such an element exists in the container, or a default-constructed mapped value, if an element does not exist in the container.

Parameters: `key` Attribute name.
Returns: A smart reference object of unspecified type.

12.
```
mapped_type operator[](key_type key) const noexcept;
```

Lookup operator

Parameters: `key` Attribute name.
Returns: If an element with the corresponding attribute name is found in the container, its mapped value is returned. Otherwise a default-constructed mapped value is returned.

13.
```
std::pair< iterator, bool > insert(key_type key, mapped_type const & data);
```

Insertion method

Parameters: `data` Pointer to the attribute. Must not be NULL.
`key` Attribute name.
Returns: A pair of values. If second is true, the insertion succeeded and the first component points to the inserted element. Otherwise the first component points to the element that prevents insertion.

14.
```
std::pair< iterator, bool > insert(const_reference value);
```

Insertion method

Parameters: `value` An element to be inserted.
Returns: A pair of values. If second is true, the insertion succeeded and the first component points to the inserted element. Otherwise the first component points to the element that prevents insertion.

15.
```
template<typename FwdIteratorT>
  void insert(FwdIteratorT begin, FwdIteratorT end);
```

Mass insertion method.

Parameters: `begin` A forward iterator that points to the first element to be inserted.
`end` A forward iterator that points to the after-the-last element to be inserted.

16.
```
template<typename FwdIteratorT, typename OutputIteratorT>
  void insert(FwdIteratorT begin, FwdIteratorT end, OutputIteratorT out);
```

Mass insertion method with ability to acquire iterators to the inserted elements.

Parameters: `begin` A forward iterator that points to the first element to be inserted.
`end` A forward iterator that points to the after-the-last element to be inserted.
`out` An output iterator that receives results of insertion of the elements

17.
```
size_type erase(key_type key) noexcept;
```

The method erases all attributes with the specified name

Parameters: `key` Attribute name.
Postconditions: All iterators to the erased elements become invalid.
Returns: Tne number of erased elements

18.
```
void erase(iterator it) noexcept;
```

The method erases the specified attribute

Parameters: `it` A valid iterator to the element to be erased.
Postconditions: All iterators to the erased element become invalid.
Returns: Tne number of erased elements

19.
```
void erase(iterator begin, iterator end) noexcept;
```

The method erases all attributes within the specified range

Parameters: `begin` An iterator that points to the first element to be erased.

`end`	An iterator that points to the after-the-last element to be erased.
Requires:	*end* is reachable from *begin* with a finite number of increments.
Postconditions:	All iterators to the erased elements become invalid.

20.
```
void clear() noexcept;
```

The method removes all elements from the container

Postconditions:	`empty() == true`

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/attributes/attribute_set.hpp>


void swap(attribute_set & left, attribute_set & right);
```

Description

Free swap overload

Header <boost/log/attributes/attribute_value.hpp>

Andrey Semashev

21.05.2010
The header contains `attribute_value` class definition.

```
namespace boost {
  namespace log {
    class attribute_value;
    void swap(attribute_value &, attribute_value &);
  }
}
```

Class attribute_value

boost::log::attribute_value — An attribute value class.

Synopsis

```
// In header: <boost/log/attributes/attribute_value.hpp>


class attribute_value {
public:
  // member classes/structs/unions

  // A base class for an attribute value implementation.

  struct impl : public attribute::impl {

    // public member functions
    virtual bool dispatch(type_dispatcher &) = 0;
    virtual intrusive_ptr< impl > detach_from_thread();
    virtual attribute_value get_value();
    virtual type_info_wrapper get_type() const;
  };
  // construct/copy/destruct
  attribute_value() = default;
  attribute_value(attribute_value const &) noexcept;
  attribute_value(attribute_value &&) noexcept;
  explicit attribute_value(intrusive_ptr< impl >) noexcept;
  attribute_value & operator=(attribute_value const &) noexcept;
  attribute_value & operator=(attribute_value &&) noexcept;

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  type_info_wrapper get_type() const;
  void detach_from_thread();
  bool dispatch(type_dispatcher &) const;
  template<typename T, typename TagT = void>
    result_of::extract< T, TagT >::type extract() const;
  template<typename T, typename TagT = void>
    result_of::extract_or_throw< T, TagT >::type extract_or_throw() const;
  template<typename T, typename TagT = void>
    result_of::extract_or_default< T, T, TagT >::type
    extract_or_default(T const &) const;
  template<typename T, typename TagT = void, typename DefaultT>
    result_of::extract_or_default< T, DefaultT, TagT >::type
    extract_or_default(DefaultT const &) const;
  template<typename T, typename VisitorT>
    visitation_result visit(VisitorT) const;
  void swap(attribute_value &) noexcept;
};
```

Description

An attribute value is an object that contains a piece of data that represents an attribute state at the point of the value acquisition. All major operations with log records, such as filtering and formatting, involve attribute values contained in a single view. Most likely an attribute value is implemented as a simple holder of some typed value. This holder implements the `attribute_value::implementation` interface and acts as a pimpl for the `attribute_value` object. The `attribute_value` class provides type dispatching support in order to allow to extract the value from the holder.

Normally, attributes and their values shall be designed in order to exclude as much interference as reasonable. Such approach allows to have more than one attribute value simultaneously, which improves scalability and allows to implement generating attributes.

However, there are cases when this approach does not help to achieve the required level of independency of attribute values and attribute itself from each other at a reasonable performance tradeoff. For example, an attribute or its values may use thread-specific

data, which is global and shared between all the instances of the attribute/value. Passing such an attribute value to another thread would be a disaster. To solve this the library defines an additional method for attribute values, namely `detach_from_thread`. The `attribute_value` class forwards the call to its pimpl, which is supposed to ensure that it no longer refers to any thread-specific data after the call. The pimpl can create a new holder as a result of this method and return it to the `attribute_value` wrapper, which will keep the returned reference for any further calls. This method is called for all attribute values that are passed to another thread.

`attribute_value` public construct/copy/destruct

1.
```
attribute_value() = default;
```

Default constructor. Creates an empty (absent) attribute value.

2.
```
attribute_value(attribute_value const & that) noexcept;
```

Copy constructor

3.
```
attribute_value(attribute_value && that) noexcept;
```

Move constructor

4.
```
explicit attribute_value(intrusive_ptr< impl > p) noexcept;
```

Initializing constructor. Creates an attribute value that refers to the specified holder.

Parameters: `p` A pointer to the attribute value holder.

5.
```
attribute_value & operator=(attribute_value const & that) noexcept;
```

Copy assignment

6.
```
attribute_value & operator=(attribute_value && that) noexcept;
```

Move assignment

`attribute_value` public member functions

1.
```
explicit operator bool() const noexcept;
```

The operator checks if the attribute value is empty

2.
```
bool operator!() const noexcept;
```

The operator checks if the attribute value is empty

3.
```
type_info_wrapper get_type() const;
```

The method returns the type information of the stored value of the attribute. The returned type info wrapper may be empty if the attribute value is empty or the information cannot be provided. If the returned value is not empty, the type can be used for value extraction.

4.
```
void detach_from_thread();
```

The method is called when the attribute value is passed to another thread (e.g. in case of asynchronous logging). The value should ensure it properly owns all thread-specific data.

Postconditions: The attribute value no longer refers to any thread-specific resources.

5.
```
bool dispatch(type_dispatcher & dispatcher) const;
```

The method dispatches the value to the given object. This method is a low level interface for attribute value visitation and extraction. For typical usage these interfaces may be more convenient.

Parameters: `dispatcher` The object that attempts to dispatch the stored value.
Returns: `true` if the value is not empty and the *dispatcher* was capable to consume the real attribute value type and `false` otherwise.

6.
```
template<typename T, typename TagT = void>
  result_of::extract< T, TagT >::type extract() const;
```

The method attempts to extract the stored value, assuming the value has the specified type. One can specify either a single type or an MPL type sequence, in which case the stored value is checked against every type in the sequence.

Note

Include `value_extraction.hpp` prior to using this method.

Returns: The extracted value, if the attribute value is not empty and the value is the same as specified. Otherwise returns an empty value. See description of the `result_of::extract` metafunction for information on the nature of the result value.

7.
```
template<typename T, typename TagT = void>
  result_of::extract_or_throw< T, TagT >::type extract_or_throw() const;
```

The method attempts to extract the stored value, assuming the value has the specified type. One can specify either a single type or an MPL type sequence, in which case the stored value is checked against every type in the sequence.

Note

Include `value_extraction.hpp` prior to using this method.

Returns: The extracted value, if the attribute value is not empty and the value is the same as specified. Otherwise an exception is thrown. See description of the `result_of::extract_or_throw` metafunction for information on the nature of the result value.

8.
```
template<typename T, typename TagT = void>
  result_of::extract_or_default< T, T, TagT >::type
  extract_or_default(T const & def_value) const;
```

The method attempts to extract the stored value, assuming the value has the specified type. One can specify either a single type or an MPL type sequence, in which case the stored value is checked against every type in the sequence. If extraction fails, the default value is returned.

Note

Include `value_extraction.hpp` prior to using this method.

Parameters: `def_value` Default value.
Returns: The extracted value, if the attribute value is not empty and the value is the same as specified. Otherwise returns the default value. See description of the `result_of::extract_or_default` metafunction for information on the nature of the result value.

9.
```
template<typename T, typename TagT = void, typename DefaultT>
  result_of::extract_or_default< T, DefaultT, TagT >::type
  extract_or_default(DefaultT const & def_value) const;
```

The method attempts to extract the stored value, assuming the value has the specified type. One can specify either a single type or an MPL type sequence, in which case the stored value is checked against every type in the sequence. If extraction fails, the default value is returned.

Note

Include `value_extraction.hpp` prior to using this method.

Parameters: `def_value` Default value.
Returns: The extracted value, if the attribute value is not empty and the value is the same as specified. Otherwise returns the default value. See description of the `result_of::extract_or_default` metafunction for information on the nature of the result value.

10.
```
template<typename T, typename VisitorT>
  visitation_result visit(VisitorT visitor) const;
```

The method attempts to extract the stored value, assuming the value has the specified type, and pass it to the *visitor* function object. One can specify either a single type or an MPL type sequence, in which case the stored value is checked against every type in the sequence.

Note

Include `value_visitation.hpp` prior to using this method.

Parameters: `visitor` A function object that will be invoked on the extracted attribute value. The visitor should be capable to be called with a single argument of any type of the specified types in `T`.
Returns: The result of visitation.

11.
```
void swap(attribute_value & that) noexcept;
```

The method swaps two attribute values

Struct impl

boost::log::attribute_value::impl — A base class for an attribute value implementation.

Synopsis

```
// In header: <boost/log/attributes/attribute_value.hpp>



// A base class for an attribute value implementation.

struct impl : public attribute::impl {

  // public member functions
  virtual bool dispatch(type_dispatcher &) = 0;
  virtual intrusive_ptr< impl > detach_from_thread();
  virtual attribute_value get_value();
  virtual type_info_wrapper get_type() const;
};
```

Description

All attribute value holders should derive from this interface.

`impl` public member functions

1. ```
   virtual bool dispatch(type_dispatcher & dispatcher) = 0;
   ```

   The method dispatches the value to the given object.

   Parameters: `dispatcher` The object that attempts to dispatch the stored value.
   Returns: true if *dispatcher* was capable to consume the real attribute value type and false otherwise.

2. ```
   virtual intrusive_ptr< impl > detach_from_thread();
   ```

   The method is called when the attribute value is passed to another thread (e.g. in case of asynchronous logging). The value should ensure it properly owns all thread-specific data.

   Returns: An actual pointer to the attribute value. It may either point to this object or another. In the latter case the returned pointer replaces the pointer used by caller to invoke this method and is considered to be a functional equivalent to the previous pointer.

3. ```
   virtual attribute_value get_value();
   ```

   Returns: The attribute value that refers to self implementation.

4. ```
   virtual type_info_wrapper get_type() const;
   ```

   Returns: The attribute value type

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/attributes/attribute_value.hpp>


void swap(attribute_value & left, attribute_value & right);
```

Description

The function swaps two attribute values

Header <boost/log/attributes/attribute_value_impl.hpp>

Andrey Semashev

24.06.2007
The header contains an implementation of a basic attribute value implementation class.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename T> class attribute_value_impl;
      template<typename T> attribute_value make_attribute_value(T &&);
    }
  }
}
```

Class template attribute_value_impl

boost::log::attributes::attribute_value_impl — Basic attribute value implementation class.

Synopsis

```
// In header: <boost/log/attributes/attribute_value_impl.hpp>

template<typename T>
class attribute_value_impl : public attribute_value::impl {
public:
  // types
  typedef T value_type;  // Value type.

  // construct/copy/destruct
  explicit attribute_value_impl(value_type const &);
  explicit attribute_value_impl(value_type &&);

  // public member functions
  virtual bool dispatch(type_dispatcher &);
  virtual type_info_wrapper get_type() const;
  value_type const & get() const;
};
```

Description

This class can be used as a boilerplate for simple attribute values. The class implements all needed interfaces of attribute values and allows to store a single value of the type specified as a template parameter. The stored value can be dispatched with type dispatching mechanism.

`attribute_value_impl` public construct/copy/destruct

1.
```
explicit attribute_value_impl(value_type const & v);
```

Constructor with initialization of the stored value

2.
```
explicit attribute_value_impl(value_type && v);
```

Constructor with initialization of the stored value

`attribute_value_impl` public member functions

1.
```
virtual bool dispatch(type_dispatcher & dispatcher);
```

Attribute value dispatching method.

Parameters: `dispatcher` The dispatcher that receives the stored value
Returns: `true` if the value has been dispatched, `false` otherwise

2.
```
virtual type_info_wrapper get_type() const;
```

Returns: The attribute value type

3.
```
value_type const & get() const;
```

Returns: Reference to the contained value.

Function template make_attribute_value

boost::log::attributes::make_attribute_value

Synopsis

```
// In header: <boost/log/attributes/attribute_value_impl.hpp>


template<typename T> attribute_value make_attribute_value(T && v);
```

Description

The function creates an attribute value from the specified object.

Header <boost/log/attributes/attribute_value_set.hpp>

Andrey Semashev

21.04.2007
This header file contains definition of attribute value set. The set is constructed from three attribute sets (global, thread-specific and source-specific) and contains attribute values.

```
namespace boost {
  namespace log {
    class attribute_value_set;
    void swap(attribute_value_set &, attribute_value_set &);
  }
}
```

Class attribute_value_set

boost::log::attribute_value_set — A set of attribute values.

Synopsis

```
// In header: <boost/log/attributes/attribute_value_set.hpp>


class attribute_value_set {
public:
  // types
  typedef attribute_name                           key_type;         // Key type.
  typedef attribute_value                          mapped_type;      // Mapped attribute type.
  typedef std::pair< const key_type, mapped_type > value_type;       // Value type.
  typedef value_type &                             reference;        // Reference type.
  typedef value_type const &                       const_reference;  // Const reference type.
  typedef value_type *                             pointer;          // Pointer type.
  typedef value_type const *                       const_pointer;    // Const pointer type.
  typedef std::size_t                              size_type;        // Size type.
  typedef std::ptrdiff_t                           difference_type;  // Pointer difference type.
  typedef implementation_defined                   const_iterator;

  // construct/copy/destruct
  explicit attribute_value_set(size_type = 8);
  attribute_value_set(attribute_value_set &&) noexcept;
  attribute_value_set(attribute_set const &, attribute_set const &,
                      attribute_set const &, size_type = 8);
  attribute_value_set(attribute_value_set const &, attribute_set const &,
                      attribute_set const &, size_type = 8);
  attribute_value_set(attribute_value_set &&, attribute_set const &,
                      attribute_set const &, size_type = 8);
  attribute_value_set(attribute_value_set const &);
  attribute_value_set & operator=(attribute_value_set) noexcept;
  ~attribute_value_set();

  // public member functions
  void swap(attribute_value_set &) noexcept;
  const_iterator begin() const;
  const_iterator end() const;
  size_type size() const;
  bool empty() const;
  const_iterator find(key_type) const;
  mapped_type operator[](key_type) const;
  template<typename DescriptorT, template< typename > class ActorT>
    result_of::extract< typename expressions::attrib↵
ute_keyword< DescriptorT, ActorT >::value_type, DescriptorT >::type
    operator[](expressions::attribute_keyword< DescriptorT, ActorT > const &) const;
  size_type count(key_type) const;
  void freeze();
```

```
  std::pair< const_iterator, bool > insert(key_type, mapped_type const &);
  std::pair< const_iterator, bool > insert(const_reference);
  template<typename FwdIteratorT> void insert(FwdIteratorT, FwdIteratorT);
  template<typename FwdIteratorT, typename OutputIteratorT>
    void insert(FwdIteratorT, FwdIteratorT, OutputIteratorT);
};
```

Description

The set of attribute values is an associative container with attribute name as a key and a pointer to attribute value object as a mapped type. This is a collection of elements with unique keys, that is, there can be only one attribute value with a given name in the set. With respect to read-only capabilities, the set interface is close to `std::unordered_map`.

The set is designed to be only capable of adding elements to it. Once added, the attribute value cannot be removed from the set.

An instance of attribute value set can be constructed from three attribute sets. The constructor attempts to accommodate values of all attributes from the sets. The situation when a same-named attribute is found in more than one attribute set is possible. This problem is solved on construction of the value set: the three attribute sets have different priorities when it comes to solving conflicts.

From the library perspective the three source attribute sets are global, thread-specific and source-specific attributes, with the latter having the highest priority. This feature allows to override attributes of wider scopes with the more specific ones.

For sake of performance, the attribute values are not immediately acquired from attribute sets at construction. Instead, on-demand acquisition is performed either on iterator dereferencing or on call to the `freeze` method. Once acquired, the attribute value stays within the set until its destruction. This nuance does not affect other set properties, such as size or lookup ability. The logging core automatically freezes the set at the right point, so users should not be bothered unless they manually create attribute value sets.

Note

The attribute sets that were used for the value set construction must not be modified or destroyed until the value set is frozen. Otherwise the behavior is undefined.

`attribute_value_set` public types

1. typedef implementation_defined const_iterator;

   Constant iterator type with bidirectional capabilities.

`attribute_value_set` public construct/copy/destruct

1. ```
   explicit attribute_value_set(size_type reserve_count = 8);
   ```

   Default constructor

   The constructor creates an empty set which can be filled later by subsequent calls of `insert` method. Optionally, the amount of storage reserved for elements to be inserted may be passed to the constructor. The constructed set is frozen.

   Parameters: `reserve_count` Number of elements to reserve space for.

2. ```
   attribute_value_set(attribute_value_set && that) noexcept;
   ```

   Move constructor

3.
```
attribute_value_set(attribute_set const & source_attrs,
                    attribute_set const & thread_attrs,
                    attribute_set const & global_attrs,
                    size_type reserve_count = 8);
```

The constructor adopts three attribute sets into the value set. The *source_attrs* attributes have the greatest preference when a same-named attribute is found in several sets, *global_attrs* has the least. The constructed set is not frozen.

Parameters:	`global_attrs`	A set of global attributes.
	`reserve_count`	Amount of elements to reserve space for, in addition to the elements in the three attribute sets provided.
	`source_attrs`	A set of source-specific attributes.
	`thread_attrs`	A set of thread-specific attributes.

4.
```
attribute_value_set(attribute_value_set const & source_attrs,
                    attribute_set const & thread_attrs,
                    attribute_set const & global_attrs,
                    size_type reserve_count = 8);
```

The constructor adopts three attribute sets into the value set. The *source_attrs* attributes have the greatest preference when a same-named attribute is found in several sets, *global_attrs* has the least. The constructed set is not frozen.

Parameters:	`global_attrs`	A set of global attributes.
	`reserve_count`	Amount of elements to reserve space for, in addition to the elements in the three attribute sets provided.
	`source_attrs`	A set of source-specific attributes.
	`thread_attrs`	A set of thread-specific attributes.
Requires:	The *source_attrs* set is frozen.	

5.
```
attribute_value_set(attribute_value_set && source_attrs,
                    attribute_set const & thread_attrs,
                    attribute_set const & global_attrs,
                    size_type reserve_count = 8);
```

The constructor adopts three attribute sets into the value set. The *source_attrs* attributes have the greatest preference when a same-named attribute is found in several sets, *global_attrs* has the least. The constructed set is not frozen.

Parameters:	`global_attrs`	A set of global attributes.
	`reserve_count`	Amount of elements to reserve space for, in addition to the elements in the three attribute sets provided.
	`source_attrs`	A set of source-specific attributes.
	`thread_attrs`	A set of thread-specific attributes.
Requires:	The *source_attrs* set is frozen.	

6.
```
attribute_value_set(attribute_value_set const & that);
```

Copy constructor.

Requires:	The original set is frozen.
Postconditions:	The constructed set is frozen, `std::equal(begin(), end(), that.begin()) == true`

7.
```
attribute_value_set & operator=(attribute_value_set that) noexcept;
```

Assignment operator

8.
```
~attribute_value_set();
```

Destructor. Releases all referenced attribute values.

`attribute_value_set` public member functions

1.
```
void swap(attribute_value_set & that) noexcept;
```

Swaps two sets

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Returns: Iterator to the first element of the set.

3.
```
const_iterator end() const;
```

Returns: Iterator to the after-the-last element of the set.

4.
```
size_type size() const;
```

Returns: Number of elements in the set.

5.
```
bool empty() const;
```

Returns: true if there are no elements in the container, false otherwise.

6.
```
const_iterator find(key_type key) const;
```

The method finds the attribute value by name.

Parameters: `key` Attribute name.
Returns: Iterator to the found element or `end()` if the attribute with such name is not found.

7.
```
mapped_type operator[](key_type key) const;
```

Alternative lookup syntax.

Parameters: `key` Attribute name.
Returns: A pointer to the attribute value if it is found with *key*, default-constructed mapped value otherwise.

8.
```
template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename expressions::attribute_keyword< DescriptorT, ActorT
>::value_type, DescriptorT >::type
  operator[](expressions::attribute_keyword< DescriptorT, ActorT > const & keyword) const;
```

Alternative lookup syntax.

Parameters: `keyword` Attribute keyword.
Returns: A `value_ref` with extracted attribute value if it is found, empty `value_ref` otherwise.

9.
```
size_type count(key_type key) const;
```

The method counts the number of the attribute value occurrences in the set. Since there can be only one attribute value with a particular key, the method always return 0 or 1.

Parameters: `key` Attribute name.
Returns: The number of times the attribute value is found in the container.

10.
```
void freeze();
```

The method acquires values of all adopted attributes.

Postconditions: The set is frozen.

11.
```
std::pair< const_iterator, bool >
insert(key_type key, mapped_type const & mapped);
```

Inserts an element into the set. The complexity of the operation is amortized constant.

Parameters: `key` The attribute name.
`mapped` The attribute value.
Requires: The set is frozen.
Returns: An iterator to the inserted element and `true` if insertion succeeded. Otherwise, if the set already contains a same-named attribute value, iterator to the existing element and `false`.

12.
```
std::pair< const_iterator, bool > insert(const_reference value);
```

Inserts an element into the set. The complexity of the operation is amortized constant.

Parameters: `value` The attribute name and value.
Requires: The set is frozen.
Returns: An iterator to the inserted element and `true` if insertion succeeded. Otherwise, if the set already contains a same-named attribute value, iterator to the existing element and `false`.

13.
```
template<typename FwdIteratorT>
  void insert(FwdIteratorT begin, FwdIteratorT end);
```

Mass insertion method. The complexity of the operation is linear to the number of elements inserted.

Parameters: `begin` A forward iterator that points to the first element to be inserted.
`end` A forward iterator that points to the after-the-last element to be inserted.
Requires: The set is frozen.

14.
```
template<typename FwdIteratorT, typename OutputIteratorT>
  void insert(FwdIteratorT begin, FwdIteratorT end, OutputIteratorT out);
```

Mass insertion method with ability to acquire iterators to the inserted elements. The complexity of the operation is linear to the number of elements inserted times the complexity of filling the *out* iterator.

Parameters: `begin` A forward iterator that points to the first element to be inserted.
`end` A forward iterator that points to the after-the-last element to be inserted.
`out` An output iterator that receives results of insertion of the elements.
Requires: The set is frozen.

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/attributes/attribute_value_set.hpp>


void swap(attribute_value_set & left, attribute_value_set & right);
```

Description

Free swap overload

Header <boost/log/attributes/clock.hpp>

Andrey Semashev

01.12.2007
The header contains wall clock attribute implementation and typedefs.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename TimeTraitsT> class basic_clock;

      typedef basic_clock< utc_time_traits > utc_clock;  // Attribute that returns current UTC ↵
time.
      typedef basic_clock< local_time_traits > local_clock;  // Attribute that returns current ↵
local time.
    }
  }
}
```

Class template basic_clock

boost::log::attributes::basic_clock — A class of an attribute that makes an attribute value of the current date and time.

Synopsis

```
// In header: <boost/log/attributes/clock.hpp>

template<typename TimeTraitsT>
class basic_clock : public attribute {
public:
  // types
  typedef TimeTraitsT::time_type value_type;  // Generated value type.

  // member classes/structs/unions

  // Attribute factory implementation.

  struct impl : public attribute::impl {

    // public member functions
    virtual attribute_value get_value();
  };

  // construct/copy/destruct
  basic_clock();
  explicit basic_clock(cast_source const &);
};
```

Description

The attribute generates current time stamp as a value. The type of the attribute value is determined with time traits passed to the class template as a template parameter. The time traits provided by the library use `boost::posix_time::ptime` as the time type.

Time traits also determine the way time is acquired. There are two types of time traits provided by the library: `utc_time_traits` and `local_time_traits`. The first returns UTC time, the second returns local time.

`basic_clock` public construct/copy/destruct

1. ```
   basic_clock();
   ```

   Default constructor

2. ```
   explicit basic_clock(cast_source const & source);
   ```

   Constructor for casting support

Struct impl

boost::log::attributes::basic_clock::impl — Attribute factory implementation.

Synopsis

```
// In header: <boost/log/attributes/clock.hpp>



// Attribute factory implementation.

struct impl : public attribute::impl {

  // public member functions
  virtual attribute_value get_value();
};
```

Description

`impl` public member functions

1. ```
   virtual attribute_value get_value();
   ```

   Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

Header <boost/log/attributes/constant.hpp>

Andrey Semashev

15.04.2007
The header contains implementation of a constant attribute.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename T> class constant;
      template<typename T> unspecified make_constant(BOOST_FWD_REF(T));
    }
  }
}
```

Class template constant

boost::log::attributes::constant — A class of an attribute that holds a single constant value.
```

Synopsis

```
// In header: <boost/log/attributes/constant.hpp>

template<typename T>
class constant : public attribute {
public:
  // types
  typedef T value_type;  // Attribute value type.

  // member classes/structs/unions

  // Factory implementation.

  class impl : public attribute_value_impl< value_type > {
  public:
    // construct/copy/destruct
    explicit impl(value_type const &);
    explicit impl(value_type &&);
  };

  // construct/copy/destruct
  explicit constant(value_type const &);
  explicit constant(value_type &&);
  explicit constant(cast_source const &);

  // public member functions
  value_type const & get() const;
};
```

Description

The constant is a simplest and one of the most frequently used types of attributes. It stores a constant value, which it eventually returns as its value each time requested.

`constant` public construct/copy/destruct

1. ```
   explicit constant(value_type const & value);
   ```

   Constructor with the stored value initialization

2. ```
   explicit constant(value_type && value);
   ```

   Constructor with the stored value initialization

3. ```
   explicit constant(cast_source const & source);
   ```

   Constructor for casting support

`constant` public member functions

1. ```
   value_type const & get() const;
   ```

   Returns: Reference to the contained value.

Class impl

boost::log::attributes::constant::impl — Factory implementation.

Synopsis

```
// In header: <boost/log/attributes/constant.hpp>



// Factory implementation.

class impl : public attribute_value_impl< value_type > {
public:
  // construct/copy/destruct
  explicit impl(value_type const &);
  explicit impl(value_type &&);
};
```

Description

`impl` public construct/copy/destruct

1. ```
   explicit impl(value_type const & value);
   ```

   Constructor with the stored value initialization

2. ```
   explicit impl(value_type && value);
   ```

   Constructor with the stored value initialization

Function template make_constant

boost::log::attributes::make_constant

Synopsis

```
// In header: <boost/log/attributes/constant.hpp>


template<typename T> unspecified make_constant(BOOST_FWD_REF(T) val);
```

Description

The function constructs a `constant` attribute containing the provided value. The function automatically converts C string arguments to `std::basic_string` objects.

Header <boost/log/attributes/counter.hpp>

Andrey Semashev

01.05.2007
The header contains implementation of the counter attribute.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename T> class counter;
    }
  }
}
```

Class template counter

boost::log::attributes::counter — A class of an attribute that counts an integral value.

Synopsis

```
// In header: <boost/log/attributes/counter.hpp>

template<typename T>
class counter : public attribute {
public:
  // types
  typedef T value_type;  // A counter value type.

  // member classes/structs/unions

  // Base class for factory implementation.

  class impl : public attribute::impl {
  };

  class impl_dec : public counter< T >::impl {
  public:
    // construct/copy/destruct
    explicit impl_dec(value_type);

    // public member functions
    attribute_value get_value();
  };

  class impl_generic : public counter< T >::impl {
  public:
    // construct/copy/destruct
    impl_generic(value_type, long);

    // public member functions
    attribute_value get_value();
  };

  class impl_inc : public counter< T >::impl {
  public:
    // construct/copy/destruct
    explicit impl_inc(value_type);

    // public member functions
    attribute_value get_value();
  };

  // construct/copy/destruct
  explicit counter(value_type = (value_type) 0, long = 1);
  explicit counter(cast_source const &);
};
```

Description

This type of attribute acts as a counter, that is, it returns a monotonously changing value each time requested. The attribute value type can be specified as a template parameter. However, the type must be an integral type of size no more than `sizeof(long)`.

`counter` public construct/copy/destruct

1.
```
explicit counter(value_type initial = (value_type) 0, long step = 1);
```

Constructor

Parameters:

`initial`	Initial value of the counter
`step`	Changing step of the counter. Each value acquired from the attribute will be greater than the previous one to this amount.

2.
```
explicit counter(cast_source const & source);
```

Constructor for casting support

Class impl

boost::log::attributes::counter::impl — Base class for factory implementation.

Synopsis

```
// In header: <boost/log/attributes/counter.hpp>



// Base class for factory implementation.

class impl : public attribute::impl {
};
```

Class impl_dec

boost::log::attributes::counter::impl_dec

Synopsis

```
// In header: <boost/log/attributes/counter.hpp>



class impl_dec : public counter< T >::impl {
public:
  // construct/copy/destruct
  explicit impl_dec(value_type);

  // public member functions
  attribute_value get_value();
};
```

Description

`impl_dec` public construct/copy/destruct

1. ```
   explicit impl_dec(value_type initial);
   ```

   Initializing constructor

`impl_dec` public member functions

1. ```
   attribute_value get_value();
   ```

Class impl_generic

boost::log::attributes::counter::impl_generic

Synopsis

```
// In header: <boost/log/attributes/counter.hpp>



class impl_generic : public counter< T >::impl {
public:
  // construct/copy/destruct
  impl_generic(value_type, long);

  // public member functions
  attribute_value get_value();
};
```

Description

`impl_generic` public construct/copy/destruct

1. ```
   impl_generic(value_type initial, long step);
   ```

   Initializing constructor

`impl_generic` public member functions

1. ```
   attribute_value get_value();
   ```

Class impl_inc

boost::log::attributes::counter::impl_inc

Synopsis

```
// In header: <boost/log/attributes/counter.hpp>



class impl_inc : public counter< T >::impl {
public:
  // construct/copy/destruct
  explicit impl_inc(value_type);

  // public member functions
  attribute_value get_value();
};
```

Description

`impl_inc` public construct/copy/destruct

1. ```
   explicit impl_inc(value_type initial);
   ```

   Initializing constructor

`impl_inc` public member functions

1. ```
   attribute_value get_value();
   ```

Header <boost/log/attributes/current_process_id.hpp>

Andrey Semashev

12.09.2009
The header contains implementation of a current process id attribute

```
namespace boost {
  namespace log {
    typedef unspecified process_id;  // Process identifier type used by the library.
    namespace attributes {
      class current_process_id;
    }
  }
}
```

Class current_process_id

boost::log::attributes::current_process_id — A class of an attribute that holds the current process identifier.

Synopsis

```
// In header: <boost/log/attributes/current_process_id.hpp>


class current_process_id : public constant< process_id > {
public:
  // construct/copy/destruct
  current_process_id();
  explicit current_process_id(cast_source const &);
};
```

Description

`current_process_id` public construct/copy/destruct

1. ```
   current_process_id();
   ```

   Constructor. Initializes the attribute with the current process identifier.

2. ```
   explicit current_process_id(cast_source const & source);
   ```

   Constructor for casting support

Header <boost/log/attributes/current_process_name.hpp>

Andrey Semashev

29.07.2012
The header contains implementation of a current process name attribute

```
namespace boost {
  namespace log {
    namespace attributes {
      class current_process_name;
    }
  }
}
```

Class current_process_name

boost::log::attributes::current_process_name — A class of an attribute that holds the current process name.

Synopsis

```
// In header: <boost/log/attributes/current_process_name.hpp>


class current_process_name : public constant< std::string > {
public:
  // construct/copy/destruct
  current_process_name();
  explicit current_process_name(cast_source const &);
};
```

Description

`current_process_name` public construct/copy/destruct

1. ```
   current_process_name();
   ```

   Constructor. Initializes the attribute with the current process name.

2. ```
   explicit current_process_name(cast_source const & source);
   ```

   Constructor for casting support

Header <boost/log/attributes/current_thread_id.hpp>

Andrey Semashev

12.09.2009
The header contains implementation of a current thread id attribute

```
namespace boost {
  namespace log {
    typedef unspecified thread_id;  // Thread identifier type.
    namespace attributes {
      class current_thread_id;
    }
  }
}
```

Class current_thread_id

boost::log::attributes::current_thread_id — A class of an attribute that always returns the current thread identifier.

Synopsis

```
// In header: <boost/log/attributes/current_thread_id.hpp>


class current_thread_id : public attribute {
public:
  // types
  typedef thread_id value_type;  // A held attribute value type.

  // member classes/structs/unions

  // Factory implementation.

  class impl : public attribute_value::impl {
  public:

    // public member functions
    virtual bool dispatch(type_dispatcher &);
    virtual intrusive_ptr< attribute_value::impl > detach_from_thread();
    virtual type_info_wrapper get_type() const;
  };

  // construct/copy/destruct
  current_thread_id();
  explicit current_thread_id(cast_source const &);
};
```

Description

Note

This attribute can be registered globally, it will still return the correct thread identifier, no matter which thread emits the log record.

`current_thread_id` public construct/copy/destruct

1. ```
   current_thread_id();
   ```

   Default constructor

2. ```
   explicit current_thread_id(cast_source const & source);
   ```

   Constructor for casting support

Class impl

boost::log::attributes::current_thread_id::impl — Factory implementation.

Synopsis

```
// In header: <boost/log/attributes/current_thread_id.hpp>



// Factory implementation.

class impl : public attribute_value::impl {
public:

  // public member functions
  virtual bool dispatch(type_dispatcher &);
  virtual intrusive_ptr< attribute_value::impl > detach_from_thread();
  virtual type_info_wrapper get_type() const;
};
```

Description

`impl` public member functions

1. ```
   virtual bool dispatch(type_dispatcher & dispatcher);
   ```

   The method dispatches the value to the given object.

   Parameters: `dispatcher` The object that attempts to dispatch the stored value.
   Returns: true if *dispatcher* was capable to consume the real attribute value type and false otherwise.

2. ```
   virtual intrusive_ptr< attribute_value::impl > detach_from_thread();
   ```

   The method is called when the attribute value is passed to another thread (e.g. in case of asynchronous logging). The value should ensure it properly owns all thread-specific data.

   Returns: An actual pointer to the attribute value. It may either point to this object or another. In the latter case the returned pointer replaces the pointer used by caller to invoke this method and is considered to be a functional equivalent to the previous pointer.

3. ```
   virtual type_info_wrapper get_type() const;
   ```

   Returns: The attribute value type

Header <boost/log/attributes/fallback_policy.hpp>

Andrey Semashev

18.08.2012
The header contains definition of fallback policies when attribute value visitation or extraction fails.

```
namespace boost {
  namespace log {
    template<typename DefaultT> struct fallback_to_default;
    struct fallback_to_none;
    struct fallback_to_throw;
  }
}
```
```

Struct template fallback_to_default

boost::log::fallback_to_default

Synopsis

```
// In header: <boost/log/attributes/fallback_policy.hpp>

template<typename DefaultT>
struct fallback_to_default {
  // types
  typedef remove_cv< typename remove_reference< DefaultT >::type >::type default_type;  // De↵
fault value type.

  enum @2 { guaranteed_result = = true };

  // construct/copy/destruct
  fallback_to_default();
  explicit fallback_to_default(default_type const &);

  // public member functions
  template<typename FunT> bool apply_default(FunT &) const;
  template<typename FunT> bool apply_default(FunT const &) const;

  // public static functions
  static void on_invalid_type(type_info_wrapper const &);
  static void on_missing_value();
};
```

Description

The `fallback_to_default` policy results in a default value if the attribute value cannot be extracted.

`fallback_to_default` public construct/copy/destruct

1. ```
   fallback_to_default();
   ```

   Default constructor.

2. ```
   explicit fallback_to_default(default_type const & def_val);
   ```

   Initializing constructor.

`fallback_to_default` public member functions

1. ```
   template<typename FunT> bool apply_default(FunT & fun) const;
   ```

   The method is called in order to apply a function object to the default value.

2. ```
   template<typename FunT> bool apply_default(FunT const & fun) const;
   ```

   The method is called in order to apply a function object to the default value.

`fallback_to_default` public static functions

1. ```
   static void on_invalid_type(type_info_wrapper const &);
   ```
   ```

The method is called when value extraction failed because the attribute value has different type than requested.

2.
```
static void on_missing_value();
```

The method is called when value extraction failed because the attribute value was not found.

Struct fallback_to_none

boost::log::fallback_to_none

Synopsis

```
// In header: <boost/log/attributes/fallback_policy.hpp>


struct fallback_to_none {

  enum @0 { guaranteed_result = = false };

  // public static functions
  template<typename FunT> static bool apply_default(FunT &);
  template<typename FunT> static bool apply_default(FunT const &);
  static void on_invalid_type(type_info_wrapper const &);
  static void on_missing_value();
};
```

Description

The `fallback_to_none` policy results in returning an empty value reference if the attribute value cannot be extracted.

`fallback_to_none` public static functions

1.
```
template<typename FunT> static bool apply_default(FunT &);
```

The method is called in order to apply a function object to the default value.

2.
```
template<typename FunT> static bool apply_default(FunT const &);
```

The method is called in order to apply a function object to the default value.

3.
```
static void on_invalid_type(type_info_wrapper const &);
```

The method is called when value extraction failed because the attribute value has different type than requested.

4.
```
static void on_missing_value();
```

The method is called when value extraction failed because the attribute value was not found.

Struct fallback_to_throw

boost::log::fallback_to_throw

Synopsis

```
// In header: <boost/log/attributes/fallback_policy.hpp>


struct fallback_to_throw {

  enum @1 { guaranteed_result = = true };

  // public static functions
  template<typename FunT> static bool apply_default(FunT &);
  template<typename FunT> static bool apply_default(FunT const &);
  static void on_invalid_type(type_info_wrapper const &);
  static void on_missing_value();
};
```

Description

The `fallback_to_throw` policy results in throwing an exception if the attribute value cannot be extracted.

`fallback_to_throw` public static functions

1. ```
   template<typename FunT> static bool apply_default(FunT &);
   ```

   The method is called in order to apply a function object to the default value.

2. ```
   template<typename FunT> static bool apply_default(FunT const &);
   ```

   The method is called in order to apply a function object to the default value.

3. ```
   static void on_invalid_type(type_info_wrapper const & t);
   ```

   The method is called when value extraction failed because the attribute value has different type than requested.

4. ```
   static void on_missing_value();
   ```

   The method is called when value extraction failed because the attribute value was not found.

Header <boost/log/attributes/fallback_policy_fwd.hpp>

Andrey Semashev

18.08.2012
The header contains forward declaration of fallback policies when attribute value visitation or extraction fails.

Header <boost/log/attributes/function.hpp>

Andrey Semashev

24.06.2007
The header contains implementation of an attribute that calls a third-party function on value acquisition.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename R> class function;
      template<typename T>
        function< typename remove_cv< typename remove_reference< typename boost::res↵
ult_of< T() >::type >::type >::type >
        make_function(T const &);
    }
  }
}
```

Class template function

boost::log::attributes::function — A class of an attribute that acquires its value from a third-party function object.

Synopsis

```
// In header: <boost/log/attributes/function.hpp>

template<typename R>
class function : public attribute {
public:
  // types
  typedef R value_type;  // The attribute value type.

  // member classes/structs/unions

  // Base class for factory implementation.

  class impl : public attribute::impl {
  };

  // Factory implementation.
  template<typename T>
  class impl_template : public function< R >::impl {
  public:
    // construct/copy/destruct
    explicit impl_template(T const &);

    // public member functions
    virtual attribute_value get_value();
  };

  // construct/copy/destruct
  template<typename T> explicit function(T const &);
  explicit function(cast_source const &);
};
```

Description

The attribute calls a stored nullary function object to acquire each value. The result type of the function object is the attribute value type.

It is not recommended to use this class directly. Use `make_function` convenience functions to construct the attribute instead.

`function` public construct/copy/destruct

1. ```
   template<typename T> explicit function(T const & fun);
   ```
   ```

Initializing constructor

2.
```
explicit function(cast_source const & source);
```

Constructor for casting support

Class impl

boost::log::attributes::function::impl — Base class for factory implementation.

Synopsis

```
// In header: <boost/log/attributes/function.hpp>


// Base class for factory implementation.

class impl : public attribute::impl {
};
```

Class template impl_template

boost::log::attributes::function::impl_template — Factory implementation.

Synopsis

```
// In header: <boost/log/attributes/function.hpp>


// Factory implementation.
template<typename T>
class impl_template : public function< R >::impl {
public:
  // construct/copy/destruct
  explicit impl_template(T const &);

  // public member functions
  virtual attribute_value get_value();
};
```

Description

`impl_template` public construct/copy/destruct

1.
```
explicit impl_template(T const & fun);
```

Constructor with the stored delegate initialization

`impl_template` public member functions

1.
```
virtual attribute_value get_value();
```

Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

Function template make_function

boost::log::attributes::make_function

Synopsis

```
// In header: <boost/log/attributes/function.hpp>


template<typename T>
  function< typename remove_cv< typename remove_reference< typename boost::res↵
ult_of< T() >::type >::type >::type >
  make_function(T const & fun);
```

Description

The function constructs `function` attribute instance with the provided function object.

Parameters: `fun` Nullary functional object that returns an actual stored value for an attribute value.
Returns: Pointer to the attribute instance

Header <boost/log/attributes/mutable_constant.hpp>

Andrey Semashev

06.11.2007
The header contains implementation of a mutable constant attribute.

```
namespace boost {
  namespace log {
    namespace attributes {
      template<typename T, typename MutexT = void,
               typename ScopedWriteLockT = auto,
               typename ScopedReadLockT = auto>
        class mutable_constant;

      template<typename T> class mutable_constant<T, void, void, void>;
    }
  }
}
```

Class template mutable_constant

boost::log::attributes::mutable_constant — A class of an attribute that holds a single constant value with ability to change it.

Synopsis

```
// In header: <boost/log/attributes/mutable_constant.hpp>

template<typename T, typename MutexT = void, typename ScopedWriteLockT = auto,
         typename ScopedReadLockT = auto>
class mutable_constant : public attribute {
public:
  // types
  typedef T value_type;  // The attribute value type.

  // member classes/structs/unions

  // Factory implementation.

  class impl : public attribute::impl {
  public:
    // construct/copy/destruct
    explicit impl(value_type const &);
    explicit impl(value_type &&);

    // public member functions
    virtual attribute_value get_value();
    void set(value_type const &);
    void set(value_type &&);
    value_type get() const;
  };

  // construct/copy/destruct
  explicit mutable_constant(value_type const &);
  explicit mutable_constant(value_type &&);
  explicit mutable_constant(cast_source const &);

  // public member functions
  void set(value_type const &);
  void set(value_type &&);
  value_type get() const;

  // protected member functions
  impl * get_impl() const;
};
```

Description

The mutable_constant attribute stores a single value of type, specified as the first template argument. This value is returned on each attribute value acquisition.

The attribute also allows to modify the stored value, even if the attribute is registered in an attribute set. In order to ensure thread safety of such modifications the `mutable_constant` class is also parametrized with three additional template arguments: mutex type, scoped write and scoped read lock types. If not specified, the lock types are automatically deduced based on the mutex type.

The implementation may avoid using these types to actually create and use the mutex, if a more efficient synchronization method is available (such as atomic operations on the value type). By default no synchronization is done.

`mutable_constant` public construct/copy/destruct

1. ```
   explicit mutable_constant(value_type const & value);
   ```

   Constructor with the stored value initialization
   ```

2. ```
   explicit mutable_constant(value_type && value);
   ```

   Constructor with the stored value initialization

3. ```
   explicit mutable_constant(cast_source const & source);
   ```

   Constructor for casting support

`mutable_constant` public member functions

1. ```
   void set(value_type const & value);
   ```

   The method sets a new attribute value. The implementation exclusively locks the mutex in order to protect the value assignment.

2. ```
   void set(value_type && value);
   ```

   The method sets a new attribute value.

3. ```
   value_type get() const;
   ```

   The method acquires the current attribute value. The implementation non-exclusively locks the mutex in order to protect the value acquisition.

`mutable_constant` protected member functions

1. ```
   impl * get_impl() const;
   ```

   Returns: Pointer to the factory implementation

Class impl

boost::log::attributes::mutable_constant::impl — Factory implementation.

Synopsis

```
// In header: <boost/log/attributes/mutable_constant.hpp>


// Factory implementation.

class impl : public attribute::impl {
public:
  // construct/copy/destruct
  explicit impl(value_type const &);
  explicit impl(value_type &&);

  // public member functions
  virtual attribute_value get_value();
  void set(value_type const &);
  void set(value_type &&);
  value_type get() const;
};
```

Description

`impl` public construct/copy/destruct

1. ```
   explicit impl(value_type const & value);
   ```

   Initializing constructor

2. ```
   explicit impl(value_type && value);
   ```

   Initializing constructor

`impl` public member functions

1. ```
   virtual attribute_value get_value();
   ```

   Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

2. ```
   void set(value_type const & value);
   ```

3. ```
   void set(value_type && value);
   ```

4. ```
   value_type get() const;
   ```

Specializations

- Class template mutable_constant<T, void, void, void>

Class template mutable_constant<T, void, void, void>

boost::log::attributes::mutable_constant<T, void, void, void> — Specialization for unlocked case.

Synopsis

```
// In header: <boost/log/attributes/mutable_constant.hpp>

template<typename T>
class mutable_constant<T, void, void, void> : public attribute {
public:
  // types
  typedef T value_type;  // The attribute value type.

  // member classes/structs/unions

  // Factory implementation.

  class impl : public attribute::impl {
  public:
    // construct/copy/destruct
    explicit impl(value_type const &);
    explicit impl(value_type &&);

    // public member functions
    virtual attribute_value get_value();
    void set(value_type const &);
    void set(value_type &&);
    value_type get() const;
  };

  // construct/copy/destruct
  explicit mutable_constant(value_type const &);
  explicit mutable_constant(value_type &&);
  explicit mutable_constant(cast_source const &);

  // public member functions
  void set(value_type const &);
  void set(value_type &&);
  value_type get() const;

  // protected member functions
  impl * get_impl() const;
};
```

Description

This version of attribute does not perform thread synchronization to access the stored value.

`mutable_constant` public construct/copy/destruct

1. ```
   explicit mutable_constant(value_type const & value);
   ```

   Constructor with the stored value initialization

2. ```
   explicit mutable_constant(value_type && value);
   ```

   Constructor with the stored value initialization

3. ```
   explicit mutable_constant(cast_source const & source);
   ```

   Constructor for casting support
   ```

`mutable_constant` public member functions

1.
```
void set(value_type const & value);
```

The method sets a new attribute value.

2.
```
void set(value_type && value);
```

The method sets a new attribute value.

3.
```
value_type get() const;
```

The method acquires the current attribute value.

`mutable_constant` protected member functions

1.
```
impl * get_impl() const;
```

Returns: Pointer to the factory implementation

Class impl

boost::log::attributes::mutable_constant<T, void, void, void>::impl — Factory implementation.

Synopsis

```
// In header: <boost/log/attributes/mutable_constant.hpp>


// Factory implementation.

class impl : public attribute::impl {
public:
  // construct/copy/destruct
  explicit impl(value_type const &);
  explicit impl(value_type &&);

  // public member functions
  virtual attribute_value get_value();
  void set(value_type const &);
  void set(value_type &&);
  value_type get() const;
};
```

Description

`impl` public construct/copy/destruct

1.
```
explicit impl(value_type const & value);
```

Initializing constructor

2.
```
explicit impl(value_type && value);
```

Initializing constructor

`impl` public member functions

1.
```
virtual attribute_value get_value();
```

Returns: The actual attribute value. It shall not return empty values (exceptions shall be used to indicate errors).

2.
```
void set(value_type const & value);
```

3.
```
void set(value_type && value);
```

4.
```
value_type get() const;
```

Header <boost/log/attributes/named_scope.hpp>

Andrey Semashev

24.06.2007
The header contains implementation of named scope container and an attribute that allows to put the named scope to log. A number of convenience macros are also provided.

```
BOOST_LOG_NAMED_SCOPE(name)
BOOST_LOG_FUNCTION()
BOOST_LOG_FUNC()
```

```
namespace boost {
  namespace log {
    namespace attributes {
      class named_scope;

      struct named_scope_entry;

      class named_scope_list;
      template<typename CharT, typename TraitsT>
        std::basic_ostream< CharT, TraitsT > &
        operator<<(std::basic_ostream< CharT, TraitsT > &,
                   named_scope_list const &);
    }
  }
}
```

Class named_scope

boost::log::attributes::named_scope — A class of an attribute that holds stack of named scopes of the current thread.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


class named_scope : public attribute {
public:
  // types
  typedef named_scope_list       value_type;   // Scope names stack (the attribute value type)
  typedef value_type::value_type scope_entry;  // Scope entry.

  // member classes/structs/unions

  // Sentry object class to automatically push and pop scopes.

  struct sentry {
    // construct/copy/destruct
    sentry(string_literal const &, string_literal const &, unsigned int,
           scope_entry::scope_name_type = scope_entry::general) noexcept;
    sentry(sentry const &) = delete;
    sentry & operator=(sentry const &) = delete;
    ~sentry();
  };

  // construct/copy/destruct
  named_scope();
  explicit named_scope(cast_source const &);

  // public static functions
  static void push_scope(scope_entry const &) noexcept;
  static void pop_scope() noexcept;
  static value_type const & get_scopes();
};
```

Description

The basic_named_scope attribute is essentially a hook to the thread-specific instance of scope list. This means that the attribute will generate different values if get_value is called in different threads. The attribute generates value with stored type `basic_named_scope_list< CharT >`.

The attribute class can also be used to gain access to the scope stack instance, e.g. to get its copy or to push or pop a scope entry. However, it is highly not recommended to maintain scope list manually. Use `BOOST_LOG_NAMED_SCOPE` or `BOOST_LOG_FUNCTION` macros instead.

named_scope public construct/copy/destruct

1. ```
   named_scope();
   ```

   Constructor. Creates an attribute.

2. ```
   explicit named_scope(cast_source const & source);
   ```

   Constructor for casting support

named_scope public static functions

1. ```
   static void push_scope(scope_entry const & entry) noexcept;
   ```
   ```

The method pushes the scope to the back of the current thread's scope list

Throws: Nothing.

2.
```
static void pop_scope() noexcept;
```

The method pops the last pushed scope from the current thread's scope list

Throws: Nothing.

3.
```
static value_type const & get_scopes();
```

Note

The returned reference is only valid until the current thread ends. The scopes in the returned container may change if the execution scope is changed (i.e. either `push_scope` or `pop_scope` is called). User has to copy the stack if he wants to keep it intact regardless of the execution scope.

Returns: The current thread's list of scopes

Struct sentry

boost::log::attributes::named_scope::sentry — Sentry object class to automatically push and pop scopes.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


// Sentry object class to automatically push and pop scopes.

struct sentry {
  // construct/copy/destruct
  sentry(string_literal const &, string_literal const &, unsigned int,
         scope_entry::scope_name_type = scope_entry::general) noexcept;
  sentry(sentry const &) = delete;
  sentry & operator=(sentry const &) = delete;
  ~sentry();
};
```

Description

`sentry` public construct/copy/destruct

1.
```
sentry(string_literal const & sn, string_literal const & fn, unsigned int ln,
       scope_entry::scope_name_type t = scope_entry::general) noexcept;
```

Constructor. Pushes the specified scope to the end of the thread-local list of scopes.

Parameters:

`fn`	File name, in which the scope is located.
`ln`	Line number in the file.
`sn`	Scope name.

2. ```
   sentry(sentry const &) = delete;
   ```

3. ```
   sentry & operator=(sentry const &) = delete;
   ```

4. ```
   ~sentry();
   ```

   Destructor. Removes the last pushed scope from the thread-local list of scopes.

Struct named_scope_entry

boost::log::attributes::named_scope_entry — The structure contains all information about a named scope.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


struct named_scope_entry {
  enum scope_name_type;
  // construct/copy/destruct
  named_scope_entry(string_literal const &, string_literal const &,
                    unsigned int, scope_name_type = general) noexcept;

  // public data members
  string_literal scope_name;
  string_literal file_name;
  unsigned int line;
  scope_name_type type;
};
```

Description

The named scope entries are stored as elements of `basic_named_scope_list` container, which in turn can be acquired either from the `basic_named_scope` attribute value or from a thread-local instance.

`named_scope_entry` public construct/copy/destruct

1. ```
   named_scope_entry(string_literal const & sn, string_literal const & fn,
                     unsigned int ln, scope_name_type t = general) noexcept;
   ```

   Initializing constructor

   Throws: Nothing.
   Postconditions: `scope_name == sn && file_name == fn && line == ln`

`named_scope_entry` public public data members

1. ```
   string_literal scope_name;
   ```

   The scope name (e.g. a function signature)

2. ```
   string_literal file_name;
   ```

The source file name

3. ```
   unsigned int line;
   ```

   The line number in the source file

4. ```
   scope_name_type type;
   ```

   The scope name type

Type scope_name_type

boost::log::attributes::named_scope_entry::scope_name_type — Scope entry type.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


enum scope_name_type { general, function };
```

Description

Describes scope name specifics

`general`	The scope name contains some unstructured string that should not be interpreted by the library.
`function`	The scope name contains a function signature.

Class named_scope_list

boost::log::attributes::named_scope_list — The class implements the list of scopes.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


class named_scope_list {
public:
  // types
  typedef std::allocator< named_scope_entry > allocator_type;          // Allocator type.
  typedef allocator_type::value_type          value_type;
  typedef allocator_type::reference           reference;
  typedef allocator_type::const_reference     const_reference;
  typedef allocator_type::pointer             pointer;
  typedef allocator_type::const_pointer       const_pointer;
  typedef allocator_type::size_type           size_type;
  typedef allocator_type::difference_type     difference_type;
  typedef implementation_defined              const_iterator;
  typedef implementation_defined              iterator;
  typedef implementation_defined              const_reverse_iterator;
  typedef implementation_defined              reverse_iterator;

  // construct/copy/destruct
  named_scope_list();
  named_scope_list(named_scope_list const &);
  named_scope_list & operator=(named_scope_list const &);
  ~named_scope_list();

  // public member functions
  const_iterator begin() const;
  const_iterator end() const;
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  size_type size() const;
  bool empty() const;
  void swap(named_scope_list &);
  const_reference back() const;
  const_reference front() const;
};
```

Description

The scope list provides a read-only access to a doubly-linked list of scopes.

`named_scope_list` public types

1. typedef implementation_defined const_iterator;

   A constant iterator to the sequence of scopes. Complies to bidirectional iterator requirements.

2. typedef implementation_defined iterator;

   An iterator to the sequence of scopes. Complies to bidirectional iterator requirements.

3. typedef implementation_defined const_reverse_iterator;

   A constant reverse iterator to the sequence of scopes. Complies to bidirectional iterator requirements.

4. typedef implementation_defined reverse_iterator;

   A reverse iterator to the sequence of scopes. Complies to bidirectional iterator requirements.

`named_scope_list` public construct/copy/destruct

1. ```
   named_scope_list();
   ```

   Default constructor

   Postconditions: `empty() == true`

2. ```
   named_scope_list(named_scope_list const & that);
   ```

   Copy constructor

   Postconditions: `std::equal(begin(), end(), that.begin()) == true`

3. ```
   named_scope_list & operator=(named_scope_list const & that);
   ```

   Assignment operator

   Postconditions: `std::equal(begin(), end(), that.begin()) == true`

4. ```
   ~named_scope_list();
   ```

   Destructor. Destroys the stored entries.

`named_scope_list` public member functions

1. ```
   const_iterator begin() const;
   ```

   Returns: Constant iterator to the first element of the container.

2. ```
   const_iterator end() const;
   ```

   Returns: Constant iterator to the after-the-last element of the container.

3. ```
   const_reverse_iterator rbegin() const;
   ```

   Returns: Constant iterator to the last element of the container.

4. ```
   const_reverse_iterator rend() const;
   ```

   Returns: Constant iterator to the before-the-first element of the container.

5. ```
   size_type size() const;
   ```

   Returns: The number of elements in the container

6. ```
   bool empty() const;
   ```

   Returns: true if the container is empty and false otherwise

7. ```
   void swap(named_scope_list & that);
   ```
   ```

Swaps two instances of the container

8.
```
const_reference back() const;
```

Returns: Last pushed scope entry

9.
```
const_reference front() const;
```

Returns: First pushed scope entry

Function template operator<<

boost::log::attributes::operator<< — Stream output operator.

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>


template<typename CharT, typename TraitsT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             named_scope_list const & sl);
```

Macro BOOST_LOG_NAMED_SCOPE

BOOST_LOG_NAMED_SCOPE

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>

BOOST_LOG_NAMED_SCOPE(name)
```

Description

Macro for scope markup. The specified scope name is pushed to the end of the current thread scope list.

Macro BOOST_LOG_FUNCTION

BOOST_LOG_FUNCTION

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>

BOOST_LOG_FUNCTION()
```

Description

Macro for function scope markup. The scope name is constructed with help of compiler and contains the current function signature. The scope name is pushed to the end of the current thread scope list.

Not all compilers have support for this macro. The exact form of the scope name may vary from one compiler to another.

Macro BOOST_LOG_FUNC

BOOST_LOG_FUNC

Synopsis

```
// In header: <boost/log/attributes/named_scope.hpp>

BOOST_LOG_FUNC()
```

Description

Macro for function scope markup. The scope name is constructed with help of compiler and contains the current function name. It may be shorter than what `BOOST_LOG_FUNCTION` macro produces. The scope name is pushed to the end of the current thread scope list.

Not all compilers have support for this macro. The exact form of the scope name may vary from one compiler to another.

Header <boost/log/attributes/scoped_attribute.hpp>

Andrey Semashev

13.05.2007
The header contains definition of facilities to define scoped attributes.

```


BOOST_LOG_SCOPED_LOGGER_ATTR(logger, attr_name, attr)
BOOST_LOG_SCOPED_LOGGER_TAG(logger, attr_name, attr_value)
BOOST_LOG_SCOPED_THREAD_ATTR(attr_name, attr)
BOOST_LOG_SCOPED_THREAD_TAG(attr_name, attr_value)
```

```
namespace boost {
  namespace log {
    typedef unspecified scoped_attribute;  // Scoped attribute guard type.
    template<typename LoggerT>
      unspecified add_scoped_logger_attribute(LoggerT &,
                                              attribute_name const &,
                                              attribute const &);
    unspecified add_scoped_thread_attribute(attribute_name const &,
                                            attribute const &);
  }
}
```

Function template add_scoped_logger_attribute

boost::log::add_scoped_logger_attribute

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>


template<typename LoggerT>
  unspecified add_scoped_logger_attribute(LoggerT & l,
                                          attribute_name const & name,
                                          attribute const & attr);
```

Description

Registers an attribute in the logger

Parameters: `attr` The attribute. Must not be NULL.
`l` Logger to register the attribute in
`name` Attribute name
Returns: An unspecified guard object which may be used to initialize a `scoped_attribute` variable.

Function add_scoped_thread_attribute

boost::log::add_scoped_thread_attribute

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>


unspecified add_scoped_thread_attribute(attribute_name const & name,
                                        attribute const & attr);
```

Description

Registers a thread-specific attribute

Parameters: `attr` The attribute. Must not be NULL.
`name` Attribute name
Returns: An unspecified guard object which may be used to initialize a `scoped_attribute` variable.

Macro BOOST_LOG_SCOPED_LOGGER_ATTR

BOOST_LOG_SCOPED_LOGGER_ATTR — The macro sets a scoped logger-wide attribute in a more compact way.

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>

BOOST_LOG_SCOPED_LOGGER_ATTR(logger, attr_name, attr)
```

Macro BOOST_LOG_SCOPED_LOGGER_TAG

BOOST_LOG_SCOPED_LOGGER_TAG — The macro sets a scoped logger-wide tag in a more compact way.

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>

BOOST_LOG_SCOPED_LOGGER_TAG(logger, attr_name, attr_value)
```

Macro BOOST_LOG_SCOPED_THREAD_ATTR

BOOST_LOG_SCOPED_THREAD_ATTR — The macro sets a scoped thread-wide attribute in a more compact way.

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>

BOOST_LOG_SCOPED_THREAD_ATTR(attr_name, attr)
```

Macro BOOST_LOG_SCOPED_THREAD_TAG

BOOST_LOG_SCOPED_THREAD_TAG — The macro sets a scoped thread-wide tag in a more compact way.

Synopsis

```
// In header: <boost/log/attributes/scoped_attribute.hpp>

BOOST_LOG_SCOPED_THREAD_TAG(attr_name, attr_value)
```

Header <boost/log/attributes/time_traits.hpp>

Andrey Semashev

01.12.2007
The header contains implementation of time traits that are used in various parts of the library to acquire current time.

```
namespace boost {
  namespace log {
    namespace attributes {
      struct basic_time_traits;
      struct local_time_traits;
      struct utc_time_traits;
    }
  }
}
```

Struct basic_time_traits

boost::log::attributes::basic_time_traits — Base class for time traits involving Boost.DateTime.

Synopsis

```
// In header: <boost/log/attributes/time_traits.hpp>


struct basic_time_traits {
  // types
  typedef posix_time::ptime        time_type;     // Time type.
  typedef posix_time::second_clock clock_source;  // Current time source.
};
```

Struct local_time_traits

boost::log::attributes::local_time_traits — Time traits that describes local time acquirement via Boost.DateTime facilities.

Synopsis

```
// In header: <boost/log/attributes/time_traits.hpp>


struct local_time_traits : public basic_time_traits {

  // public static functions
  static time_type get_clock();
};
```

Description

`local_time_traits` public static functions

1. ```
   static time_type get_clock();
   ```

   Returns: Current time stamp

Struct utc_time_traits

boost::log::attributes::utc_time_traits — Time traits that describes UTC time acquirement via Boost.DateTime facilities.

Synopsis

```
// In header: <boost/log/attributes/time_traits.hpp>


struct utc_time_traits : public basic_time_traits {

  // public static functions
  static time_type get_clock();
};
```

Description

`utc_time_traits` public static functions

1. ```
   static time_type get_clock();
   ```

Returns: Current time stamp

Header <boost/log/attributes/timer.hpp>

Andrey Semashev

02.12.2007
The header contains implementation of a stop watch attribute.

```
namespace boost {
  namespace log {
    namespace attributes {
      class timer;
    }
  }
}
```

Class timer

boost::log::attributes::timer — A class of an attribute that makes an attribute value of the time interval since construction.

Synopsis

```
// In header: <boost/log/attributes/timer.hpp>


class timer : public attribute {
public:
  // types
  typedef utc_time_traits::time_type::time_duration_type value_type;  // Attribute value type.

  // construct/copy/destruct
  timer();
  explicit timer(cast_source const &);
};
```

Description

The timer attribute calculates the time passed since its construction and returns it on value acquisition. The attribute value type is `boost::posix_time::time_duration`.

On Windows platform there are two implementations of the attribute. The default one is more precise but a bit slower. This version uses `QueryPerformanceFrequence`/`QueryPerformanceCounter` API to calculate elapsed time.

There are known problems with these functions when used with some CPUs, notably AMD Athlon with Cool'n'Quiet technology enabled. See the following links for more information and possible resolutions:

http://support.microsoft.com/?scid=kb;en-us;895980 http://support.microsoft.com/?id=896256

In case if none of these solutions apply, you are free to define `BOOST_LOG_NO_QUERY_PERFORMANCE_COUNTER` macro to fall back to another implementation based on Boost.DateTime.

`timer` public construct/copy/destruct

1. ```
   timer();
   ```

   Constructor. Starts time counting.
   ```

2.
```
explicit timer(cast_source const & source);
```

Constructor for casting support

Header <boost/log/attributes/value_extraction.hpp>

Andrey Semashev

01.03.2008

The header contains implementation of tools for extracting an attribute value from the view.

```
namespace boost {
  namespace log {
    template<typename T, typename FallbackPolicyT, typename TagT>
      class value_extractor;
    template<typename T, typename TagT = void>
      result_of::extract< T, TagT >::type
      extract(attribute_name const &, attribute_value_set const &);
    template<typename T, typename TagT = void>
      result_of::extract< T, TagT >::type
      extract(attribute_name const &, record const &);
    template<typename T, typename TagT = void>
      result_of::extract< T, TagT >::type
      extract(attribute_name const &, record_view const &);
    template<typename T, typename TagT = void>
      result_of::extract< T, TagT >::type extract(attribute_value const &);
    template<typename T, typename TagT = void>
      result_of::extract_or_throw< T, TagT >::type
      extract_or_throw(attribute_name const &, attribute_value_set const &);
    template<typename T, typename TagT = void>
      result_of::extract_or_throw< T, TagT >::type
      extract_or_throw(attribute_name const &, record const &);
    template<typename T, typename TagT = void>
      result_of::extract_or_throw< T, TagT >::type
      extract_or_throw(attribute_name const &, record_view const &);
    template<typename T, typename TagT = void>
      result_of::extract_or_throw< T, TagT >::type
      extract_or_throw(attribute_value const &);
    template<typename T, typename TagT = void, typename DefaultT>
      result_of::extract_or_default< T, DefaultT, TagT >::type
      extract_or_default(attribute_name const &, attribute_value_set const &,
                         DefaultT const &);
    template<typename T, typename TagT = void, typename DefaultT>
      result_of::extract_or_default< T, DefaultT, TagT >::type
      extract_or_default(attribute_name const &, record const &,
                         DefaultT const &);
    template<typename T, typename TagT = void, typename DefaultT>
      result_of::extract_or_default< T, DefaultT, TagT >::type
      extract_or_default(attribute_name const &, record_view const &,
                         DefaultT const &);
    template<typename T, typename TagT = void, typename DefaultT>
      result_of::extract_or_default< T, DefaultT, TagT >::type
      extract_or_default(attribute_value const &, DefaultT const &);
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
      extract(expressions::attribute_keyword< DescriptorT, ActorT > const &,
              attribute_value_set const &);
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
      extract(expressions::attribute_keyword< DescriptorT, ActorT > const &,
              record const &);
```

```
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
      extract(expressions::attribute_keyword< DescriptorT, ActorT > const &,
              record_view const &);
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
      extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                       attribute_value_set const &);
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
      extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                       record const &);
    template<typename DescriptorT, template< typename > class ActorT>
      result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
      extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                       record_view const &);
    template<typename DescriptorT, template< typename > class ActorT,
             typename DefaultT>
      result_of::extract_or_default< typename DescriptorT::value_type, De↵
faultT, DescriptorT >::type
      extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                         attribute_value_set const &, DefaultT const &);
    template<typename DescriptorT, template< typename > class ActorT,
             typename DefaultT>
      result_of::extract_or_default< typename DescriptorT::value_type, De↵
faultT, DescriptorT >::type
      extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                         record const &, DefaultT const &);
    template<typename DescriptorT, template< typename > class ActorT,
             typename DefaultT>
      result_of::extract_or_default< typename DescriptorT::value_type, De↵
faultT, DescriptorT >::type
      extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                         record_view const &, DefaultT const &);
    namespace result_of {
      template<typename T, typename TagT> struct extract;
      template<typename T, typename DefaultT, typename TagT>
        struct extract_or_default;
      template<typename T, typename TagT> struct extract_or_throw;
    }
  }
}
```

Struct template extract

boost::log::result_of::extract — A metafunction that allows to acquire the result of the value extraction.

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>

template<typename T, typename TagT>
struct extract {
  // types
  typedef value_ref< T, TagT > type;
};
```

Description

The metafunction results in a type that is in form of `value_ref< T, TagT >`.

Struct template extract_or_default

boost::log::result_of::extract_or_default — A metafunction that allows to acquire the result of the value extraction.

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>

template<typename T, typename DefaultT, typename TagT>
struct extract_or_default {
  // types
  typedef mpl::eval_if< mpl::is_sequence< T >, mpl::eval_if< mpl::contains< T, De↵
faultT >, mpl::identity< T >, mpl::push_back< T, DefaultT > >, mpl::if_< is_same< T, De↵
faultT >, T, mpl::vector2< T, DefaultT > > >::type extracted_type;
  typedef mpl::if_< mpl::is_sequence< extracted_type >, value_ref< extracted_type, TagT >, ex↵
tracted_type const & >::type                                                    ↵
                              type;
};
```

Description

The metafunction results in a type that is in form of `T const&`, if `T` is not an MPL type sequence and `DefaultT` is the same as `T`, or `value_ref< TypesT, TagT >` otherwise, with `TypesT` being a type sequence comprising the types from sequence `T` and `DefaultT`, if it is not present in `T` already.

Struct template extract_or_throw

boost::log::result_of::extract_or_throw — A metafunction that allows to acquire the result of the value extraction.

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>

template<typename T, typename TagT>
struct extract_or_throw {
  // types
  typedef mpl::if_< mpl::is_sequence< T >, value_ref< T, TagT >, T const & >::type type;
};
```

Description

The metafunction results in a type that is in form of `T const&`, if `T` is not an MPL type sequence, or `value_ref< T, TagT >` otherwise. In the latter case the value reference shall never be empty.

Class template value_extractor

boost::log::value_extractor — Generic attribute value extractor.

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>

template<typename T, typename FallbackPolicyT, typename TagT>
class value_extractor : private FallbackPolicyT {
public:
  // types
  typedef FallbackPolicyT              fallback_policy;  // Fallback policy.
  typedef T                            value_type;       // Attribute value types.
  typedef value_ref< value_type, TagT > result_type;     // Function object result type.

  // construct/copy/destruct
  value_extractor() = default;
  value_extractor(value_extractor const &);
  template<typename U> explicit value_extractor(U const &);

  // public member functions
  result_type operator()(attribute_value const &) const;
  result_type operator()(attribute_name const &, attribute_value_set const &) const;
  result_type operator()(attribute_name const &, record const &) const;
  result_type operator()(attribute_name const &, record_view const &) const;
  fallback_policy const & get_fallback_policy() const;
};
```

Description

Attribute value extractor is a functional object that attempts to find and extract the stored attribute value from the attribute values view or a log record. The extracted value is returned from the extractor.

`value_extractor` public construct/copy/destruct

1. ```
   value_extractor() = default;
   ```

   Default constructor

2. ```
   value_extractor(value_extractor const & that);
   ```

   Copy constructor

3. ```
   template<typename U> explicit value_extractor(U const & arg);
   ```

   Constructor

   Parameters: `arg` Fallback policy constructor argument

`value_extractor` public member functions

1. ```
   result_type operator()(attribute_value const & attr) const;
   ```

   Extraction operator. Attempts to acquire the stored value of one of the supported types. If extraction succeeds, the extracted value is returned.

   Parameters: `attr` The attribute value to extract from.
   Returns: The extracted value, if extraction succeeded, an empty value otherwise.

2.
```
result_type operator()(attribute_name const & name,
                       attribute_value_set const & attrs) const;
```

Extraction operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If extraction succeeds, the extracted value is returned.

Parameters: `attrs` A set of attribute values in which to look for the specified attribute value.
`name` Attribute value name.
Returns: The extracted value, if extraction succeeded, an empty value otherwise.

3.
```
result_type operator()(attribute_name const & name, record const & rec) const;
```

Extraction operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If extraction succeeds, the extracted value is returned.

Parameters: `name` Attribute value name.
`rec` A log record. The attribute value will be sought among those associated with the record.
Returns: The extracted value, if extraction succeeded, an empty value otherwise.

4.
```
result_type operator()(attribute_name const & name, record_view const & rec) const;
```

Extraction operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If extraction succeeds, the extracted value is returned.

Parameters: `name` Attribute value name.
`rec` A log record view. The attribute value will be sought among those associated with the record.
Returns: The extracted value, if extraction succeeded, an empty value otherwise.

5.
```
fallback_policy const & get_fallback_policy() const;
```

Returns: Fallback policy

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract< T, TagT >::type
  extract(attribute_name const & name, attribute_value_set const & attrs);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `attrs` A set of attribute values in which to look for the specified attribute value.
`name` The name of the attribute value to extract.
Returns: A `value_ref` that refers to the extracted value, if found. An empty value otherwise.

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract< T, TagT >::type
  extract(attribute_name const & name, record const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`name`	The name of the attribute value to extract.
	`rec`	A log record. The attribute value will be sought among those associated with the record.
Returns:	A `value_ref` that refers to the extracted value, if found. An empty value otherwise.	

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract< T, TagT >::type
  extract(attribute_name const & name, record_view const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`name`	The name of the attribute value to extract.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
Returns:	A `value_ref` that refers to the extracted value, if found. An empty value otherwise.	

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract< T, TagT >::type extract(attribute_value const & value);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `value` Attribute value.
Returns: A `value_ref` that refers to the extracted value, if found. An empty value otherwise.

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract_or_throw< T, TagT >::type
  extract_or_throw(attribute_name const & name,
                   attribute_value_set const & attrs);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `attrs` A set of attribute values in which to look for the specified attribute value.
`name` The name of the attribute value to extract.
Returns: The extracted value or a non-empty `value_ref` that refers to the value.
Throws: An exception is thrown if the requested value cannot be extracted.

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract_or_throw< T, TagT >::type
  extract_or_throw(attribute_name const & name, record const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `name` The name of the attribute value to extract.
`rec` A log record. The attribute value will be sought among those associated with the record.
Returns: The extracted value or a non-empty `value_ref` that refers to the value.
Throws: An exception is thrown if the requested value cannot be extracted.

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract_or_throw< T, TagT >::type
  extract_or_throw(attribute_name const & name, record_view const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`name`	The name of the attribute value to extract.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
Returns:	The extracted value or a non-empty `value_ref` that refers to the value.	
Throws:	An exception is thrown if the requested value cannot be extracted.	

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void>
  result_of::extract_or_throw< T, TagT >::type
  extract_or_throw(attribute_value const & value);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`value`	Attribute value.
Returns:	The extracted value or a non-empty `value_ref` that refers to the value.	
Throws:	An exception is thrown if the requested value cannot be extracted.	

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void, typename DefaultT>
  result_of::extract_or_default< T, DefaultT, TagT >::type
  extract_or_default(attribute_name const & name,
                     attribute_value_set const & attrs,
                     DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:	`attrs`	A set of attribute values in which to look for the specified attribute value.
	`def_val`	The default value
	`name`	The name of the attribute value to extract.
Returns:	The extracted value, if found. The default value otherwise.	

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void, typename DefaultT>
  result_of::extract_or_default< T, DefaultT, TagT >::type
  extract_or_default(attribute_name const & name, record const & rec,
                     DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:	`def_val`	The default value
	`name`	The name of the attribute value to extract.
	`rec`	A log record. The attribute value will be sought among those associated with the record.
Returns:	The extracted value, if found. The default value otherwise.	

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void, typename DefaultT>
  result_of::extract_or_default< T, DefaultT, TagT >::type
  extract_or_default(attribute_name const & name, record_view const & rec,
                     DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:	`def_val`	The default value
	`name`	The name of the attribute value to extract.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
Returns:	The extracted value, if found. The default value otherwise.	

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename T, typename TagT = void, typename DefaultT>
  result_of::extract_or_default< T, DefaultT, TagT >::type
  extract_or_default(attribute_value const & value, DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:	`def_val`	The default value
	`value`	Attribute value.
Returns:	The extracted value, if found. The default value otherwise.	

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
  extract(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
          attribute_value_set const & attrs);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`attrs`	A set of attribute values in which to look for the specified attribute value.
	`keyword`	The keyword of the attribute value to extract.
Returns:	A `value_ref` that refers to the extracted value, if found. An empty value otherwise.	

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
  extract(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
          record const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`keyword`	The keyword of the attribute value to extract.
	`rec`	A log record. The attribute value will be sought among those associated with the record.
Returns:	A `value_ref` that refers to the extracted value, if found. An empty value otherwise.	

Function template extract

boost::log::extract

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract< typename DescriptorT::value_type, DescriptorT >::type
  extract(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
          record_view const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`keyword`	The keyword of the attribute value to extract.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
Returns:	A `value_ref` that refers to the extracted value, if found. An empty value otherwise.	

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
  extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                   attribute_value_set const & attrs);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters:	`attrs`	A set of attribute values in which to look for the specified attribute value.
	`keyword`	The keyword of the attribute value to extract.
Returns:	The extracted value or a non-empty `value_ref` that refers to the value.	
Throws:	An exception is thrown if the requested value cannot be extracted.	

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
  extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                   record const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `keyword` The keyword of the attribute value to extract.
`rec` A log record. The attribute value will be sought among those associated with the record.
Returns: The extracted value or a non-empty `value_ref` that refers to the value.
Throws: An exception is thrown if the requested value cannot be extracted.

Function template extract_or_throw

boost::log::extract_or_throw

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  result_of::extract_or_throw< typename DescriptorT::value_type, DescriptorT >::type
  extract_or_throw(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                   record_view const & rec);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Parameters: `keyword` The keyword of the attribute value to extract.
`rec` A log record view. The attribute value will be sought among those associated with the record.
Returns: The extracted value or a non-empty `value_ref` that refers to the value.
Throws: An exception is thrown if the requested value cannot be extracted.

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename DefaultT>
  result_of::extract_or_default< typename DescriptorT::value_type, DefaultT, DescriptorT >::type
  extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                     attribute_value_set const & attrs,
                     DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be extracted.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:
- `attrs` A set of attribute values in which to look for the specified attribute value.
- `def_val` The default value
- `keyword` The keyword of the attribute value to extract.

Returns: The extracted value, if found. The default value otherwise.

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename DefaultT>
  result_of::extract_or_default< typename DescriptorT::value_type, DefaultT, DescriptorT >::type
  extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                     record const & rec, DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:
- `def_val` The default value
- `keyword` The keyword of the attribute value to extract.
- `rec` A log record. The attribute value will be sought among those associated with the record.

Returns: The extracted value, if found. The default value otherwise.

Function template extract_or_default

boost::log::extract_or_default

Synopsis

```
// In header: <boost/log/attributes/value_extraction.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename DefaultT>
  result_of::extract_or_default< typename DescriptorT::value_type, DefaultT, DescriptorT >::type
  extract_or_default(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
                     record_view const & rec, DefaultT const & def_val);
```

Description

The function extracts an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Note

Caution must be exercised if the default value is a temporary object. Because the function returns a reference, if the temporary object is destroyed, the reference may become dangling.

Parameters:	`def_val`	The default value
	`keyword`	The keyword of the attribute value to extract.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
Returns:	The extracted value, if found. The default value otherwise.	

Header <boost/log/attributes/value_extraction_fwd.hpp>

Andrey Semashev

01.03.2008
The header contains forward declaration of tools for extracting attribute values from the view.

Header <boost/log/attributes/value_visitation.hpp>

Andrey Semashev

01.03.2008
The header contains implementation of convenience tools to apply visitors to an attribute value in the view.

```
namespace boost {
  namespace log {
    template<typename T, typename FallbackPolicyT> class value_visitor_invoker;
    class visitation_result;
    template<typename T, typename VisitorT>
      visitation_result
      visit(attribute_name const &, attribute_value_set const &, VisitorT);
    template<typename T, typename VisitorT>
      visitation_result
      visit(attribute_name const &, record const &, VisitorT);
    template<typename T, typename VisitorT>
      visitation_result
      visit(attribute_name const &, record_view const &, VisitorT);
    template<typename T, typename VisitorT>
      visitation_result visit(attribute_value const &, VisitorT);
    template<typename DescriptorT, template< typename > class ActorT,
             typename VisitorT>
      visitation_result
      visit(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            attribute_value_set const &, VisitorT);
    template<typename DescriptorT, template< typename > class ActorT,
             typename VisitorT>
      visitation_result
      visit(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            record const &, VisitorT);
    template<typename DescriptorT, template< typename > class ActorT,
             typename VisitorT>
      visitation_result
      visit(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            record_view const &, VisitorT);
  }
}
```

Class template value_visitor_invoker

boost::log::value_visitor_invoker — Generic attribute value visitor invoker.

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>

template<typename T, typename FallbackPolicyT>
class value_visitor_invoker : private FallbackPolicyT {
public:
  // types
  typedef T                 value_type;       // Attribute value types.
  typedef FallbackPolicyT   fallback_policy;  // Fallback policy.
  typedef visitation_result result_type;      // Function object result type.

  // construct/copy/destruct
  value_visitor_invoker() = default;
  value_visitor_invoker(value_visitor_invoker const &);
  template<typename U> explicit value_visitor_invoker(U const &);

  // public member functions
  template<typename VisitorT>
    result_type operator()(attribute_value const &, VisitorT) const;
  template<typename VisitorT>
    result_type operator()(attribute_name const &,
                           attribute_value_set const &, VisitorT) const;
  template<typename VisitorT>
    result_type operator()(attribute_name const &, record const &, VisitorT) const;
  template<typename VisitorT>
    result_type operator()(attribute_name const &, record_view const &,
                           VisitorT) const;
  fallback_policy const & get_fallback_policy() const;
};
```

Description

Attribute value invoker is a functional object that attempts to find and extract the stored attribute value from the attribute value view or a log record. The extracted value is passed to a unary function object (the visitor) provided by user.

The invoker can be specialized on one or several attribute value types that should be specified in the second template argument.

`value_visitor_invoker` public construct/copy/destruct

1. ```
   value_visitor_invoker() = default;
   ```

   Default constructor

2. ```
   value_visitor_invoker(value_visitor_invoker const & that);
   ```

   Copy constructor

3. ```
   template<typename U> explicit value_visitor_invoker(U const & arg);
   ```

   Initializing constructor

   Parameters: `arg` Fallback policy argument
   ```

`value_visitor_invoker` public member functions

1.
```
template<typename VisitorT>
  result_type operator()(attribute_value const & attr, VisitorT visitor) const;
```

Visitation operator. Attempts to acquire the stored value of one of the supported types. If acquisition succeeds, the value is passed to *visitor*.

Parameters: `attr` An attribute value to apply the visitor to.
`visitor` A receiving function object to pass the attribute value to.
Returns: The result of visitation.

2.
```
template<typename VisitorT>
  result_type operator()(attribute_name const & name,
                         attribute_value_set const & attrs, VisitorT visitor) const;
```

Visitation operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If acquisition succeeds, the value is passed to *visitor*.

Parameters: `attrs` A set of attribute values in which to look for the specified attribute value.
`name` Attribute value name.
`visitor` A receiving function object to pass the attribute value to.
Returns: The result of visitation.

3.
```
template<typename VisitorT>
  result_type operator()(attribute_name const & name, record const & rec,
                         VisitorT visitor) const;
```

Visitation operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If acquisition succeeds, the value is passed to *visitor*.

Parameters: `name` Attribute value name.
`rec` A log record. The attribute value will be sought among those associated with the record.
`visitor` A receiving function object to pass the attribute value to.
Returns: The result of visitation.

4.
```
template<typename VisitorT>
  result_type operator()(attribute_name const & name, record_view const & rec,
                         VisitorT visitor) const;
```

Visitation operator. Looks for an attribute value with the specified name and tries to acquire the stored value of one of the supported types. If acquisition succeeds, the value is passed to *visitor*.

Parameters: `name` Attribute value name.
`rec` A log record view. The attribute value will be sought among those associated with the record.
`visitor` A receiving function object to pass the attribute value to.
Returns: The result of visitation.

5.
```
fallback_policy const & get_fallback_policy() const;
```

Returns: Fallback policy

Class visitation_result

boost::log::visitation_result — The class represents attribute value visitation result.

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


class visitation_result {
public:

  // Error codes for attribute value visitation.
  enum error_code { ok, value_not_found, value_has_invalid_type };
  // construct/copy/destruct
  visitation_result(error_code = ok) noexcept;

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  error_code code() const noexcept;
};
```

Description

The main purpose of this class is to provide a convenient interface for checking whether the attribute value visitation succeeded or not. It also allows to discover the actual cause of failure, should the operation fail.

`visitation_result` public construct/copy/destruct

1. ```
   visitation_result(error_code code = ok) noexcept;
   ```

   Initializing constructor. Creates the result that is equivalent to the specified error code.

`visitation_result` public member functions

1. ```
   explicit operator bool() const noexcept;
   ```

   Checks if the visitation was successful.

   Returns: `true` if the value was visited successfully, `false` otherwise.

2. ```
   bool operator!() const noexcept;
   ```

   Checks if the visitation was unsuccessful.

   Returns: `false` if the value was visited successfully, `true` otherwise.

3. ```
   error_code code() const noexcept;
   ```

   Returns: The actual result code of value visitation

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename T, typename VisitorT>
  visitation_result
  visit(attribute_name const & name, attribute_value_set const & attrs,
        VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`attrs`	A set of attribute values in which to look for the specified attribute value.
	`name`	The name of the attribute value to visit.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename T, typename VisitorT>
  visitation_result
  visit(attribute_name const & name, record const & rec, VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`name`	The name of the attribute value to visit.
	`rec`	A log record. The attribute value will be sought among those associated with the record.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename T, typename VisitorT>
  visitation_result
  visit(attribute_name const & name, record_view const & rec,
        VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`name`	The name of the attribute value to visit.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename T, typename VisitorT>
  visitation_result visit(attribute_value const & value, VisitorT visitor);
```

Description

The function applies a visitor to an attribute value. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`value`	The attribute value to visit.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename VisitorT>
  visitation_result
  visit(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
        attribute_value_set const & attrs, VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`attrs`	A set of attribute values in which to look for the specified attribute value.
	`keyword`	The keyword of the attribute value to visit.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename VisitorT>
  visitation_result
  visit(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
        record const & rec, VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`keyword`	The keyword of the attribute value to visit.
	`rec`	A log record. The attribute value will be sought among those associated with the record.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Function template visit

boost::log::visit

Synopsis

```
// In header: <boost/log/attributes/value_visitation.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename VisitorT>
  visitation_result
  visit(expressions::attribute_keyword< DescriptorT, ActorT > const & keyword,
        record_view const & rec, VisitorT visitor);
```

Description

The function applies a visitor to an attribute value from the view. The user has to explicitly specify the type or set of possible types of the attribute value to be visited.

Parameters:	`keyword`	The keyword of the attribute value to visit.
	`rec`	A log record view. The attribute value will be sought among those associated with the record.
	`visitor`	A receiving function object to pass the attribute value to.
Returns:	The result of visitation.	

Header <boost/log/attributes/value_visitation_fwd.hpp>

Andrey Semashev

01.03.2008
The header contains forward declaration of convenience tools to apply visitors to an attribute value in the view.

Expressions

Header <boost/log/expressions/attr.hpp>

Andrey Semashev

21.07.2012
The header contains implementation of a generic attribute placeholder in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT>
        class attribute_actor;
      template<typename T, typename FallbackPolicyT, typename TagT>
        class attribute_terminal;
      template<typename AttributeValueT>
        attribute_actor< AttributeValueT > attr(attribute_name const &);
      template<typename AttributeValueT, typename TagT>
        attribute_actor< AttributeValueT, fallback_to_none, TagT >
        attr(attribute_name const &);
    }
  }
}
```

Class template attribute_actor

boost::log::expressions::attribute_actor

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>

template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT>
class attribute_actor :
  public ActorT< attribute_terminal< T, FallbackPolicyT, TagT > >
{
public:
  // types
  typedef TagT                                                          tag_type;         ↵
     // Attribute tag type.
  typedef FallbackPolicyT                                               fallback_policy;  ↵
     // Fallback policy.
  typedef attribute_terminal< T, fallback_policy, tag_type >            terminal_type;    ↵
     // Base terminal type.
  typedef terminal_type::value_type                                     value_type;       ↵
     // Attribute value type.
  typedef ActorT< terminal_type >                                       base_type;        ↵
     // Base actor type.
  typedef attribute_actor< value_type, fallback_to_none, tag_type, ActorT >  or_none_result_type; ↵
  // Expression with cached attribute name.
  typedef attribute_actor< value_type, fallback_to_throw, tag_type, ActorT > or_throw_result_type; ↵
 // Expression with cached attribute name.

  // construct/copy/destruct
  explicit attribute_actor(base_type const &);

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  or_none_result_type or_none() const;
  or_throw_result_type or_throw() const;
  template<typename DefaultT>
    attribute_actor< value_type, fallback_to_default< DefaultT >, tag_type, ActorT >
    or_default(DefaultT const &) const;
};
```

Description

An attribute value extraction terminal actor

`attribute_actor` public construct/copy/destruct

1. ```
   explicit attribute_actor(base_type const & act);
   ```

   Initializing constructor.

`attribute_actor` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns: The attribute name

2. ```
   fallback_policy const & get_fallback_policy() const;
   ```

   Returns: Fallback policy
   ```

3. ```
   or_none_result_type or_none() const;
   ```

   Generates an expression that extracts the attribute value or a default value.

4. ```
   or_throw_result_type or_throw() const;
   ```

   Generates an expression that extracts the attribute value or throws an exception.

5. ```
   template<typename DefaultT>
     attribute_actor< value_type, fallback_to_default< DefaultT >, tag_type, ActorT >
     or_default(DefaultT const & def_val) const;
   ```

   Generates an expression that extracts the attribute value or a default value.

Class template attribute_terminal

boost::log::expressions::attribute_terminal
   ```

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>

template<typename T, typename FallbackPolicyT, typename TagT>
class attribute_terminal {
public:
  // types
  typedef void                                  _is_boost_log_terminal;  // Internal typedef ↵
for type categorization.
  typedef TagT                                  tag_type;                // Attribute tag type.
  typedef value_extractor_type::value_type      value_type;             // Attribute value type.
  typedef value_extractor_type::fallback_policy fallback_policy;        // Fallback policy type.

  // member classes/structs/unions

  // Function result type.
  template<typename >
  struct result {
  };
  template<typename ThisT, typename ContextT>
  struct result<ThisT(ContextT)> {
    // types
    typedef remove_cv< typename remove_reference< typename phoenix::result_of::env< Con↵
textT >::type >::type >::type                           env_type;
    typedef env_type::args_type                                                          ↵
                                                args_type;
    typedef unspecified                                                                  ↵
                                                cv_value_extractor_type;
   typedef boost::result_of< cv_value_extractor_type(attribute_name const &, typename fusion::res↵
ult_of::at_c< args_type, 0 >::type) >::type type;
  };

  // construct/copy/destruct
  explicit attribute_terminal(attribute_name const &);
  template<typename U> attribute_terminal(attribute_name const &, U const &);
  attribute_terminal() = delete;

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  template<typename ContextT>
    result< this_type(ContextT const &) >::type operator()(ContextT const &);
  template<typename ContextT>
    result< const this_type(ContextT const &) >::type
    operator()(ContextT const &) const;
};
```

Description

An attribute value extraction terminal

`attribute_terminal` public construct/copy/destruct

1. ```
   explicit attribute_terminal(attribute_name const & name);
   ```

   Initializing constructor

2. ```
   template<typename U>
     attribute_terminal(attribute_name const & name, U const & arg);
   ```

Initializing constructor

3.
```
attribute_terminal() = delete;
```

`attribute_terminal` public member functions

1.
```
attribute_name get_name() const;
```

Returns: Attribute value name

2.
```
fallback_policy const & get_fallback_policy() const;
```

Returns: Fallback policy

3.
```
template<typename ContextT>
  result< this_type(ContextT const &) >::type operator()(ContextT const & ctx);
```

The operator extracts attribute value

4.
```
template<typename ContextT>
  result< const this_type(ContextT const &) >::type
  operator()(ContextT const & ctx) const;
```

The operator extracts attribute value

Struct template result

boost::log::expressions::attribute_terminal::result — Function result type.

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>


// Function result type.
template<typename >
struct result {
};
```

Struct template result<ThisT(ContextT)>

boost::log::expressions::attribute_terminal::result<ThisT(ContextT)>

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>


template<typename ThisT, typename ContextT>
struct result<ThisT(ContextT)> {
  // types
  typedef remove_cv< typename remove_reference< typename phoenix::result_of::env< Con↵
textT >::type >::type >::type                          env_type;
  typedef env_type::args_type                                                      ↵
                                                 args_type;
  typedef unspecified                                                              ↵
                                                 cv_value_extractor_type;
  typedef boost::result_of< cv_value_extractor_type(attribute_name const &, typename fusion::res↵
ult_of::at_c< args_type, 0 >::type) >::type type;
};
```

Function template attr

boost::log::expressions::attr

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>


template<typename AttributeValueT>
  attribute_actor< AttributeValueT > attr(attribute_name const & name);
```

Description

The function generates a terminal node in a template expression. The node will extract the value of the attribute with the specified name and type.

Function template attr

boost::log::expressions::attr

Synopsis

```
// In header: <boost/log/expressions/attr.hpp>


template<typename AttributeValueT, typename TagT>
  attribute_actor< AttributeValueT, fallback_to_none, TagT >
  attr(attribute_name const & name);
```

Description

The function generates a terminal node in a template expression. The node will extract the value of the attribute with the specified name and type.

Header <boost/log/expressions/attr_fwd.hpp>

Andrey Semashev

21.07.2012
The header contains forward declaration of a generic attribute placeholder in template expressions.

Header <boost/log/expressions/filter.hpp>

Andrey Semashev

13.07.2012
The header contains a filter function object definition.

```
namespace boost {
  namespace log {
    class filter;
    void swap(filter &, filter &);
  }
}
```

Class filter

boost::log::filter

Synopsis

```
// In header: <boost/log/expressions/filter.hpp>


class filter {
public:
  // types
  typedef bool result_type;  // Result type.

  // member classes/structs/unions

  // Default filter, always returns true.

  struct default_filter {
    // types
    typedef bool result_type;

    // public member functions
    result_type operator()(attribute_value_set const &) const;
  };

  // construct/copy/destruct
  filter();
  filter(filter const &);
  filter(filter &&) noexcept;
  template<typename FunT> filter(FunT const &);
  filter & operator=(filter &&) noexcept;
  filter & operator=(filter const &);
  template<typename FunT> filter & operator=(FunT const &);

  // public member functions
  result_type operator()(attribute_value_set const &) const;
  void reset();
  void swap(filter &) noexcept;
};
```

Description

Log record filter function wrapper.

`filter` public construct/copy/destruct

1. ```
   filter();
   ```

   Default constructor. Creates a filter that always returns `true`.

2. ```
   filter(filter const & that);
   ```

   Copy constructor

3. ```
   filter(filter && that) noexcept;
   ```

   Move constructor. The moved-from filter is left in an unspecified state.

4. ```
   template<typename FunT> filter(FunT const & fun);
   ```

   Initializing constructor. Creates a filter which will invoke the specified function object.

5. ```
   filter & operator=(filter && that) noexcept;
   ```

   Move assignment. The moved-from filter is left in an unspecified state.

6. ```
   filter & operator=(filter const & that);
   ```

   Copy assignment.

7. ```
   template<typename FunT> filter & operator=(FunT const & fun);
   ```

   Initializing assignment. Sets the specified function object to the filter.

`filter` public member functions

1. ```
   result_type operator()(attribute_value_set const & values) const;
   ```

   Filtering operator.

   Parameters: `values` Attribute values of the log record.
   Returns: `true` if the log record passes the filter, `false` otherwise.

2. ```
   void reset();
   ```

   Resets the filter to the default. The default filter always returns `true`.

3. ```
   void swap(filter & that) noexcept;
   ```

   Swaps two filters

Struct default_filter

boost::log::filter::default_filter — Default filter, always returns `true`.

Synopsis

```
// In header: <boost/log/expressions/filter.hpp>



// Default filter, always returns true.

struct default_filter {
  // types
  typedef bool result_type;

  // public member functions
  result_type operator()(attribute_value_set const &) const;
};
```

Description

`default_filter` public member functions

1. ```
   result_type operator()(attribute_value_set const &) const;
   ```

Function swap

boost::log::swap

Synopsis

```
// In header: <boost/log/expressions/filter.hpp>


void swap(filter & left, filter & right);
```

Header <boost/log/expressions/formatter.hpp>

Andrey Semashev

13.07.2012
The header contains a formatter function object definition.

```
namespace boost {
  namespace log {
    template<typename CharT> class basic_formatter;

    typedef basic_formatter< char > formatter;
    typedef basic_formatter< wchar_t > wformatter;
    template<typename CharT>
      void swap(basic_formatter< CharT > &, basic_formatter< CharT > &);
    namespace expressions {
    }
  }
}
```
```

Class template basic_formatter

boost::log::basic_formatter

Synopsis

```
// In header: <boost/log/expressions/formatter.hpp>

template<typename CharT>
class basic_formatter {
public:
  // types
  typedef void                                    result_type;  // Result type.
  typedef CharT                                   char_type;    // Character type.
  typedef basic_formatting_ostream< char_type >   stream_type;  // Output stream type.

  // construct/copy/destruct
  basic_formatter();
  basic_formatter(basic_formatter const &);
  basic_formatter(this_type &&) noexcept;
  template<typename FunT> basic_formatter(FunT const &);
  basic_formatter & operator=(this_type &&) noexcept;
  basic_formatter & operator=(this_type const &);
  template<typename FunT> basic_formatter & operator=(FunT const &);

  // public member functions
  result_type operator()(record_view const &, stream_type &) const;
  void reset();
  void swap(basic_formatter &) noexcept;
};
```

Description

Log record formatter function wrapper.

`basic_formatter` public construct/copy/destruct

1. ```
   basic_formatter();
   ```

   Default constructor. Creates a formatter that only outputs log message.

2. ```
   basic_formatter(basic_formatter const & that);
   ```

   Copy constructor

3. ```
   basic_formatter(this_type && that) noexcept;
   ```

   Move constructor. The moved-from formatter is left in an unspecified state.

4. ```
   template<typename FunT> basic_formatter(FunT const & fun);
   ```

   Initializing constructor. Creates a formatter which will invoke the specified function object.

5. ```
   basic_formatter & operator=(this_type && that) noexcept;
   ```

   Move assignment. The moved-from formatter is left in an unspecified state.
   ```

6. ```
   basic_formatter & operator=(this_type const & that);
   ```

   Copy assignment.

7. ```
   template<typename FunT> basic_formatter & operator=(FunT const & fun);
   ```

   Initializing assignment. Sets the specified function object to the formatter.

`basic_formatter` public member functions

1. ```
   result_type operator()(record_view const & rec, stream_type & strm) const;
   ```

   Formatting operator.

   Parameters: `rec` A log record to format.
   `strm` A stream to put the formatted characters to.

2. ```
   void reset();
   ```

   Resets the formatter to the default. The default formatter only outputs message text.

3. ```
   void swap(basic_formatter & that) noexcept;
   ```

   Swaps two formatters

Function template swap

boost::log::swap

Synopsis

```
// In header: <boost/log/expressions/formatter.hpp>


template<typename CharT>
  void swap(basic_formatter< CharT > & left, basic_formatter< CharT > & right);
```

Header <boost/log/expressions/formatters.hpp>

Andrey Semashev

10.11.2012
The header includes all template expression formatters.

Header <boost/log/expressions/formatters/c_decorator.hpp>

Andrey Semashev

18.11.2012
The header contains implementation of C-style character decorators.
```

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename CharT> class c_ascii_pattern_replacer;

      unspecified c_decor;
      unspecified wc_decor;
      unspecified c_ascii_decor;
      unspecified wc_ascii_decor;
      template<typename CharT> unspecified make_c_decor();
      template<typename CharT> unspecified make_c_ascii_decor();
    }
  }
}
```

Class template c_ascii_pattern_replacer

boost::log::expressions::c_ascii_pattern_replacer

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>

template<typename CharT>
class c_ascii_pattern_replacer :
  public boost::log::expressions::pattern_replacer< CharT >
{
public:
  // types
  typedef base_type::result_type result_type;  // Result type.
  typedef base_type::char_type   char_type;    // Character type.
  typedef base_type::string_type string_type;  // String type.

  // construct/copy/destruct
  c_ascii_pattern_replacer();

  // public member functions
  result_type operator()(string_type &, typename string_type::size_type = 0) const;
};
```

Description

A character decorator implementation that escapes all non-prontable and non-ASCII characters in the output with C-style escape sequences.

`c_ascii_pattern_replacer` public construct/copy/destruct

1. ```
   c_ascii_pattern_replacer();
   ```

   Default constructor.

`c_ascii_pattern_replacer` public member functions

1. ```
   result_type operator()(string_type & str,
                          typename string_type::size_type start_pos = 0) const;
   ```

   Applies string replacements starting from the specified position.

Global c_decor

boost::log::expressions::c_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>

unspecified c_decor;
```

Description

C-style decorator generator object. The decorator replaces characters with specific meaning in C language with the corresponding escape sequences. The generator provides `operator[]` that can be used to construct the actual decorator. For example:

```
c_decor[ attr< std::string >("MyAttr") ]
```

For wide-character formatting there is the similar `wc_decor` decorator generator object.

Global wc_decor

boost::log::expressions::wc_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>

unspecified wc_decor;
```

Global c_ascii_decor

boost::log::expressions::c_ascii_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>

unspecified c_ascii_decor;
```

Description

C-style decorator generator object. Acts similarly to `c_decor`, except that `c_ascii_decor` also converts all non-ASCII and non-printable ASCII characters, except for space character, into C-style hexadecimal escape sequences. The generator provides `operator[]` that can be used to construct the actual decorator. For example:

```
c_ascii_decor[ attr< std::string >("MyAttr") ]
```

For wide-character formatting there is the similar `wc_ascii_decor` decorator generator object.

Global wc_ascii_decor

boost::log::expressions::wc_ascii_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>

unspecified wc_ascii_decor;
```

Function template make_c_decor

boost::log::expressions::make_c_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>


template<typename CharT> unspecified make_c_decor();
```

Description

The function creates a C-style decorator generator for arbitrary character type.

Function template make_c_ascii_decor

boost::log::expressions::make_c_ascii_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/c_decorator.hpp>


template<typename CharT> unspecified make_c_ascii_decor();
```

Description

The function creates a C-style decorator generator for arbitrary character type.

Header <boost/log/expressions/formatters/char_decorator.hpp>

Andrey Semashev

17.11.2012
The header contains implementation of a character decorator.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename SubactorT, typename ImplT,
               template< typename > class ActorT = phoenix::actor>
        class char_decorator_actor;
      template<typename SubactorT, typename ImplT> class char_decorator_terminal;
      template<typename CharT> class pattern_replacer;
      template<typename RangeT> unspecified char_decor(RangeT const &);
      template<typename FromRangeT, typename ToRangeT>
        unspecified char_decor(FromRangeT const &, ToRangeT const &);
    }
  }
}
```

Class template char_decorator_actor

boost::log::expressions::char_decorator_actor

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>

template<typename SubactorT, typename ImplT,
         template< typename > class ActorT = phoenix::actor>
class char_decorator_actor :
  public ActorT< char_decorator_terminal< SubactorT, ImplT > >
{
public:
  // types
  typedef char_decorator_terminal< SubactorT, ImplT > terminal_type;  // Base terminal type.
  typedef terminal_type::char_type                    char_type;      // Character type.
  typedef ActorT< terminal_type >                     base_type;      // Base actor type.

  // construct/copy/destruct
  explicit char_decorator_actor(base_type const &);

  // public member functions
  terminal_type const & get_terminal() const;
};
```

Description

Character decorator actor

`char_decorator_actor` public construct/copy/destruct

1. ```
   explicit char_decorator_actor(base_type const & act);
   ```

   Initializing constructor.

`char_decorator_actor` public member functions

1. ```
   terminal_type const & get_terminal() const;
   ```

   Returns reference to the terminal.

Class template char_decorator_terminal

boost::log::expressions::char_decorator_terminal

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>

template<typename SubactorT, typename ImplT>
class char_decorator_terminal {
public:
  // types
  typedef void                                  _is_boost_log_terminal;  // Internal typedef ↵
for type categorization.
  typedef ImplT                                impl_type;               // Implementation type.
  typedef impl_type::char_type                  char_type;                 // Character type.
  typedef impl_type::string_type                string_type;               // String type.
  typedef basic_formatting_ostream< char_type > stream_type;               // Stream type.
  typedef SubactorT                            subactor_type;           // Adopted actor type.
  typedef string_type                           result_type;              // Result type defini↵
tion.

  // construct/copy/destruct
  char_decorator_terminal(subactor_type const &, impl_type const &);
  char_decorator_terminal(char_decorator_terminal const &);
  char_decorator_terminal() = delete;

  // public member functions
  subactor_type const & get_subactor() const;
  impl_type const & get_impl() const;
  template<typename ContextT> result_type operator()(ContextT const &);
  template<typename ContextT> result_type operator()(ContextT const &) const;
};
```

Description

Character decorator terminal class. This formatter allows to modify strings generated by other formatters on character level. The most obvious application of decorators is replacing a certain set of characters with decorated equivalents to satisfy requirements of text-based sinks.

The `char_decorator_terminal` class aggregates the formatter being decorated, and a set of string pairs that are used as decorations. All decorations are applied sequentially. The `char_decorator_terminal` class is a formatter itself, so it can be used to construct more complex formatters, including nesting decorators.

`char_decorator_terminal` public construct/copy/destruct

1. ```
   char_decorator_terminal(subactor_type const & sub, impl_type const & impl);
   ```

   Initializing constructor.

2. ```
   char_decorator_terminal(char_decorator_terminal const & that);
   ```

   Copy constructor

3. ```
   char_decorator_terminal() = delete;
   ```
   ```

`char_decorator_terminal` public member functions

1. ```
   subactor_type const & get_subactor() const;
   ```

   Returns: Adopted subactor

2. ```
   impl_type const & get_impl() const;
   ```

   Returns: Implementation

3. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx);
   ```

   Invokation operator

4. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx) const;
   ```

   Invokation operator

Class template pattern_replacer

boost::log::expressions::pattern_replacer

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>

template<typename CharT>
class pattern_replacer {
public:
  // types
  typedef void                              result_type;  // Result type.
  typedef CharT                             char_type;    // Character type.
  typedef std::basic_string< char_type > string_type;  // String type.

  // member classes/structs/unions

  // Lengths of source pattern and replacement.

  struct string_lengths {

    // public data members
    unsigned int from_len;
    unsigned int to_len;
  };

  // construct/copy/destruct
  template<typename RangeT> explicit pattern_replacer(RangeT const &);
  template<typename FromRangeT, typename ToRangeT>
    pattern_replacer(FromRangeT const &, ToRangeT const &);
  pattern_replacer(pattern_replacer const &);

  // public member functions
  result_type operator()(string_type &, typename string_type::size_type = 0) const;

  // private static functions
  static char_type * string_begin(char_type *);
  static const char_type * string_begin(const char_type *);
  template<typename RangeT>
    static range_const_iterator< RangeT >::type string_begin(RangeT const &);
  static char_type * string_end(char_type *);
  static const char_type * string_end(const char_type *);
  template<typename RangeT>
    static range_const_iterator< RangeT >::type string_end(RangeT const &);
};
```

Description

A simple character decorator implementation. This implementation replaces string patterns in the source string with the fixed replacements. Source patterns and replacements can be specified at the object construction.

`pattern_replacer` public construct/copy/destruct

1. ```
   template<typename RangeT>
     explicit pattern_replacer(RangeT const & decorations);
   ```

   Initializing constructor. Creates a pattern replacer with the specified *decorations*. The provided decorations must be a sequence of `std::pair` of strings. The first element of each pair is the source pattern, and the second one is the corresponding replacement.

2. ```
   template<typename FromRangeT, typename ToRangeT>
     pattern_replacer(FromRangeT const & from, ToRangeT const & to);
   ```

Initializing constructor. Creates a pattern replacer with decorations specified in form of two same-sized string sequences. Each `i`'th decoration will be `from[i]` -> `to[i]`.

3.
```
pattern_replacer(pattern_replacer const & that);
```

Copy constructor.

`pattern_replacer` public member functions

1.
```
result_type operator()(string_type & str,
                       typename string_type::size_type start_pos = 0) const;
```

Applies string replacements starting from the specified position.

`pattern_replacer` private static functions

1.
```
static char_type * string_begin(char_type * p);
```

2.
```
static const char_type * string_begin(const char_type * p);
```

3.
```
template<typename RangeT>
  static range_const_iterator< RangeT >::type string_begin(RangeT const & r);
```

4.
```
static char_type * string_end(char_type * p);
```

5.
```
static const char_type * string_end(const char_type * p);
```

6.
```
template<typename RangeT>
  static range_const_iterator< RangeT >::type string_end(RangeT const & r);
```

Struct string_lengths

boost::log::expressions::pattern_replacer::string_lengths — Lengths of source pattern and replacement.

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>



// Lengths of source pattern and replacement.

struct string_lengths {

  // public data members
  unsigned int from_len;
  unsigned int to_len;
};
```

Function template char_decor

boost::log::expressions::char_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>


template<typename RangeT> unspecified char_decor(RangeT const & decorations);
```

Description

The function returns a decorator generator object. The generator provides `operator[]` that can be used to construct the actual decorator.

Parameters: `decorations` A sequence of string pairs that will be used as decorations. Every `decorations[i].first` substring occurrence in the output will be replaced with `decorations[i].second`.

Function template char_decor

boost::log::expressions::char_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/char_decorator.hpp>


template<typename FromRangeT, typename ToRangeT>
  unspecified char_decor(FromRangeT const & from, ToRangeT const & to);
```

Description

The function returns a decorator generator object. The generator provides `operator[]` that can be used to construct the actual decorator.

Note

The *from* and *to* sequences mush be of the same size. Every `from[i]` substring occurrence in the output will be replaced with `to[i]`.

Parameters:
- `from` A sequence of strings that will be sought in the output.
- `to` A sequence of strings that will be used as replacements.

Header <boost/log/expressions/formatters/csv_decorator.hpp>

Andrey Semashev

18.11.2012
The header contains implementation of a CSV-style character decorator. See: http://en.wikipedia.org/wiki/Comma-separated_values

```
namespace boost {
  namespace log {
    namespace expressions {
      unspecified csv_decor;
      unspecified wcsv_decor;
      template<typename CharT> unspecified make_csv_decor();
    }
  }
}
```

Global csv_decor

boost::log::expressions::csv_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/csv_decorator.hpp>

unspecified csv_decor;
```

Description

CSV-style decorator generator object. The decorator doubles double quotes that may be found in the output. See http://en.wikipedia.org/wiki/Comma-separated_values for more information on the CSV format. The generator provides `operator[]` that can be used to construct the actual decorator. For example:

```
csv_decor[ attr< std::string >("MyAttr") ]
```

For wide-character formatting there is the similar `wcsv_decor` decorator generator object.

Global wcsv_decor

boost::log::expressions::wcsv_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/csv_decorator.hpp>

unspecified wcsv_decor;
```

Function template make_csv_decor

boost::log::expressions::make_csv_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/csv_decorator.hpp>

template<typename CharT> unspecified make_csv_decor();
```

Description

The function creates an CSV-style decorator generator for arbitrary character type.

Header <boost/log/expressions/formatters/date_time.hpp>

Andrey Semashev

16.09.2012
The header contains a formatter function for date and time attribute values.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename T, typename FallbackPolicyT, typename CharT,
               template< typename > class ActorT = phoenix::actor>
        class format_date_time_actor;
      template<typename T, typename FallbackPolicyT, typename CharT>
        class format_date_time_terminal;
      template<typename AttributeValueT, typename CharT>
        format_date_time_actor< AttributeValueT, fallback_to_none, CharT >
        format_date_time(attribute_name const &, const CharT *);
      template<typename AttributeValueT, typename CharT>
        format_date_time_actor< AttributeValueT, fallback_to_none, CharT >
        format_date_time(attribute_name const &,
                         std::basic_string< CharT > const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename CharT>
        format_date_time_actor< typename DescriptorT::value_type, fall↵
back_to_none, CharT, ActorT >
        format_date_time(attribute_keyword< DescriptorT, ActorT > const &,
                         const CharT *);
      template<typename DescriptorT, template< typename > class ActorT,
               typename CharT>
        format_date_time_actor< typename DescriptorT::value_type, fall↵
back_to_none, CharT, ActorT >
        format_date_time(attribute_keyword< DescriptorT, ActorT > const &,
                         std::basic_string< CharT > const &);
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename CharT>
        format_date_time_actor< T, FallbackPolicyT, CharT, ActorT >
        format_date_time(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                         const CharT *);
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename CharT>
        format_date_time_actor< T, FallbackPolicyT, CharT, ActorT >
        format_date_time(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                         std::basic_string< CharT > const &);
    }
  }
}
```

Class template format_date_time_actor

boost::log::expressions::format_date_time_actor

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>

template<typename T, typename FallbackPolicyT, typename CharT,
         template< typename > class ActorT = phoenix::actor>
class format_date_time_actor :
  public ActorT< format_date_time_terminal< T, FallbackPolicyT, CharT > >
{
public:
  // types
  typedef T                                                                  value_type;      ↵
         // Attribute value type.
  typedef CharT                                                              char_type;       ↵
         // Character type.
  typedef FallbackPolicyT                                                    fallback_policy; ↵
         // Fallback policy.
  typedef format_date_time_terminal< value_type, fallback_policy, char_type > terminal_type;   ↵
         // Base terminal type.
  typedef terminal_type::formatter_function_type                              formatter_func↵
tion_type;  // Formatter function.
  typedef ActorT< terminal_type >                                            base_type;       ↵
         // Base actor type.

  // construct/copy/destruct
  explicit format_date_time_actor(base_type const &);

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  formatter_function_type const & get_formatter_function() const;
};
```

Description

Date and time formatter actor.

`format_date_time_actor` public construct/copy/destruct

1. ```
   explicit format_date_time_actor(base_type const & act);
   ```

   Initializing constructor.

`format_date_time_actor` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns: The attribute name

2. ```
   fallback_policy const & get_fallback_policy() const;
   ```

   Returns: Fallback policy

3. ```
   formatter_function_type const & get_formatter_function() const;
   ```

   Returns: Formatter function

Class template format_date_time_terminal

boost::log::expressions::format_date_time_terminal

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>

template<typename T, typename FallbackPolicyT, typename CharT>
class format_date_time_terminal {
public:
  // types
  typedef void                                    _is_boost_log_terminal;   // Internal typedef ↵
for type categorization.
  typedef T                                    value_type;              // Attribute value type.
  typedef FallbackPolicyT                         fallback_policy;          // Fallback policy.
  typedef CharT                                   char_type;                // Character type.
  typedef std::basic_string< char_type >          string_type;              // String type.
  typedef basic_formatting_ostream< char_type > stream_type;             // Formatting stream ↵
type.
  typedef unspecified                            formatter_function_type;  // Formatter function.
  typedef string_type                            result_type;              // Function result type.

  // construct/copy/destruct
  format_date_time_terminal(attribute_name const &, fallback_policy const &,
                            string_type const &);
  format_date_time_terminal(format_date_time_terminal const &);
  format_date_time_terminal() = delete;

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  formatter_function_type const & get_formatter_function() const;
  template<typename ContextT> result_type operator()(ContextT const &);
  template<typename ContextT> result_type operator()(ContextT const &) const;
};
```

Description

Date and time formatter terminal.

`format_date_time_terminal` public construct/copy/destruct

1. ```
   format_date_time_terminal(attribute_name const & name,
                             fallback_policy const & fallback,
                             string_type const & format);
   ```

   Initializing constructor.

2. ```
   format_date_time_terminal(format_date_time_terminal const & that);
   ```

   Copy constructor.

3. ```
   format_date_time_terminal() = delete;
   ```
   ```

`format_date_time_terminal` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns attribute name.

2. ```
   fallback_policy const & get_fallback_policy() const;
   ```

   Returns fallback policy.

3. ```
   formatter_function_type const & get_formatter_function() const;
   ```

   Retruns formatter function.

4. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx);
   ```

   Invokation operator.

5. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx) const;
   ```

   Invokation operator.

Function template format_date_time

boost::log::expressions::format_date_time

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename AttributeValueT, typename CharT>
  format_date_time_actor< AttributeValueT, fallback_to_none, CharT >
  format_date_time(attribute_name const & name, const CharT * format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`name`	Attribute name

Function template format_date_time

boost::log::expressions::format_date_time
```

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename AttributeValueT, typename CharT>
  format_date_time_actor< AttributeValueT, fallback_to_none, CharT >
  format_date_time(attribute_name const & name,
                   std::basic_string< CharT > const & format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`name`	Attribute name

Function template format_date_time

boost::log::expressions::format_date_time

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename CharT>
  format_date_time_actor< typename DescriptorT::value_type, fallback_to_none, CharT, ActorT >
  format_date_time(attribute_keyword< DescriptorT, ActorT > const & keyword,
                   const CharT * format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`keyword`	Attribute keyword

Function template format_date_time

boost::log::expressions::format_date_time

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename CharT>
  format_date_time_actor< typename DescriptorT::value_type, fallback_to_none, CharT, ActorT >
  format_date_time(attribute_keyword< DescriptorT, ActorT > const & keyword,
                   std::basic_string< CharT > const & format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`keyword`	Attribute keyword

Function template format_date_time

boost::log::expressions::format_date_time

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename CharT>
  format_date_time_actor< T, FallbackPolicyT, CharT, ActorT >
  format_date_time(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & placeholder,
                   const CharT * format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`placeholder`	Attribute placeholder

Function template format_date_time

boost::log::expressions::format_date_time

Synopsis

```
// In header: <boost/log/expressions/formatters/date_time.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename CharT>
  format_date_time_actor< T, FallbackPolicyT, CharT, ActorT >
  format_date_time(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & placeholder,
                   std::basic_string< CharT > const & format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`format`	Format string
	`placeholder`	Attribute placeholder

Header <boost/log/expressions/formatters/format.hpp>

Andrey Semashev

15.11.2012
The header contains a generic log record formatter function.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename CharT> class format_terminal;
      template<typename CharT>
        phoenix::actor< format_terminal< CharT > > format(const CharT *);
      template<typename CharT, typename TraitsT, typename AllocatorT>
        phoenix::actor< format_terminal< CharT > >
        format(std::basic_string< CharT, TraitsT, AllocatorT > const &);
    }
  }
}
```

Class template format_terminal

boost::log::expressions::format_terminal — Template expressions terminal node with Boost.Format-like formatter.

Synopsis

```
// In header: <boost/log/expressions/formatters/format.hpp>

template<typename CharT>
class format_terminal {
public:
  // types
  typedef void                          _is_boost_log_terminal;  // Internal typedef for type ↵
categorization.
  typedef CharT                           char_type;               // Character type.
  typedef unspecified                   format_type;           // Boost.Format formatter type.
  typedef std::basic_string< char_type > string_type;              // String type.
  typedef format_type::pump               result_type;             // Terminal result type.

  // construct/copy/destruct
  explicit format_terminal(const char_type *);
  format_terminal() = delete;

  // public member functions
  template<typename ContextT> result_type operator()(ContextT const &) const;
};
```

Description

`format_terminal` public construct/copy/destruct

1. ```
   explicit format_terminal(const char_type * format);
   ```

   Initializing constructor.

2. ```
   format_terminal() = delete;
   ```

`format_terminal` public member functions

1. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx) const;
   ```
   ```

Invokation operator.

Function template format

boost::log::expressions::format

Synopsis

```
// In header: <boost/log/expressions/formatters/format.hpp>


template<typename CharT>
  phoenix::actor< format_terminal< CharT > > format(const CharT * fmt);
```

Description

The function generates a terminal node in a template expression. The node will perform log record formatting according to the provided format string.

Function template format

boost::log::expressions::format

Synopsis

```
// In header: <boost/log/expressions/formatters/format.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT>
  phoenix::actor< format_terminal< CharT > >
  format(std::basic_string< CharT, TraitsT, AllocatorT > const & fmt);
```

Description

The function generates a terminal node in a template expression. The node will perform log record formatting according to the provided format string.

Header <boost/log/expressions/formatters/if.hpp>

Andrey Semashev

17.11.2012
The header contains implementation of a conditional formatter.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename CondT> unspecified if_(CondT const &);
    }
  }
}
```

Function template if_

boost::log::expressions::if_

Synopsis

```
// In header: <boost/log/expressions/formatters/if.hpp>


template<typename CondT> unspecified if_(CondT const & cond);
```

Description

The function returns a conditional formatter generator object. The generator provides `operator[]` that can be used to construct the actual formatter. The formatter must participate in a streaming expression.

Parameters: `cond` A filter expression that will be used as the condition

Header <boost/log/expressions/formatters/named_scope.hpp>

Andrey Semashev

11.11.2012
The header contains a formatter function for named scope attribute values.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename FallbackPolicyT, typename CharT,
               template< typename > class ActorT = phoenix::actor>
        class format_named_scope_actor;
      template<typename FallbackPolicyT, typename CharT>
        class format_named_scope_terminal;

      // Scope iteration directions.
      enum scope_iteration_direction { forward, reverse };
      template<typename CharT>
        format_named_scope_actor< fallback_to_none, CharT >
        format_named_scope(attribute_name const &, const CharT *);
      template<typename CharT>
        format_named_scope_actor< fallback_to_none, CharT >
        format_named_scope(attribute_name const &,
                           std::basic_string< CharT > const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename CharT>
        format_named_scope_actor< fallback_to_none, CharT, ActorT >
        format_named_scope(attribute_keyword< DescriptorT, ActorT > const &,
                           const CharT *);
      template<typename DescriptorT, template< typename > class ActorT,
               typename CharT>
        format_named_scope_actor< fallback_to_none, CharT, ActorT >
        format_named_scope(attribute_keyword< DescriptorT, ActorT > const &,
                           std::basic_string< CharT > const &);
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename CharT>
        format_named_scope_actor< FallbackPolicyT, CharT, ActorT >
        format_named_scope(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                           const CharT *);
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename CharT>
        format_named_scope_actor< FallbackPolicyT, CharT, ActorT >
        format_named_scope(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                           std::basic_string< CharT > const &);
      template<typename... ArgsT>
        unspecified format_named_scope(attribute_name const &,
                                       ArgsT...const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename... ArgsT>
        unspecified format_named_scope(attribute_keyword< DescriptorT, ActorT > const &,
                                       ArgsT...const &);
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename... ArgsT>
        unspecified format_named_scope(attribute_actor< T, FallbackPoli↵
cyT, TagT, ActorT > const &,
                                       ArgsT...const &);
    }
  }
}
```

Class template format_named_scope_actor

boost::log::expressions::format_named_scope_actor

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>

template<typename FallbackPolicyT, typename CharT,
         template< typename > class ActorT = phoenix::actor>
class format_named_scope_actor :
  public ActorT< format_named_scope_terminal< FallbackPolicyT, CharT > >
{
public:
  // types
  typedef CharT                                                  char_type;               ↵
 // Character type.
  typedef FallbackPolicyT                                        fallback_policy;         ↵
 // Fallback policy.
  typedef format_named_scope_terminal< fallback_policy, char_type > terminal_type;        ↵
 // Base terminal type.
  typedef terminal_type::value_type                              value_type;              ↵
 // Attribute value type.
  typedef terminal_type::formatter_function_type                 formatter_function_type; ↵
 // Formatter function.
  typedef ActorT< terminal_type >                                base_type;               ↵
 // Base actor type.

  // construct/copy/destruct
  explicit format_named_scope_actor(base_type const &);

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  formatter_function_type const & get_formatter_function() const;
};
```

Description

Named scope formatter actor.

`format_named_scope_actor` public construct/copy/destruct

1. ```
   explicit format_named_scope_actor(base_type const & act);
   ```

   Initializing constructor.

`format_named_scope_actor` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns: The attribute name

2. ```
   fallback_policy const & get_fallback_policy() const;
   ```

   Returns: Fallback policy

3. ```
   formatter_function_type const & get_formatter_function() const;
   ```

   Returns: Formatter function

Class template format_named_scope_terminal

boost::log::expressions::format_named_scope_terminal

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>

template<typename FallbackPolicyT, typename CharT>
class format_named_scope_terminal {
public:
  // types
  typedef void                                    _is_boost_log_terminal;   // Internal typedef ↵
for type categorization.
  typedef attributes::named_scope::value_type   value_type;             // Attribute value type.
  typedef FallbackPolicyT                         fallback_policy;          // Fallback policy.
  typedef CharT                                   char_type;                // Character type.
  typedef std::basic_string< char_type >          string_type;              // String type.
  typedef basic_formatting_ostream< char_type > stream_type;              // Formatting stream ↵
type.
  typedef unspecified                            formatter_function_type;  // Formatter function.
  typedef string_type                           result_type;             // Function result type.

  // construct/copy/destruct
  template<typename FormatT>
    format_named_scope_terminal(attribute_name const &,
                                fallback_policy const &, FormatT const &,
                                string_type const &, string_type const &,
                                string_type const &, value_type::size_type,
                                scope_iteration_direction);
  format_named_scope_terminal(format_named_scope_terminal const &);
  format_named_scope_terminal() = delete;

  // public member functions
  attribute_name get_name() const;
  fallback_policy const & get_fallback_policy() const;
  formatter_function_type const & get_formatter_function() const;
  template<typename ContextT> result_type operator()(ContextT const &);
  template<typename ContextT> result_type operator()(ContextT const &) const;
};
```

Description

Named scope formatter terminal.

`format_named_scope_terminal` public construct/copy/destruct

1. ```
   template<typename FormatT>
     format_named_scope_terminal(attribute_name const & name,
                                 fallback_policy const & fallback,
                                 FormatT const & element_format,
                                 string_type const & delimiter,
                                 string_type const & incomplete_marker,
                                 string_type const & empty_marker,
                                 value_type::size_type depth,
                                 scope_iteration_direction direction);
   ```

   Initializing constructor.

2. ```
   format_named_scope_terminal(format_named_scope_terminal const & that);
   ```

Copy constructor.

3. ```
   format_named_scope_terminal() = delete;
   ```

`format_named_scope_terminal` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns attribute name.

2. ```
   fallback_policy const & get_fallback_policy() const;
   ```

   Returns fallback policy.

3. ```
   formatter_function_type const & get_formatter_function() const;
   ```

   Retruns formatter function.

4. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx);
   ```

   Invokation operator.

5. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx) const;
   ```

   Invokation operator.

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename CharT>
  format_named_scope_actor< fallback_to_none, CharT >
  format_named_scope(attribute_name const & name,
                     const CharT * element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`name`	Attribute name

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename CharT>
  format_named_scope_actor< fallback_to_none, CharT >
  format_named_scope(attribute_name const & name,
                     std::basic_string< CharT > const & element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`name`	Attribute name

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename CharT>
  format_named_scope_actor< fallback_to_none, CharT, ActorT >
  format_named_scope(attribute_keyword< DescriptorT, ActorT > const & keyword,
                     const CharT * element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`keyword`	Attribute keyword

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename CharT>
  format_named_scope_actor< fallback_to_none, CharT, ActorT >
  format_named_scope(attribute_keyword< DescriptorT, ActorT > const & keyword,
                     std::basic_string< CharT > const & element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`keyword`	Attribute keyword

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename CharT>
  format_named_scope_actor< FallbackPolicyT, CharT, ActorT >
  format_named_scope(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & placeholder,
                     const CharT * element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`placeholder`	Attribute placeholder

Function template format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename CharT>
  format_named_scope_actor< FallbackPolicyT, CharT, ActorT >
  format_named_scope(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & placeholder,
                     std::basic_string< CharT > const & element_format);
```

Description

The function generates a manipulator node in a template expression. The manipulator must participate in a formatting expression (stream output or `format` placeholder filler).

Parameters:	`element_format`	Format string for a single named scope
	`placeholder`	Attribute placeholder

Function format_named_scope

boost::log::expressions::format_named_scope

Synopsis

```
// In header: <boost/log/expressions/formatters/named_scope.hpp>


template<typename... ArgsT>
  unspecified format_named_scope(attribute_name const & name,
                                 ArgsT...const & args);
template<typename DescriptorT, template< typename > class ActorT,
         typename... ArgsT>
  unspecified format_named_scope(attribute_keyword< DescriptorT, ActorT > const & keyword,
                                 ArgsT...const & args);
template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename... ArgsT>
  unspecified format_named_scope(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & place↵
holder,
                                 ArgsT...const & args);
```

Description

Formatter generator. Construct the named scope formatter with the specified formatting parameters.

Parameters:

`args` An set of named parameters. Supported parameters:

- `format` - A format string for named scopes. The string can contain "%n", "%f" and "%l" placeholders for the scope name, file and line number, respectively. This parameter is mandatory.
- `delimiter` - A string that is used to delimit the formatted scope names. Default: "->" or "<-", depending on the iteration direction.
- `incomplete_marker` - A string that is used to indicate that the list was printed incomplete because of depth limitation. Default: "...".
- `empty_marker` - A string that is output in case if the scope list is empty. Default: "", i.e. nothing is output.
- `iteration` - Iteration direction, see `scope_iteration_direction` enumeration. Default: forward.
- `depth` - Iteration depth. Default: unlimited.

`name` Attribute name

Header <boost/log/expressions/formatters/stream.hpp>

Andrey Semashev

24.07.2012
The header contains implementation of a stream placeholder in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef phoenix::expression::argument< 2 >::type stream_type;

      const stream_type stream;
    }
  }
}
```

Type definition stream_type

stream_type

Synopsis

```
// In header: <boost/log/expressions/formatters/stream.hpp>


typedef phoenix::expression::argument< 2 >::type stream_type;
```

Description

Stream placeholder type in formatter template expressions.

Global stream

boost::log::expressions::stream

Synopsis

```
// In header: <boost/log/expressions/formatters/stream.hpp>

const stream_type stream;
```

Description

Stream placeholder in formatter template expressions.

Header <boost/log/expressions/formatters/wrap_formatter.hpp>

Andrey Semashev

24.11.2012
The header contains a formatter function wrapper that enables third-party functions to participate in formatting expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename FunT, typename CharT,
               template< typename > class ActorT = phoenix::actor>
        class wrapped_formatter_actor;
      template<typename FunT, typename CharT> class wrapped_formatter_terminal;
      template<typename FunT> unspecified wrap_formatter(FunT const &);
      template<typename CharT, typename FunT>
        wrapped_formatter_actor< FunT, CharT > wrap_formatter(FunT const &);
    }
  }
}
```

Class template wrapped_formatter_actor

boost::log::expressions::wrapped_formatter_actor

Synopsis

```
// In header: <boost/log/expressions/formatters/wrap_formatter.hpp>

template<typename FunT, typename CharT,
         template< typename > class ActorT = phoenix::actor>
class wrapped_formatter_actor :
  public ActorT< wrapped_formatter_terminal< FunT, CharT > >
{
public:
  // types
  typedef CharT                                                 char_type;      // Character ↵
type.
  typedef FunT                                                function_type;  // Wrapped func↵
tion type.
  typedef wrapped_formatter_terminal< function_type, char_type > terminal_type;  // Base termin↵
al type.
  typedef ActorT< terminal_type >                               base_type;      // Base actor ↵
type.

  // construct/copy/destruct
  explicit wrapped_formatter_actor(base_type const &);

  // public member functions
  function_type const & get_function() const;
};
```

Description

Wrapped formatter function actor.

`wrapped_formatter_actor` public construct/copy/destruct

1. ```
   explicit wrapped_formatter_actor(base_type const & act);
   ```

   Initializing constructor.

`wrapped_formatter_actor` public member functions

1. ```
   function_type const & get_function() const;
   ```

   Returns: The wrapped function

Class template wrapped_formatter_terminal

boost::log::expressions::wrapped_formatter_terminal

Synopsis

```
// In header: <boost/log/expressions/formatters/wrap_formatter.hpp>

template<typename FunT, typename CharT>
class wrapped_formatter_terminal {
public:
  // types
  typedef void                                    _is_boost_log_terminal;  // Internal typedef ↵
for type categorization.
  typedef CharT                                   char_type;               // Character type.
  typedef std::basic_string< char_type >          string_type;             // String type.
  typedef basic_formatting_ostream< char_type > stream_type;             // Formatting stream ↵
type.
  typedef FunT                                  function_type;          // Wrapped function type.
  typedef string_type                           result_type;            // Formatter result type.

  // construct/copy/destruct
  explicit wrapped_formatter_terminal(function_type const &);
  wrapped_formatter_terminal(wrapped_formatter_terminal const &);

  // public member functions
  function_type const & get_function() const;
  template<typename ContextT> result_type operator()(ContextT const &);
  template<typename ContextT> result_type operator()(ContextT const &) const;
};
```

Description

Formatter function wrapper terminal.

`wrapped_formatter_terminal` public construct/copy/destruct

1. ```
   explicit wrapped_formatter_terminal(function_type const & fun);
   ```

   Initializing construction.

2. ```
   wrapped_formatter_terminal(wrapped_formatter_terminal const & that);
   ```

   Copy constructor.

`wrapped_formatter_terminal` public member functions

1. ```
   function_type const & get_function() const;
   ```

   Returns the wrapped function.

2. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx);
   ```

   Invokation operator.

3. ```
   template<typename ContextT> result_type operator()(ContextT const & ctx) const;
   ```

   Invokation operator.
   ```

Function template wrap_formatter

boost::log::expressions::wrap_formatter

Synopsis

```
// In header: <boost/log/expressions/formatters/wrap_formatter.hpp>


template<typename FunT> unspecified wrap_formatter(FunT const & fun);
```

Description

The function wraps a function object in order it to be able to participate in formatting expressions. The wrapped function object must be compatible with the following signature:

<preformatted> void (record_view const&, basic_formatting_ostream< CharT >&) </preformatted>

where `CharT` is the character type of the formatting expression.

Function template wrap_formatter

boost::log::expressions::wrap_formatter

Synopsis

```
// In header: <boost/log/expressions/formatters/wrap_formatter.hpp>


template<typename CharT, typename FunT>
  wrapped_formatter_actor< FunT, CharT > wrap_formatter(FunT const & fun);
```

Description

The function wraps a function object in order it to be able to participate in formatting expressions. The wrapped function object must be compatible with the following signature:

<preformatted> void (record_view const&, basic_formatting_ostream< CharT >&) </preformatted>

where `CharT` is the character type of the formatting expression.

Header <boost/log/expressions/formatters/xml_decorator.hpp>

Andrey Semashev

18.11.2012
The header contains implementation of a XML-style character decorator.

```
namespace boost {
  namespace log {
    namespace expressions {
      unspecified xml_decor;
      unspecified wxml_decor;
      template<typename CharT> unspecified make_xml_decor();
    }
  }
}
```

Global xml_decor

boost::log::expressions::xml_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/xml_decorator.hpp>

unspecified xml_decor;
```

Description

XML-style decorator generator object. The decorator replaces characters that have special meaning in XML documents with the corresponding decorated counterparts. The generator provides `operator[]` that can be used to construct the actual decorator. For example:

```
xml_decor[ attr< std::string >("MyAttr") ]
```

For wide-character formatting there is the similar `wxml_decor` decorator generator object.

Global wxml_decor

boost::log::expressions::wxml_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/xml_decorator.hpp>

unspecified wxml_decor;
```

Function template make_xml_decor

boost::log::expressions::make_xml_decor

Synopsis

```
// In header: <boost/log/expressions/formatters/xml_decorator.hpp>


template<typename CharT> unspecified make_xml_decor();
```

Description

The function creates an XML-style decorator generator for arbitrary character type.

Header <boost/log/expressions/is_keyword_descriptor.hpp>

Andrey Semashev

14.07.2012
The header contains attribute keyword descriptor detection trait.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename T, typename VoidT = void> struct is_keyword_descriptor;
      struct keyword_descriptor;
    }
  }
}
```

Struct template is_keyword_descriptor

boost::log::expressions::is_keyword_descriptor

Synopsis

```
// In header: <boost/log/expressions/is_keyword_descriptor.hpp>

template<typename T, typename VoidT = void>
struct is_keyword_descriptor : public false_ {
};
```

Description

The metafunction detects if the type `T` is a keyword descriptor

Struct keyword_descriptor

boost::log::expressions::keyword_descriptor

Synopsis

```
// In header: <boost/log/expressions/is_keyword_descriptor.hpp>


struct keyword_descriptor {
};
```

Description

Base class for keyword descriptors. All keyword descriptors must derive from this class to support the `is_keyword_descriptor` trait.

Header <boost/log/expressions/keyword.hpp>

Andrey Semashev

29.01.2012
The header contains attribute keyword declaration.

```


BOOST_LOG_ATTRIBUTE_KEYWORD_TYPE(keyword_, name_, value_type_)
BOOST_LOG_ATTRIBUTE_KEYWORD(keyword_, name_, value_type_)
```

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename DescriptorT, template< typename > class ActorT>
        struct attribute_keyword;
    }
  }
}
```

Struct template attribute_keyword

boost::log::expressions::attribute_keyword — This class implements an expression template keyword.

Synopsis

```
// In header: <boost/log/expressions/keyword.hpp>

template<typename DescriptorT, template< typename > class ActorT>
struct attribute_keyword {
  // types
  typedef attribute_keyword                                                    this_type; ↵
            // Self type.
  typedef DescriptorT                                                           ↵
 descriptor_type;       // Attribute descriptor type.
  typedef descriptor_type::value_type                                        value_type; ↵
           // Attribute value type.
  typedef attribute_actor< value_type, fallback_to_none, descriptor_type, ActorT >  or_none_res↵
ult_type;   // Expression with cached attribute name.
  typedef attribute_actor< value_type, fallback_to_throw, descriptor_type, ActorT > or_throw_res↵
ult_type;  // Expression with cached attribute name.

  // public static functions
  static attribute_name get_name();
  static or_none_result_type or_none();
  static or_throw_result_type or_throw();
  template<typename DefaultT>
    static attribute_actor< value_type, fallback_to_default< DefaultT >, descriptor_type, ActorT >
    or_default(DefaultT const &);
};
```

Description

This class implements an expression template keyword. It is used to start template expressions involving attribute values.

`attribute_keyword` public static functions

1. ```
   static attribute_name get_name();
   ```

   Returns attribute name.

2. ```
   static or_none_result_type or_none();
   ```

   Generates an expression that extracts the attribute value or a default value.

3. ```
   static or_throw_result_type or_throw();
   ```

   Generates an expression that extracts the attribute value or throws an exception.
   ```

4.
```
template<typename DefaultT>
  static attribute_actor< value_type, fallback_to_default< DefaultT >, descriptor_type, ActorT
 >
  or_default(DefaultT const & def_val);
```

Generates an expression that extracts the attribute value or a default value.

Macro BOOST_LOG_ATTRIBUTE_KEYWORD_TYPE

BOOST_LOG_ATTRIBUTE_KEYWORD_TYPE — The macro declares an attribute keyword type.

Synopsis

```
// In header: <boost/log/expressions/keyword.hpp>

BOOST_LOG_ATTRIBUTE_KEYWORD_TYPE(keyword_, name_, value_type_)
```

Description

The macro should be used at a namespace scope. It expands into an attribute keyword type definition, including the `tag` namespace and the keyword tag type within which has the following layout:

```
namespace tag
{
  struct keyword_ :
    public boost::log::expressions::keyword_descriptor
  {
    typedef value_type_ value_type;
    static boost::log::attribute_name get_name();
  };
}

typedef boost::log::expressions::attribute_keyword< tag::keyword_ > keyword_type;
```

The `get_name` method returns the attribute name.

Note

This macro only defines the type of the keyword. To also define the keyword object, use the `BOOST_LOG_ATTRIBUTE_KEYWORD` macro instead.

Parameters:		
	`keyword_`	Keyword name
	`name_`	Attribute name string
	`value_type_`	Attribute value type

Macro BOOST_LOG_ATTRIBUTE_KEYWORD

BOOST_LOG_ATTRIBUTE_KEYWORD — The macro declares an attribute keyword.

Synopsis

```
// In header: <boost/log/expressions/keyword.hpp>

BOOST_LOG_ATTRIBUTE_KEYWORD(keyword_, name_, value_type_)
```

Description

The macro provides definitions similar to `BOOST_LOG_ATTRIBUTE_KEYWORD_TYPE` and additionally defines the keyword object.

Parameters:	`keyword_`	Keyword name
	`name_`	Attribute name string
	`value_type_`	Attribute value type

Header <boost/log/expressions/keyword_fwd.hpp>

Andrey Semashev

29.01.2012
The header contains attribute keyword forward declaration.

Header <boost/log/expressions/message.hpp>

Andrey Semashev

13.07.2012
The header contains log message keyword declaration.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef attribute_keyword< tag::message > message_type;
      typedef attribute_keyword< tag::smessage > smessage_type;
      typedef attribute_keyword< tag::wmessage > wmessage_type;

      const message_type message;
      const smessage_type smessage;
      const wmessage_type wmessage;
      namespace tag {
        struct message;
        struct smessage;
        struct wmessage;
      }
    }
  }
}
```

Struct message

boost::log::expressions::tag::message

Synopsis

```
// In header: <boost/log/expressions/message.hpp>


struct message : public keyword_descriptor {
  // types
  typedef void                                        attribute_type;
  typedef mpl::vector2< std::string, std::wstring > value_type;

  // public static functions
  static attribute_name get_name();
};
```

Description

Generic log message attribute descriptor.

`message` public static functions

1. ```
   static attribute_name get_name();
   ```

Struct smessage

boost::log::expressions::tag::smessage

Synopsis

```
// In header: <boost/log/expressions/message.hpp>


struct smessage : public keyword_descriptor {
  // types
  typedef void        attribute_type;
  typedef std::string value_type;

  // public static functions
  static attribute_name get_name();
};
```

Description

Narrow character log message attribute descriptor.

`smessage` public static functions

1. ```
   static attribute_name get_name();
   ```

Struct wmessage

boost::log::expressions::tag::wmessage

Synopsis

```
// In header: <boost/log/expressions/message.hpp>


struct wmessage : public keyword_descriptor {
  // types
  typedef void         attribute_type;
  typedef std::wstring value_type;

  // public static functions
  static attribute_name get_name();
};
```

Description

Wide character log message attribute descriptor.

`wmessage` public static functions

1. ```
   static attribute_name get_name();
   ```

Type definition message_type

message_type

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

typedef attribute_keyword< tag::message > message_type;
```

Description

Generic message keyword type.

Type definition smessage_type

smessage_type

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

typedef attribute_keyword< tag::smessage > smessage_type;
```

Description

Narrow message keyword type.

Type definition wmessage_type

wmessage_type

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

typedef attribute_keyword< tag::wmessage > wmessage_type;
```

Description

Wide message keyword type.
```

Global message

boost::log::expressions::message

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

const message_type message;
```

Description

Generic message keyword.

Global smessage

boost::log::expressions::smessage

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

const smessage_type smessage;
```

Description

Narrow message keyword.

Global wmessage

boost::log::expressions::wmessage

Synopsis

```
// In header: <boost/log/expressions/message.hpp>

const wmessage_type wmessage;
```

Description

Wide message keyword.

Header <boost/log/expressions/predicates.hpp>

Andrey Semashev

29.01.2012
The header includes all template expression predicates.

Header <boost/log/expressions/predicates/begins_with.hpp>

Andrey Semashev

02.09.2012
The header contains implementation of a `begins_with` predicate in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef unspecified attribute_begins_with;
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename SubstringT>
        unspecified begins_with(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                                SubstringT const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename SubstringT>
        unspecified begins_with(attribute_keyword< DescriptorT, ActorT > const &,
                                SubstringT const &);
      template<typename T, typename SubstringT>
        unspecified begins_with(attribute_name const &, SubstringT const &);
    }
  }
}
```

Type definition attribute_begins_with

attribute_begins_with

Synopsis

```
// In header: <boost/log/expressions/predicates/begins_with.hpp>


typedef unspecified attribute_begins_with;
```

Description

The predicate checks if the attribute value begins with a substring. The attribute value is assumed to be of a string type.

Function template begins_with

boost::log::expressions::begins_with

Synopsis

```
// In header: <boost/log/expressions/predicates/begins_with.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename SubstringT>
  unspecified begins_with(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & attr,
                          SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, begins with the specified substring.

Function template begins_with

boost::log::expressions::begins_with

Synopsis

```
// In header: <boost/log/expressions/predicates/begins_with.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename SubstringT>
  unspecified begins_with(attribute_keyword< DescriptorT, ActorT > const &,
                          SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, begins with the specified substring.

Function template begins_with

boost::log::expressions::begins_with

Synopsis

```
// In header: <boost/log/expressions/predicates/begins_with.hpp>


template<typename T, typename SubstringT>
  unspecified begins_with(attribute_name const & name,
                          SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, begins with the specified substring.

Header <boost/log/expressions/predicates/channel_severity_filter.hpp>

Andrey Semashev

25.11.2012
The header contains implementation of a minimal severity per channel filter.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename ChannelT, typename SeverityT,
               typename ChannelFallbackT = fallback_to_none,
               typename SeverityFallbackT = fallback_to_none,
               typename ChannelOrderT = less,
               typename SeverityCompareT = greater_equal,
               typename AllocatorT = std::allocator< void >,
               template< typename > class ActorT = phoenix::actor>
        class channel_severity_filter_actor;
      template<typename ChannelT, typename SeverityT,
               typename ChannelFallbackT = fallback_to_none,
               typename SeverityFallbackT = fallback_to_none,
               typename ChannelOrderT = less,
               typename SeverityCompareT = greater_equal,
               typename AllocatorT = std::allocator< void > >
        class channel_severity_filter_terminal;
      template<typename ChannelT, typename SeverityT>
        channel_severity_filter_actor< ChannelT, SeverityT >
        channel_severity_filter(attribute_name const &,
                                attribute_name const &);
      template<typename SeverityT, typename ChannelDescriptorT,
               template< typename > class ActorT>
       channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, less, greater_equal, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_name const &);
      template<typename ChannelT, typename SeverityDescriptorT,
               template< typename > class ActorT>
       channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, less, greater_equal, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &);
      template<typename ChannelDescriptorT, typename SeverityDescriptorT,
               template< typename > class ActorT>
        channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Sever↵
ityDescriptorT::value_type, fallback_to_none, fallback_to_none, less, greater_equal, std::alloc↵
ator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &);
      template<typename SeverityT, typename ChannelT,
               typename ChannelFallbackT, typename ChannelTagT,
               template< typename > class ActorT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fall↵
back_to_none, less, greater_equal, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_name const &);
      template<typename ChannelT, typename SeverityT,
               typename SeverityFallbackT, typename SeverityTagT,
               template< typename > class ActorT>
        channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFall↵
backT, less, greater_equal, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &);
      template<typename ChannelT, typename ChannelFallbackT,
               typename ChannelTagT, typename SeverityT,
               typename SeverityFallbackT, typename SeverityTagT,
               template< typename > class ActorT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
backT, less, greater_equal, std::allocator< void >, ActorT >
```

```
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &);
      template<typename ChannelT, typename SeverityT,
               typename SeverityCompareT>
        channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, fall↵
back_to_none, less, SeverityCompareT >
        channel_severity_filter(attribute_name const &,
                                attribute_name const &,
                                SeverityCompareT const &);
      template<typename SeverityT, typename ChannelDescriptorT,
               template< typename > class ActorT, typename SeverityCompareT>
       channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_name const &,
                                SeverityCompareT const &);
      template<typename ChannelT, typename SeverityDescriptorT,
               template< typename > class ActorT, typename SeverityCompareT>
       channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &,
                                SeverityCompareT const &);
      template<typename ChannelDescriptorT, typename SeverityDescriptorT,
               template< typename > class ActorT, typename SeverityCompareT>
        channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Sever↵
ityDescriptorT::value_type, fallback_to_none, fallback_to_none, less, SeverityCompareT, std::al↵
locator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &,
                                SeverityCompareT const &);
      template<typename SeverityT, typename ChannelT,
               typename ChannelFallbackT, typename ChannelTagT,
               template< typename > class ActorT, typename SeverityCompareT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fall↵
back_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_name const &,
                                SeverityCompareT const &);
      template<typename ChannelT, typename SeverityT,
               typename SeverityFallbackT, typename SeverityTagT,
               template< typename > class ActorT, typename SeverityCompareT>
        channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFall↵
backT, less, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &,
                                SeverityCompareT const &);
      template<typename ChannelT, typename ChannelFallbackT,
               typename ChannelTagT, typename SeverityT,
               typename SeverityFallbackT, typename SeverityTagT,
               template< typename > class ActorT, typename SeverityCompareT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
backT, less, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &,
                                SeverityCompareT const &);
      template<typename ChannelT, typename SeverityT,
```

```
              typename SeverityCompareT, typename ChannelOrderT>
        channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, fall↵
back_to_none, ChannelOrderT, SeverityCompareT >
        channel_severity_filter(attribute_name const &,
                                attribute_name const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename SeverityT, typename ChannelDescriptorT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
       channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_name const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename ChannelT, typename SeverityDescriptorT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
       channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename ChannelDescriptorT, typename SeverityDescriptorT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
       channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Sever↵
ityDescriptorT::value_type, fallback_to_none, fallback_to_none, ChannelOrderT, Severity↵
CompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const &,
                                attribute_keyword< SeverityDescriptorT, ActorT > const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename SeverityT, typename ChannelT,
              typename ChannelFallbackT, typename ChannelTagT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fall↵
back_to_none, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_name const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename ChannelT, typename SeverityT,
              typename SeverityFallbackT, typename SeverityTagT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
        channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFall↵
backT, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_name const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
      template<typename ChannelT, typename ChannelFallbackT,
              typename ChannelTagT, typename SeverityT,
              typename SeverityFallbackT, typename SeverityTagT,
              template< typename > class ActorT, typename SeverityCompareT,
              typename ChannelOrderT>
        channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
```

```
backT, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
        channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const &,
                                attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const &,
                                SeverityCompareT const &,
                                ChannelOrderT const &);
    }
  }
}
```

Class template channel_severity_filter_actor

boost::log::expressions::channel_severity_filter_actor

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>

template<typename ChannelT, typename SeverityT,
         typename ChannelFallbackT = fallback_to_none,
         typename SeverityFallbackT = fallback_to_none,
         typename ChannelOrderT = less,
         typename SeverityCompareT = greater_equal,
         typename AllocatorT = std::allocator< void >,
         template< typename > class ActorT = phoenix::actor>
class channel_severity_filter_actor : public ActorT< channel_severity_filter_terminal< ChannelT, ↵
SeverityT, ChannelFallbackT, SeverityFallbackT, ChannelOrderT, SeverityCompareT, AllocatorT > >
{
public:
  // types
  typedef channel_severity_filter_terminal< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
backT, ChannelOrderT, SeverityCompareT, AllocatorT > terminal_type;            // Terminal type.
  typedef ActorT< terminal_type >                                                          ↵
                                                  base_type;                  // Base actor ↵
type.
  typedef terminal_type::channel_value_type                                                ↵
                                                   channel_value_type;         // Channel at↵
tribute value type.
  typedef terminal_type::channel_fallback_policy                                           ↵
                                                 channel_fallback_policy;    // Channel fall↵
back policy.
  typedef terminal_type::severity_value_type                                               ↵
                                                   severity_value_type;        // Severity ↵
level attribute value type.
  typedef terminal_type::severity_fallback_policy                                          ↵
                                                   severity_fallback_policy;  // Severity ↵
level fallback policy.

  // member classes/structs/unions

  // An auxiliary pseudo-reference to implement insertion through subscript
  // operator.

  class subscript_result {
  public:
    // construct/copy/destruct
    subscript_result(channel_severity_filter_actor &,
                     channel_value_type const &);
    void operator=(severity_value_type const &);
```

```
  };

  // construct/copy/destruct
  explicit channel_severity_filter_actor(base_type const &);
  channel_severity_filter_actor(channel_severity_filter_actor const &);

  // public member functions
  this_type & set_default(bool);
  this_type & add(channel_value_type const &, severity_value_type const &);
  subscript_result operator[](channel_value_type const &);
};
```

Description

`channel_severity_filter_actor` public construct/copy/destruct

1. ```
   explicit channel_severity_filter_actor(base_type const & act);
   ```

   Initializing constructor.

2. ```
   channel_severity_filter_actor(channel_severity_filter_actor const & that);
   ```

   Copy constructor.

`channel_severity_filter_actor` public member functions

1. ```
   this_type & set_default(bool def);
   ```

   Sets the default function result.

2. ```
   this_type & add(channel_value_type const & channel,
                   severity_value_type const & severity);
   ```

   Adds a new element to the mapping.

3. ```
   subscript_result operator[](channel_value_type const & channel);
   ```

   Alternative interface for adding a new element to the mapping.

Class subscript_result

boost::log::expressions::channel_severity_filter_actor::subscript_result — An auxiliary pseudo-reference to implement insertion through subscript operator.
   ```

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>



// An auxiliary pseudo-reference to implement insertion through subscript
// operator.

class subscript_result {
public:
  // construct/copy/destruct
  subscript_result(channel_severity_filter_actor &,
                   channel_value_type const &);
  void operator=(severity_value_type const &);
};
```

Description

`subscript_result` public construct/copy/destruct

1.
```
subscript_result(channel_severity_filter_actor & owner,
                 channel_value_type const & channel);
```

2.
```
void operator=(severity_value_type const & severity);
```

Class template channel_severity_filter_terminal

boost::log::expressions::channel_severity_filter_terminal

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>

template<typename ChannelT, typename SeverityT,
         typename ChannelFallbackT = fallback_to_none,
         typename SeverityFallbackT = fallback_to_none,
         typename ChannelOrderT = less,
         typename SeverityCompareT = greater_equal,
         typename AllocatorT = std::allocator< void > >
class channel_severity_filter_terminal {
public:
  // types
  typedef void              _is_boost_log_terminal;    // Internal typedef for type categoriza↵
tion.
  typedef bool              result_type;               // Function result type.
  typedef ChannelT          channel_value_type;        // Channel attribute value type.
  typedef ChannelFallbackT  channel_fallback_policy;   // Channel fallback policy.
  typedef SeverityT         severity_value_type;       // Severity level attribute value type.
  typedef SeverityFallbackT severity_fallback_policy;  // Severity level fallback policy.

  // member classes/structs/unions

  // Channel visitor.
  template<typename ArgT>
  struct channel_visitor {
    // types
    typedef void result_type;

    // construct/copy/destruct
    channel_visitor(channel_severity_filter_terminal const &, ArgT, bool &);

    // public member functions
    result_type operator()(channel_value_type const &) const;
  };

  // Severity level visitor.

  struct severity_visitor {
    // types
    typedef void result_type;

    // construct/copy/destruct
    severity_visitor(channel_severity_filter_terminal const &,
                     severity_value_type const &, bool &);

    // public member functions
    result_type operator()(severity_value_type const &) const;
  };

  // construct/copy/destruct
  channel_severity_filter_terminal(attribute_name const &,
                                   attribute_name const &,
                                 channel_fallback_policy const & = channel_fallback_policy(),
                               severity_fallback_policy const & = severity_fallback_policy(),
                                   ChannelOrderT const & = ChannelOrderT(),
                                   SeverityCompareT const & = SeverityCompareT());

  // public member functions
  void add(channel_value_type const &, severity_value_type const &);
  void set_default(bool);
  template<typename ContextT> result_type operator()(ContextT const &) const;
```

```
  // private member functions
  template<typename ArgT>
    void visit_channel(channel_value_type const &, ArgT const &, bool &) const;
  void visit_severity(severity_value_type const &,
                      severity_value_type const &, bool &) const;
};
```

Description

`channel_severity_filter_terminal` public construct/copy/destruct

1.
```
channel_severity_filter_terminal(attribute_name const & channel_name,
                                 attribute_name const & severity_name,
                               channel_fallback_policy const & channel_fallback = channel_fall↵
back_policy(),
                                 severity_fallback_policy const & severity_fallback = sever↵
ity_fallback_policy(),
                                 ChannelOrderT const & channel_order = ChannelOrderT(),
                                 SeverityCompareT const & severity_compare = Severity↵
CompareT());
```

Initializing constructor.

`channel_severity_filter_terminal` public member functions

1.
```
void add(channel_value_type const & channel,
         severity_value_type const & severity);
```

Adds a new element to the mapping.

2.
```
void set_default(bool def);
```

Sets the default result of the predicate.

3.
```
template<typename ContextT> result_type operator()(ContextT const & ctx) const;
```

Invokation operator.

`channel_severity_filter_terminal` private member functions

1.
```
template<typename ArgT>
  void visit_channel(channel_value_type const & channel, ArgT const & arg,
                     bool & res) const;
```

Visits channel name.

2.
```
void visit_severity(severity_value_type const & left,
                    severity_value_type const & right, bool & res) const;
```

Visits severity level.

Struct template channel_visitor

boost::log::expressions::channel_severity_filter_terminal::channel_visitor — Channel visitor.

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>



// Channel visitor.
template<typename ArgT>
struct channel_visitor {
  // types
  typedef void result_type;

  // construct/copy/destruct
  channel_visitor(channel_severity_filter_terminal const &, ArgT, bool &);

  // public member functions
  result_type operator()(channel_value_type const &) const;
};
```

Description

`channel_visitor` public construct/copy/destruct

1. ```
   channel_visitor(channel_severity_filter_terminal const & self, ArgT arg,
                   bool & res);
   ```

`channel_visitor` public member functions

1. ```
   result_type operator()(channel_value_type const & channel) const;
   ```

Struct severity_visitor

boost::log::expressions::channel_severity_filter_terminal::severity_visitor — Severity level visitor.

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>



// Severity level visitor.

struct severity_visitor {
  // types
  typedef void result_type;

  // construct/copy/destruct
  severity_visitor(channel_severity_filter_terminal const &,
                   severity_value_type const &, bool &);

  // public member functions
  result_type operator()(severity_value_type const &) const;
};
```

Description

`severity_visitor` public construct/copy/destruct

1.
```
severity_visitor(channel_severity_filter_terminal const & self,
                 severity_value_type const & severity, bool & res);
```

`severity_visitor` public member functions

1.
```
result_type operator()(severity_value_type const & severity) const;
```

Function channel_severity_filter

boost::log::expressions::channel_severity_filter

Synopsis

```
// In header: <boost/log/expressions/predicates/channel_severity_filter.hpp>


template<typename ChannelT, typename SeverityT>
  channel_severity_filter_actor< ChannelT, SeverityT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_name const & severity_name);
template<typename SeverityT, typename ChannelDescriptorT,
         template< typename > class ActorT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, less, greater_equal, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_name const & severity_name);
template<typename ChannelT, typename SeverityDescriptorT,
         template< typename > class ActorT>
  channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, less, greater_equal, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword);
template<typename ChannelDescriptorT, typename SeverityDescriptorT,
         template< typename > class ActorT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Severity↵
DescriptorT::value_type, fallback_to_none, fallback_to_none, less, greater_equal, std::allocat↵
or< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword);
template<typename SeverityT, typename ChannelT, typename ChannelFallbackT,
         typename ChannelTagT, template< typename > class ActorT>
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fall↵
back_to_none, less, greater_equal, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_name const & severity_name);
template<typename ChannelT, typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT>
  channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFall↵
backT, less, greater_equal, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
```

```
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder);
template<typename ChannelT, typename ChannelFallbackT, typename ChannelTagT,
         typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT>
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
backT, less, greater_equal, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder);
template<typename ChannelT, typename SeverityT, typename SeverityCompareT>
  channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, fall↵
back_to_none, less, SeverityCompareT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare);
template<typename SeverityT, typename ChannelDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare);
template<typename ChannelT, typename SeverityDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT>
  channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword,
                          SeverityCompareT const & severity_compare);
template<typename ChannelDescriptorT, typename SeverityDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Severity↵
DescriptorT::value_type, fallback_to_none, fallback_to_none, less, SeverityCompareT, std::alloc↵
ator< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword,
                          SeverityCompareT const & severity_compare);
template<typename SeverityT, typename ChannelT, typename ChannelFallbackT,
         typename ChannelTagT, template< typename > class ActorT,
         typename SeverityCompareT>
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fall↵
back_to_none, less, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare);
template<typename ChannelT, typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT,
         typename SeverityCompareT>
  channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFall↵
backT, less, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder,
                          SeverityCompareT const & severity_compare);
template<typename ChannelT, typename ChannelFallbackT, typename ChannelTagT,
         typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT,
         typename SeverityCompareT>
```

```
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFall↵
backT, less, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder,
                          SeverityCompareT const & severity_compare);
template<typename ChannelT, typename SeverityT, typename SeverityCompareT,
         typename ChannelOrderT>
  channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, fallback_to_none, Chan↵
nelOrderT, SeverityCompareT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename SeverityT, typename ChannelDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT,
         typename ChannelOrderT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, SeverityT, fall↵
back_to_none, fallback_to_none, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename ChannelT, typename SeverityDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT,
         typename ChannelOrderT>
  channel_severity_filter_actor< ChannelT, typename SeverityDescriptorT::value_type, fall↵
back_to_none, fallback_to_none, ChannelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename ChannelDescriptorT, typename SeverityDescriptorT,
         template< typename > class ActorT, typename SeverityCompareT,
         typename ChannelOrderT>
  channel_severity_filter_actor< typename ChannelDescriptorT::value_type, typename Severity↵
DescriptorT::value_type, fallback_to_none, fallback_to_none, ChannelOrderT, Severity↵
CompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_keyword< ChannelDescriptorT, ActorT > const & channel_keyword,
                          attribute_keyword< SeverityDescriptorT, ActorT > const & sever↵
ity_keyword,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename SeverityT, typename ChannelT, typename ChannelFallbackT,
         typename ChannelTagT, template< typename > class ActorT,
         typename SeverityCompareT, typename ChannelOrderT>
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, fallback_to_none, Chan↵
nelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_name const & severity_name,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename ChannelT, typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT,
         typename SeverityCompareT, typename ChannelOrderT>
  channel_severity_filter_actor< ChannelT, SeverityT, fallback_to_none, SeverityFallbackT, Chan↵
nelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_name const & channel_name,
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder,
```

```
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
template<typename ChannelT, typename ChannelFallbackT, typename ChannelTagT,
         typename SeverityT, typename SeverityFallbackT,
         typename SeverityTagT, template< typename > class ActorT,
         typename SeverityCompareT, typename ChannelOrderT>
  channel_severity_filter_actor< ChannelT, SeverityT, ChannelFallbackT, SeverityFallbackT, Chan↵
nelOrderT, SeverityCompareT, std::allocator< void >, ActorT >
  channel_severity_filter(attribute_actor< ChannelT, ChannelFallbackT, Chan↵
nelTagT, ActorT > const & channel_placeholder,
                          attribute_actor< SeverityT, SeverityFallbackT, Sever↵
ityTagT, ActorT > const & severity_placeholder,
                          SeverityCompareT const & severity_compare,
                          ChannelOrderT const & channel_order);
```

Description

The function generates a filtering predicate that checks the severity levels of log records in different channels. The predicate will return `true` if the record severity level is not less than the threshold for the channel the record belongs to.

Header <boost/log/expressions/predicates/contains.hpp>

Andrey Semashev

02.09.2012
The header contains implementation of a `contains` predicate in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef unspecified attribute_contains;
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename SubstringT>
        unspecified contains(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                             SubstringT const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename SubstringT>
        unspecified contains(attribute_keyword< DescriptorT, ActorT > const &,
                             SubstringT const &);
      template<typename T, typename SubstringT>
        unspecified contains(attribute_name const &, SubstringT const &);
    }
  }
}
```

Type definition attribute_contains

attribute_contains

Synopsis

```
// In header: <boost/log/expressions/predicates/contains.hpp>


typedef unspecified attribute_contains;
```

Description

The predicate checks if the attribute value contains a substring. The attribute value is assumed to be of a string type.

Function template contains

boost::log::expressions::contains

Synopsis

```
// In header: <boost/log/expressions/predicates/contains.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename SubstringT>
  unspecified contains(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & attr,
                       SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, contains the specified substring.

Function template contains

boost::log::expressions::contains

Synopsis

```
// In header: <boost/log/expressions/predicates/contains.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename SubstringT>
  unspecified contains(attribute_keyword< DescriptorT, ActorT > const &,
                       SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, contains the specified substring.

Function template contains

boost::log::expressions::contains

Synopsis

```
// In header: <boost/log/expressions/predicates/contains.hpp>


template<typename T, typename SubstringT>
  unspecified contains(attribute_name const & name,
                       SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, contains the specified substring.

Header <boost/log/expressions/predicates/ends_with.hpp>

Andrey Semashev

02.09.2012
The header contains implementation of a `ends_with` predicate in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef unspecified attribute_ends_with;
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename SubstringT>
        unspecified ends_with(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                              SubstringT const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename SubstringT>
        unspecified ends_with(attribute_keyword< DescriptorT, ActorT > const &,
                              SubstringT const &);
      template<typename T, typename SubstringT>
        unspecified ends_with(attribute_name const &, SubstringT const &);
    }
  }
}
```

Type definition attribute_ends_with

attribute_ends_with

Synopsis

```
// In header: <boost/log/expressions/predicates/ends_with.hpp>


typedef unspecified attribute_ends_with;
```

Description

The predicate checks if the attribute value ends with a substring. The attribute value is assumed to be of a string type.

Function template ends_with

boost::log::expressions::ends_with

Synopsis

```
// In header: <boost/log/expressions/predicates/ends_with.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename SubstringT>
  unspecified ends_with(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & attr,
                        SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, ends with the specified substring.

Function template ends_with

boost::log::expressions::ends_with

Synopsis

```
// In header: <boost/log/expressions/predicates/ends_with.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename SubstringT>
  unspecified ends_with(attribute_keyword< DescriptorT, ActorT > const &,
                        SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, ends with the specified substring.

Function template ends_with

boost::log::expressions::ends_with

Synopsis

```
// In header: <boost/log/expressions/predicates/ends_with.hpp>


template<typename T, typename SubstringT>
  unspecified ends_with(attribute_name const & name,
                        SubstringT const & substring);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, ends with the specified substring.

Header <boost/log/expressions/predicates/has_attr.hpp>

Andrey Semashev

23.07.2012
The header contains implementation of a generic attribute presence checker in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename T> class has_attribute;

      template<> class has_attribute<void>;
      template<typename AttributeValueT>
        unspecified has_attr(attribute_name const &);
      template<typename DescriptorT, template< typename > class ActorT>
        unspecified has_attr(attribute_keyword< DescriptorT, ActorT > const &);
    }
  }
}
```

Class template has_attribute

boost::log::expressions::has_attribute

Synopsis

```
// In header: <boost/log/expressions/predicates/has_attr.hpp>

template<typename T>
class has_attribute {
public:
  // types
  typedef bool result_type;  // Function result_type.
  typedef T    value_type;   // Expected attribute value type.

  // construct/copy/destruct
  explicit has_attribute(attribute_name const &);

  // public member functions
  template<typename ArgT> result_type operator()(ArgT const &) const;
};
```

Description

An attribute value presence checker.

`has_attribute` public construct/copy/destruct

1. ```
   explicit has_attribute(attribute_name const & name);
   ```

   Initializing constructor

   Parameters: `name` Attribute name

`has_attribute` public member functions

1. ```
   template<typename ArgT> result_type operator()(ArgT const & arg) const;
   ```

   Checking operator

   Parameters: `arg` A set of attribute values or a log record

Returns: `true` if the log record contains the sought attribute value, `false` otherwise

Specializations

- Class has_attribute<void>

Class has_attribute<void>

boost::log::expressions::has_attribute<void>

Synopsis

```
// In header: <boost/log/expressions/predicates/has_attr.hpp>


class has_attribute<void> {
public:
  // types
  typedef bool result_type;  // Function result_type.
  typedef void value_type;   // Expected attribute value type.

  // construct/copy/destruct
  explicit has_attribute(attribute_name const &);

  // public member functions
  result_type operator()(attribute_value_set const &) const;
  result_type operator()(boost::log::record_view const &) const;
};
```

Description

An attribute value presence checker. This specialization does not check the type of the attribute value.

`has_attribute` public construct/copy/destruct

1. ```
   explicit has_attribute(attribute_name const & name);
   ```

   Initializing constructor

   Parameters: `name` Attribute name

`has_attribute` public member functions

1. ```
   result_type operator()(attribute_value_set const & attrs) const;
   ```

   Checking operator

   Parameters: `attrs` A set of attribute values
   Returns: `true` if the log record contains the sought attribute value, `false` otherwise

2. ```
   result_type operator()(boost::log::record_view const & rec) const;
   ```

   Checking operator

   Parameters: `rec` A log record
   Returns: `true` if the log record contains the sought attribute value, `false` otherwise
   ```

Function template has_attr

boost::log::expressions::has_attr

Synopsis

```
// In header: <boost/log/expressions/predicates/has_attr.hpp>


template<typename AttributeValueT>
  unspecified has_attr(attribute_name const & name);
```

Description

The function generates a terminal node in a template expression. The node will check for the attribute value presence in a log record. The node will also check that the attribute value has the specified type, if present.

The function generates a terminal node in a template expression. The node will check for the attribute value presence in a log record.

Function template has_attr

boost::log::expressions::has_attr

Synopsis

```
// In header: <boost/log/expressions/predicates/has_attr.hpp>


template<typename DescriptorT, template< typename > class ActorT>
  unspecified has_attr(attribute_keyword< DescriptorT, ActorT > const &);
```

Description

The function generates a terminal node in a template expression. The node will check for the attribute value presence in a log record. The node will also check that the attribute value has the specified type, if present.

Header <boost/log/expressions/predicates/is_debugger_present.hpp>

Andrey Semashev

05.12.2012
The header contains implementation of the `is_debugger_present` predicate in template expressions.

Header <boost/log/expressions/predicates/is_in_range.hpp>

Andrey Semashev

02.09.2012
The header contains implementation of an `is_in_range` predicate in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef unspecified attribute_is_in_range;
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename BoundaryT>
        unspecified is_in_range(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                                BoundaryT const &, BoundaryT const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename BoundaryT>
        unspecified is_in_range(attribute_keyword< DescriptorT, ActorT > const &,
                                BoundaryT const &, BoundaryT const &);
      template<typename T, typename BoundaryT>
        unspecified is_in_range(attribute_name const &, BoundaryT const &,
                                BoundaryT const &);
    }
  }
}
```

Type definition attribute_is_in_range

attribute_is_in_range

Synopsis

```
// In header: <boost/log/expressions/predicates/is_in_range.hpp>


typedef unspecified attribute_is_in_range;
```

Description

The predicate checks if the attribute value contains a substring. The attribute value is assumed to be of a string type.

Function template is_in_range

boost::log::expressions::is_in_range

Synopsis

```
// In header: <boost/log/expressions/predicates/is_in_range.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename BoundaryT>
  unspecified is_in_range(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & attr,
                          BoundaryT const & least, BoundaryT const & most);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value is in the specified range. The range must be half-open, that is the predicate will be equivalent to `least <= attr < most`.

Function template is_in_range

boost::log::expressions::is_in_range

Synopsis

```
// In header: <boost/log/expressions/predicates/is_in_range.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename BoundaryT>
  unspecified is_in_range(attribute_keyword< DescriptorT, ActorT > const &,
                          BoundaryT const & least, BoundaryT const & most);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value is in the specified range. The range must be half-open, that is the predicate will be equivalent to `least <= attr < most`.

Function template is_in_range

boost::log::expressions::is_in_range

Synopsis

```
// In header: <boost/log/expressions/predicates/is_in_range.hpp>


template<typename T, typename BoundaryT>
  unspecified is_in_range(attribute_name const & name,
                          BoundaryT const & least, BoundaryT const & most);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value is in the specified range. The range must be half-open, that is the predicate will be equivalent to `least <= attr < most`.

Header <boost/log/expressions/predicates/matches.hpp>

Andrey Semashev

02.09.2012
The header contains implementation of a `matches` predicate in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      template<typename T, typename RegexT,
               typename FallbackPolicyT = fallback_to_none>
        class attribute_matches;
      template<typename T, typename FallbackPolicyT, typename TagT,
               template< typename > class ActorT, typename RegexT>
        unspecified matches(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const &,
                            RegexT const &);
      template<typename DescriptorT, template< typename > class ActorT,
               typename RegexT>
        unspecified matches(attribute_keyword< DescriptorT, ActorT > const &,
                            RegexT const &);
      template<typename T, typename RegexT>
        unspecified matches(attribute_name const &, RegexT const &);
    }
  }
}
```

Class template attribute_matches

boost::log::expressions::attribute_matches

Synopsis

```
// In header: <boost/log/expressions/predicates/matches.hpp>

template<typename T, typename RegexT,
         typename FallbackPolicyT = fallback_to_none>
class attribute_matches {
public:
  // construct/copy/destruct
  attribute_matches(attribute_name const &, RegexT const &);
  template<typename U>
    attribute_matches(attribute_name const &, RegexT const &, U const &);
};
```

Description

The predicate checks if the attribute value matches a regular expression. The attribute value is assumed to be of a string type.

`attribute_matches` public construct/copy/destruct

1. ```
   attribute_matches(attribute_name const & name, RegexT const & rex);
   ```

   Initializing constructor

   Parameters: `name` Attribute name
   `rex` The regular expression to match the attribute value against

2. ```
   template<typename U>
     attribute_matches(attribute_name const & name, RegexT const & rex,
                       U const & arg);
   ```

   Initializing constructor

   Parameters: `arg` Additional parameter for the fallback policy

`name`	Attribute name
`rex`	The regular expression to match the attribute value against

Function template matches

boost::log::expressions::matches

Synopsis

```
// In header: <boost/log/expressions/predicates/matches.hpp>


template<typename T, typename FallbackPolicyT, typename TagT,
         template< typename > class ActorT, typename RegexT>
  unspecified matches(attribute_actor< T, FallbackPolicyT, TagT, ActorT > const & attr,
                      RegexT const & rex);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, matches the specified regular expression.

Function template matches

boost::log::expressions::matches

Synopsis

```
// In header: <boost/log/expressions/predicates/matches.hpp>


template<typename DescriptorT, template< typename > class ActorT,
         typename RegexT>
  unspecified matches(attribute_keyword< DescriptorT, ActorT > const &,
                      RegexT const & rex);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, matches the specified regular expression.

Function template matches

boost::log::expressions::matches

Synopsis

```
// In header: <boost/log/expressions/predicates/matches.hpp>


template<typename T, typename RegexT>
  unspecified matches(attribute_name const & name, RegexT const & rex);
```

Description

The function generates a terminal node in a template expression. The node will check if the attribute value, which is assumed to be a string, matches the specified regular expression.

Header <boost/log/expressions/record.hpp>

Andrey Semashev

25.07.2012
The header contains implementation of a log record placeholder in template expressions.

```
namespace boost {
  namespace log {
    namespace expressions {
      typedef phoenix::expression::argument< 1 >::type record_type;

      const record_type record;
    }
  }
}
```

Type definition record_type

record_type

Synopsis

```
// In header: <boost/log/expressions/record.hpp>


typedef phoenix::expression::argument< 1 >::type record_type;
```

Description

Log record placeholder type in formatter template expressions.

Global record

boost::log::expressions::record

Synopsis

```
// In header: <boost/log/expressions/record.hpp>

const record_type record;
```

Description

Log record placeholder in formatter template expressions.

Logging sources

Header <boost/log/sources/basic_logger.hpp>

Andrey Semashev

08.03.2007
The header contains implementation of a base class for loggers. Convenience macros for defining custom loggers are also provided.

```
BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS(class_type)
BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS_TEMPLATE(class_type)
BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT(class_type)
BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT_TEMPLATE(class_type)
BOOST_LOG_FORWARD_LOGGER_MEMBERS(class_type)
BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE(class_type)
BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, char_type, base_seq, threading)
BOOST_LOG_DECLARE_LOGGER(type_name, base_seq)
BOOST_LOG_DECLARE_LOGGER_MT(type_name, base_seq)
BOOST_LOG_DECLARE_WLOGGER(type_name, base_seq)
BOOST_LOG_DECLARE_WLOGGER_MT(type_name, base_seq)
```

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename CharT, typename FinalT, typename ThreadingModelT,
               typename FeaturesT>
        class basic_composite_logger;

      template<typename CharT, typename FinalT, typename FeaturesT>
        class basic_composite_logger<CharT, FinalT, single_thread_model, FeaturesT>;

      template<typename CharT, typename FinalT, typename ThreadingModelT>
        class basic_logger;
      template<typename CharT, typename FinalT, typename ThreadingModelT>
        void swap(basic_logger< CharT, FinalT, ThreadingModelT > &,
                  basic_logger< CharT, FinalT, ThreadingModelT > &);
    }
  }
}
```

Class template basic_composite_logger

boost::log::sources::basic_composite_logger — A composite logger that inherits a number of features.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

template<typename CharT, typename FinalT, typename ThreadingModelT,
         typename FeaturesT>
class basic_composite_logger {
public:
  // types
  typedef base_type::threading_model threading_model;  // Threading model being used.

  // construct/copy/destruct
  basic_composite_logger();
  basic_composite_logger(basic_composite_logger const &);
  basic_composite_logger(logger_base &&);
  template<typename ArgsT> explicit basic_composite_logger(ArgsT const &);

  // public member functions
  std::pair< attribute_set::iterator, bool >
  add_attribute(attribute_name const &, attribute const &);
  void remove_attribute(attribute_set::iterator);
  void remove_all_attributes();
  attribute_set get_attributes() const;
  void set_attributes(attribute_set const &);
  record open_record();
  template<typename ArgsT> record open_record(ArgsT const &);
  void push_record(record &&);
  void swap(basic_composite_logger &);

  // protected member functions
  FinalT & assign(FinalT const &);
};
```

Description

The composite logger is a helper class that simplifies feature composition into the final logger. The user's logger class is expected to derive from the composite logger class, instantiated with the character type, the user's logger class, the threading model and the list of the required features. The former three parameters are passed to the `basic_logger` class template. The feature list must be an MPL type sequence, where each element is a unary MPL metafunction class, that upon applying on its argument results in a logging feature class that derives from the argument. Every logger feature provided by the library can participate in the feature list.

`basic_composite_logger` public construct/copy/destruct

1. ```
   basic_composite_logger();
   ```

   Default constructor (default-constructs all features)

2. ```
   basic_composite_logger(basic_composite_logger const & that);
   ```

   Copy constructor

3. ```
   basic_composite_logger(logger_base && that);
   ```

   Move constructor

4. ```
   template<typename ArgsT> explicit basic_composite_logger(ArgsT const & args);
   ```

Constructor with named parameters

`basic_composite_logger` public member functions

1.
```
std::pair< attribute_set::iterator, bool >
add_attribute(attribute_name const & name, attribute const & attr);
```

The method adds an attribute to the source-specific attribute set. The attribute will be implicitly added to every log record made with the current logger.

Parameters: `attr` The attribute factory.
`name` The attribute name.
Returns: A pair of values. If the second member is `true`, then the attribute is added and the first member points to the attribute. Otherwise the attribute was not added and the first member points to the attribute that prevents addition.

2.
```
void remove_attribute(attribute_set::iterator it);
```

The method removes an attribute from the source-specific attribute set.

Parameters: `it` Iterator to the previously added attribute.
Requires: The attribute was added with the add_attribute call for this instance of the logger.
Postconditions: The attribute is no longer registered as a source-specific attribute for this logger. The iterator is invalidated after removal.

3.
```
void remove_all_attributes();
```

The method removes all attributes from the logger. All iterators and references to the removed attributes are invalidated.

4.
```
attribute_set get_attributes() const;
```

The method retrieves a copy of a set with all attributes from the logger.

Returns: The copy of the attribute set. Attributes are shallow-copied.

5.
```
void set_attributes(attribute_set const & attrs);
```

The method installs the whole attribute set into the logger. All iterators and references to elements of the previous set are invalidated. Iterators to the *attrs* set are not valid to be used with the logger (that is, the logger owns a copy of *attrs* after completion).

Parameters: `attrs` The set of attributes to install into the logger. Attributes are shallow-copied.

6.
```
record open_record();
```

The method opens a new log record in the logging core.

Returns: A valid record handle if the logging record is opened successfully, an invalid handle otherwise.

7.
```
template<typename ArgsT> record open_record(ArgsT const & args);
```

The method opens a new log record in the logging core.

Parameters: `args` A set of additional named arguments. The parameter is ignored.
Returns: A valid record handle if the logging record is opened successfully, an invalid handle otherwise.

8. ```
   void push_record(record && rec);
   ```

   The method pushes the constructed message to the logging core

   Parameters: `rec` The log record with the formatted message

9. ```
   void swap(basic_composite_logger & that);
   ```

   Thread-safe implementation of swap

`basic_composite_logger` protected member functions

1. ```
   FinalT & assign(FinalT const & that);
   ```

   Assignment for the final class. Threadsafe, provides strong exception guarantee.

Specializations

- Class template basic_composite_logger<CharT, FinalT, single_thread_model, FeaturesT>

Class template basic_composite_logger<CharT, FinalT, single_thread_model, FeaturesT>

boost::log::sources::basic_composite_logger<CharT, FinalT, single_thread_model, FeaturesT> — An optimized composite logger version with no multithreading support.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

template<typename CharT, typename FinalT, typename FeaturesT>
class basic_composite_logger<CharT, FinalT, single_thread_model, FeaturesT> {
public:
  // types
  typedef base_type::threading_model threading_model;

  // construct/copy/destruct
  basic_composite_logger();
  basic_composite_logger(basic_composite_logger const &);
  basic_composite_logger(logger_base &&);
  template<typename ArgsT> explicit basic_composite_logger(ArgsT const &);

  // public member functions
  std::pair< attribute_set::iterator, bool >
  add_attribute(attribute_name const &, attribute const &);
  void remove_attribute(attribute_set::iterator);
  void remove_all_attributes();
  attribute_set get_attributes() const;
  void set_attributes(attribute_set const &);
  record open_record();
  template<typename ArgsT> record open_record(ArgsT const &);
  void push_record(record &&);
  void swap(basic_composite_logger &);

  // protected member functions
  FinalT & assign(FinalT);
};
```
```

Description

`basic_composite_logger` public construct/copy/destruct

1.
```
basic_composite_logger();
```

2.
```
basic_composite_logger(basic_composite_logger const & that);
```

3.
```
basic_composite_logger(logger_base && that);
```

4.
```
template<typename ArgsT> explicit basic_composite_logger(ArgsT const & args);
```

`basic_composite_logger` public member functions

1.
```
std::pair< attribute_set::iterator, bool >
add_attribute(attribute_name const & name, attribute const & attr);
```

2.
```
void remove_attribute(attribute_set::iterator it);
```

3.
```
void remove_all_attributes();
```

4.
```
attribute_set get_attributes() const;
```

5.
```
void set_attributes(attribute_set const & attrs);
```

6.
```
record open_record();
```

7.
```
template<typename ArgsT> record open_record(ArgsT const & args);
```

8.
```
void push_record(record && rec);
```

9.
```
void swap(basic_composite_logger & that);
```

`basic_composite_logger` protected member functions

1.
```
FinalT & assign(FinalT that);
```

Class template basic_logger

boost::log::sources::basic_logger — Basic logger class.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

template<typename CharT, typename FinalT, typename ThreadingModelT>
class basic_logger : public ThreadingModelT {
public:
  // types
  typedef CharT                     char_type;                  // Character type.
  typedef FinalT                    final_type;                 // Final logger type.
  typedef ThreadingModelT           threading_model;            // Threading model type.
  typedef unspecified               swap_lock;                  // Lock requirement for the ↵
swap_unlocked method.
  typedef unspecified               add_attribute_lock;         // Lock requirement for the ↵
add_attribute_unlocked method.
  typedef unspecified               remove_attribute_lock;      // Lock requirement for the ↵
remove_attribute_unlocked method.
  typedef unspecified               remove_all_attributes_lock; // Lock requirement for the ↵
remove_all_attributes_unlocked method.
  typedef unspecified               get_attributes_lock;        // Lock requirement for the ↵
get_attributes method.
  typedef unspecified               open_record_lock;           // Lock requirement for the ↵
open_record_unlocked method.
  typedef unspecified               set_attributes_lock;        // Lock requirement for the ↵
set_attributes method.
  typedef no_lock< threading_model > push_record_lock;          // Lock requirement for the ↵
push_record_unlocked method.

  // construct/copy/destruct
  basic_logger();
  basic_logger(basic_logger const &);
  basic_logger(basic_logger &&);
  template<typename ArgsT> explicit basic_logger(ArgsT const &);
  basic_logger & operator=(basic_logger const &) = delete;

  // protected member functions
  core_ptr const & core() const;
  attribute_set & attributes();
  attribute_set const & attributes() const;
  threading_model & get_threading_model();
  threading_model const & get_threading_model() const;
  final_type * final_this();
  final_type const * final_this() const;
  void swap_unlocked(basic_logger &);
  std::pair< attribute_set::iterator, bool >
  add_attribute_unlocked(attribute_name const &, attribute const &);
  void remove_attribute_unlocked(attribute_set::iterator);
  void remove_all_attributes_unlocked();
  record open_record_unlocked();
  template<typename ArgsT> record open_record_unlocked(ArgsT const &);
  void push_record_unlocked(record &&);
  attribute_set get_attributes_unlocked() const;
  void set_attributes_unlocked(attribute_set const &);
};
```

Description

The `basic_logger` class template serves as a base class for all loggers provided by the library. It can also be used as a base for user-defined loggers. The template parameters are:

- `CharT` - logging character type
- `FinalT` - final type of the logger that eventually derives from the `basic_logger`. There may be other classes in the hierarchy between the final class and `basic_logger`.
- `ThreadingModelT` - threading model policy. Must provide methods of the Boost.Thread locking concept used in `basic_logger` class and all its derivatives in the hierarchy up to the `FinalT` class. The `basic_logger` class itself requires methods of the SharedLockable concept. The threading model policy must also be default and copy-constructible and support member function `swap`. There are currently two policies provided: `single_thread_model` and `multi_thread_model`.

The logger implements fundamental facilities of loggers, such as storing source-specific attribute set and formatting log record messages. The basic logger interacts with the logging core in order to apply filtering and pass records to sinks.

`basic_logger` public construct/copy/destruct

1. ```
   basic_logger();
   ```

   Constructor. Initializes internal data structures of the basic logger class, acquires reference to the logging core.

2. ```
   basic_logger(basic_logger const & that);
   ```

   Copy constructor. Copies all attributes from the source logger.

   > **Note**
   >
   > Not thread-safe. The source logger must be locked in the final class before copying.

   Parameters: `that` Source logger

3. ```
   basic_logger(basic_logger && that);
   ```

   Move constructor. Moves all attributes from the source logger.

   > **Note**
   >
   > Not thread-safe. The source logger must be locked in the final class before copying.

   Parameters: `that` Source logger

4. ```
   template<typename ArgsT> explicit basic_logger(ArgsT const &);
   ```

   Constructor with named arguments. The constructor ignores all arguments. The result of construction is equivalent to default construction.

5. ```
   basic_logger & operator=(basic_logger const &) = delete;
   ```

   Assignment is closed (should be implemented through copy and swap in the final class)
   ```

`basic_logger` protected member functions

1.
```
core_ptr const & core() const;
```
An accessor to the logging system pointer

2.
```
attribute_set & attributes();
```
An accessor to the logger attributes

3.
```
attribute_set const & attributes() const;
```
An accessor to the logger attributes

4.
```
threading_model & get_threading_model();
```
An accessor to the threading model base

5.
```
threading_model const & get_threading_model() const;
```
An accessor to the threading model base

6.
```
final_type * final_this();
```
An accessor to the final logger

7.
```
final_type const * final_this() const;
```
An accessor to the final logger

8.
```
void swap_unlocked(basic_logger & that);
```
Unlocked `swap`

9.
```
std::pair< attribute_set::iterator, bool >
add_attribute_unlocked(attribute_name const & name, attribute const & attr);
```
Unlocked `add_attribute`

10.
```
void remove_attribute_unlocked(attribute_set::iterator it);
```
Unlocked `remove_attribute`

11.
```
void remove_all_attributes_unlocked();
```
Unlocked `remove_all_attributes`

12.
```
record open_record_unlocked();
```
Unlocked `open_record`

13. ```
template<typename ArgsT> record open_record_unlocked(ArgsT const &);
```

Unlocked `open_record`

14. ```
void push_record_unlocked(record && rec);
```

Unlocked `push_record`

15. ```
attribute_set get_attributes_unlocked() const;
```

Unlocked `get_attributes`

16. ```
void set_attributes_unlocked(attribute_set const & attrs);
```

Unlocked `set_attributes`

Function template swap

boost::log::sources::swap

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>


template<typename CharT, typename FinalT, typename ThreadingModelT>
  void swap(basic_logger< CharT, FinalT, ThreadingModelT > & left,
            basic_logger< CharT, FinalT, ThreadingModelT > & right);
```

Description

Free-standing swap for all loggers

Macro BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS

BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS(class_type)
```

Macro BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS_TEMPLATE

BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS_TEMPLATE

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_CONSTRUCTORS_TEMPLATE(class_type)
```

Macro BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT

BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT(class_type)
```

Macro BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT_TEMPLATE

BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT_TEMPLATE

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_ASSIGNMENT_TEMPLATE(class_type)
```

Macro BOOST_LOG_FORWARD_LOGGER_MEMBERS

BOOST_LOG_FORWARD_LOGGER_MEMBERS

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_MEMBERS(class_type)
```

Macro BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE

BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_FORWARD_LOGGER_MEMBERS_TEMPLATE(class_type)
```

Macro BOOST_LOG_DECLARE_LOGGER_TYPE

BOOST_LOG_DECLARE_LOGGER_TYPE — The macro declares a logger class that inherits a number of base classes.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, char_type, base_seq, threading)
```

Description

Parameters:	`base_seq`	A Boost.Preprocessor sequence of type identifiers of the base classes templates
	`char_type`	The character type of the logger. Either char or wchar_t expected.
	`threading`	A threading model class
	`type_name`	The name of the logger class to declare

Macro BOOST_LOG_DECLARE_LOGGER

BOOST_LOG_DECLARE_LOGGER — The macro declares a narrow-char logger class that inherits a number of base classes.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_DECLARE_LOGGER(type_name, base_seq)
```

Description

Equivalent to BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, char, base_seq, single_thread_model)

Parameters:	`base_seq`	A Boost.Preprocessor sequence of type identifiers of the base classes templates
	`type_name`	The name of the logger class to declare

Macro BOOST_LOG_DECLARE_LOGGER_MT

BOOST_LOG_DECLARE_LOGGER_MT — The macro declares a narrow-char thread-safe logger class that inherits a number of base classes.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_DECLARE_LOGGER_MT(type_name, base_seq)
```

Description

Equivalent to `BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, char, base_seq, multi_thread_model< shared_mutex >)`

Parameters:	`base_seq`	A Boost.Preprocessor sequence of type identifiers of the base classes templates
	`type_name`	The name of the logger class to declare

Macro BOOST_LOG_DECLARE_WLOGGER

BOOST_LOG_DECLARE_WLOGGER — The macro declares a wide-char logger class that inherits a number of base classes.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_DECLARE_WLOGGER(type_name, base_seq)
```

Description

Equivalent to BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, wchar_t, base_seq, single_thread_model)

Parameters:

`base_seq`	A Boost.Preprocessor sequence of type identifiers of the base classes templates
`type_name`	The name of the logger class to declare

Macro BOOST_LOG_DECLARE_WLOGGER_MT

BOOST_LOG_DECLARE_WLOGGER_MT — The macro declares a wide-char thread-safe logger class that inherits a number of base classes.

Synopsis

```
// In header: <boost/log/sources/basic_logger.hpp>

BOOST_LOG_DECLARE_WLOGGER_MT(type_name, base_seq)
```

Description

Equivalent to `BOOST_LOG_DECLARE_LOGGER_TYPE(type_name, wchar_t, base_seq, multi_thread_model< shared_mutex >)`

Parameters:

`base_seq`	A Boost.Preprocessor sequence of type identifiers of the base classes templates
`type_name`	The name of the logger class to declare

Header <boost/log/sources/channel_feature.hpp>

Andrey Semashev

28.02.2008
The header contains implementation of a channel support feature.

```


BOOST_LOG_STREAM_CHANNEL(logger, chan)
BOOST_LOG_CHANNEL(logger, chan)
```

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename BaseT, typename ChannelT> class basic_channel_logger;

      template<typename ChannelT = std::string> struct channel;
    }
  }
}
```

Class template basic_channel_logger

boost::log::sources::basic_channel_logger — Channel feature implementation.

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>

template<typename BaseT, typename ChannelT>
class basic_channel_logger : public BaseT {
public:
  // types
  typedef base_type::char_type                          char_type;          // Character type.
  typedef base_type::final_type                         final_type;         // Final type.
  typedef base_type::threading_model                    threading_model;    // Threading model ↵
being used.
  typedef ChannelT                                      channel_type;       // Channel type.
  typedef attributes::mutable_constant< channel_type > channel_attribute;  // Channel attribute ↵
type.
  typedef unspecified                                   open_record_lock;   // Lock requirement ↵
for the open_record_unlocked method.
  typedef unspecified                                   swap_lock;          // Lock requirement ↵
for the swap_unlocked method.

  // member classes/structs/unions

  // Default channel name generator.

  struct make_default_channel_name {
    // types
    typedef channel_type result_type;

    // public member functions
    result_type operator()() const;
  };

  // construct/copy/destruct
  basic_channel_logger();
  basic_channel_logger(basic_channel_logger const &);
  basic_channel_logger(basic_channel_logger &&);
  template<typename ArgsT> explicit basic_channel_logger(ArgsT const &);

  // public member functions
  channel_type channel() const;
  void channel(channel_type const &);

  // protected member functions
  channel_attribute const & get_channel_attribute() const;
  template<typename ArgsT> record open_record_unlocked(ArgsT const &);
  void swap_unlocked(basic_channel_logger &);

  // private member functions
  template<typename ArgsT, typename T>
    record open_record_with_channel_unlocked(ArgsT const &, T const &);
  template<typename ArgsT>
    record open_record_with_channel_unlocked(ArgsT const &, parameter::void_);
};
```

Description

`basic_channel_logger` public construct/copy/destruct

1. ```
   basic_channel_logger();
   ```

   Default constructor. The constructed logger has the default-constructed channel name.

2. ```
   basic_channel_logger(basic_channel_logger const & that);
   ```

   Copy constructor

3. ```
   basic_channel_logger(basic_channel_logger && that);
   ```

   Move constructor

4. ```
   template<typename ArgsT> explicit basic_channel_logger(ArgsT const & args);
   ```

   Constructor with arguments. Allows to register a channel name attribute on construction.

   Parameters: `args` A set of named arguments. The following arguments are supported:

   - `channel` - a string that represents the channel name

`basic_channel_logger` public member functions

1. ```
   channel_type channel() const;
   ```

   The observer of the channel name

   Returns: The channel name that was set by the logger

2. ```
   void channel(channel_type const & ch);
   ```

   The setter of the channel name

   Parameters: `ch` The channel name to be set for the logger

`basic_channel_logger` protected member functions

1. ```
   channel_attribute const & get_channel_attribute() const;
   ```

   Channel attribute accessor

2. ```
   template<typename ArgsT> record open_record_unlocked(ArgsT const & args);
   ```

   Unlocked `open_record`

3. ```
   void swap_unlocked(basic_channel_logger & that);
   ```

   Unlocked swap
   ```

`basic_channel_logger` private member functions

1.
```
template<typename ArgsT, typename T>
  record open_record_with_channel_unlocked(ArgsT const & args, T const & ch);
```

The `open_record` implementation for the case when the channel is specified in log statement.

2.
```
template<typename ArgsT>
  record open_record_with_channel_unlocked(ArgsT const & args,
                                           parameter::void_);
```

The `open_record` implementation for the case when the channel is not specified in log statement.

Struct make_default_channel_name

boost::log::sources::basic_channel_logger::make_default_channel_name — Default channel name generator.

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>



// Default channel name generator.

struct make_default_channel_name {
  // types
  typedef channel_type result_type;

  // public member functions
  result_type operator()() const;
};
```

Description

`make_default_channel_name` public member functions

1.
```
result_type operator()() const;
```

Struct template channel

boost::log::sources::channel — Channel support feature.

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>

template<typename ChannelT = std::string>
struct channel {
  // member classes/structs/unions
  template<typename BaseT>
  struct apply {
    // types
    typedef basic_channel_logger< BaseT, ChannelT > type;
  };
};
```

Description

The logger with this feature automatically registers an attribute with the specified on construction value, which is a channel name. The channel name can be modified through the logger life time, either by calling the `channel` method or by specifying the name in the logging statement.

The type of the channel name can be customized by providing it as a template parameter to the feature template. By default, a string will be used.

Struct template apply

boost::log::sources::channel::apply

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>


template<typename BaseT>
struct apply {
  // types
  typedef basic_channel_logger< BaseT, ChannelT > type;
};
```

Macro BOOST_LOG_STREAM_CHANNEL

BOOST_LOG_STREAM_CHANNEL — The macro allows to put a record with a specific channel name into log.

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>

BOOST_LOG_STREAM_CHANNEL(logger, chan)
```

Macro BOOST_LOG_CHANNEL

BOOST_LOG_CHANNEL — An equivalent to BOOST_LOG_STREAM_CHANNEL(logger, chan)

Synopsis

```
// In header: <boost/log/sources/channel_feature.hpp>

BOOST_LOG_CHANNEL(logger, chan)
```

Header <boost/log/sources/channel_logger.hpp>

Andrey Semashev

28.02.2008
The header contains implementation of a logger with channel support.

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename ChannelT = std::string> class channel_logger;
      template<typename ChannelT = std::string> class channel_logger_mt;
      template<typename ChannelT = std::wstring> class wchannel_logger;
      template<typename ChannelT = std::wstring> class wchannel_logger_mt;
    }
  }
}
```

Class template channel_logger

boost::log::sources::channel_logger — Narrow-char logger. Functionally equivalent to `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/channel_logger.hpp>

template<typename ChannelT = std::string>
class channel_logger : public basic_composite_logger< char, channel_logger< ChannelT >, ↵
single_thread_model, features< channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  channel_logger();
  channel_logger(channel_logger const &);
  template<typename... ArgsT> explicit channel_logger(ArgsT...const &);
  explicit channel_logger(ChannelT const &);
  channel_logger & operator=(channel_logger const &);
};
```

Description

See `channel` class template for a more detailed description

`channel_logger` public construct/copy/destruct

1. ```
   channel_logger();
   ```

   Default constructor
   ```

2. `channel_logger(channel_logger const & that);`

   Copy constructor

3. `template<typename... ArgsT> explicit channel_logger(ArgsT...const & args);`

   Constructor with named arguments

4. `explicit channel_logger(ChannelT const & channel);`

   The constructor creates the logger with the specified channel name

   Parameters: `channel` The channel name

5. `channel_logger & operator=(channel_logger const & that);`

   Assignment operator

   Swaps two loggers

Class template channel_logger_mt

boost::log::sources::channel_logger_mt — Narrow-char thread-safe logger. Functionally equivalent to `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/channel_logger.hpp>

template<typename ChannelT = std::string>
class channel_logger_mt : public basic_composite_logger< char, channel_logger_mt< ChannelT >, ↵
multi_thread_model< implementation_defined >, features< channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  channel_logger_mt();
  channel_logger_mt(channel_logger_mt const &);
  template<typename... ArgsT> explicit channel_logger_mt(ArgsT...const &);
  explicit channel_logger_mt(ChannelT const &);
  channel_logger_mt & operator=(channel_logger_mt const &);
};
```

Description

See `channel` class template for a more detailed description

`channel_logger_mt` public construct/copy/destruct

1. `channel_logger_mt();`

   Default constructor

2. `channel_logger_mt(channel_logger_mt const & that);`

   Copy constructor

3. `template<typename... ArgsT> explicit channel_logger_mt(ArgsT...const & args);`

   Constructor with named arguments

4. `explicit channel_logger_mt(ChannelT const & channel);`

   The constructor creates the logger with the specified channel name

   Parameters: `channel` The channel name

5. `channel_logger_mt & operator=(channel_logger_mt const & that);`

   Assignment operator

   Swaps two loggers

Class template wchannel_logger

boost::log::sources::wchannel_logger — Wide-char logger. Functionally equivalent to `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/channel_logger.hpp>

template<typename ChannelT = std::wstring>
class wchannel_logger : public basic_composite_logger< wchar_t, wchannel_logger< ChannelT >, ↵
single_thread_model, features< channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  wchannel_logger();
  wchannel_logger(wchannel_logger const &);
  template<typename... ArgsT> explicit wchannel_logger(ArgsT...const &);
  explicit wchannel_logger(ChannelT const &);
  wchannel_logger & operator=(wchannel_logger const &);
};
```

Description

See `channel` class template for a more detailed description

`wchannel_logger` public construct/copy/destruct

1. `wchannel_logger();`

   Default constructor

2. `wchannel_logger(wchannel_logger const & that);`

   Copy constructor

3. `template<typename... ArgsT> explicit wchannel_logger(ArgsT...const & args);`

   Constructor with named arguments

4. ```
   explicit wchannel_logger(ChannelT const & channel);
   ```

   The constructor creates the logger with the specified channel name

   Parameters: `channel` The channel name

5. ```
   wchannel_logger & operator=(wchannel_logger const & that);
   ```

   Assignment operator

   Swaps two loggers

Class template wchannel_logger_mt

boost::log::sources::wchannel_logger_mt — Wide-char thread-safe logger. Functionally equivalent to `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/channel_logger.hpp>

template<typename ChannelT = std::wstring>
class wchannel_logger_mt : public basic_composite_logger< wchar_t, wchannel_logger< ChannelT >, ↵
multi_thread_model< implementation_defined >, features< channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  wchannel_logger_mt();
  wchannel_logger_mt(wchannel_logger_mt const &);
  template<typename... ArgsT> explicit wchannel_logger_mt(ArgsT...const &);
  explicit wchannel_logger_mt(ChannelT const &);
  wchannel_logger_mt & operator=(wchannel_logger_mt const &);
};
```

Description

See `channel` class template for a more detailed description

`wchannel_logger_mt` public construct/copy/destruct

1. ```
   wchannel_logger_mt();
   ```

   Default constructor

2. ```
   wchannel_logger_mt(wchannel_logger_mt const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT> explicit wchannel_logger_mt(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   explicit wchannel_logger_mt(ChannelT const & channel);
   ```

   The constructor creates the logger with the specified channel name

Parameters: `channel` The channel name

5.
```
wchannel_logger_mt & operator=(wchannel_logger_mt const & that);
```

Assignment operator

Swaps two loggers

Header <boost/log/sources/exception_handler_feature.hpp>

Andrey Semashev

17.07.2009
The header contains implementation of an exception handler support feature.

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename BaseT> class basic_exception_handler_logger;

      struct exception_handler;
    }
  }
}
```

Class template basic_exception_handler_logger

boost::log::sources::basic_exception_handler_logger — Exception handler feature implementation.

Synopsis

```
// In header: <boost/log/sources/exception_handler_feature.hpp>

template<typename BaseT>
class basic_exception_handler_logger : public BaseT {
public:
  // types
  typedef base_type::threading_model                                                ↵
    threading_model;         // Threading model being used.
  typedef base_type::final_type                                                     ↵
    final_type;              // Final logger type.
  typedef unspecified                                                               ↵
    exception_handler_type;  // Exception handler function type.
  typedef strictest_lock< typename base_type::open_record_lock, no_lock< threading_mod↵
el > >::type open_record_lock;        // Lock requirement for the open_record_unlocked method.
  typedef strictest_lock< typename base_type::push_record_lock, no_lock< threading_mod↵
el > >::type push_record_lock;        // Lock requirement for the push_record_unlocked method.
  typedef unspecified                                                               ↵
    swap_lock;               // Lock requirement for the swap_unlocked method.

  // construct/copy/destruct
  basic_exception_handler_logger();
  basic_exception_handler_logger(basic_exception_handler_logger const &);
  basic_exception_handler_logger(basic_exception_handler_logger &&);
  template<typename ArgsT>
    explicit basic_exception_handler_logger(ArgsT const &);

  // public member functions
  template<typename HandlerT> void set_exception_handler(HandlerT const &);

  // protected member functions
  template<typename ArgsT> record open_record_unlocked(ArgsT const &);
  void push_record_unlocked(record &&);
  void swap_unlocked(basic_exception_handler_logger &);
};
```

Description

`basic_exception_handler_logger` public construct/copy/destruct

1. ```
   basic_exception_handler_logger();
   ```

   Default constructor. The constructed logger does not have an exception handler.

2. ```
   basic_exception_handler_logger(basic_exception_handler_logger const & that);
   ```

   Copy constructor

3. ```
   basic_exception_handler_logger(basic_exception_handler_logger && that);
   ```

   Move constructor

4. ```
   template<typename ArgsT>
     explicit basic_exception_handler_logger(ArgsT const & args);
   ```

   Constructor with arguments. Passes arguments to other features.

`basic_exception_handler_logger` public member functions

1.
```
template<typename HandlerT>
  void set_exception_handler(HandlerT const & handler);
```

The method sets exception handler function. The function will be called with no arguments in case if an exception occurs during either `open_record` or `push_record` method execution. Since exception handler is called from a `catch` statement, the exception can be rethrown in order to determine its type.

By default no handler is installed, thus any exception is propagated as usual.

See Also:

`utility/exception_handler.hpp`

Note

The exception handler can be invoked in several threads concurrently.

Note

Thread interruptions are not affected by exception handlers.

Parameters: `handler` Exception handling function

`basic_exception_handler_logger` protected member functions

1.
```
template<typename ArgsT> record open_record_unlocked(ArgsT const & args);
```

Unlocked `open_record`

2.
```
void push_record_unlocked(record && rec);
```

Unlocked `push_record`

3.
```
void swap_unlocked(basic_exception_handler_logger & that);
```

Unlocked swap

Struct exception_handler

boost::log::sources::exception_handler — Exception handler support feature.

Synopsis

```
// In header: <boost/log/sources/exception_handler_feature.hpp>


struct exception_handler {
  // member classes/structs/unions
  template<typename BaseT>
  struct apply {
    // types
    typedef basic_exception_handler_logger< BaseT > type;
  };
};
```

Description

The logger with this feature will provide an additional method to install an exception handler functional object. This functional object will be called if during either opening or pushing a record an exception is thrown from the logging core.

Struct template apply

boost::log::sources::exception_handler::apply

Synopsis

```
// In header: <boost/log/sources/exception_handler_feature.hpp>


template<typename BaseT>
struct apply {
  // types
  typedef basic_exception_handler_logger< BaseT > type;
};
```

Header <boost/log/sources/features.hpp>

Andrey Semashev

17.07.2009
The header contains definition of a features list class template.

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename... FeaturesT> struct features;
    }
  }
}
```

Struct template features

boost::log::sources::features — A type sequence of logger features.

Synopsis

```
// In header: <boost/log/sources/features.hpp>

template<typename... FeaturesT>
struct features {
};
```

Description

This class template can be used to specify logger features in a `basic_composite_logger` instantiation.

Header <boost/log/sources/global_logger_storage.hpp>

Andrey Semashev

21.04.2008
The header contains implementation of facilities to declare global loggers.

```
BOOST_LOG_GLOBAL_LOGGER(tag_name, logger)
BOOST_LOG_GLOBAL_LOGGER_INIT(tag_name, logger)
BOOST_LOG_GLOBAL_LOGGER_DEFAULT(tag_name, logger)
BOOST_LOG_GLOBAL_LOGGER_CTOR_ARGS(tag_name, logger, args)
BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(tag_name, logger)
BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(tag_name, logger)
BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS(tag_name, logger, args)
```

Macro BOOST_LOG_GLOBAL_LOGGER

BOOST_LOG_GLOBAL_LOGGER — The macro forward-declares a global logger with a custom initialization.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_GLOBAL_LOGGER(tag_name, logger)
```

Macro BOOST_LOG_GLOBAL_LOGGER_INIT

BOOST_LOG_GLOBAL_LOGGER_INIT — The macro defines a global logger initialization routine.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_GLOBAL_LOGGER_INIT(tag_name, logger)
```

Macro BOOST_LOG_GLOBAL_LOGGER_DEFAULT

BOOST_LOG_GLOBAL_LOGGER_DEFAULT — The macro defines a global logger initializer that will default-construct the logger.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_GLOBAL_LOGGER_DEFAULT(tag_name, logger)
```

Macro BOOST_LOG_GLOBAL_LOGGER_CTOR_ARGS

BOOST_LOG_GLOBAL_LOGGER_CTOR_ARGS — The macro defines a global logger initializer that will construct the logger with the specified constructor arguments.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_GLOBAL_LOGGER_CTOR_ARGS(tag_name, logger, args)
```

Macro BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT

BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT — The macro declares a global logger with a custom initialization.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT(tag_name, logger)
```

Macro BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT — The macro declares a global logger that will be default-constructed.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(tag_name, logger)
```

Macro BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS

BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS — The macro declares a global logger that will be constructed with the specified arguments.

Synopsis

```
// In header: <boost/log/sources/global_logger_storage.hpp>

BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS(tag_name, logger, args)
```

Header <boost/log/sources/logger.hpp>

Andrey Semashev

08.03.2007
The header contains implementation of a simplistic logger with no features.

```
namespace boost {
  namespace log {
    namespace sources {
      class logger;
      class logger_mt;
      class wlogger;
      class wlogger_mt;
    }
  }
}
```

Class logger

boost::log::sources::logger — Narrow-char logger. Functionally equivalent to `basic_logger`.

Synopsis

```
// In header: <boost/log/sources/logger.hpp>


class logger : public basic_composite_logger< char, logger, single_thread_model, features<  > >
{
};
```

Description

See `basic_logger` class template for a more detailed description.

Class logger_mt

boost::log::sources::logger_mt — Narrow-char thread-safe logger. Functionally equivalent to `basic_logger`.

Synopsis

```
// In header: <boost/log/sources/logger.hpp>


class logger_mt {
};
```

Description

See `basic_logger` class template for a more detailed description.

Class wlogger

boost::log::sources::wlogger — Wide-char logger. Functionally equivalent to `basic_logger`.

Synopsis

```
// In header: <boost/log/sources/logger.hpp>


class wlogger : public basic_composite_logger< wchar_t, wlogger, single_thread_model, features< ↵
 > >
{
};
```

Description

See basic_logger class template for a more detailed description.

Class wlogger_mt

boost::log::sources::wlogger_mt — Wide-char thread-safe logger. Functionally equivalent to basic_logger.

Synopsis

```
// In header: <boost/log/sources/logger.hpp>


class wlogger_mt {
};
```

Description

See basic_logger class template for a more detailed description.

Header <boost/log/sources/record_ostream.hpp>

Andrey Semashev

09.03.2009
This header contains a wrapper class around a logging record that allows to compose the record message with a streaming expression.

```


BOOST_LOG_STREAM(logger)
BOOST_LOG_STREAM_WITH_PARAMS(logger, params_seq)
BOOST_LOG(logger)
BOOST_LOG_WITH_PARAMS(logger, params_seq)
```

```
namespace boost {
  namespace log {
    template<typename CharT> class basic_record_ostream;

    typedef basic_record_ostream< char > record_ostream;  // Convenience typedef for narrow-
character logging.
    typedef basic_record_ostream< wchar_t > wrecord_ostream;  // Convenience typedef for wide-
character logging.
  }
}
```

Class template basic_record_ostream

boost::log::basic_record_ostream — Logging record adapter with a streaming capability.

Synopsis

```
// In header: <boost/log/sources/record_ostream.hpp>

template<typename CharT>
class basic_record_ostream : public basic_formatting_ostream< CharT > {
public:
  // types
  typedef CharT                            char_type;    // Character type.
  typedef std::basic_string< char_type >  string_type;  // String type to be used as a message ↵
text holder.
  typedef std::basic_ostream< char_type > stream_type;  // Stream type.

  // construct/copy/destruct
  basic_record_ostream() noexcept;
  explicit basic_record_ostream(record &);
  basic_record_ostream(basic_record_ostream const &) = delete;
  basic_record_ostream & operator=(basic_record_ostream const &) = delete;
  ~basic_record_ostream();

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  record & get_record();
  record const & get_record() const;
  void attach_record(record &);
  void detach_from_record() noexcept;

  // private member functions
  void init_stream();
};
```

Description

This class allows to compose the logging record message by streaming operations. It aggregates the log record and provides the standard output stream interface.

`basic_record_ostream` public construct/copy/destruct

1. ```
   basic_record_ostream() noexcept;
   ```

   Default constructor. Creates an empty record that is equivalent to the invalid record handle. The stream capability is not available after construction.

   Postconditions: `!*this == true`

2. ```
   explicit basic_record_ostream(record & rec);
   ```

   Constructor from a record object. Attaches to the provided record.

   Parameters: `rec` The record handle being attached to
   Requires: `!!rec == true`
   Postconditions: `&this->get_record() == &rec`

3. ```
   basic_record_ostream(basic_record_ostream const &) = delete;
   ```

4. ```
   basic_record_ostream & operator=(basic_record_ostream const &) = delete;
   ```

5. ```
   ~basic_record_ostream();
   ```

   Destructor. Destroys the record, releases any sinks and attribute values that were involved in processing this record.

`basic_record_ostream` public member functions

1. ```
   explicit operator bool() const noexcept;
   ```

   Conversion to an unspecified boolean type

   Returns: `true`, if stream is valid and ready for formatting, `false`, if the stream is not valid. The latter also applies to the case when the stream is not attached to a log record.

2. ```
   bool operator!() const noexcept;
   ```

   Inverted conversion to an unspecified boolean type

   Returns: `false`, if stream is valid and ready for formatting, `true`, if the stream is not valid. The latter also applies to the case when the stream is not attached to a log record.

3. ```
   record & get_record();
   ```

   Flushes internal buffers to complete all pending formatting operations and returns the aggregated log record

   Returns: The aggregated record object

4. ```
   record const & get_record() const;
   ```

   Flushes internal buffers to complete all pending formatting operations and returns the aggregated log record

   Returns: The aggregated record object

5. ```
   void attach_record(record & rec);
   ```

   If the stream is attached to a log record, flushes internal buffers to complete all pending formatting operations. Then reattaches the stream to another log record.

   Parameters: `rec` New log record to attach to

6. ```
   void detach_from_record() noexcept;
   ```

   The function resets the stream into a detached (default initialized) state.

`basic_record_ostream` private member functions

1. ```
   void init_stream();
   ```

The function initializes the stream and the stream buffer.

Macro BOOST_LOG_STREAM

BOOST_LOG_STREAM — The macro writes a record to the log.

Synopsis

```
// In header: <boost/log/sources/record_ostream.hpp>

BOOST_LOG_STREAM(logger)
```

Macro BOOST_LOG_STREAM_WITH_PARAMS

BOOST_LOG_STREAM_WITH_PARAMS — The macro writes a record to the log and allows to pass additional named arguments to the logger.

Synopsis

```
// In header: <boost/log/sources/record_ostream.hpp>

BOOST_LOG_STREAM_WITH_PARAMS(logger, params_seq)
```

Macro BOOST_LOG

BOOST_LOG — An equivalent to BOOST_LOG_STREAM(logger)

Synopsis

```
// In header: <boost/log/sources/record_ostream.hpp>

BOOST_LOG(logger)
```

Macro BOOST_LOG_WITH_PARAMS

BOOST_LOG_WITH_PARAMS — An equivalent to BOOST_LOG_STREAM_WITH_PARAMS(logger, params_seq)

Synopsis

```
// In header: <boost/log/sources/record_ostream.hpp>

BOOST_LOG_WITH_PARAMS(logger, params_seq)
```

Header <boost/log/sources/severity_channel_logger.hpp>

Andrey Semashev

28.02.2008
The header contains implementation of a logger with severity level and channel support.

```
BOOST_LOG_STREAM_CHANNEL_SEV(logger, chan, lvl)
BOOST_LOG_CHANNEL_SEV(logger, chan, lvl)
```

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename LevelT = int, typename ChannelT = std::string>
        class severity_channel_logger;
      template<typename LevelT = int, typename ChannelT = std::string>
        class severity_channel_logger_mt;
      template<typename LevelT = int, typename ChannelT = std::wstring>
        class wseverity_channel_logger;
      template<typename LevelT = int, typename ChannelT = std::wstring>
        class wseverity_channel_logger_mt;
    }
  }
}
```

Class template severity_channel_logger

boost::log::sources::severity_channel_logger — Narrow-char logger. Functionally equivalent to `basic_severity_logger` and `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

template<typename LevelT = int, typename ChannelT = std::string>
class severity_channel_logger : public basic_composite_logger< char, severity_channel_logger< ↵
LevelT, ChannelT >, single_thread_model, features< severity< LevelT >, channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  severity_channel_logger();
  severity_channel_logger(severity_channel_logger const &);
  template<typename... ArgsT>
    explicit severity_channel_logger(ArgsT...const &);
  severity_channel_logger & operator=(severity_channel_logger const &);
};
```

Description

See `severity` and `channel` class templates for a more detailed description

`severity_channel_logger` public construct/copy/destruct

1. ```
   severity_channel_logger();
   ```

   Default constructor

2. ```
   severity_channel_logger(severity_channel_logger const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT>
     explicit severity_channel_logger(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   severity_channel_logger & operator=(severity_channel_logger const & that);
   ```

   Assignment operator

   Swaps two loggers

Class template severity_channel_logger_mt

boost::log::sources::severity_channel_logger_mt — Narrow-char thread-safe logger. Functionally equivalent to `basic_severity_logger` and `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

template<typename LevelT = int, typename ChannelT = std::string>
class severity_channel_logger_mt : public basic_composite_logger< char, severity_channel_log↵
ger_mt< LevelT, ChannelT >, multi_thread_model< implementation_defined >, features< severity< ↵
LevelT >, channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  severity_channel_logger_mt();
  severity_channel_logger_mt(severity_channel_logger_mt const &);
  template<typename... ArgsT>
    explicit severity_channel_logger_mt(ArgsT...const &);
  severity_channel_logger_mt & operator=(severity_channel_logger_mt const &);
};
```

Description

See `severity` and `channel` class templates for a more detailed description

`severity_channel_logger_mt` public construct/copy/destruct

1. ```
   severity_channel_logger_mt();
   ```

   Default constructor

2. ```
   severity_channel_logger_mt(severity_channel_logger_mt const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT>
     explicit severity_channel_logger_mt(ArgsT...const & args);
   ```

   Constructor with named arguments
   ```

4.
```
severity_channel_logger_mt &
operator=(severity_channel_logger_mt const & that);
```

Assignment operator

Swaps two loggers

Class template wseverity_channel_logger

boost::log::sources::wseverity_channel_logger — Wide-char logger. Functionally equivalent to `basic_severity_logger` and `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

template<typename LevelT = int, typename ChannelT = std::wstring>
class wseverity_channel_logger : public basic_composite_logger< wchar_t, wseverity_channel_log↵
ger< LevelT, ChannelT >, single_thread_model, features< severity< LevelT >, channel< ChannelT ↵
> > >
{
public:
  // construct/copy/destruct
  wseverity_channel_logger();
  wseverity_channel_logger(wseverity_channel_logger const &);
  template<typename... ArgsT>
    explicit wseverity_channel_logger(ArgsT...const &);
  wseverity_channel_logger & operator=(wseverity_channel_logger const &);
};
```

Description

See `severity` and `channel` class templates for a more detailed description

`wseverity_channel_logger` public construct/copy/destruct

1.
```
wseverity_channel_logger();
```

Default constructor

2.
```
wseverity_channel_logger(wseverity_channel_logger const & that);
```

Copy constructor

3.
```
template<typename... ArgsT>
  explicit wseverity_channel_logger(ArgsT...const & args);
```

Constructor with named arguments

4.
```
wseverity_channel_logger & operator=(wseverity_channel_logger const & that);
```

Assignment operator

Swaps two loggers

Class template wseverity_channel_logger_mt

boost::log::sources::wseverity_channel_logger_mt — Wide-char thread-safe logger. Functionally equivalent to `basic_severity_logger` and `basic_channel_logger`.

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

template<typename LevelT = int, typename ChannelT = std::wstring>
class wseverity_channel_logger_mt : public basic_composite_logger< wchar_t, wseverity_channel_log↵
ger_mt< LevelT, ChannelT >, multi_thread_model< implementation_defined >, features< severity< ↵
LevelT >, channel< ChannelT > > >
{
public:
  // construct/copy/destruct
  wseverity_channel_logger_mt();
  wseverity_channel_logger_mt(wseverity_channel_logger_mt const &);
  template<typename... ArgsT>
    explicit wseverity_channel_logger_mt(ArgsT...const &);
  wseverity_channel_logger_mt & operator=(wseverity_channel_logger_mt const &);
};
```

Description

See `severity` and `channel` class templates for a more detailed description

`wseverity_channel_logger_mt` public construct/copy/destruct

1. ```
   wseverity_channel_logger_mt();
   ```

   Default constructor

2. ```
   wseverity_channel_logger_mt(wseverity_channel_logger_mt const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT>
     explicit wseverity_channel_logger_mt(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   wseverity_channel_logger_mt &
   operator=(wseverity_channel_logger_mt const & that);
   ```

   Assignment operator

   Swaps two loggers

Macro BOOST_LOG_STREAM_CHANNEL_SEV

BOOST_LOG_STREAM_CHANNEL_SEV — The macro allows to put a record with a specific channel name into log.

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

BOOST_LOG_STREAM_CHANNEL_SEV(logger, chan, lvl)
```

Macro BOOST_LOG_CHANNEL_SEV

BOOST_LOG_CHANNEL_SEV — An equivalent to BOOST_LOG_STREAM_CHANNEL_SEV(logger, chan, lvl)

Synopsis

```
// In header: <boost/log/sources/severity_channel_logger.hpp>

BOOST_LOG_CHANNEL_SEV(logger, chan, lvl)
```

Header <boost/log/sources/severity_feature.hpp>

Andrey Semashev

08.03.2007
The header contains implementation of a severity level support feature.

```


BOOST_LOG_STREAM_SEV(logger, lvl)
BOOST_LOG_SEV(logger, lvl)
```

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename BaseT, typename LevelT = int> class basic_severity_logger;

      template<typename LevelT = int> struct severity;
    }
  }
}
```

Class template basic_severity_logger

boost::log::sources::basic_severity_logger — Severity level feature implementation.

Synopsis

```
// In header: <boost/log/sources/severity_feature.hpp>

template<typename BaseT, typename LevelT = int>
class basic_severity_logger : public BaseT {
public:
  // types
  typedef base_type::char_type                                                 ↵
    char_type;           // Character type.
  typedef base_type::final_type                                                ↵
    final_type;          // Final type.
  typedef base_type::threading_model                                           ↵
    threading_model;     // Threading model being used.
  typedef LevelT                                                               ↵
    severity_level;      // Severity level type.
  typedef unspecified                                                          ↵
    severity_attribute;  // Severity attribute type.
  typedef strictest_lock< typename base_type::open_record_lock, no_lock< threading_mod↵
el > >::type open_record_lock;    // Lock requirement for the open_record_unlocked method.
  typedef unspecified                                                          ↵
    swap_lock;           // Lock requirement for the swap_unlocked method.

  // construct/copy/destruct
  basic_severity_logger();
  basic_severity_logger(basic_severity_logger const &);
  basic_severity_logger(basic_severity_logger &&);
  template<typename ArgsT> explicit basic_severity_logger(ArgsT const &);

  // public member functions
  severity_level default_severity() const;

  // protected member functions
  severity_attribute const & get_severity_attribute() const;
  template<typename ArgsT> record open_record_unlocked(ArgsT const &);
  void swap_unlocked(basic_severity_logger &);
};
```

Description

`basic_severity_logger` public construct/copy/destruct

1. ```
   basic_severity_logger();
   ```

   Default constructor. The constructed logger will have a severity attribute registered. The default level for log records will be 0.

2. ```
   basic_severity_logger(basic_severity_logger const & that);
   ```

   Copy constructor

3. ```
   basic_severity_logger(basic_severity_logger && that);
   ```

   Move constructor

4. ```
   template<typename ArgsT> explicit basic_severity_logger(ArgsT const & args);
   ```

   Constructor with named arguments. Allows to setup the default level for log records.

Parameters: `args` A set of named arguments. The following arguments are supported:

- `severity` - default severity value

`basic_severity_logger` public member functions

1.
```
severity_level default_severity() const;
```

Default severity value getter

`basic_severity_logger` protected member functions

1.
```
severity_attribute const & get_severity_attribute() const;
```

Severity attribute accessor

2.
```
template<typename ArgsT> record open_record_unlocked(ArgsT const & args);
```

Unlocked `open_record`

3.
```
void swap_unlocked(basic_severity_logger & that);
```

Unlocked `swap`.

Struct template severity

boost::log::sources::severity — Severity level support feature.

Synopsis

```
// In header: <boost/log/sources/severity_feature.hpp>

template<typename LevelT = int>
struct severity {
  // member classes/structs/unions
  template<typename BaseT>
  struct apply {
    // types
    typedef basic_severity_logger< BaseT, LevelT > type;
  };
};
```

Description

The logger with this feature registers a special attribute with an integral value type on construction. This attribute will provide severity level for each log record being made through the logger. The severity level can be omitted on logging record construction, in which case the default level will be used. The default level can also be customized by passing it to the logger constructor.

The type of the severity level attribute can be specified as a template parameter for the feature template. By default, `int` will be used.

Struct template apply

boost::log::sources::severity::apply

Synopsis

```
// In header: <boost/log/sources/severity_feature.hpp>


template<typename BaseT>
struct apply {
  // types
  typedef basic_severity_logger< BaseT, LevelT > type;
};
```

Macro BOOST_LOG_STREAM_SEV

BOOST_LOG_STREAM_SEV — The macro allows to put a record with a specific severity level into log.

Synopsis

```
// In header: <boost/log/sources/severity_feature.hpp>

BOOST_LOG_STREAM_SEV(logger, lvl)
```

Macro BOOST_LOG_SEV

BOOST_LOG_SEV — An equivalent to BOOST_LOG_STREAM_SEV(logger, lvl)

Synopsis

```
// In header: <boost/log/sources/severity_feature.hpp>

BOOST_LOG_SEV(logger, lvl)
```

Header <boost/log/sources/severity_logger.hpp>

Andrey Semashev

08.03.2007
The header contains implementation of a logger with severity level support.

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename LevelT = int> class severity_logger;
      template<typename LevelT = int> class severity_logger_mt;
      template<typename LevelT = int> class wseverity_logger;
      template<typename LevelT = int> class wseverity_logger_mt;
    }
  }
}
```

Class template severity_logger

boost::log::sources::severity_logger — Narrow-char logger. Functionally equivalent to `basic_severity_logger`.

Synopsis

```
// In header: <boost/log/sources/severity_logger.hpp>

template<typename LevelT = int>
class severity_logger : public basic_composite_logger< char, severity_logger< LevelT >, ↵
single_thread_model, features< severity< LevelT > > >
{
public:
  // construct/copy/destruct
  severity_logger();
  severity_logger(severity_logger const &);
  template<typename... ArgsT> explicit severity_logger(ArgsT...const &);
  explicit severity_logger(LevelT);
  severity_logger & operator=(severity_logger const &);
};
```

Description

See `severity` class template for a more detailed description

`severity_logger` public construct/copy/destruct

1. ```
   severity_logger();
   ```

   Default constructor

2. ```
   severity_logger(severity_logger const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT> explicit severity_logger(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   explicit severity_logger(LevelT level);
   ```

   The constructor creates the logger with the specified default severity level

   Parameters: `level` The default severity level

5. ```
   severity_logger & operator=(severity_logger const & that);
   ```

   Assignment operator

   Swaps two loggers

Class template severity_logger_mt

boost::log::sources::severity_logger_mt — Narrow-char thread-safe logger. Functionally equivalent to `basic_severity_logger`.
   ```

Synopsis

```
// In header: <boost/log/sources/severity_logger.hpp>

template<typename LevelT = int>
class severity_logger_mt : public basic_composite_logger< char, severity_logger_mt< LevelT >, ↵
multi_thread_model< implementation_defined >, features< severity< LevelT > > >
{
public:
  // construct/copy/destruct
  severity_logger_mt();
  severity_logger_mt(severity_logger_mt const &);
  template<typename... ArgsT> explicit severity_logger_mt(ArgsT...const &);
  explicit severity_logger_mt(LevelT);
  severity_logger_mt & operator=(severity_logger_mt const &);
};
```

Description

See `severity` class template for a more detailed description

`severity_logger_mt` public construct/copy/destruct

1. ```
   severity_logger_mt();
   ```

   Default constructor

2. ```
   severity_logger_mt(severity_logger_mt const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT> explicit severity_logger_mt(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   explicit severity_logger_mt(LevelT level);
   ```

   The constructor creates the logger with the specified default severity level

   Parameters: `level` The default severity level

5. ```
   severity_logger_mt & operator=(severity_logger_mt const & that);
   ```

   Assignment operator

   Swaps two loggers

Class template wseverity_logger

boost::log::sources::wseverity_logger — Wide-char logger. Functionally equivalent to `basic_severity_logger`.
   ```

Synopsis

```
// In header: <boost/log/sources/severity_logger.hpp>

template<typename LevelT = int>
class wseverity_logger : public basic_composite_logger< wchar_t, wseverity_logger< LevelT >, ↵
single_thread_model, features< severity< LevelT > > >
{
public:
  // construct/copy/destruct
  wseverity_logger();
  wseverity_logger(wseverity_logger const &);
  template<typename... ArgsT> explicit wseverity_logger(ArgsT...const &);
  explicit wseverity_logger(LevelT);
  wseverity_logger & operator=(wseverity_logger const &);
};
```

Description

See `severity` class template for a more detailed description

`wseverity_logger` public construct/copy/destruct

1. ```
   wseverity_logger();
   ```

   Default constructor

2. ```
   wseverity_logger(wseverity_logger const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT> explicit wseverity_logger(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   explicit wseverity_logger(LevelT level);
   ```

   The constructor creates the logger with the specified default severity level

   Parameters: `level` The default severity level

5. ```
   wseverity_logger & operator=(wseverity_logger const & that);
   ```

   Assignment operator

   Swaps two loggers

Class template wseverity_logger_mt

boost::log::sources::wseverity_logger_mt — Wide-char thread-safe logger. Functionally equivalent to `basic_severity_logger`.
   ```

Synopsis

```
// In header: <boost/log/sources/severity_logger.hpp>

template<typename LevelT = int>
class wseverity_logger_mt : public basic_composite_logger< wchar_t, wseverity_logger_mt< LevelT ↵
>, multi_thread_model< implementation_defined >, features< severity< LevelT > > >
{
public:
  // construct/copy/destruct
  wseverity_logger_mt();
  wseverity_logger_mt(wseverity_logger_mt const &);
  template<typename... ArgsT> explicit wseverity_logger_mt(ArgsT...const &);
  explicit wseverity_logger_mt(LevelT);
  wseverity_logger_mt & operator=(wseverity_logger_mt const &);
};
```

Description

See `severity` class template for a more detailed description

`wseverity_logger_mt` public construct/copy/destruct

1. ```
   wseverity_logger_mt();
   ```

   Default constructor

2. ```
   wseverity_logger_mt(wseverity_logger_mt const & that);
   ```

   Copy constructor

3. ```
   template<typename... ArgsT> explicit wseverity_logger_mt(ArgsT...const & args);
   ```

   Constructor with named arguments

4. ```
   explicit wseverity_logger_mt(LevelT level);
   ```

   The constructor creates the logger with the specified default severity level

   Parameters: `level` The default severity level

5. ```
   wseverity_logger_mt & operator=(wseverity_logger_mt const & that);
   ```

   Assignment operator

   Swaps two loggers

Header <boost/log/sources/threading_models.hpp>

Andrey Semashev

04.10.2008
The header contains definition of threading models that can be used in loggers. The header also provides a number of tags that can be used to express lock requirements on a function callee.
   ```

```
namespace boost {
  namespace log {
    namespace sources {
      template<typename MutexT> struct multi_thread_model;
      struct single_thread_model;
    }
  }
}
```

Struct template multi_thread_model

boost::log::sources::multi_thread_model — Multi-thread locking model with maximum locking capabilities.

Synopsis

```
// In header: <boost/log/sources/threading_models.hpp>

template<typename MutexT>
struct multi_thread_model {
  // construct/copy/destruct
  multi_thread_model();
  multi_thread_model(multi_thread_model const &);
  multi_thread_model & operator=(multi_thread_model const &);

  // public member functions
  void lock_shared() const;
  bool try_lock_shared() const;
  template<typename TimeT> bool timed_lock_shared(TimeT const &) const;
  void unlock_shared() const;
  void lock() const;
  bool try_lock() const;
  template<typename TimeT> bool timed_lock(TimeT const &) const;
  void unlock() const;
  void lock_upgrade() const;
  bool try_lock_upgrade() const;
  template<typename TimeT> bool timed_lock_upgrade(TimeT const &) const;
  void unlock_upgrade() const;
  void unlock_upgrade_and_lock() const;
  void unlock_and_lock_upgrade() const;
  void unlock_and_lock_shared() const;
  void unlock_upgrade_and_lock_shared() const;
  void swap(multi_thread_model &);
};
```

Description

`multi_thread_model` public construct/copy/destruct

1. ```
   multi_thread_model();
   ```

2. ```
   multi_thread_model(multi_thread_model const &);
   ```

3. ```
   multi_thread_model & operator=(multi_thread_model const &);
   ```
   ```

`multi_thread_model` public member functions

1. ```
   void lock_shared() const;
   ```

2. ```
   bool try_lock_shared() const;
   ```

3. ```
   template<typename TimeT> bool timed_lock_shared(TimeT const & t) const;
   ```

4. ```
   void unlock_shared() const;
   ```

5. ```
   void lock() const;
   ```

6. ```
   bool try_lock() const;
   ```

7. ```
   template<typename TimeT> bool timed_lock(TimeT const & t) const;
   ```

8. ```
   void unlock() const;
   ```

9. ```
   void lock_upgrade() const;
   ```

10. ```
    bool try_lock_upgrade() const;
    ```

11. ```
    template<typename TimeT> bool timed_lock_upgrade(TimeT const & t) const;
    ```

12. ```
    void unlock_upgrade() const;
    ```

13. ```
    void unlock_upgrade_and_lock() const;
    ```

14. ```
    void unlock_and_lock_upgrade() const;
    ```

15. ```
    void unlock_and_lock_shared() const;
    ```

16. ```
    void unlock_upgrade_and_lock_shared() const;
    ```

17.
```
void swap(multi_thread_model &);
```

Struct single_thread_model

boost::log::sources::single_thread_model — Single thread locking model.

Synopsis

```
// In header: <boost/log/sources/threading_models.hpp>


struct single_thread_model {

  // public member functions
  void lock_shared() const;
  bool try_lock_shared() const;
  template<typename TimeT> bool timed_lock_shared(TimeT const &) const;
  void unlock_shared() const;
  void lock() const;
  bool try_lock() const;
  template<typename TimeT> bool timed_lock(TimeT const &) const;
  void unlock() const;
  void lock_upgrade() const;
  bool try_lock_upgrade() const;
  template<typename TimeT> bool timed_lock_upgrade(TimeT const &) const;
  void unlock_upgrade() const;
  void unlock_upgrade_and_lock() const;
  void unlock_and_lock_upgrade() const;
  void unlock_and_lock_shared() const;
  void unlock_upgrade_and_lock_shared() const;
  void swap(single_thread_model &);
};
```

Description

`single_thread_model` public member functions

1.
```
void lock_shared() const;
```

2.
```
bool try_lock_shared() const;
```

3.
```
template<typename TimeT> bool timed_lock_shared(TimeT const &) const;
```

4.
```
void unlock_shared() const;
```

5.
```
void lock() const;
```

6.
```
bool try_lock() const;
```

7. `template<typename TimeT> bool timed_lock(TimeT const &) const;`
8. `void unlock() const;`
9. `void lock_upgrade() const;`
10. `bool try_lock_upgrade() const;`
11. `template<typename TimeT> bool timed_lock_upgrade(TimeT const &) const;`
12. `void unlock_upgrade() const;`
13. `void unlock_upgrade_and_lock() const;`
14. `void unlock_and_lock_upgrade() const;`
15. `void unlock_and_lock_shared() const;`
16. `void unlock_upgrade_and_lock_shared() const;`
17. `void swap(single_thread_model &);`

Sinks

Header <boost/log/sinks/async_frontend.hpp>

Andrey Semashev

14.07.2009
The header contains implementation of asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename SinkBackendT,
               typename QueueingStrategyT = unbounded_fifo_queue>
        class asynchronous_sink;
    }
  }
}
```

Class template asynchronous_sink

boost::log::sinks::asynchronous_sink — Asynchronous logging sink frontend.

Synopsis

```
// In header: <boost/log/sinks/async_frontend.hpp>

template<typename SinkBackendT,
         typename QueueingStrategyT = unbounded_fifo_queue>
class asynchronous_sink :
  public basic_sink_frontend, public QueueingStrategyT
{
public:
  // types
  typedef SinkBackendT           sink_backend_type;   // Sink implementation type.
  typedef implementation_defined locked_backend_ptr;  // A pointer type that locks the backend ↵
until it's destroyed.

  // member classes/structs/unions

  // A scope guard that resets a flag on destructor.

  class scoped_flag {
  public:
    // construct/copy/destruct
    explicit scoped_flag(frontend_mutex_type &, condition_variable_any &,
                         volatile bool &);
    scoped_flag(scoped_flag const &);
    scoped_flag & operator=(scoped_flag const &);
    ~scoped_flag();
  };

  // A scope guard that implements thread ID management.

  class scoped_thread_id {
  public:
    // construct/copy/destruct
    scoped_thread_id(frontend_mutex_type &, condition_variable_any &,
                     thread::id &, bool volatile &);
    scoped_thread_id(unique_lock< frontend_mutex_type > &,
                     condition_variable_any &, thread::id &, bool volatile &);
    scoped_thread_id(scoped_thread_id const &);
    scoped_thread_id & operator=(scoped_thread_id const &);
    ~scoped_thread_id();
  };

  // construct/copy/destruct
  asynchronous_sink(bool = true);
  explicit asynchronous_sink(shared_ptr< sink_backend_type > const &,
                             bool = true);
  ~asynchronous_sink();

  // public member functions
  locked_backend_ptr locked_backend();
  virtual void consume(record_view const &);
  virtual bool try_consume(record_view const &);
  void run();
  void stop();
  void feed_records();
  virtual void flush();
};
```

Description

The frontend starts a separate thread on construction. All logging records are passed to the backend in this dedicated thread only.

`asynchronous_sink` public construct/copy/destruct

1. ```
   asynchronous_sink(bool start_thread = true);
   ```

   Default constructor. Constructs the sink backend instance. Requires the backend to be default-constructible.

   Parameters: `start_thread` If `true`, the frontend creates a thread to feed log records to the backend. Otherwise no thread is started and it is assumed that the user will call either `run` or `feed_records` himself.

2. ```
   explicit asynchronous_sink(shared_ptr< sink_backend_type > const & backend,
                              bool start_thread = true);
   ```

   Constructor attaches user-constructed backend instance

   Parameters: `backend` Pointer to the backend instance.
   `start_thread` If `true`, the frontend creates a thread to feed log records to the backend. Otherwise no thread is started and it is assumed that the user will call either `run` or `feed_records` himself.

   Requires: *backend* is not `NULL`.

3. ```
   ~asynchronous_sink();
   ```

   Destructor. Implicitly stops the dedicated feeding thread, if one is running.

`asynchronous_sink` public member functions

1. ```
   locked_backend_ptr locked_backend();
   ```

   Locking accessor to the attached backend

2. ```
   virtual void consume(record_view const & rec);
   ```

   Enqueues the log record to the backend

3. ```
   virtual bool try_consume(record_view const & rec);
   ```

   The method attempts to pass logging record to the backend

4. ```
   void run();
   ```

   The method starts record feeding loop and effectively blocks until either of this happens:

   - the thread is interrupted due to either standard thread interruption or a call to `stop`
   - an exception is thrown while processing a log record in the backend, and the exception is not terminated by the exception handler, if one is installed

   Requires: The sink frontend must be constructed without spawning a dedicated thread

5. ```
   void stop();
   ```

The method softly interrupts record feeding loop. This method must be called when the `run` method execution has to be interrupted. Unlike regular thread interruption, calling `stop` will not interrupt the record processing in the middle. Instead, the sink frontend will attempt to finish its business with the record in progress and return afterwards. This method can be called either if the sink was created with a dedicated thread, or if the feeding loop was initiated by user.

Note

Returning from this method does not guarantee that there are no records left buffered in the sink frontend. It is possible that log records keep coming during and after this method is called. At some point of execution of this method log records stop being processed, and all records that come after this point are put into the queue. These records will be processed upon further calls to `run` or `feed_records`.

6.
```
void feed_records();
```

The method feeds log records that may have been buffered to the backend and returns

Requires: The sink frontend must be constructed without spawning a dedicated thread

7.
```
virtual void flush();
```

The method feeds all log records that may have been buffered to the backend and returns. Unlike `feed_records`, in case of ordering queueing the method also feeds records that were enqueued during the ordering window, attempting to empty the queue completely.

Requires: The sink frontend must be constructed without spawning a dedicated thread

Class scoped_flag

boost::log::sinks::asynchronous_sink::scoped_flag — A scope guard that resets a flag on destructor.

Synopsis

```
// In header: <boost/log/sinks/async_frontend.hpp>


// A scope guard that resets a flag on destructor.

class scoped_flag {
public:
  // construct/copy/destruct
  explicit scoped_flag(frontend_mutex_type &, condition_variable_any &,
                       volatile bool &);
  scoped_flag(scoped_flag const &);
  scoped_flag & operator=(scoped_flag const &);
  ~scoped_flag();
};
```

Description

`scoped_flag` public construct/copy/destruct

1.
```
explicit scoped_flag(frontend_mutex_type & mut, condition_variable_any & cond,
                     volatile bool & f);
```

2. ```
   scoped_flag(scoped_flag const &);
   ```

3. ```
   scoped_flag & operator=(scoped_flag const &);
   ```

4. ```
   ~scoped_flag();
   ```

Class scoped_thread_id

boost::log::sinks::asynchronous_sink::scoped_thread_id — A scope guard that implements thread ID management.

Synopsis

```
// In header: <boost/log/sinks/async_frontend.hpp>



// A scope guard that implements thread ID management.

class scoped_thread_id {
public:
  // construct/copy/destruct
  scoped_thread_id(frontend_mutex_type &, condition_variable_any &,
                   thread::id &, bool volatile &);
  scoped_thread_id(unique_lock< frontend_mutex_type > &,
                   condition_variable_any &, thread::id &, bool volatile &);
  scoped_thread_id(scoped_thread_id const &);
  scoped_thread_id & operator=(scoped_thread_id const &);
  ~scoped_thread_id();
};
```

Description

`scoped_thread_id` public construct/copy/destruct

1. ```
   scoped_thread_id(frontend_mutex_type & mut, condition_variable_any & cond,
                   thread::id & tid, bool volatile & sr);
   ```

   Initializing constructor.

2. ```
   scoped_thread_id(unique_lock< frontend_mutex_type > & l,
                   condition_variable_any & cond, thread::id & tid,
                   bool volatile & sr);
   ```

   Initializing constructor.

3. ```
   scoped_thread_id(scoped_thread_id const &);
   ```

4. ```
   scoped_thread_id & operator=(scoped_thread_id const &);
   ```
   ```

5. `~scoped_thread_id();`

   Destructor.

Header <boost/log/sinks/attribute_mapping.hpp>

Andrey Semashev

07.11.2008
The header contains facilities that are used in different sinks to map attribute values used throughout the application to values used with the specific native logging API. These tools are mostly needed to map application severity levels on native levels, required by OS-specific sink backends.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename MappedT, typename AttributeValueT = int>
        class basic_custom_mapping;
      template<typename MappedT, typename AttributeValueT = int>
        class basic_direct_mapping;

      template<typename MappedT> struct basic_mapping;
    }
  }
}
```

Class template basic_custom_mapping

boost::log::sinks::basic_custom_mapping — Customizable mapping.

Synopsis

```
// In header: <boost/log/sinks/attribute_mapping.hpp>

template<typename MappedT, typename AttributeValueT = int>
class basic_custom_mapping :
  public boost::log::sinks::basic_mapping< MappedT >
{
public:
  // types
  typedef AttributeValueT        attribute_value_type;  // Attribute contained value type.
  typedef base_type::mapped_type mapped_type;           // Mapped value type.

  // construct/copy/destruct
  explicit basic_custom_mapping(attribute_name const &, mapped_type const &);

  // public member functions
  mapped_type operator()(record_view const &) const;
  implementation_defined operator[](attribute_value_type const &);
};
```

Description

The class allows to setup a custom mapping between an attribute and native values. The mapping should be initialized similarly to the standard `map` container, by using indexing operator and assignment.

Note

Unlike many other components of the library, exact type of the attribute value must be specified in the template parameter `AttributeValueT`. Type sequences are not supported.

`basic_custom_mapping` public construct/copy/destruct

1.
```
explicit basic_custom_mapping(attribute_name const & name,
                              mapped_type const & default_value);
```

Constructor

Parameters:	`default_value`	The default native value that is returned if the conversion cannot be performed
	`name`	Attribute name

`basic_custom_mapping` public member functions

1.
```
mapped_type operator()(record_view const & rec) const;
```

Extraction operator. Extracts the attribute value and attempts to map it onto the native value.

Parameters:	`rec`	A log record to extract value from
Returns:	A mapped value, if mapping was successful, or the default value if mapping did not succeed.	

2.
```
implementation_defined operator[](attribute_value_type const & key);
```

Insertion operator

Parameters:	`key`	Attribute value to be mapped
Returns:	An object of unspecified type that allows to insert a new mapping through assignment. The *key* argument becomes the key attribute value, and the assigned value becomes the mapped native value.	

Class template basic_direct_mapping

boost::log::sinks::basic_direct_mapping — Straightforward mapping.

Synopsis

```
// In header: <boost/log/sinks/attribute_mapping.hpp>

template<typename MappedT, typename AttributeValueT = int>
class basic_direct_mapping :
  public boost::log::sinks::basic_mapping< MappedT >
{
public:
  // types
  typedef AttributeValueT        attribute_value_type;  // Attribute contained value type.
  typedef base_type::mapped_type mapped_type;           // Mapped value type.

  // construct/copy/destruct
  explicit basic_direct_mapping(attribute_name const &, mapped_type const &);

  // public member functions
  mapped_type operator()(record_view const &) const;
};
```

Description

This type of mapping assumes that attribute with a particular name always provides values that map directly onto the native values. The mapping simply returns the extracted attribute value converted to the native value.

`basic_direct_mapping` public construct/copy/destruct

1.
```
explicit basic_direct_mapping(attribute_name const & name,
                              mapped_type const & default_value);
```

Constructor

Parameters: `default_value` The default native value that is returned if the attribute value is not found
`name` Attribute name

`basic_direct_mapping` public member functions

1.
```
mapped_type operator()(record_view const & rec) const;
```

Extraction operator

Parameters: `rec` A log record to extract value from
Returns: An extracted attribute value

Struct template basic_mapping

boost::log::sinks::basic_mapping — Base class for attribute mapping function objects.

Synopsis

```
// In header: <boost/log/sinks/attribute_mapping.hpp>

template<typename MappedT>
struct basic_mapping {
  // types
  typedef MappedT     mapped_type;  // Mapped value type.
  typedef mapped_type result_type;  // Result type.
};
```

Header <boost/log/sinks/basic_sink_backend.hpp>

Andrey Semashev

04.11.2007
The header contains implementation of base classes for sink backends.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename CharT,
               typename FrontendRequirementsT = synchronized_feeding>
        struct basic_formatted_sink_backend;
      template<typename FrontendRequirementsT> struct basic_sink_backend;
    }
  }
}
```

Struct template basic_formatted_sink_backend

boost::log::sinks::basic_formatted_sink_backend — A base class for a logging sink backend with message formatting support.

Synopsis

```
// In header: <boost/log/sinks/basic_sink_backend.hpp>

template<typename CharT,
         typename FrontendRequirementsT = synchronized_feeding>
struct basic_formatted_sink_backend : public basic_sink_backend< combine_requirements< FrontendRe↵
quirementsT, formatted_records >::type >
{
  // types
  typedef CharT                              char_type;              // Character type.
  typedef std::basic_string< char_type >     string_type;            // Formatted string type.
  typedef base_type::frontend_requirements frontend_requirements;  // Frontend requirements.
};
```

Description

The `basic_formatted_sink_backend` class template indicates to the frontend that the backend requires logging record formatting.

The class allows to request encoding conversion in case if the sink backend requires the formatted string in some particular encoding (e.g. if underlying API supports only narrow or wide characters). In order to perform conversion one should specify the desired final character type in the `TargetCharT` template parameter.

Struct template basic_sink_backend

boost::log::sinks::basic_sink_backend — Base class for a logging sink backend.

Synopsis

```
// In header: <boost/log/sinks/basic_sink_backend.hpp>

template<typename FrontendRequirementsT>
struct basic_sink_backend {
  // types
  typedef FrontendRequirementsT frontend_requirements;  // Frontend requirements tag.

  // construct/copy/destruct
  basic_sink_backend() = default;
  basic_sink_backend(basic_sink_backend const &) = delete;
  basic_sink_backend & operator=(basic_sink_backend const &) = delete;
};
```

Description

The `basic_sink_backend` class template defines a number of types that all sink backends are required to define. All sink backends have to derive from the class.

`basic_sink_backend` public construct/copy/destruct

1. ```
   basic_sink_backend() = default;
   ```

2. ```
   basic_sink_backend(basic_sink_backend const &) = delete;
   ```

3.
```
basic_sink_backend & operator=(basic_sink_backend const &) = delete;
```

Header <boost/log/sinks/basic_sink_frontend.hpp>

Andrey Semashev

14.07.2009
The header contains implementation of a base class for sink frontends.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename CharT> class basic_formatting_sink_frontend;
      class basic_sink_frontend;
    }
  }
}
```

Class template basic_formatting_sink_frontend

boost::log::sinks::basic_formatting_sink_frontend — A base class for a logging sink frontend with formatting support.

Synopsis

```
// In header: <boost/log/sinks/basic_sink_frontend.hpp>

template<typename CharT>
class basic_formatting_sink_frontend : public basic_sink_frontend {
public:
  // types
  typedef CharT                           char_type;       // Character type.
  typedef std::basic_string< char_type > string_type;      // Formatted string type.
  typedef basic_formatter< char_type >   formatter_type;  // Formatter function object type.
  typedef formatter_type::stream_type    stream_type;     // Output stream type.

  // member classes/structs/unions

  struct formatting_context {
    // construct/copy/destruct
    formatting_context();
    formatting_context(unsigned int, std::locale const &,
                       formatter_type const &);

    // public data members
    const unsigned int m_Version;    // Object version.
    string_type m_FormattedRecord;    // Formatted log record storage.
    stream_type m_FormattingStream;    // Formatting stream.
    formatter_type m_Formatter;    // Formatter functor.
  };

  // construct/copy/destruct
  explicit basic_formatting_sink_frontend(bool);

  // public member functions
  template<typename FunT> void set_formatter(FunT const &);
  void reset_formatter();
  std::locale getloc() const;
  void imbue(std::locale const &);

  // protected member functions
  formatter_type & formatter();
  template<typename BackendMutexT, typename BackendT>
    void feed_record(record_view const &, BackendMutexT &, BackendT &);
  template<typename BackendMutexT, typename BackendT>
    bool try_feed_record(record_view const &, BackendMutexT &, BackendT &);
};
```

Description

`basic_formatting_sink_frontend` public construct/copy/destruct

1. ```
   explicit basic_formatting_sink_frontend(bool cross_thread);
   ```

   Initializing constructor.

   Parameters: `cross_thread` The flag indicates whether the sink passes log records between different threads

`basic_formatting_sink_frontend` public member functions

1. ```
   template<typename FunT> void set_formatter(FunT const & formatter);
   ```

   The method sets sink-specific formatter function object

2. ```
   void reset_formatter();
   ```

   The method resets the formatter

3. ```
   std::locale getloc() const;
   ```

   The method returns the current locale used for formatting

4. ```
   void imbue(std::locale const & loc);
   ```

   The method sets the locale used for formatting

`basic_formatting_sink_frontend` protected member functions

1. ```
   formatter_type & formatter();
   ```

   Returns reference to the formatter.

2. ```
   template<typename BackendMutexT, typename BackendT>
     void feed_record(record_view const & rec, BackendMutexT & backend_mutex,
                      BackendT & backend);
   ```

   Feeds log record to the backend.

3. ```
   template<typename BackendMutexT, typename BackendT>
     bool try_feed_record(record_view const & rec, BackendMutexT & backend_mutex,
                          BackendT & backend);
   ```

   Attempts to feeds log record to the backend, does not block if *backend_mutex* is locked.

Struct formatting_context

boost::log::sinks::basic_formatting_sink_frontend::formatting_context

Synopsis

```
// In header: <boost/log/sinks/basic_sink_frontend.hpp>



struct formatting_context {
  // construct/copy/destruct
  formatting_context();
  formatting_context(unsigned int, std::locale const &,
                     formatter_type const &);

  // public data members
  const unsigned int m_Version;  // Object version.
  string_type m_FormattedRecord;  // Formatted log record storage.
  stream_type m_FormattingStream;  // Formatting stream.
  formatter_type m_Formatter;  // Formatter functor.
};
```

Description

`formatting_context` public construct/copy/destruct

1.
```
formatting_context();
```

2.
```
formatting_context(unsigned int version, std::locale const & loc,
                   formatter_type const & formatter);
```

Class basic_sink_frontend

boost::log::sinks::basic_sink_frontend — A base class for a logging sink frontend.

Synopsis

```
// In header: <boost/log/sinks/basic_sink_frontend.hpp>


class basic_sink_frontend : public sink {
public:
  // types
  typedef base_type::exception_handler_type exception_handler_type;  // An exception handler type.

  // construct/copy/destruct
  explicit basic_sink_frontend(bool);

  // public member functions
  template<typename FunT> void set_filter(FunT const &);
  void reset_filter();
  template<typename FunT> void set_exception_handler(FunT const &);
  void reset_exception_handler();
  virtual bool will_consume(attribute_value_set const &);

  // protected member functions
  mutex_type & frontend_mutex() const;
  exception_handler_type & exception_handler();
  exception_handler_type const & exception_handler() const;
  template<typename BackendMutexT, typename BackendT>
    void feed_record(record_view const &, BackendMutexT &, BackendT &);
  template<typename BackendMutexT, typename BackendT>
    bool try_feed_record(record_view const &, BackendMutexT &, BackendT &);
  template<typename BackendMutexT, typename BackendT>
    void flush_backend(BackendMutexT &, BackendT &);

  // private member functions
  template<typename BackendMutexT, typename BackendT>
    void flush_backend_impl(BackendMutexT &, BackendT &, mpl::true_);
  template<typename BackendMutexT, typename BackendT>
    void flush_backend_impl(BackendMutexT &, BackendT &, mpl::false_);
};
```

Description

`basic_sink_frontend` public construct/copy/destruct

1.
```
explicit basic_sink_frontend(bool cross_thread);
```

Initializing constructor.

Parameters: `cross_thread` The flag indicates whether the sink passes log records between different threads

`basic_sink_frontend` public member functions

1.
```
template<typename FunT> void set_filter(FunT const & filter);
```

The method sets sink-specific filter functional object

2.
```
void reset_filter();
```

The method resets the filter

3.
```
template<typename FunT> void set_exception_handler(FunT const & handler);
```

The method sets an exception handler function

4.
```
void reset_exception_handler();
```

The method resets the exception handler function

5.
```
virtual bool will_consume(attribute_value_set const & attrs);
```

The method returns `true` if no filter is set or the attribute values pass the filter

Parameters: `attrs` A set of attribute values of a logging record

`basic_sink_frontend` protected member functions

1.
```
mutex_type & frontend_mutex() const;
```

Returns reference to the frontend mutex.

2.
```
exception_handler_type & exception_handler();
```

Returns reference to the exception handler.

3.
```
exception_handler_type const & exception_handler() const;
```

Returns reference to the exception handler.

4.
```
template<typename BackendMutexT, typename BackendT>
  void feed_record(record_view const & rec, BackendMutexT & backend_mutex,
                   BackendT & backend);
```

Feeds log record to the backend.

5.
```
template<typename BackendMutexT, typename BackendT>
  bool try_feed_record(record_view const & rec, BackendMutexT & backend_mutex,
                       BackendT & backend);
```

Attempts to feeds log record to the backend, does not block if *backend_mutex* is locked.

6.
```
template<typename BackendMutexT, typename BackendT>
  void flush_backend(BackendMutexT & backend_mutex, BackendT & backend);
```

Flushes record buffers in the backend, if one supports it.

`basic_sink_frontend` private member functions

1.
```
template<typename BackendMutexT, typename BackendT>
  void flush_backend_impl(BackendMutexT & backend_mutex, BackendT & backend,
                          mpl::true_);
```

Flushes record buffers in the backend (the actual implementation)

2.
```
template<typename BackendMutexT, typename BackendT>
  void flush_backend_impl(BackendMutexT &, BackendT &, mpl::false_);
```

Flushes record buffers in the backend (stub for backends that don't support flushing)

Header <boost/log/sinks/block_on_overflow.hpp>

Andrey Semashev

04.01.2012
The header contains implementation of `block_on_overflow` strategy for handling queue overflows in bounded queues for the asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      class block_on_overflow;
    }
  }
}
```

Class block_on_overflow

boost::log::sinks::block_on_overflow — Blocking strategy for handling log record queue overflows.

Synopsis

```
// In header: <boost/log/sinks/block_on_overflow.hpp>


class block_on_overflow {
};
```

Description

This strategy will cause enqueueing threads to block when the log record queue overflows. The blocked threads will be woken as soon as there appears free space in the queue, in the same order they attempted to enqueue records.

Header <boost/log/sinks/bounded_fifo_queue.hpp>

Andrey Semashev

04.01.2012

The header contains implementation of bounded FIFO queueing strategy for the asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<std::size_t MaxQueueSizeV, typename OverflowStrategyT>
        class bounded_fifo_queue;
    }
  }
}
```

Class template bounded_fifo_queue

boost::log::sinks::bounded_fifo_queue — Bounded FIFO log record queueing strategy.

Synopsis

```
// In header: <boost/log/sinks/bounded_fifo_queue.hpp>

template<std::size_t MaxQueueSizeV, typename OverflowStrategyT>
class bounded_fifo_queue : private OverflowStrategyT {
public:
  // construct/copy/destruct
  bounded_fifo_queue();
  template<typename ArgsT> explicit bounded_fifo_queue(ArgsT const &);

  // protected member functions
  void enqueue(record_view const &);
  bool try_enqueue(record_view const &);
  bool try_dequeue_ready(record_view &);
  bool try_dequeue(record_view &);
  bool dequeue_ready(record_view &);
  void interrupt_dequeue();
};
```

Description

The `bounded_fifo_queue` class is intended to be used with the `asynchronous_sink` frontend as a log record queueing strategy.

This strategy describes log record queueing logic. The queue has a limited capacity, upon reaching which the enqueue operation will invoke the overflow handling strategy specified in the `OverflowStrategyT` template parameter to handle the situation. The library provides overflow handling strategies for most common cases: `drop_on_overflow` will silently discard the log record, and `block_on_overflow` will put the enqueueing thread to wait until there is space in the queue.

The log record queue imposes no ordering over the queued elements aside from the order in which they are enqueued.

`bounded_fifo_queue` public construct/copy/destruct

1. ```
   bounded_fifo_queue();
   ```

   Default constructor.

2. ```
   template<typename ArgsT> explicit bounded_fifo_queue(ArgsT const &);
   ```

   Initializing constructor.

`bounded_fifo_queue` protected member functions

1. ```
   void enqueue(record_view const & rec);
   ```

   Enqueues log record to the queue.

2. ```
   bool try_enqueue(record_view const & rec);
   ```

   Attempts to enqueue log record to the queue.

3. ```
   bool try_dequeue_ready(record_view & rec);
   ```

   Attempts to dequeue a log record ready for processing from the queue, does not block if the queue is empty.

4. ```
   bool try_dequeue(record_view & rec);
   ```

   Attempts to dequeue log record from the queue, does not block if the queue is empty.

5. ```
   bool dequeue_ready(record_view & rec);
   ```

   Dequeues log record from the queue, blocks if the queue is empty.

6. ```
   void interrupt_dequeue();
   ```

   Wakes a thread possibly blocked in the `dequeue` method.

Header <boost/log/sinks/bounded_ordering_queue.hpp>

Andrey Semashev

06.01.2012
The header contains implementation of bounded ordering queueing strategy for the asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename OrderT, std::size_t MaxQueueSizeV,
               typename OverflowStrategyT>
        class bounded_ordering_queue;
    }
  }
}
```

Class template bounded_ordering_queue

boost::log::sinks::bounded_ordering_queue — Bounded ordering log record queueing strategy.

Synopsis

```
// In header: <boost/log/sinks/bounded_ordering_queue.hpp>

template<typename OrderT, std::size_t MaxQueueSizeV,
         typename OverflowStrategyT>
class bounded_ordering_queue : private OverflowStrategyT {
public:
  // construct/copy/destruct
  template<typename ArgsT> explicit bounded_ordering_queue(ArgsT const &);

  // public member functions
  posix_time::time_duration get_ordering_window() const;

  // public static functions
  static posix_time::time_duration get_default_ordering_window();

  // protected member functions
  void enqueue(record_view const &);
  bool try_enqueue(record_view const &);
  bool try_dequeue_ready(record_view &);
  bool try_dequeue(record_view &);
  bool dequeue_ready(record_view &);
  void interrupt_dequeue();
};
```

Description

The `bounded_ordering_queue` class is intended to be used with the `asynchronous_sink` frontend as a log record queueing strategy.

This strategy provides the following properties to the record queueing mechanism:

- The queue has limited capacity specified by the `MaxQueueSizeV` template parameter.
- Upon reaching the size limit, the queue invokes the overflow handling strategy specified in the `OverflowStrategyT` template parameter to handle the situation. The library provides overflow handling strategies for most common cases: `drop_on_overflow` will silently discard the log record, and `block_on_overflow` will put the enqueueing thread to wait until there is space in the queue.
- The queue has a fixed latency window. This means that each log record put into the queue will normally not be dequeued for a certain period of time.
- The queue performs stable record ordering within the latency window. The ordering predicate can be specified in the `OrderT` template parameter.

`bounded_ordering_queue` public construct/copy/destruct

1. ```
   template<typename ArgsT> explicit bounded_ordering_queue(ArgsT const & args);
   ```

   Initializing constructor.

`bounded_ordering_queue` public member functions

1. ```
   posix_time::time_duration get_ordering_window() const;
   ```

   Returns ordering window size specified during initialization

`bounded_ordering_queue` public static functions

1. ```
   static posix_time::time_duration get_default_ordering_window();
   ```

   Returns default ordering window size. The default window size is specific to the operating system thread scheduling mechanism.

`bounded_ordering_queue` protected member functions

1. ```
   void enqueue(record_view const & rec);
   ```

   Enqueues log record to the queue.

2. ```
   bool try_enqueue(record_view const & rec);
   ```

   Attempts to enqueue log record to the queue.

3. ```
   bool try_dequeue_ready(record_view & rec);
   ```

   Attempts to dequeue a log record ready for processing from the queue, does not block if the queue is empty.

4. ```
   bool try_dequeue(record_view & rec);
   ```

   Attempts to dequeue log record from the queue, does not block if the queue is empty.

5. ```
   bool dequeue_ready(record_view & rec);
   ```

   Dequeues log record from the queue, blocks if the queue is empty.

6. ```
   void interrupt_dequeue();
   ```

   Wakes a thread possibly blocked in the `dequeue` method.

Header <boost/log/sinks/debug_output_backend.hpp>

Andrey Semashev

07.11.2008
The header contains a logging sink backend that outputs log records to the debugger.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename CharT> class basic_debug_output_backend;

     typedef basic_debug_output_backend< char > debug_output_backend;  // Convenience typedef ↵
for narrow-character logging.
      typedef basic_debug_output_backend< wchar_t > wdebug_output_backend;  // Convenience ty↵
pedef for wide-character logging.
    }
  }
}
```
```

Class template basic_debug_output_backend

boost::log::sinks::basic_debug_output_backend — An implementation of a logging sink backend that outputs to the debugger.

Synopsis

```
// In header: <boost/log/sinks/debug_output_backend.hpp>

template<typename CharT>
class basic_debug_output_backend :
  public basic_formatted_sink_backend< CharT, concurrent_feeding >
{
public:
  // types
  typedef base_type::char_type   char_type;    // Character type.
  typedef base_type::string_type string_type;  // String type to be used as a message text holder.

  // construct/copy/destruct
  basic_debug_output_backend();
  ~basic_debug_output_backend();

  // public member functions
  void consume(record_view const &, string_type const &);
};
```

Description

The sink uses Windows API in order to write log records as debug messages, if the application process is run under debugger. The sink backend also provides a specific filter that allows to check whether the debugger is available and thus elide unnecessary formatting.

`basic_debug_output_backend` public construct/copy/destruct

1. ```
   basic_debug_output_backend();
   ```

   Constructor. Initializes the sink backend.

2. ```
   ~basic_debug_output_backend();
   ```

   Destructor

`basic_debug_output_backend` public member functions

1. ```
   void consume(record_view const & rec, string_type const & formatted_message);
   ```

   The method passes the formatted message to debugger

Header <boost/log/sinks/drop_on_overflow.hpp>

Andrey Semashev

04.01.2012
The header contains implementation of `drop_on_overflow` strategy for handling queue overflows in bounded queues for the asynchronous sink frontend.
   ```

```
namespace boost {
  namespace log {
    namespace sinks {
      class drop_on_overflow;
    }
  }
}
```

Class drop_on_overflow

boost::log::sinks::drop_on_overflow — Log record dropping strategy.

Synopsis

```
// In header: <boost/log/sinks/drop_on_overflow.hpp>


class drop_on_overflow {
};
```

Description

This strategy will cause log records to be discarded in case of queue overflow in bounded asynchronous sinks. It should not be used if losing log records is not acceptable.

Header <boost/log/sinks/event_log_backend.hpp>

Andrey Semashev

07.11.2008
The header contains a logging sink backend that uses Windows NT event log API for signaling application events.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename CharT> class basic_event_log_backend;
      template<typename CharT> class basic_simple_event_log_backend;

     typedef basic_simple_event_log_backend< char > simple_event_log_backend;  // Convenience typedef for narrow-character logging.
     typedef basic_event_log_backend< char > event_log_backend;  // Convenience typedef for narrow-character logging.
      typedef basic_simple_event_log_backend< wchar_t > wsimple_event_log_backend;  // Convenience typedef for wide-character logging.
      typedef basic_event_log_backend< wchar_t > wevent_log_backend;  // Convenience typedef for wide-character logging.
      namespace event_log {
        template<typename CharT> class basic_event_composer;
        template<typename AttributeValueT = int>
          class custom_event_category_mapping;
        template<typename AttributeValueT = int> class custom_event_id_mapping;
        template<typename AttributeValueT = int> class custom_event_type_mapping;
        template<typename AttributeValueT = int>
          class direct_event_category_mapping;
        template<typename AttributeValueT = int> class direct_event_id_mapping;
        template<typename AttributeValueT = int> class direct_event_type_mapping;

        // Event log source registration modes.
        enum registration_mode { never, on_demand, forced };

       typedef basic_event_composer< char > event_composer;  // Convenience typedef for narrow-character logging.
       typedef basic_event_composer< wchar_t > wevent_composer;  // Convenience typedef for wide-character logging.
      }
    }
  }
}
```

Class template basic_event_composer

boost::log::sinks::event_log::basic_event_composer — An event composer.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename CharT>
class basic_event_composer {
public:
  // types
  typedef CharT                           char_type;             // Character type.
  typedef std::basic_string< char_type > string_type;          // String type to be used as a ↵
message text holder.
  typedef unspecified                   event_id_mapper_type;  // Event identifier mapper type.
  typedef basic_formatter< char_type >   formatter_type;       // Type of an insertion composer ↵
(a formatter)
  typedef std::vector< string_type >     insertion_list;        // Type of the composed inser↵
tions list.

  // construct/copy/destruct
  explicit basic_event_composer(event_id_mapper_type const &);
  basic_event_composer(basic_event_composer const &);
  basic_event_composer & operator=(basic_event_composer);
  ~basic_event_composer();

  // public member functions
  void swap(basic_event_composer &);
  event_map_reference operator[](event_id);
  event_map_reference operator[](int);
  event_id operator()(record_view const &, insertion_list &) const;
};
```

Description

This class is a function object that extracts event identifier from the attribute values set and formats insertion strings for the particular event. Each insertion string is formatted with a distinct formatter, which can be created just like regular sinks formatters.

Before using, the composer must be initialized with the following information:

- Event identifier extraction logic. One can use `basic_direct_event_id_mapping` or `basic_custom_event_id_mapping` classes in order to create such extractor and pass it to the composer constructor.
- Event identifiers and insertion string formatters. The composer provides the following syntax to provide this information:

```
event_composer comp;
comp[MY_EVENT_ID1] % formatter1 % ... % formatterN;
comp[MY_EVENT_ID2] % formatter1 % ... % formatterN;
...
```

The event identifiers in square brackets are provided by the message compiler generated header (the actual names are specified in the .mc file). The formatters represent the insertion strings that will be used to replace placeholders in event messages, thus the number and the order of the formatters must correspond to the message definition.

`basic_event_composer` public construct/copy/destruct

1. ```
   explicit basic_event_composer(event_id_mapper_type const & id_mapper);
   ```

   Default constructor. Creates an empty map of events.

   Parameters: `id_mapper` An event identifier mapping function that will be used to extract event ID from attribute values
   ```

2. ```
   basic_event_composer(basic_event_composer const & that);
   ```

   Copy constructor. Performs a deep copy of the object.

3. ```
   basic_event_composer & operator=(basic_event_composer that);
   ```

   Assignment. Provides strong exception guarantee.

4. ```
   ~basic_event_composer();
   ```

   Destructor

`basic_event_composer` public member functions

1. ```
   void swap(basic_event_composer & that);
   ```

   Swaps `*this` and `that` objects.

2. ```
   event_map_reference operator[](event_id id);
   ```

   Initiates creation of a new event description. The result of the operator can be used to add formatters for insertion strings construction. The returned reference type is implementation detail.

   Parameters: `id` Event identifier.

3. ```
   event_map_reference operator[](int id);
   ```

   Initiates creation of a new event description. The result of the operator can be used to add formatters for insertion strings construction. The returned reference type is implementation detail.

   Parameters: `id` Event identifier.

4. ```
   event_id operator()(record_view const & rec, insertion_list & insertions) const;
   ```

   Event composition operator. Extracts an event identifier from the attribute values by calling event ID mapper. Then runs all formatters that were registered for the event with the extracted ID. The results of formatting are returned in the *insertions* parameter.

   Parameters: `insertions` A sequence of formatted insertion strings
   `rec` Log record view
   Returns: An event identifier that was extracted from `attributes`

Class template custom_event_category_mapping

boost::log::sinks::event_log::custom_event_category_mapping — Customizable event category mapping.
   ```

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class custom_event_category_mapping :
  public basic_custom_mapping< event_category, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit custom_event_category_mapping(attribute_name const &);
};
```

Description

The class allows to setup a custom mapping between an attribute and event categories. The mapping should be initialized similarly to the standard `map` container, by using indexing operator and assignment.

`custom_event_category_mapping` public construct/copy/destruct

1. ```
   explicit custom_event_category_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name

Class template custom_event_id_mapping

boost::log::sinks::event_log::custom_event_id_mapping — Customizable event ID mapping.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class custom_event_id_mapping :
  public basic_custom_mapping< event_id, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit custom_event_id_mapping(attribute_name const &);
};
```

Description

The class allows to setup a custom mapping between an attribute and event identifiers. The mapping should be initialized similarly to the standard `map` container, by using indexing operator and assignment.

`custom_event_id_mapping` public construct/copy/destruct

1. ```
   explicit custom_event_id_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name

Class template custom_event_type_mapping

boost::log::sinks::event_log::custom_event_type_mapping — Customizable event type mapping.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class custom_event_type_mapping :
  public basic_custom_mapping< event_type, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit custom_event_type_mapping(attribute_name const &);
};
```

Description

The class allows to setup a custom mapping between an attribute and native event types. The mapping should be initialized similarly to the standard `map` container, by using indexing operator and assignment.

`custom_event_type_mapping` public construct/copy/destruct

1. ```
   explicit custom_event_type_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name

Class template direct_event_category_mapping

boost::log::sinks::event_log::direct_event_category_mapping — Straightforward event category mapping.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class direct_event_category_mapping :
  public basic_direct_mapping< event_category, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit direct_event_category_mapping(attribute_name const &);
};
```

Description

This type of mapping assumes that attribute with a particular name always provides values that map directly onto the event categories. The mapping simply returns the extracted attribute value converted to the event category.

`direct_event_category_mapping` public construct/copy/destruct

1. ```
   explicit direct_event_category_mapping(attribute_name const & name);
   ```

Constructor

Parameters: `name` Attribute name

Class template direct_event_id_mapping

boost::log::sinks::event_log::direct_event_id_mapping — Straightforward event ID mapping.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class direct_event_id_mapping :
  public basic_direct_mapping< event_id, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit direct_event_id_mapping(attribute_name const &);
};
```

Description

This type of mapping assumes that attribute with a particular name always provides values that map directly onto the event identifiers. The mapping simply returns the extracted attribute value converted to the event ID.

`direct_event_id_mapping` public construct/copy/destruct

1. ```
   explicit direct_event_id_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name

Class template direct_event_type_mapping

boost::log::sinks::event_log::direct_event_type_mapping — Straightforward event type mapping.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename AttributeValueT = int>
class direct_event_type_mapping :
  public basic_direct_mapping< event_type, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit direct_event_type_mapping(attribute_name const &);
};
```

Description

This type of mapping assumes that attribute with a particular name always provides values that map directly onto the native event types. The mapping simply returns the extracted attribute value converted to the native event type.
```

`direct_event_type_mapping` public construct/copy/destruct

1.
```
explicit direct_event_type_mapping(attribute_name const & name);
```

Constructor

Parameters: `name` Attribute name

Class template basic_event_log_backend

boost::log::sinks::basic_event_log_backend — An implementation of a logging sink backend that emits events into Windows NT event log.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename CharT>
class basic_event_log_backend :
  public basic_sink_backend< synchronized_feeding >
{
public:
  // types
  typedef CharT                           char_type;                  // Character type.
  typedef std::basic_string< char_type > string_type;                 // String type.
  typedef std::vector< string_type >     insertion_list;              // Type of the composed ↵
insertions list.
  typedef unspecified                     event_type_mapper_type;     // Mapper type for the ↵
event type.
  typedef unspecified                     event_category_mapper_type; // Mapper type for the ↵
event category.
  typedef unspecified                     event_composer_type;        // Event composer type.

  // construct/copy/destruct
  template<typename T>
    explicit basic_event_log_backend(std::basic_string< T > const &);
  explicit basic_event_log_backend(filesystem::path const &);
  template<typename... ArgsT>
    explicit basic_event_log_backend(ArgsT...const &);
  ~basic_event_log_backend();

  // public member functions
  void consume(record_view const &);
  void set_event_type_mapper(event_type_mapper_type const &);
  void set_event_category_mapper(event_category_mapper_type const &);
  void set_event_composer(event_composer_type const &);

  // public static functions
  static string_type get_default_log_name();
  static string_type get_default_source_name();
};
```

Description

The sink uses Windows NT 5 (Windows 2000) and later event log API to emit events to an event log. The sink acts as an event source. Unlike `basic_simple_event_log_backend`, this sink backend allows users to specify the custom event message file and supports mapping attribute values onto several insertion strings. Although it requires considerably more scaffolding than the simple backend, this allows to support localizable event descriptions.

Besides the file name of the module with event resources, the backend provides the following customizations:

- Remote server UNC address, log name and source name. These parameters have similar meaning to `basic_simple_event_log_backend`.
- Event type and category mappings. These are function object that allow to map attribute values to the according event parameters. One can use mappings in the `event_log` namespace.
- Event composer. This function object extracts event identifier and formats string insertions, that will be used by the API to compose the final event message text.

`basic_event_log_backend` public construct/copy/destruct

1.
```
template<typename T>
  explicit basic_event_log_backend(std::basic_string< T > const & message_file_name);
```

Constructor. Registers event source with name based on the application executable file name in the Application log. If such a registration is already present, it is not overridden.

2.
```
explicit basic_event_log_backend(filesystem::path const & message_file_name);
```

Constructor. Registers event source with name based on the application executable file name in the Application log. If such a registration is already present, it is not overridden.

3.
```
template<typename... ArgsT>
  explicit basic_event_log_backend(ArgsT...const & args);
```

Constructor. Registers event log source with the specified parameters. The following named parameters are supported:

- `message_file` - Specifies the file name that contains resources that describe events and categories.
- `target` - Specifies an UNC path to the remote server to which log records should be sent to. The local machine will be used to process log records, if not specified.
- `log_name` - Specifies the log in which the source should be registered. The result of `get_default_log_name` is used, if the parameter is not specified.
- `log_source` - Specifies the source name. The result of `get_default_source_name` is used, if the parameter is not specified.
- `registration` - Specifies the event source registration mode in the Windows registry. Can have values of the `registration_mode` enum. Default value: `on_demand`.

Parameters: `args` A set of named parameters.

4.
```
~basic_event_log_backend();
```

Destructor. Unregisters event source. The log source description is not removed from the Windows registry.

`basic_event_log_backend` public member functions

1.
```
void consume(record_view const & rec);
```

The method creates an event in the event log

Parameters: `rec` Log record to consume

2.
```
void set_event_type_mapper(event_type_mapper_type const & mapper);
```

The method installs the function object that maps application severity levels to WinAPI event types

3. ```
   void set_event_category_mapper(event_category_mapper_type const & mapper);
   ```

   The method installs the function object that extracts event category from attribute values

4. ```
   void set_event_composer(event_composer_type const & composer);
   ```

   The method installs the function object that extracts event identifier from the attributes and creates insertion strings that will replace placeholders in the event message.

`basic_event_log_backend` public static functions

1. ```
   static string_type get_default_log_name();
   ```

   Returns: Default log name: Application

2. ```
   static string_type get_default_source_name();
   ```

   Returns: Default log source name that is based on the application executable file name and the sink name

Class template basic_simple_event_log_backend

boost::log::sinks::basic_simple_event_log_backend — An implementation of a simple logging sink backend that emits events into Windows NT event log.

Synopsis

```
// In header: <boost/log/sinks/event_log_backend.hpp>

template<typename CharT>
class basic_simple_event_log_backend :
  public basic_formatted_sink_backend< CharT, concurrent_feeding >
{
public:
  // types
  typedef base_type::char_type   char_type;               // Character type.
  typedef base_type::string_type string_type;           // String type to be used as a message ↵
text holder.
  typedef unspecified            event_type_mapper_type;  // Mapper type for the event type.

  // construct/copy/destruct
  basic_simple_event_log_backend();
  template<typename... ArgsT>
    explicit basic_simple_event_log_backend(ArgsT...const &);
  ~basic_simple_event_log_backend();

  // public member functions
  void set_event_type_mapper(event_type_mapper_type const &);
  void consume(record_view const &, string_type const &);

  // public static functions
  static string_type get_default_log_name();
  static string_type get_default_source_name();
};
```

Description

The sink uses Windows NT 5 (Windows 2000) and later event log API to emit events to an event log. The sink acts as an event source in terms of the API, it implements all needed resources and source registration in the Windows registry that is needed for the event delivery.

The backend performs message text formatting. The composed text is then passed as the first and only string parameter of the event. The resource embedded into the backend describes the event so that the parameter is inserted into the event description text, thus making it visible in the event log.

The backend allows to customize mapping of application severity levels to the native Windows event types. This allows to write portable code even if OS-specific sinks, such as this one, are used.

Note

Since the backend registers itself into Windows registry as the resource file that contains event description, it is important to keep the library binary in a stable place of the filesystem. Otherwise Windows might not be able to load event resources from the library and display events correctly.

Note

It is known that Windows is not able to find event resources in the application executable, which is linked against the static build of the library. Users are advised to use dynamic builds of the library to solve this problem.

`basic_simple_event_log_backend` public construct/copy/destruct

1. ```
   basic_simple_event_log_backend();
   ```

   Default constructor. Registers event source with name based on the application executable file name in the Application log. If such a registration is already present, it is not overridden.

2. ```
   template<typename... ArgsT>
     explicit basic_simple_event_log_backend(ArgsT...const & args);
   ```

   Constructor. Registers event log source with the specified parameters. The following named parameters are supported:

   - `target` - Specifies an UNC path to the remote server which log records should be sent to. The local machine will be used to process log records, if not specified.
   - `log_name` - Specifies the log in which the source should be registered. The result of `get_default_log_name` is used, if the parameter is not specified.
   - `log_source` - Specifies the source name. The result of `get_default_source_name` is used, if the parameter is not specified.
   - `registration` - Specifies the event source registration mode in the Windows registry. Can have values of the `registration_mode` enum. Default value: `on_demand`.

   Parameters: `args` A set of named parameters.

3. ```
   ~basic_simple_event_log_backend();
   ```

   Destructor. Unregisters event source. The log source description is not removed from the Windows registry.
   ```

`basic_simple_event_log_backend` public member functions

1. ```
   void set_event_type_mapper(event_type_mapper_type const & mapper);
   ```

   The method installs the function object that maps application severity levels to WinAPI event types

2. ```
   void consume(record_view const & rec, string_type const & formatted_message);
   ```

   The method puts the formatted message to the event log

`basic_simple_event_log_backend` public static functions

1. ```
   static string_type get_default_log_name();
   ```

   Returns: Default log name: Application

2. ```
   static string_type get_default_source_name();
   ```

   Returns: Default log source name that is based on the application executable file name and the sink name

Header <boost/log/sinks/event_log_constants.hpp>

Andrey Semashev

07.11.2008
The header contains definition of constants related to Windows NT Event Log API. The constants can be used in other places without the event log backend.

```
namespace boost {
  namespace log {
    namespace sinks {
      namespace event_log {

        // Windows event types.
        enum event_type { success = = 0, info = = 4, warning = = 2,
                          error = = 1 };

        typedef unspecified event_id;  // A tagged integral type that represents event identifi↵
er for the Windows API.
        typedef unspecified event_category;  // A tagged integral type that represents event cat↵
egory for the Windows API.
        event_id make_event_id(unsigned int);
        event_category make_event_category(unsigned short);
        event_type make_event_type(unsigned short);
      }
    }
  }
}
```

Function make_event_id

boost::log::sinks::event_log::make_event_id

Synopsis

```
// In header: <boost/log/sinks/event_log_constants.hpp>

event_id make_event_id(unsigned int id);
```

Description

The function constructs event identifier from an integer

Function make_event_category

boost::log::sinks::event_log::make_event_category

Synopsis

```
// In header: <boost/log/sinks/event_log_constants.hpp>

event_category make_event_category(unsigned short cat);
```

Description

The function constructs event category from an integer

Function make_event_type

boost::log::sinks::event_log::make_event_type

Synopsis

```
// In header: <boost/log/sinks/event_log_constants.hpp>

event_type make_event_type(unsigned short lev);
```

Description

The function constructs log record level from an integer

Header <boost/log/sinks/frontend_requirements.hpp>

Andrey Semashev

22.04.2007
The header contains definition of requirement tags that sink backend may declare with regard to frontends. These requirements ensure that a backend will not be used with an incompatible frontend.

```
BOOST_LOG_COMBINE_REQUIREMENTS_LIMIT
```

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename... RequirementsT> struct combine_requirements;
      struct concurrent_feeding;
      struct flushing;
      struct formatted_records;
      template<typename TestedT, typename RequiredT> struct has_requirement;
      struct synchronized_feeding;
    }
  }
}
```

Struct template combine_requirements

boost::log::sinks::combine_requirements

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>

template<typename... RequirementsT>
struct combine_requirements {
};
```

Description

The metafunction combines multiple requirement tags into one type. The resulting type will satisfy all specified requirements (i.e. `has_requirement` metafunction will return positive result).

Struct concurrent_feeding

boost::log::sinks::concurrent_feeding

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>


struct concurrent_feeding : public synchronized_feeding {
};
```

Description

The sink backend ensures all needed synchronization, it is capable to handle multithreaded calls

Struct flushing

boost::log::sinks::flushing

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>


struct flushing {
};
```

Description

The sink backend supports flushing

Struct formatted_records

boost::log::sinks::formatted_records

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>


struct formatted_records {
};
```

Description

The sink backend requires the frontend to perform log record formatting before feeding

Struct template has_requirement

boost::log::sinks::has_requirement

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>

template<typename TestedT, typename RequiredT>
struct has_requirement : public is_base_of< RequiredT, TestedT > {
};
```

Description

A helper metafunction to check if a requirement is satisfied. The `TestedT` template argument should be the type combining one or several requirements and `RequiredT` is the requirement to test against. The metafunction will yield a positive result if `TestedT` supports `RequiredT`.

Struct synchronized_feeding

boost::log::sinks::synchronized_feeding

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>


struct synchronized_feeding {
};
```

Description

The sink backend expects pre-synchronized calls, all needed synchronization is implemented in the frontend (IOW, only one thread is feeding records to the backend concurrently, but it is possible for several threads to write sequentially). Note that if a frontend supports synchronized record feeding, it will also report capable of concurrent record feeding.

Macro BOOST_LOG_COMBINE_REQUIREMENTS_LIMIT

BOOST_LOG_COMBINE_REQUIREMENTS_LIMIT — The macro specifies the maximum number of requirements that can be combined with the `combine_requirements` metafunction.

Synopsis

```
// In header: <boost/log/sinks/frontend_requirements.hpp>

BOOST_LOG_COMBINE_REQUIREMENTS_LIMIT
```

Header <boost/log/sinks/sink.hpp>

Andrey Semashev

22.04.2007
The header contains an interface declaration for all sinks. This interface is used by the logging core to feed log records to sinks.

```
namespace boost {
  namespace log {
    namespace sinks {
      class sink;
    }
  }
}
```

Class sink

boost::log::sinks::sink — A base class for a logging sink frontend.

Synopsis

```
// In header: <boost/log/sinks/sink.hpp>


class sink {
public:
  // types
  typedef unspecified exception_handler_type;  // An exception handler type.

  // construct/copy/destruct
  explicit sink(bool);
  sink(sink const &) = delete;
  sink & operator=(sink const &) = delete;
  ~sink();

  // public member functions
  virtual bool will_consume(attribute_value_set const &) = 0;
  virtual void consume(record_view const &) = 0;
  virtual bool try_consume(record_view const &);
  virtual void flush() = 0;
  bool is_cross_thread() const noexcept;
};
```

Description

`sink` public construct/copy/destruct

1. ```
   explicit sink(bool cross_thread);
   ```

   Default constructor

2. ```
   sink(sink const &) = delete;
   ```

3. ```
   sink & operator=(sink const &) = delete;
   ```

4. ```
   ~sink();
   ```

   Virtual destructor

`sink` public member functions

1. ```
   virtual bool will_consume(attribute_value_set const & attributes) = 0;
   ```

   The method returns `true` if no filter is set or the attribute values pass the filter

   Parameters: `attributes` A set of attribute values of a logging record

2. ```
   virtual void consume(record_view const & rec) = 0;
   ```

   The method puts logging record to the sink

   Parameters: `rec` Logging record to consume

3. ```
   virtual bool try_consume(record_view const & rec);
   ```

   The method attempts to put logging record to the sink. The method may be used by the core in order to determine the most efficient order of sinks to feed records to in case of heavy contention. Sink implementations may implement try/backoff logic in order to improve overall logging throughput.

   Parameters: `rec` Logging record to consume
   Returns: `true`, if the record was consumed, `false`, if not.

4. ```
   virtual void flush() = 0;
   ```

   The method performs flushing of any internal buffers that may hold log records. The method may take considerable time to complete and may block both the calling thread and threads attempting to put new records into the sink while this call is in progress.

5. ```
   bool is_cross_thread() const noexcept;
   ```

   The method indicates that the sink passes log records between different threads. This information is needed by the logging core to detach log records from all thread-specific resources before passing it to the sink.

Header <boost/log/sinks/sync_frontend.hpp>

Andrey Semashev

14.07.2009
The header contains implementation of synchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename SinkBackendT> class synchronous_sink;
    }
  }
}
```

Class template synchronous_sink

boost::log::sinks::synchronous_sink — Synchronous logging sink frontend.
```

Synopsis

```
// In header: <boost/log/sinks/sync_frontend.hpp>

template<typename SinkBackendT>
class synchronous_sink : public basic_sink_frontend {
public:
  // types
  typedef SinkBackendT           sink_backend_type;   // Sink implementation type.
  typedef implementation_defined locked_backend_ptr;  // A pointer type that locks the backend ↵
until it's destroyed.

  // construct/copy/destruct
  synchronous_sink();
  explicit synchronous_sink(shared_ptr< sink_backend_type > const &);

  // public member functions
  locked_backend_ptr locked_backend();
  virtual void consume(record_view const &);
  virtual bool try_consume(record_view const &);
  virtual void flush();
};
```

Description

The sink frontend serializes threads before passing logging records to the backend

`synchronous_sink` public construct/copy/destruct

1. ```
   synchronous_sink();
   ```

   Default constructor. Constructs the sink backend instance. Requires the backend to be default-constructible.

2. ```
   explicit synchronous_sink(shared_ptr< sink_backend_type > const & backend);
   ```

   Constructor attaches user-constructed backend instance

   Parameters: `backend` Pointer to the backend instance
   Requires: *backend* is not `NULL`.

`synchronous_sink` public member functions

1. ```
   locked_backend_ptr locked_backend();
   ```

   Locking accessor to the attached backend

2. ```
   virtual void consume(record_view const & rec);
   ```

   Passes the log record to the backend

3. ```
   virtual bool try_consume(record_view const & rec);
   ```

   The method attempts to pass logging record to the backend

4. ```
   virtual void flush();
   ```

The method performs flushing of any internal buffers that may hold log records. The method may take considerable time to complete and may block both the calling thread and threads attempting to put new records into the sink while this call is in progress.

Header <boost/log/sinks/syslog_backend.hpp>

Andrey Semashev

08.01.2008
The header contains implementation of a Syslog sink backend along with its setup facilities.

```
namespace boost {
  namespace log {
    namespace sinks {
      class syslog_backend;

      // Supported IP protocol versions.
      enum ip_versions { v4, v6 };
      namespace syslog {
        template<typename AttributeValueT = int> class custom_severity_mapping;
        template<typename AttributeValueT = int> class direct_severity_mapping;

        // The enumeration defined the possible implementation types for the syslog backend.
        enum impl_types { native = = 0, udp_socket_based = = 1 };
      }
    }
  }
}
```

Class template custom_severity_mapping

boost::log::sinks::syslog::custom_severity_mapping — Customizable severity level mapping.

Synopsis

```
// In header: <boost/log/sinks/syslog_backend.hpp>

template<typename AttributeValueT = int>
class custom_severity_mapping :
  public basic_custom_mapping< level, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit custom_severity_mapping(attribute_name const &);
};
```

Description

The class allows to setup a custom mapping between an attribute and Syslog severity levels. The mapping should be initialized similarly to the standard `map` container, by using indexing operator and assignment.

`custom_severity_mapping` public construct/copy/destruct

1. ```
   explicit custom_severity_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name
   ```

Class template direct_severity_mapping

boost::log::sinks::syslog::direct_severity_mapping — Straightforward severity level mapping.

Synopsis

```
// In header: <boost/log/sinks/syslog_backend.hpp>

template<typename AttributeValueT = int>
class direct_severity_mapping :
  public basic_direct_mapping< level, AttributeValueT >
{
public:
  // construct/copy/destruct
  explicit direct_severity_mapping(attribute_name const &);
};
```

Description

This type of mapping assumes that attribute with a particular name always provides values that map directly onto the Syslog levels. The mapping simply returns the extracted attribute value converted to the Syslog severity level.

`direct_severity_mapping` public construct/copy/destruct

1. ```
   explicit direct_severity_mapping(attribute_name const & name);
   ```

   Constructor

   Parameters: `name` Attribute name

Class syslog_backend

boost::log::sinks::syslog_backend — An implementation of a syslog sink backend.
   ```

Synopsis

```
// In header: <boost/log/sinks/syslog_backend.hpp>


class syslog_backend : public basic_formatted_sink_backend< char > {
public:
  // types
  typedef base_type::char_type   char_type;             // Character type.
  typedef base_type::string_type string_type;          // String type that is used to pass mes↵
sage test.
  typedef unspecified            severity_mapper_type;  // Syslog severity level mapper type.

  // construct/copy/destruct
  syslog_backend();
  template<typename... ArgsT> explicit syslog_backend(ArgsT...const &);
  ~syslog_backend();

  // public member functions
  void set_severity_mapper(severity_mapper_type const &);
  void set_local_address(std::string const &, unsigned short = 514);
  void set_local_address(boost::asio::ip::address const &,
                         unsigned short = 514);
  void set_target_address(std::string const &, unsigned short = 514);
  void set_target_address(boost::asio::ip::address const &,
                          unsigned short = 514);
  void consume(record_view const &, string_type const &);
};
```

Description

The backend provides support for the syslog protocol, defined in RFC3164. The backend sends log records to a remote host via UDP. The host name can be specified by calling the `set_target_address` method. By default log records will be sent to localhost:514. The local address can be specified as well, by calling the `set_local_address` method. By default syslog packets will be sent from any local address available.

It is safe to create several sink backends with the same local addresses - the backends within the process will share the same socket. The same applies to different processes that use the syslog backends to send records from the same socket. However, it is not guaranteed to work if some third party facility is using the socket.

On systems with native syslog implementation it may be preferable to utilize the POSIX syslog API instead of direct socket management in order to bypass possible security limitations that may be in action. To do so one has to pass the `use_impl = native` to the backend constructor. Note, however, that in that case you will only have one chance to specify syslog facility and process identification string - on the first native syslog backend construction. Other native syslog backends will ignore these parameters. Obviously, the `set_local_address` and `set_target_address` methods have no effect for native backends. Using `use_impl = native` on platforms with no native support for POSIX syslog API will have no effect.

`syslog_backend` public construct/copy/destruct

1. ```
   syslog_backend();
   ```

   Constructor. Creates a UDP socket-based backend with `syslog::user` facility code. IPv4 protocol will be used.

2. ```
   template<typename... ArgsT> explicit syslog_backend(ArgsT...const & args);
   ```

   Constructor. Creates a sink backend with the specified named parameters. The following named parameters are supported:

   - `facility` - Specifies the facility code. If not specified, `syslog::user` will be used.

- `use_impl` - Specifies the backend implementation. Can be one of:
- `native` - Use the native syslog API, if available. If no native API is available, it is equivalent to `udp_socket_based`.
- `udp_socket_based` - Use the UDP socket-based implementation, conforming to RFC3164 protocol specification. This is the default.
- `ip_version` - Specifies IP protocol version to use, in case if socket-based implementation is used. Can be either `v4` (the default one) or `v6`.
- `ident` - Process identification string. This parameter is only supported by native syslog implementation.

3.
```
~syslog_backend();
```

Destructor

`syslog_backend` public member functions

1.
```
void set_severity_mapper(severity_mapper_type const & mapper);
```

The method installs the function object that maps application severity levels to syslog levels

2.
```
void set_local_address(std::string const & addr, unsigned short port = 514);
```

The method sets the local host name which log records will be sent from. The host name is resolved to obtain the final IP address.

Note

Does not have effect if the backend was constructed to use native syslog API

Parameters:
`addr` The local address
`port` The local port number

3.
```
void set_local_address(boost::asio::ip::address const & addr,
                       unsigned short port = 514);
```

The method sets the local address which log records will be sent from.

Note

Does not have effect if the backend was constructed to use native syslog API

Parameters:
`addr` The local address
`port` The local port number

4.
```
void set_target_address(std::string const & addr, unsigned short port = 514);
```

The method sets the remote host name where log records will be sent to. The host name is resolved to obtain the final IP address.

Note

Does not have effect if the backend was constructed to use native syslog API

Parameters: `addr` The remote host address
`port` The port number on the remote host

5.
```
void set_target_address(boost::asio::ip::address const & addr,
                        unsigned short port = 514);
```

The method sets the address of the remote host where log records will be sent to.

Note

Does not have effect if the backend was constructed to use native syslog API

Parameters: `addr` The remote host address
`port` The port number on the remote host

6.
```
void consume(record_view const & rec, string_type const & formatted_message);
```

The method passes the formatted message to the syslog API or sends to a syslog server

Header <boost/log/sinks/syslog_constants.hpp>

Andrey Semashev

08.01.2008
The header contains definition of constants related to Syslog API. The constants can be used in other places without the Syslog backend.

```
namespace boost {
  namespace log {
    namespace sinks {
      namespace syslog {

        // Syslog record levels.
        enum level { emergency = = 0, alert = = 1, critical = = 2,
                     error = = 3, warning = = 4, notice = = 5, info = = 6,
                     debug = = 7 };

        // Syslog facility codes.
        enum facility { kernel = = 0 * 8, user = = 1 * 8, mail = = 2 * 8,
                        daemon = = 3 * 8, security0 = = 4 * 8,
                        syslogd = = 5 * 8, printer = = 6 * 8, news = = 7 * 8,
                        uucp = = 8 * 8, clock0 = = 9 * 8,
                        security1 = = 10 * 8, ftp = = 11 * 8, ntp = = 12 * 8,
                        log_audit = = 13 * 8, log_alert = = 14 * 8,
                        clock1 = = 15 * 8, local0 = = 16 * 8,
                        local1 = = 17 * 8, local2 = = 18 * 8,
                        local3 = = 19 * 8, local4 = = 20 * 8,
                        local5 = = 21 * 8, local6 = = 22 * 8,
                        local7 = = 23 * 8 };
        level make_level(int);
        facility make_facility(int);
      }
    }
  }
}
```

Function make_level

boost::log::sinks::syslog::make_level

Synopsis

```
// In header: <boost/log/sinks/syslog_constants.hpp>


level make_level(int lev);
```

Description

The function constructs log record level from an integer

Function make_facility

boost::log::sinks::syslog::make_facility

Synopsis

```
// In header: <boost/log/sinks/syslog_constants.hpp>


facility make_facility(int fac);
```

Description

The function constructs log source facility from an integer

Header <boost/log/sinks/text_file_backend.hpp>

Andrey Semashev

09.06.2009
The header contains implementation of a text file sink backend.

```
namespace boost {
  namespace log {
    namespace sinks {
      class text_file_backend;
      namespace file {
        struct collector;

        class rotation_at_time_interval;
        class rotation_at_time_point;

        // The enumeration of the stored files scan methods.
        enum scan_method { no_scan, scan_matching, scan_all };
        template<typename... ArgsT>
          shared_ptr< collector > make_collector(ArgsT...const &);
      }
    }
  }
}
```

Struct collector

boost::log::sinks::file::collector — Base class for file collectors.

Synopsis

```
// In header: <boost/log/sinks/text_file_backend.hpp>


struct collector {
  // construct/copy/destruct
  collector() = default;
  collector(collector const &) = delete;
  collector & operator=(collector const &) = delete;
  ~collector();

  // public member functions
  virtual void store_file(filesystem::path const &) = 0;
  virtual uintmax_t
  scan_for_files(scan_method, filesystem::path const & = filesystem::path(),
                 unsigned int * = 0) = 0;
};
```

Description

All file collectors, supported by file sink backends, should inherit this class.

`collector` public construct/copy/destruct

1. ```
   collector() = default;
   ```

   Default constructor

2. ```
   collector(collector const &) = delete;
   ```

3. ```
   collector & operator=(collector const &) = delete;
   ```

4. ```
   ~collector();
   ```

   Virtual destructor

`collector` public member functions

1. ```
   virtual void store_file(filesystem::path const & src_path) = 0;
   ```

   The function stores the specified file in the storage. May lead to an older file deletion and a long file moving.

   Parameters: `src_path` The name of the file to be stored

2. ```
   virtual uintmax_t
   scan_for_files(scan_method method,
                  filesystem::path const & pattern = filesystem::path(),
                  unsigned int * counter = 0) = 0;
   ```

Scans the target directory for the files that have already been stored. The found files are added to the collector in order to be tracked and erased, if needed.

The function may scan the directory in two ways: it will either consider every file in the directory a log file, or will only consider files with names that match the specified pattern. The pattern may contain the following placeholders:

- y, Y, m, d - date components, in Boost.DateTime meaning.
- H, M, S, f - time components, in Boost.DateTime meaning.
- N - numeric file counter. May also contain width specification in printf-compatible form (e.g. %5N). The resulting number will always be zero-filled.
- %% - a percent sign

All other placeholders are not supported.

Note

In case if *method* is `scan_matching` the effect of this function is highly dependent on the *pattern* definition. It is recommended to choose patterns with easily distinguished placeholders (i.e. having delimiters between them). Otherwise either some files can be mistakenly found or not found, which in turn may lead to an incorrect file deletion.

Parameters:	`counter`	If not `NULL` and *method* is `scan_matching`, the method suggests initial value of a file counter that may be used in the file name pattern. The parameter is not used otherwise.
	`method`	The method of scanning. If `no_scan` is specified, the call has no effect.
	`pattern`	The file name pattern if *method* is `scan_matching`. Otherwise the parameter is not used.
Returns:	The number of found files.	

Class rotation_at_time_interval

boost::log::sinks::file::rotation_at_time_interval

Synopsis

```
// In header: <boost/log/sinks/text_file_backend.hpp>


class rotation_at_time_interval {
public:
  // types
  typedef bool result_type;

  // construct/copy/destruct
  explicit rotation_at_time_interval(posix_time::time_duration const &);

  // public member functions
  bool operator()() const;
};
```

Description

The class represents the time interval of log file rotation. The log file will be rotated after the specified time interval has passed.

`rotation_at_time_interval` public construct/copy/destruct

1. ```
   explicit rotation_at_time_interval(posix_time::time_duration const & interval);
   ```
   ```

Creates a rotation time interval of the specified duration

Parameters: `interval` The interval of the rotation, should be no less than 1 second

`rotation_at_time_interval` public member functions

1.
```
bool operator()() const;
```

Checks if it's time to rotate the file

Class rotation_at_time_point

boost::log::sinks::file::rotation_at_time_point

Synopsis

```
// In header: <boost/log/sinks/text_file_backend.hpp>


class rotation_at_time_point {
public:
  // types
  typedef bool result_type;

  // construct/copy/destruct
  explicit rotation_at_time_point(unsigned char, unsigned char, unsigned char);
  explicit rotation_at_time_point(date_time::weekdays, unsigned char = 0,
                                  unsigned char = 0, unsigned char = 0);
  explicit rotation_at_time_point(gregorian::greg_day, unsigned char = 0,
                                  unsigned char = 0, unsigned char = 0);

  // public member functions
  bool operator()() const;
};
```

Description

The class represents the time point of log file rotation. One can specify one of three types of time point based rotation:

- rotation takes place every day, at the specified time
- rotation takes place on the specified day of every week, at the specified time
- rotation takes place on the specified day of every month, at the specified time

The time points are considered to be local time.

`rotation_at_time_point` public construct/copy/destruct

1.
```
explicit rotation_at_time_point(unsigned char hour, unsigned char minute,
                                unsigned char second);
```

Creates a rotation time point of every day at the specified time

Parameters:

`hour`	The rotation hour, should be within 0 and 23
`minute`	The rotation minute, should be within 0 and 59
`second`	The rotation second, should be within 0 and 59

2.
```
explicit rotation_at_time_point(date_time::weekdays wday,
                                unsigned char hour = 0,
                                unsigned char minute = 0,
                                unsigned char second = 0);
```

Creates a rotation time point of each specified weekday at the specified time

Parameters:

`hour`	The rotation hour, should be within 0 and 23
`minute`	The rotation minute, should be within 0 and 59
`second`	The rotation second, should be within 0 and 59
`wday`	The weekday of the rotation

3.
```
explicit rotation_at_time_point(gregorian::greg_day mday,
                                unsigned char hour = 0,
                                unsigned char minute = 0,
                                unsigned char second = 0);
```

Creates a rotation time point of each specified day of month at the specified time

Parameters:

`hour`	The rotation hour, should be within 0 and 23
`mday`	The monthday of the rotation, should be within 1 and 31
`minute`	The rotation minute, should be within 0 and 59
`second`	The rotation second, should be within 0 and 59

`rotation_at_time_point` public member functions

1.
```
bool operator()() const;
```

Checks if it's time to rotate the file

Function template make_collector

boost::log::sinks::file::make_collector

Synopsis

```
// In header: <boost/log/sinks/text_file_backend.hpp>


template<typename... ArgsT>
  shared_ptr< collector > make_collector(ArgsT...const & args);
```

Description

The function creates a file collector for the specified target directory. Each target directory is managed by a single file collector, so if this function is called several times for the same directory, it will return a reference to the same file collector. It is safe to use the same collector in different sinks, even in a multithreaded application.

One can specify certain restrictions for the stored files, such as maximum total size or minimum free space left in the target directory. If any of the specified restrictions is not met, the oldest stored file is deleted. If the same collector is requested more than once with different restrictions, the collector will act according to the most strict combination of all specified restrictions.

The following named parameters are supported:

- `target` - Specifies the target directory for the files being stored in. This parameter is mandatory.

- `max_size` - Specifies the maximum total size, in bytes, of stored files that the collector will try not to exceed. If the size exceeds this threshold the oldest file(s) is deleted to free space. Note that the threshold may be exceeded if the size of individual files exceed the `max_size` value. The threshold is not maintained, if not specified.
- `min_free_space` - Specifies the minimum free space, in bytes, in the target directory that the collector tries to maintain. If the threshold is exceeded, the oldest file(s) is deleted to free space. The threshold is not maintained, if not specified.

Returns: The file collector.

Class text_file_backend

boost::log::sinks::text_file_backend — An implementation of a text file logging sink backend.

Synopsis

```
// In header: <boost/log/sinks/text_file_backend.hpp>


class text_file_backend : public basic_formatted_sink_backend< char, combine_requirements< syn↵
chronized_feeding, flushing >::type >
{
public:
  // types
  typedef base_type::char_type           char_type;                    // Character type.
  typedef base_type::string_type         string_type;                  // String type to be ↵
used as a message text holder.
  typedef std::basic_ostream< char_type > stream_type;                  // Stream type.
  typedef unspecified                    open_handler_type;            // File open handler.
  typedef unspecified                    close_handler_type;           // File close handler.
  typedef unspecified                    time_based_rotation_predicate;  // Predicate that ↵
defines the time-based condition for file rotation.

  // construct/copy/destruct
  text_file_backend();
  template<typename... ArgsT> explicit text_file_backend(ArgsT...const &);
  ~text_file_backend();

  // public member functions
  template<typename PathT> void set_file_name_pattern(PathT const &);
  void set_open_mode(std::ios_base::openmode);
  void set_file_collector(shared_ptr< file::collector > const &);
  void set_open_handler(open_handler_type const &);
  void set_close_handler(close_handler_type const &);
  void set_rotation_size(uintmax_t);
  void set_time_based_rotation(time_based_rotation_predicate const &);
  void auto_flush(bool = true);
  uintmax_t scan_for_files(file::scan_method = file::scan_matching,
                           bool = true);
  void consume(record_view const &, string_type const &);
  void flush();
  void rotate_file();
};
```

Description

The sink backend puts formatted log records to a text file. The sink supports file rotation and advanced file control, such as size and file count restriction.

`text_file_backend` public construct/copy/destruct

1.
```
text_file_backend();
```

Default constructor. The constructed sink backend uses default values of all the parameters.

2.
```
template<typename... ArgsT> explicit text_file_backend(ArgsT...const & args);
```

Constructor. Creates a sink backend with the specified named parameters. The following named parameters are supported:

- `file_name` - Specifies the file name pattern where logs are actually written to. The pattern may contain directory and file name portions, but only the file name may contain placeholders. The backend supports Boost.DateTime placeholders for injecting current time and date into the file name. Also, an additional N placeholder is supported, it will be replaced with an integral increasing file counter. The placeholder may also contain width specification in the printf-compatible form (e.g. %5N). The printed file counter will always be zero-filled. If `file_name` is not specified, pattern "%5N.log" will be used.
- `open_mode` - File open mode. The mode should be presented in form of mask compatible to `std::ios_base::openmode`. If not specified, `trunc | out` will be used.
- `rotation_size` - Specifies the approximate size, in characters written, of the temporary file upon which the file is passed to the file collector. Note the size does not count any possible character conversions that may take place during writing to the file. If not specified, the file won't be rotated upon reaching any size.
- `time_based_rotation` - Specifies the predicate for time-based file rotation. No time-based file rotations will be performed, if not specified.
- `auto_flush` - Specifies a flag, whether or not to automatically flush the file after each written log record. By default, is `false`.

Note

Read the caution note regarding file name pattern in the `sinks::file::collector::scan_for_files` documentation.

3.
```
~text_file_backend();
```

Destructor

`text_file_backend` public member functions

1.
```
template<typename PathT> void set_file_name_pattern(PathT const & pattern);
```

The method sets file name wildcard for the files being written. The wildcard supports date and time injection into the file name.

Parameters: `pattern` The name pattern for the file being written.

2.
```
void set_open_mode(std::ios_base::openmode mode);
```

The method sets the file open mode

Parameters: `mode` File open mode

3.
```
void set_file_collector(shared_ptr< file::collector > const & collector);
```

The method sets the log file collector function. The function is called on file rotation and is being passed the written file name.

Parameters: `collector` The file collector function object

4. ```
   void set_open_handler(open_handler_type const & handler);
   ```

   The method sets file opening handler. The handler will be called every time the backend opens a new temporary file. The handler may write a header to the opened file in order to maintain file validity.

   Parameters: `handler` The file open handler function object

5. ```
   void set_close_handler(close_handler_type const & handler);
   ```

   The method sets file closing handler. The handler will be called every time the backend closes a temporary file. The handler may write a footer to the opened file in order to maintain file validity.

   Parameters: `handler` The file close handler function object

6. ```
   void set_rotation_size(uintmax_t size);
   ```

   The method sets maximum file size. When the size is reached, file rotation is performed.

   Note

   The size does not count any possible character translations that may happen in the underlying API. This may result in greater actual sizes of the written files.

   Parameters: `size` The maximum file size, in characters.

7. ```
   void set_time_based_rotation(time_based_rotation_predicate const & predicate);
   ```

   The method sets the predicate that defines the time-based condition for file rotation.

   Note

   The rotation always occurs on writing a log record, so the rotation is not strictly bound to the specified condition.

   Parameters: `predicate` The predicate that defines the time-based condition for file rotation. If empty, no time-based rotation will take place.

8. ```
   void auto_flush(bool f = true);
   ```

   Sets the flag to automatically flush buffers of all attached streams after each log record

9. ```
   uintmax_t scan_for_files(file::scan_method method = file::scan_matching,
                            bool update_counter = true);
   ```

   Performs scanning of the target directory for log files that may have been left from previous runs of the application. The found files are considered by the file collector as if they were rotated.

   The file scan can be performed in two ways: either all files in the target directory will be considered as log files, or only those files that satisfy the file name pattern. See documentation on `sinks::file::collector::scan_for_files` for more information.

Note

The method essentially delegates to the same-named function of the file collector.

Parameters:
- `method` — File scanning method
- `update_counter` — If `true` and *method* is `scan_matching`, the method attempts to update the internal file counter according to the found files. The counter is unaffected otherwise.

Requires: File collector and the proper file name pattern have already been set.
Returns: The number of files found.

10.
```
void consume(record_view const & rec, string_type const & formatted_message);
```

The method writes the message to the sink

11.
```
void flush();
```

The method flushes the currently open log file

12.
```
void rotate_file();
```

The method rotates the file

Header <boost/log/sinks/text_multifile_backend.hpp>

Andrey Semashev

09.06.2009
The header contains implementation of a text multi-file sink backend.

```
namespace boost {
  namespace log {
    namespace sinks {
      class text_multifile_backend;
      namespace file {
        template<typename FormatterT> class file_name_composer_adapter;
        template<typename FormatterT>
          file_name_composer_adapter< FormatterT >
          as_file_name_composer(FormatterT const &,
                                std::locale const & = std::locale());
      }
    }
  }
}
```

Class template file_name_composer_adapter

boost::log::sinks::file::file_name_composer_adapter

Synopsis

```
// In header: <boost/log/sinks/text_multifile_backend.hpp>

template<typename FormatterT>
class file_name_composer_adapter {
public:
  // types
  typedef filesystem::path                              result_type;      // Functor result type.
  typedef result_type::string_type::value_type          native_char_type;  // File name charac↵
ter type.
  typedef FormatterT                                    formatter_type;    // The adopted format↵
ter type.
  typedef basic_formatting_ostream< native_char_type > stream_type;       // Formatting stream ↵
type.

  // construct/copy/destruct
  explicit file_name_composer_adapter(formatter_type const &,
                                      std::locale const & = std::locale());
  file_name_composer_adapter(file_name_composer_adapter const &);
  file_name_composer_adapter & operator=(file_name_composer_adapter const &);

  // public member functions
  result_type operator()(record_view const &) const;
};
```

Description

An adapter class that allows to use regular formatters as file name generators.

`file_name_composer_adapter` public construct/copy/destruct

1. ```
   explicit file_name_composer_adapter(formatter_type const & formatter,
                                       std::locale const & loc = std::locale());
   ```

   Initializing constructor

2. ```
   file_name_composer_adapter(file_name_composer_adapter const & that);
   ```

   Copy constructor

3. ```
   file_name_composer_adapter &
   operator=(file_name_composer_adapter const & that);
   ```

   Assignment

`file_name_composer_adapter` public member functions

1. ```
   result_type operator()(record_view const & rec) const;
   ```

   The operator generates a file name based on the log record

Function template as_file_name_composer

boost::log::sinks::file::as_file_name_composer

Synopsis

```
// In header: <boost/log/sinks/text_multifile_backend.hpp>


template<typename FormatterT>
  file_name_composer_adapter< FormatterT >
  as_file_name_composer(FormatterT const & fmt,
                        std::locale const & loc = std::locale());
```

Description

The function adopts a log record formatter into a file name generator

Class text_multifile_backend

boost::log::sinks::text_multifile_backend — An implementation of a text multiple files logging sink backend.

Synopsis

```
// In header: <boost/log/sinks/text_multifile_backend.hpp>


class text_multifile_backend : public basic_formatted_sink_backend< char > {
public:
  // types
  typedef base_type::char_type   char_type;               // Character type.
  typedef base_type::string_type string_type;             // String type to be used as a mes↵
sage text holder.
  typedef unspecified            file_name_composer_type;  // File name composer functor type.

  // construct/copy/destruct
  text_multifile_backend();
  ~text_multifile_backend();

  // public member functions
  template<typename ComposerT> void set_file_name_composer(ComposerT const &);
  void consume(record_view const &, string_type const &);
};
```

Description

The sink backend puts formatted log records to one of the text files. The particular file is chosen upon each record's attribute values, which allows to distribute records into individual files or to group records related to some entity or process in a separate file.

`text_multifile_backend` public construct/copy/destruct

1. ```
   text_multifile_backend();
   ```

   Default constructor. The constructed sink backend has no file name composer and thus will not write any files.

2. ```
   ~text_multifile_backend();
   ```

   Destructor

`text_multifile_backend` public member functions

1.
```
template<typename ComposerT>
  void set_file_name_composer(ComposerT const & composer);
```

The method sets file name composer functional object. Log record formatters are accepted, too.

Parameters: `composer` File name composer functor

2.
```
void consume(record_view const & rec, string_type const & formatted_message);
```

The method writes the message to the sink

Header <boost/log/sinks/text_ostream_backend.hpp>

Andrey Semashev

22.04.2007
The header contains implementation of a text output stream sink backend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename CharT> class basic_text_ostream_backend;

     typedef basic_text_ostream_backend< char > text_ostream_backend;  // Convenience typedef ↵
for narrow-character logging.
      typedef basic_text_ostream_backend< wchar_t > wtext_ostream_backend;  // Convenience ty↵
pedef for wide-character logging.
    }
  }
}
```

Class template basic_text_ostream_backend

boost::log::sinks::basic_text_ostream_backend — An implementation of a text output stream logging sink backend.

Synopsis

```
// In header: <boost/log/sinks/text_ostream_backend.hpp>

template<typename CharT>
class basic_text_ostream_backend : public basic_formatted_sink_backend< CharT, combine_require↵
ments< synchronized_feeding, flushing >::type >
{
public:
  // types
  typedef base_type::char_type            char_type;    // Character type.
  typedef base_type::string_type         string_type;  // String type to be used as a message ↵
text holder.
  typedef std::basic_ostream< char_type > stream_type;  // Output stream type.

  // construct/copy/destruct
  basic_text_ostream_backend();
  ~basic_text_ostream_backend();

  // public member functions
  void add_stream(shared_ptr< stream_type > const &);
  void remove_stream(shared_ptr< stream_type > const &);
  void auto_flush(bool = true);
  void consume(record_view const &, string_type const &);
  void flush();
};
```

Description

The sink backend puts formatted log records to one or more text streams.

`basic_text_ostream_backend` public construct/copy/destruct

1. ```
   basic_text_ostream_backend();
   ```

   Constructor. No streams attached to the constructed backend, auto flush feature disabled.

2. ```
   ~basic_text_ostream_backend();
   ```

   Destructor

`basic_text_ostream_backend` public member functions

1. ```
   void add_stream(shared_ptr< stream_type > const & strm);
   ```

   The method adds a new stream to the sink.

   Parameters: `strm` Pointer to the stream. Must not be NULL.

2. ```
   void remove_stream(shared_ptr< stream_type > const & strm);
   ```

   The method removes a stream from the sink. If the stream is not attached to the sink, the method has no effect.

   Parameters: `strm` Pointer to the stream. Must not be NULL.

3. ```
   void auto_flush(bool f = true);
   ```
   ```

Sets the flag to automatically flush buffers of all attached streams after each log record

4. ```
   void consume(record_view const & rec, string_type const & formatted_message);
   ```

   The method writes the message to the sink

5. ```
   void flush();
   ```

   The method flushes the associated streams

Header <boost/log/sinks/unbounded_fifo_queue.hpp>

Andrey Semashev

24.07.2011
The header contains implementation of unbounded FIFO queueing strategy for the asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      class unbounded_fifo_queue;
    }
  }
}
```

Class unbounded_fifo_queue

boost::log::sinks::unbounded_fifo_queue — Unbounded FIFO log record queueing strategy.

Synopsis

```
// In header: <boost/log/sinks/unbounded_fifo_queue.hpp>


class unbounded_fifo_queue {
public:
  // construct/copy/destruct
  unbounded_fifo_queue();
  template<typename ArgsT> explicit unbounded_fifo_queue(ArgsT const &);

  // protected member functions
  void enqueue(record_view const &);
  bool try_enqueue(record_view const &);
  bool try_dequeue_ready(record_view &);
  bool try_dequeue(record_view &);
  bool dequeue_ready(record_view &);
  void interrupt_dequeue();
};
```

Description

The `unbounded_fifo_queue` class is intended to be used with the `asynchronous_sink` frontend as a log record queueing strategy.

This strategy implements the simplest logic of log record buffering between threads: the queue has no limits and imposes no ordering over the queued elements aside from the order in which they are enqueued. Because of this the queue provides decent performance and scalability, however if sink backends can't consume log records fast enough the queue may grow uncontrollably. When this is an issue, it is recommended to use one of the bounded strategies.

`unbounded_fifo_queue` public construct/copy/destruct

1. ```
   unbounded_fifo_queue();
   ```

   Default constructor.

2. ```
   template<typename ArgsT> explicit unbounded_fifo_queue(ArgsT const &);
   ```

   Initializing constructor.

`unbounded_fifo_queue` protected member functions

1. ```
   void enqueue(record_view const & rec);
   ```

   Enqueues log record to the queue.

2. ```
   bool try_enqueue(record_view const & rec);
   ```

   Attempts to enqueue log record to the queue.

3. ```
   bool try_dequeue_ready(record_view & rec);
   ```

   Attempts to dequeue a log record ready for processing from the queue, does not block if the queue is empty.

4. ```
   bool try_dequeue(record_view & rec);
   ```

   Attempts to dequeue log record from the queue, does not block if the queue is empty.

5. ```
   bool dequeue_ready(record_view & rec);
   ```

   Dequeues log record from the queue, blocks if the queue is empty.

6. ```
   void interrupt_dequeue();
   ```

   Wakes a thread possibly blocked in the `dequeue` method.

Header <boost/log/sinks/unbounded_ordering_queue.hpp>

Andrey Semashev

24.07.2011
The header contains implementation of unbounded ordering record queueing strategy for the asynchronous sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename OrderT> class unbounded_ordering_queue;
    }
  }
}
```

Class template unbounded_ordering_queue

boost::log::sinks::unbounded_ordering_queue — Unbounded ordering log record queueing strategy.

Synopsis

```
// In header: <boost/log/sinks/unbounded_ordering_queue.hpp>

template<typename OrderT>
class unbounded_ordering_queue {
public:
  // construct/copy/destruct
  template<typename ArgsT> explicit unbounded_ordering_queue(ArgsT const &);

  // public member functions
  posix_time::time_duration get_ordering_window() const;

  // public static functions
  static posix_time::time_duration get_default_ordering_window();

  // protected member functions
  void enqueue(record_view const &);
  bool try_enqueue(record_view const &);
  bool try_dequeue_ready(record_view &);
  bool try_dequeue(record_view &);
  bool dequeue_ready(record_view &);
  void interrupt_dequeue();

  // private member functions
  void enqueue_unlocked(record_view const &);
};
```

Description

The `unbounded_ordering_queue` class is intended to be used with the `asynchronous_sink` frontend as a log record queueing strategy.

This strategy provides the following properties to the record queueing mechanism:

- The queue has no size limits.
- The queue has a fixed latency window. This means that each log record put into the queue will normally not be dequeued for a certain period of time.
- The queue performs stable record ordering within the latency window. The ordering predicate can be specified in the `OrderT` template parameter.

Since this queue has no size limits, it may grow uncontrollably if sink backends dequeue log records not fast enough. When this is an issue, it is recommended to use one of the bounded strategies.

`unbounded_ordering_queue` public construct/copy/destruct

1. ```
   template<typename ArgsT> explicit unbounded_ordering_queue(ArgsT const & args);
   ```

   Initializing constructor.

`unbounded_ordering_queue` public member functions

1. ```
   posix_time::time_duration get_ordering_window() const;
   ```

Returns ordering window size specified during initialization

`unbounded_ordering_queue` public static functions

1.
```
static posix_time::time_duration get_default_ordering_window();
```

Returns default ordering window size. The default window size is specific to the operating system thread scheduling mechanism.

`unbounded_ordering_queue` protected member functions

1.
```
void enqueue(record_view const & rec);
```

Enqueues log record to the queue.

2.
```
bool try_enqueue(record_view const & rec);
```

Attempts to enqueue log record to the queue.

3.
```
bool try_dequeue_ready(record_view & rec);
```

Attempts to dequeue a log record ready for processing from the queue, does not block if no log records are ready to be processed.

4.
```
bool try_dequeue(record_view & rec);
```

Attempts to dequeue log record from the queue, does not block.

5.
```
bool dequeue_ready(record_view & rec);
```

Dequeues log record from the queue, blocks if no log records are ready to be processed.

6.
```
void interrupt_dequeue();
```

Wakes a thread possibly blocked in the `dequeue` method.

`unbounded_ordering_queue` private member functions

1.
```
void enqueue_unlocked(record_view const & rec);
```

Enqueues a log record.

Header <boost/log/sinks/unlocked_frontend.hpp>

Andrey Semashev

14.07.2009
The header contains declaration of an unlocked sink frontend.

```
namespace boost {
  namespace log {
    namespace sinks {
      template<typename SinkBackendT> class unlocked_sink;
    }
  }
}
```

Class template unlocked_sink

boost::log::sinks::unlocked_sink — Non-blocking logging sink frontend.

Synopsis

```
// In header: <boost/log/sinks/unlocked_frontend.hpp>

template<typename SinkBackendT>
class unlocked_sink : public basic_sink_frontend {
public:
  // types
  typedef SinkBackendT                    sink_backend_type;   // Sink implementation type.
  typedef shared_ptr< sink_backend_type > locked_backend_ptr;  // Type of pointer to the backend.

  // construct/copy/destruct
  unlocked_sink();
  explicit unlocked_sink(shared_ptr< sink_backend_type > const &);

  // public member functions
  locked_backend_ptr locked_backend();
  virtual void consume(record_view const &);
  virtual void flush();
};
```

Description

The sink frontend does not perform thread synchronization and simply passes logging records to the sink backend.

`unlocked_sink` public construct/copy/destruct

1. ```
   unlocked_sink();
   ```

   Default constructor. Constructs the sink backend instance. Requires the backend to be default-constructible.

2. ```
   explicit unlocked_sink(shared_ptr< sink_backend_type > const & backend);
   ```

   Constructor attaches user-constructed backend instance

   Parameters: `backend` Pointer to the backend instance
   Requires: *backend* is not `NULL`.

`unlocked_sink` public member functions

1. ```
   locked_backend_ptr locked_backend();
   ```

   Locking accessor to the attached backend.

   Note

   Does not do any actual locking, provided only for interface consistency with other frontends.

2. ```
   virtual void consume(record_view const & rec);
   ```

Passes the log record to the backend

3.
```
virtual void flush();
```

The method performs flushing of any internal buffers that may hold log records. The method may take considerable time to complete and may block both the calling thread and threads attempting to put new records into the sink while this call is in progress.

Utilities

Header <boost/log/utility/empty_deleter.hpp>

Andrey Semashev

22.04.2007
This header is deprecated, use boost/utility/empty_deleter.hpp instead. The header is left for backward compatibility and will be removed in future versions.

```
namespace boost {
  namespace log {
    typedef boost::null_deleter empty_deleter;
  }
}
```

Header <boost/log/utility/exception_handler.hpp>

Andrey Semashev

12.07.2009
This header contains tools for exception handlers support in different parts of the library.

```
BOOST_LOG_MAX_EXCEPTION_TYPES
```

```
namespace boost {
  namespace log {
    template<typename SequenceT, typename HandlerT> class exception_handler;
    template<typename SequenceT, typename HandlerT>
      class nothrow_exception_handler;
    nop make_exception_suppressor();
    template<typename HandlerT>
      exception_handler< typename HandlerT::exception_types, HandlerT >
      make_exception_handler(HandlerT const &);
    template<typename HandlerT>
      nothrow_exception_handler< typename HandlerT::exception_types, HandlerT >
      make_exception_handler(HandlerT const &, std::nothrow_t const &);
    template<typename... ExceptionsT, typename HandlerT>
      exception_handler< MPL_sequence_of_ExceptionsT, HandlerT >
      make_exception_handler(HandlerT const &);
    template<typename... ExceptionsT, typename HandlerT>
      nothrow_exception_handler< MPL_sequence_of_ExceptionsT, HandlerT >
      make_exception_handler(HandlerT const &, std::nothrow_t const &);
  }
}
```

Class template exception_handler

boost::log::exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>

template<typename SequenceT, typename HandlerT>
class exception_handler {
public:
  // types
  typedef HandlerT handler_type;  // The exception handler type.
  typedef void     result_type;   // The handler result type.

  // construct/copy/destruct
  explicit exception_handler(handler_type const &);

  // public member functions
  void operator()() const;
};
```

Description

An exception handler functional object. The handler aggregates a user-defined functional object that will be called when one of the specified exception types is caught.

`exception_handler` public construct/copy/destruct

1. ```
   explicit exception_handler(handler_type const & handler);
   ```

   Initializing constructor. Creates an exception handler with the specified function object that will receive the exception.

`exception_handler` public member functions

1. ```
   void operator()() const;
   ```

   Exception launcher. Rethrows the current exception in order to detect its type and pass it to the aggregated function object.

   Note

   Must be called from within a `catch` statement.

Class template nothrow_exception_handler

boost::log::nothrow_exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>

template<typename SequenceT, typename HandlerT>
class nothrow_exception_handler :
  public boost::log::exception_handler< SequenceT, HandlerT >
{
public:
  // types
  typedef HandlerT handler_type;  // The exception handler type.
  typedef void     result_type;   // The handler result type.

  // construct/copy/destruct
  explicit nothrow_exception_handler(handler_type const &);

  // public member functions
  void operator()() const;
};
```

Description

A no-throw exception handler functional object. Acts similar to `exception_handler`, but in case if the exception cannot be handled the exception is not propagated from the handler. Instead the user-defined functional object is called with no parameters.

`nothrow_exception_handler` public construct/copy/destruct

1. ```
   explicit nothrow_exception_handler(handler_type const & handler);
   ```

   Initializing constructor. Creates an exception handler with the specified function object that will receive the exception.

`nothrow_exception_handler` public member functions

1. ```
   void operator()() const;
   ```

   Exception launcher. Rethrows the current exception in order to detect its type and pass it to the aggregated function object. If the type of the exception could not be detected, the user-defined handler is called with no arguments.

   Note

   Must be called from within a `catch` statement.

Function make_exception_suppressor

boost::log::make_exception_suppressor

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>


nop make_exception_suppressor();
```

Description

The function creates an empty exception handler that effectively suppresses any exception

Function template make_exception_handler

boost::log::make_exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>


template<typename HandlerT>
  exception_handler< typename HandlerT::exception_types, HandlerT >
  make_exception_handler(HandlerT const & handler);
```

Description

The function creates an exception handler functional object. The handler will call to the user-specified functional object with an exception as its argument.

Note

This form requires the user-defined functional object to have an `exception_types` nested type. This type should be an MPL sequence of all expected exception types.

Parameters: `handler` User-defined functional object that will receive exceptions.
Returns: A nullary functional object that should be called from within a `catch` statement.

Function template make_exception_handler

boost::log::make_exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>


template<typename HandlerT>
  nothrow_exception_handler< typename HandlerT::exception_types, HandlerT >
  make_exception_handler(HandlerT const & handler, std::nothrow_t const &);
```

Description

The function creates an exception handler functional object. The handler will call to the user-specified functional object with an exception as its argument. If the exception type cannot be identified, the handler will call the user-defined functor with no arguments, instead of propagating exception to the caller.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Note

This form requires the user-defined functional object to have an `exception_types` nested type. This type should be an MPL sequence of all expected exception types.

Parameters: `handler` User-defined functional object that will receive exceptions.
Returns: A nullary functional object that should be called from within a `catch` statement.

Function template make_exception_handler

boost::log::make_exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>


template<typename... ExceptionsT, typename HandlerT>
  exception_handler< MPL_sequence_of_ExceptionsT, HandlerT >
  make_exception_handler(HandlerT const & handler);
```

Description

The function creates an exception handler functional object. The handler will call to the user-specified functional object with an exception as its argument. All expected exception types should be specified as first template parameters explicitly, in the order they would be specified in a corresponding `try/catch` statement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters: `handler` User-defined functional object that will receive exceptions.
Returns: A nullary functional object that should be called from within a `catch` statement.

Function template make_exception_handler

boost::log::make_exception_handler

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>


template<typename... ExceptionsT, typename HandlerT>
  nothrow_exception_handler< MPL_sequence_of_ExceptionsT, HandlerT >
  make_exception_handler(HandlerT const & handler, std::nothrow_t const &);
```

Description

The function creates an exception handler functional object. The handler will call to the user-specified functional object with an exception as its argument. If the exception type cannot be identified, the handler will call the user-defined functor with no arguments, instead of propagating exception to the caller. All expected exception types should be specified as first template parameters explicitly, in the order they would be specified in a corresponding `try/catch` statement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters: `handler` User-defined functional object that will receive exceptions.
Returns: A nullary functional object that should be called from within a `catch` statement.

Macro BOOST_LOG_MAX_EXCEPTION_TYPES

BOOST_LOG_MAX_EXCEPTION_TYPES — Maximum number of exception types that can be specified for exception handlers.

Synopsis

```
// In header: <boost/log/utility/exception_handler.hpp>

BOOST_LOG_MAX_EXCEPTION_TYPES
```

Header <boost/log/utility/explicit_operator_bool.hpp>

Andrey Semashev

08.03.2009
This header is deprecated, use boost/utility/explicit_operator_bool.hpp instead. The header is left for backward compatibility and will be removed in future versions.

```


BOOST_LOG_EXPLICIT_OPERATOR_BOOL()
```

Macro BOOST_LOG_EXPLICIT_OPERATOR_BOOL

BOOST_LOG_EXPLICIT_OPERATOR_BOOL — The macro defines an explicit operator of conversion to `bool`.

Synopsis

```
// In header: <boost/log/utility/explicit_operator_bool.hpp>

BOOST_LOG_EXPLICIT_OPERATOR_BOOL()
```

Description

The macro should be used inside the definition of a class that has to support the conversion. The class should also implement `operator!`, in terms of which the conversion operator will be implemented.

Header <boost/log/utility/formatting_ostream.hpp>

Andrey Semashev

11.07.2012
The header contains implementation of a string stream used for log record formatting.

```
namespace boost {
  namespace log {
    template<typename CharT, typename TraitsT, typename AllocatorT>
      class basic_formatting_ostream;
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &,
                 T const &);
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &,
                 T &);
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &&,
                 T const &);
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &&,
                 T &);
  }
}
```

Class template basic_formatting_ostream

boost::log::basic_formatting_ostream — Stream wrapper for log records formatting.

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>

template<typename CharT, typename TraitsT, typename AllocatorT>
class basic_formatting_ostream {
public:
  // types
  typedef CharT                                          char_type;       // Character type.
  typedef TraitsT                                        traits_type;     // Character traits.
  typedef AllocatorT                                     allocator_type;  // Memory allocator.
  typedef unspecified                                    streambuf_type;  // Stream buffer type.
  typedef streambuf_type::string_type                    string_type;     // Target string type.
  typedef std::basic_ostream< char_type, traits_type > ostream_type;    // Stream type.
  typedef ostream_type::pos_type                         pos_type;       // Stream position type.
  typedef ostream_type::off_type                          off_type;       // Stream offset type.
  typedef ostream_type::int_type                         int_type;       // Integer type for char↵
acters.
  typedef ostream_type::failure                          failure;
  typedef ostream_type::fmtflags                         fmtflags;
  typedef ostream_type::iostate                          iostate;
  typedef ostream_type::openmode                         openmode;
  typedef ostream_type::seekdir                          seekdir;
  typedef ostream_type::Init                             Init;
  typedef ostream_type::event                            event;
  typedef ostream_type::event_callback                   event_callback;

  // member classes/structs/unions

  class sentry : public sentry {
  public:
    // construct/copy/destruct
    explicit sentry(basic_formatting_ostream &);
    sentry(sentry const &) = delete;
    sentry & operator=(sentry const &) = delete;
  };

  // construct/copy/destruct
  basic_formatting_ostream();
  explicit basic_formatting_ostream(string_type &);
  basic_formatting_ostream(basic_formatting_ostream const &) = delete;
  basic_formatting_ostream &
  operator=(basic_formatting_ostream const &) = delete;
  ~basic_formatting_ostream();

  // public member functions
  void attach(string_type &);
  void detach();
  string_type const & str() const;
  ostream_type & stream();
  ostream_type const & stream() const;
  fmtflags flags() const;
  fmtflags flags(fmtflags);
  fmtflags setf(fmtflags);
  fmtflags setf(fmtflags, fmtflags);
  void unsetf(fmtflags);
  std::streamsize precision() const;
  std::streamsize precision(std::streamsize);
  std::streamsize width() const;
  std::streamsize width(std::streamsize);
  std::locale getloc() const;
  std::locale imbue(std::locale const &);
```

```
long & iword(int);
void *& pword(int);
void register_callback(event_callback, int);
explicit operator bool() const;
bool operator!() const;
iostate rdstate() const;
void clear(iostate = goodbit);
void setstate(iostate);
bool good() const;
bool eof() const;
bool fail() const;
bool bad() const;
iostate exceptions() const;
void exceptions(iostate);
ostream_type * tie() const;
ostream_type * tie(ostream_type *);
streambuf_type * rdbuf() const;
basic_formatting_ostream &
copyfmt(std::basic_ios< char_type, traits_type > &);
basic_formatting_ostream & copyfmt(basic_formatting_ostream &);
char_type fill() const;
char_type fill(char_type);
char narrow(char_type, char) const;
char_type widen(char) const;
basic_formatting_ostream & flush();
pos_type tellp();
basic_formatting_ostream & seekp(pos_type);
basic_formatting_ostream & seekp(off_type, std::ios_base::seekdir);
basic_formatting_ostream & put(char_type);
template<typename OtherCharT> unspecified put(OtherCharT);
basic_formatting_ostream & write(const char_type *, std::streamsize);
template<typename OtherCharT>
  unspecified write(const OtherCharT *, std::streamsize);
basic_formatting_ostream & operator<<(ios_base_manip);
basic_formatting_ostream & operator<<(basic_ios_manip);
basic_formatting_ostream & operator<<(stream_manip);
basic_formatting_ostream & operator<<(char);
basic_formatting_ostream & operator<<(const char *);
basic_formatting_ostream & operator<<(wchar_t);
basic_formatting_ostream & operator<<(const wchar_t *);
basic_formatting_ostream & operator<<(char16_t);
basic_formatting_ostream & operator<<(const char16_t *);
basic_formatting_ostream & operator<<(char32_t);
basic_formatting_ostream & operator<<(const char32_t *);
basic_formatting_ostream & operator<<(bool);
basic_formatting_ostream & operator<<(signed char);
basic_formatting_ostream & operator<<(unsigned char);
basic_formatting_ostream & operator<<(short);
basic_formatting_ostream & operator<<(unsigned short);
basic_formatting_ostream & operator<<(int);
basic_formatting_ostream & operator<<(unsigned int);
basic_formatting_ostream & operator<<(long);
basic_formatting_ostream & operator<<(unsigned long);
basic_formatting_ostream & operator<<(long long);
basic_formatting_ostream & operator<<(unsigned long long);
basic_formatting_ostream & operator<<(float);
basic_formatting_ostream & operator<<(double);
basic_formatting_ostream & operator<<(long double);
basic_formatting_ostream & operator<<(const void *);
basic_formatting_ostream &
operator<<(std::basic_streambuf< char_type, traits_type > *);

// public static functions
```

```
  static int xalloc();
  static bool sync_with_stdio(bool = true);

  // private member functions
  void init_stream();
  basic_formatting_ostream &
  formatted_write(const char_type *, std::streamsize);
  template<typename OtherCharT>
    basic_formatting_ostream &
    formatted_write(const OtherCharT *, std::streamsize);
  void aligned_write(const char_type *, std::streamsize);
  template<typename OtherCharT>
    void aligned_write(const OtherCharT *, std::streamsize);

  // public data members
  static constexpr fmtflags boolalpha;
  static constexpr fmtflags dec;
  static constexpr fmtflags fixed;
  static constexpr fmtflags hex;
  static constexpr fmtflags internal;
  static constexpr fmtflags left;
  static constexpr fmtflags oct;
  static constexpr fmtflags right;
  static constexpr fmtflags scientific;
  static constexpr fmtflags showbase;
  static constexpr fmtflags showpoint;
  static constexpr fmtflags skipws;
  static constexpr fmtflags unitbuf;
  static constexpr fmtflags uppercase;
  static constexpr fmtflags adjustfield;
  static constexpr fmtflags basefield;
  static constexpr fmtflags floatfield;
  static constexpr iostate badbit;
  static constexpr iostate eofbit;
  static constexpr iostate failbit;
  static constexpr iostate goodbit;
  static constexpr openmode app;
  static constexpr openmode ate;
  static constexpr openmode binary;
  static constexpr openmode in;
  static constexpr openmode out;
  static constexpr openmode trunc;
  static constexpr seekdir beg;
  static constexpr seekdir cur;
  static constexpr seekdir end;
  static constexpr event erase_event;
  static constexpr event imbue_event;
  static constexpr event copyfmt_event;
};
```

Description

Stream for log records formatting.

This stream wrapper is used by the library for log record formatting. It implements the standard string stream interface with a few differences:

- It does not derive from standard types `std::basic_ostream`, `std::basic_ios` and `std::ios_base`, although it tries to implement their interfaces closely. There are a few small differences, mostly regarding `rdbuf` and `str` signatures, as well as the supported insertion operator overloads. The actual wrapped stream can be accessed through the `stream` methods.
- By default, `bool` values are formatted using alphabetical representation rather than numeric.

- The stream supports writing strings of character types different from the stream character type. The stream will perform character code conversion as needed using the imbued locale.
- The stream operates on an external string object rather than on the embedded one. The string can be attached or detached from the stream dynamically.

Although `basic_formatting_ostream` does not derive from `std::basic_ostream`, users are not required to add special overloads of `operator<<` for it since the stream will by default reuse the operators for `std::basic_ostream`. However, one can define special overloads of `operator<<` for `basic_formatting_ostream` if a certain type needs special formatting when output to log.

`basic_formatting_ostream` public construct/copy/destruct

1.
```
basic_formatting_ostream();
```

Default constructor. Creates an empty record that is equivalent to the invalid record handle. The stream capability is not available after construction.

Postconditions: `!*this == true`

2.
```
explicit basic_formatting_ostream(string_type & str);
```

Initializing constructor. Attaches the string to the constructed stream. The string will be used to store the formatted characters.

Parameters: `str` The string buffer to attach.
Postconditions: `!*this == false`

3.
```
basic_formatting_ostream(basic_formatting_ostream const & that) = delete;
```

Copy constructor (closed)

4.
```
basic_formatting_ostream &
operator=(basic_formatting_ostream const & that) = delete;
```

Assignment (closed)

5.
```
~basic_formatting_ostream();
```

Destructor. Destroys the record, releases any sinks and attribute values that were involved in processing this record.

`basic_formatting_ostream` public member functions

1.
```
void attach(string_type & str);
```

Attaches the stream to the string. The string will be used to store the formatted characters.

Parameters: `str` The string buffer to attach.

2.
```
void detach();
```

Detaches the stream from the string. Any buffered data is flushed to the string.

3.
```
string_type const & str() const;
```

Returns: Reference to the attached string. The string must be attached before calling this method.

4.
```
ostream_type & stream();
```

Returns: Reference to the wrapped stream

5.
```
ostream_type const & stream() const;
```

Returns: Reference to the wrapped stream

6.
```
fmtflags flags() const;
```

7.
```
fmtflags flags(fmtflags f);
```

8.
```
fmtflags setf(fmtflags f);
```

9.
```
fmtflags setf(fmtflags f, fmtflags mask);
```

10.
```
void unsetf(fmtflags f);
```

11.
```
std::streamsize precision() const;
```

12.
```
std::streamsize precision(std::streamsize p);
```

13.
```
std::streamsize width() const;
```

14.
```
std::streamsize width(std::streamsize w);
```

15.
```
std::locale getloc() const;
```

16.
```
std::locale imbue(std::locale const & loc);
```

17.
```
long & iword(int index);
```

18.
```
void *& pword(int index);
```

```
19. void register_callback(event_callback fn, int index);

20. explicit operator bool() const;

21. bool operator!() const;

22. iostate rdstate() const;

23. void clear(iostate state = goodbit);

24. void setstate(iostate state);

25. bool good() const;

26. bool eof() const;

27. bool fail() const;

28. bool bad() const;

29. iostate exceptions() const;

30. void exceptions(iostate s);

31. ostream_type * tie() const;

32. ostream_type * tie(ostream_type * strm);

33. streambuf_type * rdbuf() const;

34. basic_formatting_ostream &
    copyfmt(std::basic_ios< char_type, traits_type > & rhs);
```

```
basic_formatting_ostream & copyfmt(basic_formatting_ostream & rhs);
```

```
char_type fill() const;
```

```
char_type fill(char_type ch);
```

```
char narrow(char_type ch, char def) const;
```

```
char_type widen(char ch) const;
```

```
basic_formatting_ostream & flush();
```

```
pos_type tellp();
```

```
basic_formatting_ostream & seekp(pos_type pos);
```

```
basic_formatting_ostream & seekp(off_type off, std::ios_base::seekdir dir);
```

```
basic_formatting_ostream & put(char_type c);
```

```
template<typename OtherCharT> unspecified put(OtherCharT c);
```

```
basic_formatting_ostream & write(const char_type * p, std::streamsize size);
```

```
template<typename OtherCharT>
  unspecified write(const OtherCharT * p, std::streamsize size);
```

```
basic_formatting_ostream & operator<<(ios_base_manip manip);
```

```
basic_formatting_ostream & operator<<(basic_ios_manip manip);
```

```
basic_formatting_ostream & operator<<(stream_manip manip);
```

```
basic_formatting_ostream & operator<<(char c);
basic_formatting_ostream & operator<<(const char * p);
basic_formatting_ostream & operator<<(wchar_t c);
basic_formatting_ostream & operator<<(const wchar_t * p);
basic_formatting_ostream & operator<<(char16_t c);
basic_formatting_ostream & operator<<(const char16_t * p);
basic_formatting_ostream & operator<<(char32_t c);
basic_formatting_ostream & operator<<(const char32_t * p);
basic_formatting_ostream & operator<<(bool value);
basic_formatting_ostream & operator<<(signed char value);
basic_formatting_ostream & operator<<(unsigned char value);
basic_formatting_ostream & operator<<(short value);
basic_formatting_ostream & operator<<(unsigned short value);
basic_formatting_ostream & operator<<(int value);
basic_formatting_ostream & operator<<(unsigned int value);
basic_formatting_ostream & operator<<(long value);
```

67.
```
basic_formatting_ostream & operator<<(unsigned long value);
```

68.
```
basic_formatting_ostream & operator<<(long long value);
```

69.
```
basic_formatting_ostream & operator<<(unsigned long long value);
```

70.
```
basic_formatting_ostream & operator<<(float value);
```

71.
```
basic_formatting_ostream & operator<<(double value);
```

72.
```
basic_formatting_ostream & operator<<(long double value);
```

73.
```
basic_formatting_ostream & operator<<(const void * value);
```

74.
```
basic_formatting_ostream &
operator<<(std::basic_streambuf< char_type, traits_type > * buf);
```

`basic_formatting_ostream` public static functions

1.
```
static int xalloc();
```

2.
```
static bool sync_with_stdio(bool sync = true);
```

`basic_formatting_ostream` private member functions

1.
```
void init_stream();
```

2.
```
basic_formatting_ostream &
formatted_write(const char_type * p, std::streamsize size);
```

3.
```
template<typename OtherCharT>
  basic_formatting_ostream &
  formatted_write(const OtherCharT * p, std::streamsize size);
```

4.
```
void aligned_write(const char_type * p, std::streamsize size);
```

5.
```
template<typename OtherCharT>
  void aligned_write(const OtherCharT * p, std::streamsize size);
```

Class sentry

boost::log::basic_formatting_ostream::sentry

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>


class sentry : public sentry {
public:
  // construct/copy/destruct
  explicit sentry(basic_formatting_ostream &);
  sentry(sentry const &) = delete;
  sentry & operator=(sentry const &) = delete;
};
```

Description

`sentry` public construct/copy/destruct

1.
```
explicit sentry(basic_formatting_ostream & strm);
```

2.
```
sentry(sentry const &) = delete;
```

3.
```
sentry & operator=(sentry const &) = delete;
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > & strm,
             T const & value);
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > & strm,
             T & value);
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > && strm,
             T const & value);
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/formatting_ostream.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > && strm,
             T & value);
```

Header <boost/log/utility/formatting_ostream_fwd.hpp>

Andrey Semashev

11.07.2012
The header contains forward declaration of a string stream used for log record formatting.

```
namespace boost {
  namespace log {
    typedef basic_formatting_ostream< char > formatting_ostream;
    typedef basic_formatting_ostream< wchar_t > wformatting_ostream;
  }
}
```

Header <boost/log/utility/functional.hpp>

Andrey Semashev

30.03.2008
This header includes all functional helpers.

Header <boost/log/utility/functional/as_action.hpp>

Andrey Semashev

30.03.2008
This header contains function object adapter for compatibility with Boost.Spirit actions interface requirements.

```
namespace boost {
  namespace log {
    template<typename FunT> struct as_action_adapter;
    template<typename FunT> as_action_adapter< FunT > as_action(FunT const &);
  }
}
```

Struct template as_action_adapter

boost::log::as_action_adapter — Function object adapter for Boost.Spirit actions.

Synopsis

```
// In header: <boost/log/utility/functional/as_action.hpp>

template<typename FunT>
struct as_action_adapter {
  // types
  typedef FunT::result_type result_type;

  // construct/copy/destruct
  as_action_adapter() = default;
  explicit as_action_adapter(FunT const &);

  // public member functions
  template<typename AttributeT, typename ContextT>
    result_type operator()(AttributeT const &, ContextT const &, bool &) const;
};
```

Description

`as_action_adapter` public construct/copy/destruct

1. ```
   as_action_adapter() = default;
   ```

2. ```
   explicit as_action_adapter(FunT const & fun);
   ```

`as_action_adapter` public member functions

1.
```
template<typename AttributeT, typename ContextT>
  result_type operator()(AttributeT const & attr, ContextT const & ctx,
                         bool & pass) const;
```

Function template as_action

boost::log::as_action

Synopsis

```
// In header: <boost/log/utility/functional/as_action.hpp>


template<typename FunT> as_action_adapter< FunT > as_action(FunT const & fun);
```

Header <boost/log/utility/functional/begins_with.hpp>

Andrey Semashev

30.03.2008
This header contains a predicate for checking if the provided string begins with a substring.

```
namespace boost {
  namespace log {
    struct begins_with_fun;
  }
}
```

Struct begins_with_fun

boost::log::begins_with_fun — The `begins_with` functor.

Synopsis

```
// In header: <boost/log/utility/functional/begins_with.hpp>


struct begins_with_fun {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;
};
```

Description

`begins_with_fun` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

Header <boost/log/utility/functional/bind.hpp>

Andrey Semashev

30.03.2008
This header contains function object adapters. This is a lightweight alternative to what Boost.Phoenix and Boost.Bind provides.

```
namespace boost {
  namespace log {
    template<typename FunT, typename FirstArgT> struct binder1st;

    template<typename FunT, typename FirstArgT>
      struct binder1st<FunT &, FirstArgT>;

    template<typename FunT, typename SecondArgT> struct binder2nd;

    template<typename FunT, typename SecondArgT>
      struct binder2nd<FunT &, SecondArgT>;

    template<typename FunT, typename ThirdArgT> struct binder3rd;

    template<typename FunT, typename ThirdArgT>
      struct binder3rd<FunT &, ThirdArgT>;
    template<typename FunT, typename FirstArgT>
      binder1st< FunT, FirstArgT > bind1st(FunT, FirstArgT const &);
    template<typename FunT, typename FirstArgT>
      binder1st< FunT, FirstArgT > bind1st(FunT, FirstArgT &);
    template<typename FunT, typename SecondArgT>
      binder2nd< FunT, SecondArgT > bind2nd(FunT, SecondArgT const &);
    template<typename FunT, typename SecondArgT>
      binder2nd< FunT, SecondArgT > bind2nd(FunT, SecondArgT &);
    template<typename FunT, typename ThirdArgT>
      binder3rd< FunT, ThirdArgT > bind3rd(FunT, ThirdArgT const &);
    template<typename FunT, typename ThirdArgT>
      binder3rd< FunT, ThirdArgT > bind3rd(FunT, ThirdArgT &);
  }
}
```

Struct template binder1st

boost::log::binder1st — First argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename FirstArgT>
struct binder1st : private FunT {
  // types
  typedef FunT::result_type result_type;

  // construct/copy/destruct
  binder1st(FunT const &, unspecified);

  // public member functions
  result_type operator()() const;
  template<typename T0> result_type operator()(T0 const &) const;
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```

Description

`binder1st` public construct/copy/destruct

1.
```
binder1st(FunT const & fun, unspecified arg);
```

`binder1st` public member functions

1.
```
result_type operator()() const;
```

2.
```
template<typename T0> result_type operator()(T0 const & arg0) const;
```

3.
```
template<typename T0, typename T1>
  result_type operator()(T0 const & arg0, T1 const & arg1) const;
```

Struct template binder1st<FunT &, FirstArgT>

boost::log::binder1st<FunT &, FirstArgT> — First argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename FirstArgT>
struct binder1st<FunT &, FirstArgT> {
  // types
  typedef remove_cv< FunT >::type::result_type result_type;

  // construct/copy/destruct
  binder1st(FunT &, unspecified);

  // public member functions
  result_type operator()() const;
  template<typename T0> result_type operator()(T0 const &) const;
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```

Description

`binder1st` public construct/copy/destruct

1.
```
binder1st(FunT & fun, unspecified arg);
```

`binder1st` public member functions

1.
```
result_type operator()() const;
```

2.
```
template<typename T0> result_type operator()(T0 const & arg0) const;
```

3.
```
template<typename T0, typename T1>
  result_type operator()(T0 const & arg0, T1 const & arg1) const;
```

Struct template binder2nd

boost::log::binder2nd — Second argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename SecondArgT>
struct binder2nd : private FunT {
  // types
  typedef FunT::result_type result_type;

  // construct/copy/destruct
  binder2nd(FunT const &, unspecified);

  // public member functions
  template<typename T> result_type operator()(T const &) const;
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```

Description

`binder2nd` public construct/copy/destruct

1.
```
binder2nd(FunT const & fun, unspecified arg);
```

`binder2nd` public member functions

1.
```
template<typename T> result_type operator()(T const & arg) const;
```

2.
```
template<typename T0, typename T1>
  result_type operator()(T0 const & arg0, T1 const & arg1) const;
```

Struct template binder2nd<FunT &, SecondArgT>

boost::log::binder2nd<FunT &, SecondArgT> — Second argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename SecondArgT>
struct binder2nd<FunT &, SecondArgT> {
  // types
  typedef remove_cv< FunT >::type::result_type result_type;

  // construct/copy/destruct
  binder2nd(FunT &, unspecified);

  // public member functions
  template<typename T> result_type operator()(T const &) const;
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```

Description

`binder2nd` public construct/copy/destruct

1. ```
   binder2nd(FunT & fun, unspecified arg);
   ```

`binder2nd` public member functions

1. ```
   template<typename T> result_type operator()(T const & arg) const;
   ```

2. ```
   template<typename T0, typename T1>
     result_type operator()(T0 const & arg0, T1 const & arg1) const;
   ```

Struct template binder3rd

boost::log::binder3rd — Third argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename ThirdArgT>
struct binder3rd : private FunT {
  // types
  typedef FunT::result_type result_type;

  // construct/copy/destruct
  binder3rd(FunT const &, unspecified);

  // public member functions
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```
```

Description

`binder3rd` public construct/copy/destruct

1. ```
   binder3rd(FunT const & fun, unspecified arg);
   ```

`binder3rd` public member functions

1. ```
   template<typename T0, typename T1>
     result_type operator()(T0 const & arg0, T1 const & arg1) const;
   ```

Struct template binder3rd<FunT &, ThirdArgT>

boost::log::binder3rd<FunT &, ThirdArgT> — Third argument binder.

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>

template<typename FunT, typename ThirdArgT>
struct binder3rd<FunT &, ThirdArgT> {
  // types
  typedef remove_cv< FunT >::type::result_type result_type;

  // construct/copy/destruct
  binder3rd(FunT &, unspecified);

  // public member functions
  template<typename T0, typename T1>
    result_type operator()(T0 const &, T1 const &) const;
};
```

Description

`binder3rd` public construct/copy/destruct

1. ```
   binder3rd(FunT & fun, unspecified arg);
   ```

`binder3rd` public member functions

1. ```
   template<typename T0, typename T1>
     result_type operator()(T0 const & arg0, T1 const & arg1) const;
   ```

Function template bind1st

boost::log::bind1st

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename FirstArgT>
  binder1st< FunT, FirstArgT > bind1st(FunT fun, FirstArgT const & arg);
```

Function template bind1st

boost::log::bind1st

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename FirstArgT>
  binder1st< FunT, FirstArgT > bind1st(FunT fun, FirstArgT & arg);
```

Function template bind2nd

boost::log::bind2nd

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename SecondArgT>
  binder2nd< FunT, SecondArgT > bind2nd(FunT fun, SecondArgT const & arg);
```

Function template bind2nd

boost::log::bind2nd

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename SecondArgT>
  binder2nd< FunT, SecondArgT > bind2nd(FunT fun, SecondArgT & arg);
```

Function template bind3rd

boost::log::bind3rd

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename ThirdArgT>
  binder3rd< FunT, ThirdArgT > bind3rd(FunT fun, ThirdArgT const & arg);
```

Function template bind3rd

boost::log::bind3rd

Synopsis

```
// In header: <boost/log/utility/functional/bind.hpp>


template<typename FunT, typename ThirdArgT>
  binder3rd< FunT, ThirdArgT > bind3rd(FunT fun, ThirdArgT & arg);
```

Header <boost/log/utility/functional/bind_assign.hpp>

Andrey Semashev

30.03.2008
This header contains a function object that assigns the received value to the bound object. This is a lightweight alternative to what Boost.Phoenix and Boost.Lambda provides.

```
namespace boost {
  namespace log {
    struct assign_fun;
    template<typename AssigneeT>
      binder1st< assign_fun, AssigneeT & > bind_assign(AssigneeT &);
  }
}
```

Struct assign_fun

boost::log::assign_fun — The function object that assigns its second operand to the first one.

Synopsis

```
// In header: <boost/log/utility/functional/bind_assign.hpp>


struct assign_fun {
  // types
  typedef void result_type;

  // public member functions
  template<typename LeftT, typename RightT>
    void operator()(LeftT &, RightT const &) const;
};
```

Description

`assign_fun` public member functions

1.
```
template<typename LeftT, typename RightT>
  void operator()(LeftT & assignee, RightT const & val) const;
```

Function template bind_assign

boost::log::bind_assign

Synopsis

```
// In header: <boost/log/utility/functional/bind_assign.hpp>


template<typename AssigneeT>
  binder1st< assign_fun, AssigneeT & > bind_assign(AssigneeT & assignee);
```

Header <boost/log/utility/functional/bind_output.hpp>

Andrey Semashev

30.03.2008
This header contains a function object that puts the received value to the bound stream. This is a lightweight alternative to what Boost.Phoenix and Boost.Lambda provides.

```
namespace boost {
  namespace log {
    struct output_fun;
    template<typename StreamT>
      binder1st< output_fun, StreamT & > bind_output(StreamT &);
  }
}
```

Struct output_fun

boost::log::output_fun — The function object that outputs its second operand to the first one.

Synopsis

```
// In header: <boost/log/utility/functional/bind_output.hpp>


struct output_fun {
  // types
  typedef void result_type;

  // public member functions
  template<typename StreamT, typename T>
    void operator()(StreamT &, T const &) const;
};
```

Description

`output_fun` public member functions

1.
```
template<typename StreamT, typename T>
  void operator()(StreamT & strm, T const & val) const;
```

Function template bind_output

boost::log::bind_output

Synopsis

```
// In header: <boost/log/utility/functional/bind_output.hpp>


template<typename StreamT>
  binder1st< output_fun, StreamT & > bind_output(StreamT & strm);
```

Header <boost/log/utility/functional/bind_to_log.hpp>

Andrey Semashev

06.11.2012
This header contains a function object that puts the received value to the bound stream using the `to_log` manipulator. This is a lightweight alternative to what Boost.Phoenix and Boost.Lambda provides.

```
namespace boost {
  namespace log {
    template<typename TagT = void> struct to_log_fun;

    template<> struct to_log_fun<void>;
    template<typename StreamT>
      binder1st< to_log_fun<  >, StreamT & > bind_to_log(StreamT &);
    template<typename TagT, typename StreamT>
      binder1st< to_log_fun< TagT >, StreamT & > bind_to_log(StreamT &);
  }
}
```

Struct template to_log_fun

boost::log::to_log_fun — The function object that outputs its second operand to the first one.

Synopsis

```
// In header: <boost/log/utility/functional/bind_to_log.hpp>

template<typename TagT = void>
struct to_log_fun {
  // types
  typedef void result_type;

  // public member functions
  template<typename StreamT, typename T>
    void operator()(StreamT &, T const &) const;
};
```

Description

`to_log_fun` public member functions

1.
```
template<typename StreamT, typename T>
  void operator()(StreamT & strm, T const & val) const;
```

Struct to_log_fun<void>

boost::log::to_log_fun<void> — The function object that outputs its second operand to the first one.

Synopsis

```
// In header: <boost/log/utility/functional/bind_to_log.hpp>


struct to_log_fun<void> {
  // types
  typedef void result_type;

  // public member functions
  template<typename StreamT, typename T>
    void operator()(StreamT &, T const &) const;
};
```

Description

`to_log_fun` public member functions

1.
```
template<typename StreamT, typename T>
  void operator()(StreamT & strm, T const & val) const;
```

Function template bind_to_log

boost::log::bind_to_log

Synopsis

```
// In header: <boost/log/utility/functional/bind_to_log.hpp>


template<typename StreamT>
  binder1st< to_log_fun<  >, StreamT & > bind_to_log(StreamT & strm);
```

Function template bind_to_log

boost::log::bind_to_log

Synopsis

```
// In header: <boost/log/utility/functional/bind_to_log.hpp>


template<typename TagT, typename StreamT>
  binder1st< to_log_fun< TagT >, StreamT & > bind_to_log(StreamT & strm);
```

Header <boost/log/utility/functional/contains.hpp>

Andrey Semashev

30.03.2008
This header contains a predicate for checking if the provided string contains a substring.

```
namespace boost {
  namespace log {
    struct contains_fun;
  }
}
```

Struct contains_fun

boost::log::contains_fun — The `contains` functor.

Synopsis

```
// In header: <boost/log/utility/functional/contains.hpp>


struct contains_fun {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;
};
```

Description

`contains_fun` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

Header <boost/log/utility/functional/ends_with.hpp>

Andrey Semashev

30.03.2008
This header contains a predicate for checking if the provided string ends with a substring.

```
namespace boost {
  namespace log {
    struct ends_with_fun;
  }
}
```

Struct ends_with_fun

boost::log::ends_with_fun — The `ends_with` functor.

Synopsis

```
// In header: <boost/log/utility/functional/ends_with.hpp>


struct ends_with_fun {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;
};
```

Description

`ends_with_fun` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

Header <boost/log/utility/functional/fun_ref.hpp>

Andrey Semashev

30.03.2008
This header contains function object reference adapter. The adapter stores a reference to external function object and forwards all calls to the referred function.

```
namespace boost {
  namespace log {
    template<typename FunT> struct function_reference_wrapper;
    template<typename FunT> function_reference_wrapper< FunT > fun_ref(FunT &);
  }
}
```

Struct template function_reference_wrapper

boost::log::function_reference_wrapper — Reference wrapper for function objects.

Synopsis

```
// In header: <boost/log/utility/functional/fun_ref.hpp>

template<typename FunT>
struct function_reference_wrapper {
  // types
  typedef FunT::result_type result_type;

  // construct/copy/destruct
  explicit function_reference_wrapper(FunT &);

  // public member functions
  result_type operator()() const;
  template<typename... ArgsT> result_type operator()(ArgsT const &...) const;
};
```

Description

`function_reference_wrapper` public construct/copy/destruct

1. ```
   explicit function_reference_wrapper(FunT & fun);
   ```

`function_reference_wrapper` public member functions

1. ```
   result_type operator()() const;
   ```

2. ```
   template<typename... ArgsT>
     result_type operator()(ArgsT const &... args) const;
   ```

Function template fun_ref

boost::log::fun_ref

Synopsis

```
// In header: <boost/log/utility/functional/fun_ref.hpp>


template<typename FunT> function_reference_wrapper< FunT > fun_ref(FunT & fun);
```
```

Header <boost/log/utility/functional/in_range.hpp>

Andrey Semashev

30.03.2008
This header contains a predicate for checking if the provided value is within a half-open range.

```
namespace boost {
  namespace log {
    struct in_range_fun;
  }
}
```

Struct in_range_fun

boost::log::in_range_fun — The in_range functor.

Synopsis

```
// In header: <boost/log/utility/functional/in_range.hpp>


struct in_range_fun {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U>
    bool operator()(T const &, std::pair< U, U > const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, std::pair< U, U > const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, std::pair< U, U > const &, mpl::true_ const &);
};
```

Description

`in_range_fun` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & value, std::pair< U, U > const & rng) const;
```

`in_range_fun` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & value, std::pair< U, U > const & rng,
                 mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & value, std::pair< U, U > const & rng,
                 mpl::true_ const &);
```

Header <boost/log/utility/functional/logical.hpp>

Andrey Semashev

30.03.2008

This header contains logical predicates for value comparison, analogous to `std::less`, `std::greater` and others. The main difference from the standard equivalents is that the predicates defined in this header are not templates and therefore do not require a fixed argument type. Furthermore, both arguments may have different types, in which case the comparison is performed without type conversion.

Note

In case if arguments are integral, the conversion is performed according to the standard C++ rules in order to avoid warnings from the compiler.

```
namespace boost {
  namespace log {
    struct equal_to;
    struct greater;
    struct greater_equal;
    struct less;
    struct less_equal;
    struct not_equal_to;
  }
}
```

Struct equal_to

boost::log::equal_to — Equality predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct equal_to {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`equal_to` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`equal_to` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Struct greater

boost::log::greater — Greater predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct greater {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`greater` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`greater` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Struct greater_equal

boost::log::greater_equal — Greater or equal predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct greater_equal {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`greater_equal` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`greater_equal` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Struct less

boost::log::less — Less predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct less {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`less` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`less` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Struct less_equal

boost::log::less_equal — Less or equal predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct less_equal {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`less_equal` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`less_equal` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Struct not_equal_to

boost::log::not_equal_to — Inequality predicate.

Synopsis

```
// In header: <boost/log/utility/functional/logical.hpp>


struct not_equal_to {
  // types
  typedef bool result_type;

  // public member functions
  template<typename T, typename U> bool operator()(T const &, U const &) const;

  // private static functions
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::false_ const &);
  template<typename T, typename U>
    static bool op(T const &, U const &, mpl::true_ const &);
};
```

Description

`not_equal_to` public member functions

1.
```
template<typename T, typename U>
  bool operator()(T const & left, U const & right) const;
```

`not_equal_to` private static functions

1.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::false_ const &);
```

2.
```
template<typename T, typename U>
  static bool op(T const & left, U const & right, mpl::true_ const &);
```

Header <boost/log/utility/functional/matches.hpp>

Andrey Semashev

30.03.2008
This header contains a predicate for checking if the provided string matches a regular expression.

```
namespace boost {
  namespace log {
    struct matches_fun;
  }
}
```

Struct matches_fun

boost::log::matches_fun — The regex matching functor.

Synopsis

```
// In header: <boost/log/utility/functional/matches.hpp>


struct matches_fun {
  // types
  typedef bool result_type;

  // public member functions
  template<typename StringT, typename ExpressionT>
    bool operator()(StringT const &, ExpressionT const &) const;
  template<typename StringT, typename ExpressionT, typename ArgT>
    bool operator()(StringT const &, ExpressionT const &, ArgT const &) const;
};
```

Description

`matches_fun` public member functions

1.
```
template<typename StringT, typename ExpressionT>
  bool operator()(StringT const & str, ExpressionT const & expr) const;
```

2.
```
template<typename StringT, typename ExpressionT, typename ArgT>
  bool operator()(StringT const & str, ExpressionT const & expr,
                  ArgT const & arg) const;
```

Header <boost/log/utility/functional/nop.hpp>

Andrey Semashev

30.03.2008
This header contains a function object that does nothing.

```
namespace boost {
  namespace log {
    struct nop;
  }
}
```

Struct nop

boost::log::nop — The function object that does nothing.

Synopsis

```
// In header: <boost/log/utility/functional/nop.hpp>


struct nop {
  // types
  typedef void result_type;

  // public member functions
  void operator()() const noexcept;
  template<typename... ArgsT> void operator()(ArgsT const &) const noexcept;
};
```

Description

nop public member functions

1.
```
void operator()() const noexcept;
```

2.
```
template<typename... ArgsT> void operator()(ArgsT const & ...) const noexcept;
```

Header <boost/log/utility/functional/save_result.hpp>

Andrey Semashev

19.01.2013
This header contains function object adapter that saves the result of the adopted function to an external variable.

```
namespace boost {
  namespace log {
    template<typename FunT, typename AssigneeT> struct save_result_wrapper;
    template<typename FunT, typename AssigneeT>
      save_result_wrapper< FunT, AssigneeT >
      save_result(FunT const &, AssigneeT &);
  }
}
```

Struct template save_result_wrapper

boost::log::save_result_wrapper — Function object wrapper for saving the adopted function object result.

Synopsis

```
// In header: <boost/log/utility/functional/save_result.hpp>

template<typename FunT, typename AssigneeT>
struct save_result_wrapper {
  // types
  typedef void result_type;

  // construct/copy/destruct
  save_result_wrapper(FunT, AssigneeT &);

  // public member functions
  template<typename ArgT> result_type operator()(ArgT const &) const;
};
```

Description

`save_result_wrapper` public construct/copy/destruct

1. ```
   save_result_wrapper(FunT fun, AssigneeT & assignee);
   ```

`save_result_wrapper` public member functions

1. ```
   template<typename ArgT> result_type operator()(ArgT const & arg) const;
   ```

Function template save_result

boost::log::save_result

Synopsis

```
// In header: <boost/log/utility/functional/save_result.hpp>


template<typename FunT, typename AssigneeT>
  save_result_wrapper< FunT, AssigneeT >
  save_result(FunT const & fun, AssigneeT & assignee);
```

Header <boost/log/utility/intrusive_ref_counter.hpp>

Andrey Semashev

12.03.2009
This header is deprecated, use boost/smart_ptr/intrusive_ref_counter.hpp instead. The header is left for backward compatibility and will be removed in future versions.

```
namespace boost {
  namespace log {
    typedef unspecified intrusive_ref_counter;
  }
}
```

Header <boost/log/utility/manipulators.hpp>

Andrey Semashev

06.11.2012
This header includes all manipulators.

Header <boost/log/utility/manipulators/add_value.hpp>

Andrey Semashev

26.11.2012
This header contains the `add_value` manipulator.

```
namespace boost {
  namespace log {
    template<typename RefT> class add_value_manip;
    template<typename CharT, typename RefT>
      basic_record_ostream< CharT > &
      operator<<(basic_record_ostream< CharT > &,
                 add_value_manip< RefT > const &);
    template<typename T>
      add_value_manip< T && > add_value(attribute_name const &, T &&);
    template<typename DescriptorT, template< typename > class ActorT>
      add_value_manip< typename DescriptorT::value_type && >
      add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                typename DescriptorT::value_type &&);
    template<typename DescriptorT, template< typename > class ActorT>
      add_value_manip< typename DescriptorT::value_type & >
      add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                typename DescriptorT::value_type &);
    template<typename DescriptorT, template< typename > class ActorT>
      add_value_manip< typename DescriptorT::value_type const & >
      add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
                typename DescriptorT::value_type const &);
  }
}
```

Class template add_value_manip

boost::log::add_value_manip — Attribute value manipulator.

Synopsis

```
// In header: <boost/log/utility/manipulators/add_value.hpp>

template<typename RefT>
class add_value_manip {
public:
  // types
  typedef RefT                                                                    reference_type; ↵
 // Stored reference type.
  typedef remove_cv< typename remove_reference< reference_type >::type >::type value_type;     ↵
 // Attribute value type.

  // construct/copy/destruct
  add_value_manip(attribute_name const &, reference_type);

  // public member functions
  attribute_name get_name() const;
  get_value_result_type get_value() const;
};
```

Description

`add_value_manip` public construct/copy/destruct

1. ```
   add_value_manip(attribute_name const & name, reference_type value);
   ```

   Initializing constructor.

`add_value_manip` public member functions

1. ```
   attribute_name get_name() const;
   ```

   Returns attribute name.

2. ```
   get_value_result_type get_value() const;
   ```

   Returns attribute value.

Function template operator<<

boost::log::operator<< — The operator attaches an attribute value to the log record.

Synopsis

```
// In header: <boost/log/utility/manipulators/add_value.hpp>


template<typename CharT, typename RefT>
  basic_record_ostream< CharT > &
  operator<<(basic_record_ostream< CharT > & strm,
             add_value_manip< RefT > const & manip);
```
```

Function add_value

boost::log::add_value — The function creates a manipulator that attaches an attribute value to a log record.

Synopsis

```
// In header: <boost/log/utility/manipulators/add_value.hpp>


template<typename T>
  add_value_manip< T && > add_value(attribute_name const & name, T && value);
template<typename DescriptorT, template< typename > class ActorT>
  add_value_manip< typename DescriptorT::value_type && >
  add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            typename DescriptorT::value_type && value);
template<typename DescriptorT, template< typename > class ActorT>
  add_value_manip< typename DescriptorT::value_type & >
  add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            typename DescriptorT::value_type & value);
template<typename DescriptorT, template< typename > class ActorT>
  add_value_manip< typename DescriptorT::value_type const & >
  add_value(expressions::attribute_keyword< DescriptorT, ActorT > const &,
            typename DescriptorT::value_type const & value);
```

Header <boost/log/utility/manipulators/dump.hpp>

Andrey Semashev

03.05.2013
This header contains the `dump` output manipulator.

```
namespace boost {
  namespace log {
    class bounded_dump_manip;
    class dump_manip;
    template<typename CharT, typename TraitsT>
      std::basic_ostream< CharT, TraitsT > &
      operator<<(std::basic_ostream< CharT, TraitsT > &, dump_manip const &);
    template<typename CharT, typename TraitsT>
      std::basic_ostream< CharT, TraitsT > &
      operator<<(std::basic_ostream< CharT, TraitsT > &,
                 bounded_dump_manip const &);
    template<typename T> unspecified dump(T *, std::size_t);
    template<typename T> dump_manip dump_elements(T *, std::size_t);
    template<typename T> unspecified dump(T *, std::size_t, std::size_t);
    template<typename T>
      bounded_dump_manip dump_elements(T *, std::size_t, std::size_t);
  }
}
```

Class bounded_dump_manip

boost::log::bounded_dump_manip — Manipulator for printing binary representation of the data with a size limit.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


class bounded_dump_manip : public dump_manip {
public:
  // construct/copy/destruct
  bounded_dump_manip(const void *, std::size_t, std::size_t) noexcept;
  bounded_dump_manip(bounded_dump_manip const &) noexcept;

  // public member functions
  std::size_t get_max_size() const noexcept;
};
```

Description

`bounded_dump_manip` public construct/copy/destruct

1. ```
   bounded_dump_manip(const void * data, std::size_t size, std::size_t max_size) noexcept;
   ```

2. ```
   bounded_dump_manip(bounded_dump_manip const & that) noexcept;
   ```

`bounded_dump_manip` public member functions

1. ```
   std::size_t get_max_size() const noexcept;
   ```

Class dump_manip

boost::log::dump_manip — Manipulator for printing binary representation of the data.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


class dump_manip {
public:
  // construct/copy/destruct
  dump_manip(const void *, std::size_t) noexcept;
  dump_manip(dump_manip const &) noexcept;

  // public member functions
  const void * get_data() const noexcept;
  std::size_t get_size() const noexcept;
};
```

Description

`dump_manip` public construct/copy/destruct

1. ```
   dump_manip(const void * data, std::size_t size) noexcept;
   ```

2.
```
dump_manip(dump_manip const & that) noexcept;
```

`dump_manip` public member functions

1.
```
const void * get_data() const noexcept;
```

2.
```
std::size_t get_size() const noexcept;
```

Function template operator<<

boost::log::operator<< — The operator outputs binary data to a stream.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename CharT, typename TraitsT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             dump_manip const & manip);
```

Function template operator<<

boost::log::operator<< — The operator outputs binary data to a stream.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename CharT, typename TraitsT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             bounded_dump_manip const & manip);
```

Function template dump

boost::log::dump — Creates a stream manipulator that will output contents of a memory region in hexadecimal form.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename T> unspecified dump(T * data, std::size_t size);
```

Description

Parameters: `data` The pointer to the beginning of the region

`size` The size of the region, in bytes
Returns: The manipulator that is to be put to a stream

Function template dump_elements

boost::log::dump_elements — Creates a stream manipulator that will dump elements of an array in hexadecimal form.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename T> dump_manip dump_elements(T * data, std::size_t count);
```

Description

Parameters: `count` The size of the region, in number of `T` elements
`data` The pointer to the beginning of the array
Returns: The manipulator that is to be put to a stream

Function template dump

boost::log::dump — Creates a stream manipulator that will output contents of a memory region in hexadecimal form.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename T>
  unspecified dump(T * data, std::size_t size, std::size_t max_size);
```

Description

Parameters: `data` The pointer to the beginning of the region
`size` The size of the region, in bytes max_size The maximum number of bytes of the region to output
Returns: The manipulator that is to be put to a stream

Function template dump_elements

boost::log::dump_elements — Creates a stream manipulator that will dump elements of an array in hexadecimal form.

Synopsis

```
// In header: <boost/log/utility/manipulators/dump.hpp>


template<typename T>
  bounded_dump_manip
  dump_elements(T * data, std::size_t count, std::size_t max_count);
```

Description

Parameters: `count` The size of the region, in number of `T` elements max_count The maximum number of elements to output
`data` The pointer to the beginning of the array

Returns: The manipulator that is to be put to a stream

Header <boost/log/utility/manipulators/to_log.hpp>

Andrey Semashev

06.11.2012
This header contains the `to_log` output manipulator.

```
namespace boost {
  namespace log {
    template<typename T, typename TagT = void> class to_log_manip;
    template<typename CharT, typename TraitsT, typename T, typename TagT>
      std::basic_ostream< CharT, TraitsT > &
      operator<<(std::basic_ostream< CharT, TraitsT > &,
                 to_log_manip< T, TagT >);
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T, typename TagT>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &,
                 to_log_manip< T, TagT >);
    template<typename T> to_log_manip< T > to_log(T const &);
    template<typename TagT, typename T>
      to_log_manip< T, TagT > to_log(T const &);
  }
}
```

Class template to_log_manip

boost::log::to_log_manip — Generic manipulator for customizing output to log.

Synopsis

```
// In header: <boost/log/utility/manipulators/to_log.hpp>

template<typename T, typename TagT = void>
class to_log_manip {
public:
  // types
  typedef T    value_type;  // Output value type.
  typedef TagT tag_type;    // Value tag type.

  // construct/copy/destruct
  explicit to_log_manip(value_type const &);
  to_log_manip(to_log_manip const &);

  // public member functions
  value_type const & get() const;
};
```

Description

`to_log_manip` public construct/copy/destruct

1. ```
   explicit to_log_manip(value_type const & value);
   ```
   ```

2.
```
to_log_manip(to_log_manip const & that);
```

`to_log_manip` public member functions

1.
```
value_type const & get() const;
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/manipulators/to_log.hpp>


template<typename CharT, typename TraitsT, typename T, typename TagT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             to_log_manip< T, TagT > manip);
```

Function template operator<<

boost::log::operator<<

Synopsis

```
// In header: <boost/log/utility/manipulators/to_log.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T,
         typename TagT>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > & strm,
             to_log_manip< T, TagT > manip);
```

Function template to_log

boost::log::to_log

Synopsis

```
// In header: <boost/log/utility/manipulators/to_log.hpp>


template<typename T> to_log_manip< T > to_log(T const & value);
```

Function template to_log

boost::log::to_log

Synopsis

```
// In header: <boost/log/utility/manipulators/to_log.hpp>


template<typename TagT, typename T>
  to_log_manip< T, TagT > to_log(T const & value);
```

Header <boost/log/utility/once_block.hpp>

The header defines classes and macros for once-blocks.

Andrey Semashev

23.06.2010

```


BOOST_LOG_ONCE_BLOCK_INIT
BOOST_LOG_ONCE_BLOCK_FLAG(flag_var)
BOOST_LOG_ONCE_BLOCK()
```

```
namespace boost {
  namespace log {
    struct once_block_flag;
  }
}
```

Struct once_block_flag

boost::log::once_block_flag — A flag to detect if a code block has already been executed.

Synopsis

```
// In header: <boost/log/utility/once_block.hpp>


struct once_block_flag {
};
```

Description

This structure should be used in conjunction with the `BOOST_LOG_ONCE_BLOCK_FLAG` macro. Usage example:

```
once_block_flag flag = BOOST_LOG_ONCE_BLOCK_INIT;

void foo() { BOOST_LOG_ONCE_BLOCK_FLAG(flag) { puts("Hello, world once!"); } }
```

Macro BOOST_LOG_ONCE_BLOCK_INIT

BOOST_LOG_ONCE_BLOCK_INIT

Synopsis

```
// In header: <boost/log/utility/once_block.hpp>

BOOST_LOG_ONCE_BLOCK_INIT
```

Description

The static initializer for `once_block_flag`.

Macro BOOST_LOG_ONCE_BLOCK_FLAG

BOOST_LOG_ONCE_BLOCK_FLAG

Synopsis

```
// In header: <boost/log/utility/once_block.hpp>

BOOST_LOG_ONCE_BLOCK_FLAG(flag_var)
```

Description

Begins a code block to be executed only once, with protection against thread concurrency. User has to provide the flag variable that controls whether the block has already been executed.

Macro BOOST_LOG_ONCE_BLOCK

BOOST_LOG_ONCE_BLOCK

Synopsis

```
// In header: <boost/log/utility/once_block.hpp>

BOOST_LOG_ONCE_BLOCK()
```

Description

Begins a code block to be executed only once, with protection against thread concurrency.

Header <boost/log/utility/record_ordering.hpp>

Andrey Semashev

23.08.2009
This header contains ordering predicates for logging records.

```
namespace boost {
  namespace log {
    template<typename FunT = less> class abstract_ordering;
    template<typename ValueT, typename FunT = less>
      class attribute_value_ordering;
    template<typename ValueT, typename FunT>
      attribute_value_ordering< ValueT, FunT >
      make_attr_ordering(attribute_name const &, FunT const &);
    template<typename FunT>
      unspecified make_attr_ordering(attribute_name const &, FunT const &);
  }
}
```

Class template abstract_ordering

boost::log::abstract_ordering — Ordering predicate, based on opaque pointers to the record view implementation data.

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>

template<typename FunT = less>
class abstract_ordering : private FunT {
public:
  // types
  typedef bool result_type;  // Result type.

  // construct/copy/destruct
  abstract_ordering();
  explicit abstract_ordering(FunT const &);

  // public member functions
  result_type operator()(record_view const &, record_view const &) const;
};
```

Description

Since record views only refer to a shared implementation data, this predicate is able to order the views by comparing the pointers to the data. Therefore two views are considered to be equivalent if they refer to the same implementation data. Otherwise it is not specified whether one record is ordered before the other until the predicate is applied. Note that the ordering may change every time the application runs.

This kind of ordering may be useful if log records are to be stored in an associative container with as least performance overhead as possible, when the particular order is not important.

The `FunT` template argument is the predicate that is used to actually compare pointers. It should be able to compare `const void*` pointers. The compared pointers may refer to distinct memory regions, the pointers must not be interpreted in any way.

`abstract_ordering` public construct/copy/destruct

1. ```
   abstract_ordering();
   ```

   Default constructor. Requires `FunT` to be default constructible.

2. ```
   explicit abstract_ordering(FunT const & fun);
   ```

   Initializing constructor. Constructs `FunT` instance as a copy of the *fun* argument.

`abstract_ordering` public member functions

1. ```
   result_type operator()(record_view const & left, record_view const & right) const;
   ```

   Ordering operator

Class template attribute_value_ordering

boost::log::attribute_value_ordering — Ordering predicate, based on attribute values associated with records.

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>

template<typename ValueT, typename FunT = less>
class attribute_value_ordering : private FunT {
public:
  // types
  typedef bool   result_type;  // Result type.
  typedef ValueT value_type;   // Compared attribute value type.

  // member classes/structs/unions

  struct l1_visitor {
    // types
    typedef void result_type;

    // construct/copy/destruct
    l1_visitor(attribute_value_ordering const &, record_view const &, bool &);

    // public member functions
    template<typename LeftT> result_type operator()(LeftT const &) const;
  };
  template<typename LeftT>
  struct l2_visitor {
    // types
    typedef void result_type;

    // construct/copy/destruct
    l2_visitor(FunT const &, LeftT const &, bool &);

    // public member functions
    template<typename RightT> result_type operator()(RightT const &) const;
  };

  // construct/copy/destruct
  explicit attribute_value_ordering(attribute_name const &,
                                    FunT const & = FunT());

  // public member functions
  result_type operator()(record_view const &, record_view const &) const;
};
```

Description

This predicate allows to order log records based on values of a specifically named attribute associated with them. Two given log records being compared should both have the specified attribute value of the specified type to be able to be ordered properly. As a special case, if neither of the records have the value, these records are considered equivalent. Otherwise, the ordering results are unspecified.
```

`attribute_value_ordering` public construct/copy/destruct

1.
```
explicit attribute_value_ordering(attribute_name const & name,
                                  FunT const & fun = FunT());
```

Initializing constructor.

Parameters:

`fun`	The ordering functor
`name`	The attribute value name to be compared

`attribute_value_ordering` public member functions

1.
```
result_type operator()(record_view const & left, record_view const & right) const;
```

Ordering operator

Struct l1_visitor

boost::log::attribute_value_ordering::l1_visitor

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>



struct l1_visitor {
  // types
  typedef void result_type;

  // construct/copy/destruct
  l1_visitor(attribute_value_ordering const &, record_view const &, bool &);

  // public member functions
  template<typename LeftT> result_type operator()(LeftT const &) const;
};
```

Description

`l1_visitor` public construct/copy/destruct

1.
```
l1_visitor(attribute_value_ordering const & owner, record_view const & right,
           bool & result);
```

`l1_visitor` public member functions

1.
```
template<typename LeftT> result_type operator()(LeftT const & left) const;
```

Struct template l2_visitor

boost::log::attribute_value_ordering::l2_visitor

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>


template<typename LeftT>
struct l2_visitor {
  // types
  typedef void result_type;

  // construct/copy/destruct
  l2_visitor(FunT const &, LeftT const &, bool &);

  // public member functions
  template<typename RightT> result_type operator()(RightT const &) const;
};
```

Description

`l2_visitor` public construct/copy/destruct

1. ```
   l2_visitor(FunT const & fun, LeftT const & left, bool & result);
   ```

`l2_visitor` public member functions

1. ```
   template<typename RightT> result_type operator()(RightT const & right) const;
   ```

Function template make_attr_ordering

boost::log::make_attr_ordering

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>


template<typename ValueT, typename FunT>
  attribute_value_ordering< ValueT, FunT >
  make_attr_ordering(attribute_name const & name, FunT const & fun);
```

Description

The function constructs a log record ordering predicate

Function template make_attr_ordering

boost::log::make_attr_ordering

Synopsis

```
// In header: <boost/log/utility/record_ordering.hpp>


template<typename FunT>
  unspecified make_attr_ordering(attribute_name const & name,
                                 FunT const & fun);
```

Description

The function constructs a log record ordering predicate

Header <boost/log/utility/setup.hpp>

Andrey Semashev

16.02.2013
This header includes all library setup helpers.

Header <boost/log/utility/setup/common_attributes.hpp>

Andrey Semashev

16.05.2008
The header contains implementation of convenience functions for registering commonly used attributes.

```
namespace boost {
  namespace log {
    void add_common_attributes();
  }
}
```

Function add_common_attributes

boost::log::add_common_attributes — Simple attribute initialization routine.

Synopsis

```
// In header: <boost/log/utility/setup/common_attributes.hpp>


void add_common_attributes();
```

Description

The function adds commonly used attributes to the logging system. Specifically, the following attributes are registered globally:

- LineID - logging records counter with value type `unsigned int`
- TimeStamp - local time generator with value type `boost::posix_time::ptime`
- ProcessID - current process identifier with value type `attributes::current_process_id::value_type`
- ThreadID - in multithreaded builds, current thread identifier with value type `attributes::current_thread_id::value_type`

Header <boost/log/utility/setup/console.hpp>

Andrey Semashev

16.05.2008
The header contains implementation of convenience functions for enabling logging to console.

```
namespace boost {
  namespace log {
    template<typename CharT, typename... ArgsT>
      shared_ptr< sinks::synchronous_sink< sinks::basic_text_ostream_backend< CharT > >>
      add_console_log(std::basic_ostream< CharT > &, ArgsT...const &);
    template<typename CharT, typename... ArgsT>
      shared_ptr< sinks::synchronous_sink< sinks::basic_text_ostream_backend< CharT > >>
      add_console_log(ArgsT...const &);
    shared_ptr< sinks::synchronous_sink< sinks::text_ostream_backend >>
    add_console_log();
    shared_ptr< sinks::synchronous_sink< sinks::wtext_ostream_backend >>
    wadd_console_log();
  }
}
```

Function template add_console_log

boost::log::add_console_log

Synopsis

```
// In header: <boost/log/utility/setup/console.hpp>


template<typename CharT, typename... ArgsT>
  shared_ptr< sinks::synchronous_sink< sinks::basic_text_ostream_backend< CharT > >>
  add_console_log(std::basic_ostream< CharT > & strm, ArgsT...const & args);
```

Description

The function constructs sink for the specified console stream and adds it to the core

Parameters:

`args` Optional additional named arguments for the sink initialization. The following arguments are supported:

- `filter` Specifies a filter to install into the sink. May be a string that represents a filter, or a filter lambda expression.
- `format` Specifies a formatter to install into the sink. May be a string that represents a formatter, or a formatter lambda expression (either streaming or Boost.Format-like notation).
- `auto_flush` A boolean flag that shows whether the sink should automatically flush the stream after each written record.

`strm` One of the standard console streams: `std::cout`, `std::cerr` or `std::clog` (or the corresponding wide-character analogues).

Returns: Pointer to the constructed sink.

Function template add_console_log

boost::log::add_console_log

Synopsis

```
// In header: <boost/log/utility/setup/console.hpp>


template<typename CharT, typename... ArgsT>
  shared_ptr< sinks::synchronous_sink< sinks::basic_text_ostream_backend< CharT > >>
  add_console_log(ArgsT...const & args);
```

Description

Equivalent to: `add_console_log(std::clog);` or `add_console_log(std::wclog);`, depending on the `CharT` type.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Function add_console_log

boost::log::add_console_log

Synopsis

```
// In header: <boost/log/utility/setup/console.hpp>


shared_ptr< sinks::synchronous_sink< sinks::text_ostream_backend >>
add_console_log();
```

Description

The function constructs sink for the `std::clog` stream and adds it to the core

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Returns: Pointer to the constructed sink.

Function wadd_console_log

boost::log::wadd_console_log

Synopsis

```
// In header: <boost/log/utility/setup/console.hpp>


shared_ptr< sinks::synchronous_sink< sinks::wtext_ostream_backend >>
wadd_console_log();
```

Description

The function constructs sink for the `std::wclog` stream and adds it to the core

Returns: Pointer to the constructed sink.

Header <boost/log/utility/setup/file.hpp>

Andrey Semashev

16.05.2008
The header contains implementation of convenience functions for enabling logging to a file.

```
namespace boost {
  namespace log {
    template<typename... ArgsT>
      shared_ptr< sinks::synchronous_sink< sinks::text_file_backend > >
      add_file_log(ArgsT...const &);
  }
}
```

Function template add_file_log

boost::log::add_file_log

Synopsis

```
// In header: <boost/log/utility/setup/file.hpp>


template<typename... ArgsT>
  shared_ptr< sinks::synchronous_sink< sinks::text_file_backend > >
  add_file_log(ArgsT...const & args);
```

Description

The function initializes the logging library to write logs to a file stream.

Parameters: `args` A number of named arguments. The following parameters are supported:

- `file_name` The file name or its pattern. This parameter is mandatory.
- `open_mode` The mask that describes the open mode for the file. See `std::ios_base::openmode`.
- `rotation_size` The size of the file at which rotation should occur. See `basic_text_file_backend`.
- `time_based_rotation` The predicate for time-based file rotations. See `basic_text_file_backend`.
- `auto_flush` A boolean flag that shows whether the sink should automatically flush the file after each written record.
- `target` The target directory to store rotated files in. See `sinks::file::make_collector`.
- `max_size` The maximum total size of rotated files in the target directory. See `sinks::file::make_collector`.
- `min_free_space` Minimum free space in the target directory. See `sinks::file::make_collector`.
- `scan_method` The method of scanning the target directory for log files. See `sinks::file::scan_method`.
- `filter` Specifies a filter to install into the sink. May be a string that represents a filter, or a filter lambda expression.

- `format` Specifies a formatter to install into the sink. May be a string that represents a formatter, or a formatter lambda expression (either streaming or Boost.Format-like notation).

Returns: Pointer to the constructed sink.

Header <boost/log/utility/setup/filter_parser.hpp>

Andrey Semashev

31.03.2008
The header contains definition of a filter parser function.

```
namespace boost {
  namespace log {
    template<typename CharT, typename AttributeValueT>
      class basic_filter_factory;

    template<typename CharT> struct filter_factory;
    template<typename CharT>
      void register_filter_factory(attribute_name const &,
                                   shared_ptr< filter_factory< CharT > > const &);
    template<typename FactoryT>
      enable_if< is_base_and_derived< filter_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
      register_filter_factory(attribute_name const &,
                              shared_ptr< FactoryT > const &);
    template<typename AttributeValueT, typename CharT>
      void register_simple_filter_factory(attribute_name const &);
    template<typename AttributeValueT>
      void register_simple_filter_factory(attribute_name const &);
    template<typename CharT, typename DescriptorT,
             template< typename > class ActorT>
      void register_simple_filter_factory(expressions::attrib↵
ute_keyword< DescriptorT, ActorT > const &);
    template<typename CharT> filter parse_filter(const CharT *, const CharT *);
    template<typename CharT, typename TraitsT, typename AllocatorT>
      filter parse_filter(std::basic_string< CharT, TraitsT, AllocatorT > const &);
    template<typename CharT> filter parse_filter(const CharT *);
  }
}
```

Class template basic_filter_factory

boost::log::basic_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>

template<typename CharT, typename AttributeValueT>
class basic_filter_factory : public boost::log::filter_factory< CharT > {
public:
  // types
  typedef AttributeValueT       value_type;   // The type(s) of the attribute value expected.
  typedef base_type::string_type string_type;

  // public member functions
  virtual filter on_exists_test(attribute_name const &);
  virtual filter
  on_equality_relation(attribute_name const &, string_type const &);
  virtual filter
  on_inequality_relation(attribute_name const &, string_type const &);
  virtual filter on_less_relation(attribute_name const &, string_type const &);
  virtual filter
  on_greater_relation(attribute_name const &, string_type const &);
  virtual filter
  on_less_or_equal_relation(attribute_name const &, string_type const &);
  virtual filter
  on_greater_or_equal_relation(attribute_name const &, string_type const &);
  virtual filter
  on_custom_relation(attribute_name const &, string_type const &,
                     string_type const &);
  virtual value_type parse_argument(string_type const &);
};
```

Description

The base class for filter factories. The class defines default implementations for most filter expressions. In order to be able to construct filters, the attribute value type must support reading from a stream. Also, the default filters will rely on relational operators for the type, so these operators must also be defined.

`basic_filter_factory` public member functions

1. ```
   virtual filter on_exists_test(attribute_name const & name);
   ```

   The callback for filter for the attribute existence test

2. ```
   virtual filter
   on_equality_relation(attribute_name const & name, string_type const & arg);
   ```

   The callback for equality relation filter

3. ```
   virtual filter
   on_inequality_relation(attribute_name const & name, string_type const & arg);
   ```

   The callback for inequality relation filter

4. ```
   virtual filter
   on_less_relation(attribute_name const & name, string_type const & arg);
   ```

   The callback for less relation filter

5.
```
virtual filter
on_greater_relation(attribute_name const & name, string_type const & arg);
```

The callback for greater relation filter

6.
```
virtual filter
on_less_or_equal_relation(attribute_name const & name,
                          string_type const & arg);
```

The callback for less or equal relation filter

7.
```
virtual filter
on_greater_or_equal_relation(attribute_name const & name,
                             string_type const & arg);
```

The callback for greater or equal relation filter

8.
```
virtual filter
on_custom_relation(attribute_name const & name, string_type const & rel,
                   string_type const & arg);
```

The callback for custom relation filter

9.
```
virtual value_type parse_argument(string_type const & arg);
```

The function parses the argument value for a binary relation

Struct template filter_factory

boost::log::filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>

template<typename CharT>
struct filter_factory {
  // types
  typedef CharT                           char_type;    // Character type.
  typedef std::basic_string< char_type > string_type;  // String type.

  // construct/copy/destruct
  filter_factory() = default;
  filter_factory(filter_factory const &) = delete;
  filter_factory & operator=(filter_factory const &) = delete;
  ~filter_factory();

  // public member functions
  virtual filter on_exists_test(attribute_name const &);
  virtual filter
  on_equality_relation(attribute_name const &, string_type const &);
  virtual filter
  on_inequality_relation(attribute_name const &, string_type const &);
  virtual filter on_less_relation(attribute_name const &, string_type const &);
  virtual filter
  on_greater_relation(attribute_name const &, string_type const &);
  virtual filter
  on_less_or_equal_relation(attribute_name const &, string_type const &);
  virtual filter
  on_greater_or_equal_relation(attribute_name const &, string_type const &);
  virtual filter
  on_custom_relation(attribute_name const &, string_type const &,
                     string_type const &);
};
```

Description

The interface class for all filter factories.

`filter_factory` public construct/copy/destruct

1. ```
   filter_factory() = default;
   ```

   Default constructor

2. ```
   filter_factory(filter_factory const &) = delete;
   ```

3. ```
   filter_factory & operator=(filter_factory const &) = delete;
   ```

4. ```
   ~filter_factory();
   ```

   Virtual destructor

`filter_factory` public member functions

1. ```
   virtual filter on_exists_test(attribute_name const & name);
   ```
   ```

The callback for filter for the attribute existence test

2.
```
virtual filter
on_equality_relation(attribute_name const & name, string_type const & arg);
```

The callback for equality relation filter

3.
```
virtual filter
on_inequality_relation(attribute_name const & name, string_type const & arg);
```

The callback for inequality relation filter

4.
```
virtual filter
on_less_relation(attribute_name const & name, string_type const & arg);
```

The callback for less relation filter

5.
```
virtual filter
on_greater_relation(attribute_name const & name, string_type const & arg);
```

The callback for greater relation filter

6.
```
virtual filter
on_less_or_equal_relation(attribute_name const & name,
                          string_type const & arg);
```

The callback for less or equal relation filter

7.
```
virtual filter
on_greater_or_equal_relation(attribute_name const & name,
                             string_type const & arg);
```

The callback for greater or equal relation filter

8.
```
virtual filter
on_custom_relation(attribute_name const & name, string_type const & rel,
                   string_type const & arg);
```

The callback for custom relation filter

Function template register_filter_factory

boost::log::register_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename CharT>
  void register_filter_factory(attribute_name const & name,
                               shared_ptr< filter_factory< CharT > > const & factory);
```

Description

The function registers a filter factory object for the specified attribute name. The factory will be used to construct a filter during parsing the filter string.

Parameters: `factory` The filter factory
`name` Attribute name to associate the factory with
Requires: `name != NULL && factory != NULL`, `name` points to a zero-terminated string

Function template register_filter_factory

boost::log::register_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename FactoryT>
  enable_if< is_base_and_derived< filter_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
  register_filter_factory(attribute_name const & name,
                          shared_ptr< FactoryT > const & factory);
```

Description

The function registers a filter factory object for the specified attribute name. The factory will be used to construct a filter during parsing the filter string.

Parameters: `factory` The filter factory
`name` Attribute name to associate the factory with
Requires: `name != NULL && factory != NULL`, `name` points to a zero-terminated string

Function template register_simple_filter_factory

boost::log::register_simple_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename AttributeValueT, typename CharT>
  void register_simple_filter_factory(attribute_name const & name);
```

Description

The function registers a simple filter factory object for the specified attribute name. The factory will support attribute values of type `AttributeValueT`, which must support all relation operations, such as equality comparison and less/greater ordering, and also extraction from stream.

Parameters: `name` Attribute name to associate the factory with
Requires: `name != NULL`, `name` points to a zero-terminated string

Function template register_simple_filter_factory

boost::log::register_simple_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename AttributeValueT>
  void register_simple_filter_factory(attribute_name const & name);
```

Description

The function registers a simple filter factory object for the specified attribute name. The factory will support attribute values of type `AttributeValueT`, which must support all relation operations, such as equality comparison and less/greater ordering, and also extraction from stream.

Parameters: `name` Attribute name to associate the factory with
Requires: `name != NULL`, `name` points to a zero-terminated string

Function template register_simple_filter_factory

boost::log::register_simple_filter_factory

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename CharT, typename DescriptorT,
         template< typename > class ActorT>
  void register_simple_filter_factory(expressions::attrib↵
ute_keyword< DescriptorT, ActorT > const & keyword);
```

Description

The function registers a simple filter factory object for the specified attribute keyword. The factory will support attribute values described by the keyword. The values must support all relation operations, such as equality comparison and less/greater ordering, and also extraction from stream.

Parameters: `keyword` Attribute keyword to associate the factory with
Requires: `name != NULL`, `name` points to a zero-terminated string

Function template parse_filter

boost::log::parse_filter

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename CharT>
  filter parse_filter(const CharT * begin, const CharT * end);
```

Description

The function parses a filter from the sequence of characters

Throws: An `std::exception`-based exception, if a filter cannot be recognized in the character sequence.

Parameters:	`begin`	Pointer to the first character of the sequence
	`end`	Pointer to the after-the-last character of the sequence
Requires:	`begin <= end`, both pointers must not be `NULL`	
Returns:	A function object that can be used as a filter.	

Function template parse_filter

boost::log::parse_filter

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT>
  filter parse_filter(std::basic_string< CharT, TraitsT, AllocatorT > const & str);
```

Description

The function parses a filter from the string

Throws: An `std::exception`-based exception, if a filter cannot be recognized in the character sequence.

Parameters:	`str`	A string that contains filter description
Returns:	A function object that can be used as a filter.	

Function template parse_filter

boost::log::parse_filter

Synopsis

```
// In header: <boost/log/utility/setup/filter_parser.hpp>


template<typename CharT> filter parse_filter(const CharT * str);
```

Description

The function parses a filter from the string

Throws: An `std::exception`-based exception, if a filter cannot be recognized in the character sequence.

Parameters:	`str`	A string that contains filter description.
Requires:	`str != NULL`, `str` points to a zero-terminated string.	
Returns:	A function object that can be used as a filter.	

Header <boost/log/utility/setup/formatter_parser.hpp>

Andrey Semashev

07.04.2008
The header contains definition of a formatter parser function, along with facilities to add support for custom formatters.

```
namespace boost {
  namespace log {
    template<typename CharT, typename AttributeValueT>
      class basic_formatter_factory;

    template<typename CharT> struct formatter_factory;
    template<typename CharT>
      void register_formatter_factory(attribute_name const &,
                                      shared_ptr< formatter_factory< CharT > > const &);
    template<typename FactoryT>
      enable_if< is_base_and_derived< formatter_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
      register_formatter_factory(attribute_name const &,
                                 shared_ptr< FactoryT > const &);
    template<typename AttributeValueT, typename CharT>
      void register_simple_formatter_factory(attribute_name const &);
    template<typename CharT>
      basic_formatter< CharT > parse_formatter(const CharT *, const CharT *);
    template<typename CharT, typename TraitsT, typename AllocatorT>
      basic_formatter< CharT >
      parse_formatter(std::basic_string< CharT, TraitsT, AllocatorT > const &);
    template<typename CharT>
      basic_formatter< CharT > parse_formatter(const CharT *);
  }
}
```

Class template basic_formatter_factory

boost::log::basic_formatter_factory

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>

template<typename CharT, typename AttributeValueT>
class basic_formatter_factory : public boost::log::formatter_factory< CharT > {
public:
  // types
  typedef AttributeValueT           value_type;      // Attribute value type.
  typedef base_type::formatter_type formatter_type;
  typedef base_type::args_map       args_map;

  // public member functions
  virtual formatter_type
  create_formatter(attribute_name const &, args_map const &);
};
```

Description

Base class for formatter factories. This class provides default implementation of formatter expressions for types supporting stream output. The factory does not take into account any additional parameters that may be specified.

`basic_formatter_factory` public member functions

1.
```
virtual formatter_type
create_formatter(attribute_name const & name, args_map const & args);
```

The function creates a formatter for the specified attribute.

Parameters: `args` Formatter arguments
`name` Attribute name

Struct template formatter_factory

boost::log::formatter_factory

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>

template<typename CharT>
struct formatter_factory {
  // types
  typedef CharT                                 char_type;       // Character type.
  typedef std::basic_string< char_type >        string_type;     // String type.
  typedef basic_formatter< char_type >        formatter_type;  // The formatter function object.
  typedef std::map< string_type, string_type > args_map;

  // construct/copy/destruct
  formatter_factory() = default;
  formatter_factory(formatter_factory const &) = delete;
  formatter_factory & operator=(formatter_factory const &) = delete;
  ~formatter_factory();

  // public member functions
  virtual formatter_type
  create_formatter(attribute_name const &, args_map const &) = 0;
};
```

Description

Formatter factory base interface.

`formatter_factory` public types

1. typedef std::map< string_type, string_type > args_map;

   Type of the map of formatter factory arguments [argument name -> argument value]. This type of maps will be passed to formatter factories on attempt to create a formatter.

`formatter_factory` public construct/copy/destruct

1. ```
   formatter_factory() = default;
   ```

   Default constructor

2. ```
   formatter_factory(formatter_factory const &) = delete;
   ```

3. ```
   formatter_factory & operator=(formatter_factory const &) = delete;
   ```

4. ```
   ~formatter_factory();
   ```

Virtual destructor

`formatter_factory` public member functions

1.
```
virtual formatter_type
create_formatter(attribute_name const & name, args_map const & args) = 0;
```

The function creates a formatter for the specified attribute.

Parameters:	`args`	Formatter arguments
	`name`	Attribute name

Function template register_formatter_factory

boost::log::register_formatter_factory — The function registers a user-defined formatter factory.

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename CharT>
  void register_formatter_factory(attribute_name const & attr_name,
                                  shared_ptr< formatter_factory< CharT > > const & factory);
```

Description

The function registers a user-defined formatter factory. The registered factory function will be called when the formatter parser detects the specified attribute name in the formatter string.

Parameters:	`attr_name`	Attribute name
	`factory`	Pointer to the formatter factory
Requires:	`!!attr_name && !!factory`.	

Function template register_formatter_factory

boost::log::register_formatter_factory — The function registers a user-defined formatter factory.

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename FactoryT>
  enable_if< is_base_and_derived< formatter_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
  register_formatter_factory(attribute_name const & attr_name,
                             shared_ptr< FactoryT > const & factory);
```

Description

The function registers a user-defined formatter factory. The registered factory function will be called when the formatter parser detects the specified attribute name in the formatter string.

Parameters:	`attr_name`	Attribute name
	`factory`	Pointer to the formatter factory

Requires: `!!attr_name && !!factory`.

Function template register_simple_formatter_factory

boost::log::register_simple_formatter_factory — The function registers a simple formatter factory.

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename AttributeValueT, typename CharT>
  void register_simple_formatter_factory(attribute_name const & attr_name);
```

Description

The function registers a simple formatter factory. The registered factory will generate formatters that will be equivalent to the `log::expressions::attr` formatter (i.e. that will use the native `operator<<` to format the attribute value). The factory does not use any arguments from the format string, if specified.

Parameters: `attr_name` Attribute name
Requires: `!!attr_name`.

Function template parse_formatter

boost::log::parse_formatter

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename CharT>
  basic_formatter< CharT >
  parse_formatter(const CharT * begin, const CharT * end);
```

Description

The function parses a formatter from the sequence of characters

Throws: An `std::exception`-based exception, if a formatter cannot be recognized in the character sequence.

Parameters: `begin` Pointer to the first character of the sequence
`end` Pointer to the after-the-last character of the sequence
Requires: `begin <= end`, both pointers must not be NULL
Returns: The parsed formatter.

Function template parse_formatter

boost::log::parse_formatter

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT>
  basic_formatter< CharT >
  parse_formatter(std::basic_string< CharT, TraitsT, AllocatorT > const & str);
```

Description

The function parses a formatter from the string

Throws: An `std::exception`-based exception, if a formatter cannot be recognized in the character sequence.

Parameters: `str` A string that contains format description
Returns: The parsed formatter.

Function template parse_formatter

boost::log::parse_formatter

Synopsis

```
// In header: <boost/log/utility/setup/formatter_parser.hpp>


template<typename CharT>
  basic_formatter< CharT > parse_formatter(const CharT * str);
```

Description

The function parses a formatter from the string

Throws: An `std::exception`-based exception, if a formatter cannot be recognized in the character sequence.

Parameters: `str` A string that contains format description.
Requires: `str != NULL`, `str` points to a zero-terminated string
Returns: The parsed formatter.

Header <boost/log/utility/setup/from_settings.hpp>

Andrey Semashev

11.10.2009
The header contains definition of facilities that allows to initialize the library from settings.

```
namespace boost {
  namespace log {
    template<typename CharT> struct sink_factory;
    template<typename CharT>
      void init_from_settings(basic_settings_section< CharT > const &);
    template<typename CharT>
      void register_sink_factory(const char *,
                                 shared_ptr< sink_factory< CharT > > const &);
    template<typename CharT>
      void register_sink_factory(std::string const &,
                                 shared_ptr< sink_factory< CharT > > const &);
    template<typename FactoryT>
      enable_if< is_base_and_derived< sink_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
      register_sink_factory(const char *, shared_ptr< FactoryT > const &);
    template<typename FactoryT>
      enable_if< is_base_and_derived< sink_factory< typename FactoryT::char_type >, Fact↵
oryT >>::type
      register_sink_factory(std::string const &,
                            shared_ptr< FactoryT > const &);
  }
}
```

Struct template sink_factory

boost::log::sink_factory

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>

template<typename CharT>
struct sink_factory {
  // types
  typedef CharT                                 char_type;         // Character type.
  typedef std::basic_string< char_type >        string_type;       // String type.
  typedef basic_settings_section< char_type > settings_section;  // Settings section type.

  // construct/copy/destruct
  sink_factory() = default;
  sink_factory(sink_factory const &) = delete;
  sink_factory & operator=(sink_factory const &) = delete;
  ~sink_factory();

  // public member functions
  virtual shared_ptr< sinks::sink > create_sink(settings_section const &) = 0;
};
```

Description

Sink factory base interface

`sink_factory` public construct/copy/destruct

1. ```
   sink_factory() = default;
   ```

   Default constructor
   ```

2. ```
   sink_factory(sink_factory const &) = delete;
   ```

3. ```
   sink_factory & operator=(sink_factory const &) = delete;
   ```

4. ```
   ~sink_factory();
   ```

   Virtual destructor

`sink_factory` public member functions

1. ```
   virtual shared_ptr< sinks::sink >
   create_sink(settings_section const & settings) = 0;
   ```

   The function creates a formatter for the specified attribute.

   Parameters: `settings` Sink parameters

Function template init_from_settings

boost::log::init_from_settings

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>


template<typename CharT>
  void init_from_settings(basic_settings_section< CharT > const & setts);
```

Description

The function initializes the logging library from a settings container

Throws: An `std::exception`-based exception if the provided settings are not valid.

Parameters: `setts` Library settings container

Function template register_sink_factory

boost::log::register_sink_factory — The function registers a factory for a custom sink.

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>


template<typename CharT>
  void register_sink_factory(const char * sink_name,
                             shared_ptr< sink_factory< CharT > > const & factory);
```

Description

The function registers a factory for a sink. The factory will be called to create sink instance when the parser discovers the specified sink type in the settings file. The factory must accept a map of parameters [parameter name -> parameter value] that it may use to initialize the sink. The factory must return a non-NULL pointer to the constructed sink instance.

Parameters:	`factory`	Pointer to the custom sink factory. Must not be NULL.
	`sink_name`	The custom sink name. Must point to a zero-terminated sequence of characters, must not be NULL.

Function template register_sink_factory

boost::log::register_sink_factory — The function registers a factory for a custom sink.

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>


template<typename CharT>
  void register_sink_factory(std::string const & sink_name,
                             shared_ptr< sink_factory< CharT > > const & factory);
```

Description

The function registers a factory for a sink. The factory will be called to create sink instance when the parser discovers the specified sink type in the settings file. The factory must accept a map of parameters [parameter name -> parameter value] that it may use to initialize the sink. The factory must return a non-NULL pointer to the constructed sink instance.

Parameters:	`factory`	Pointer to the custom sink factory. Must not be NULL.
	`sink_name`	The custom sink name

Function template register_sink_factory

boost::log::register_sink_factory — The function registers a factory for a custom sink.

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>


template<typename FactoryT>
  enable_if< is_base_and_derived< sink_factory< typename FactoryT::char_type >, FactoryT >>::type
  register_sink_factory(const char * sink_name,
                        shared_ptr< FactoryT > const & factory);
```

Description

The function registers a factory for a sink. The factory will be called to create sink instance when the parser discovers the specified sink type in the settings file. The factory must accept a map of parameters [parameter name -> parameter value] that it may use to initialize the sink. The factory must return a non-NULL pointer to the constructed sink instance.

Parameters:	`factory`	Pointer to the custom sink factory. Must not be NULL.
	`sink_name`	The custom sink name. Must point to a zero-terminated sequence of characters, must not be NULL.

Function template register_sink_factory

boost::log::register_sink_factory — The function registers a factory for a custom sink.

Synopsis

```
// In header: <boost/log/utility/setup/from_settings.hpp>


template<typename FactoryT>
  enable_if< is_base_and_derived< sink_factory< typename FactoryT::char_type >, FactoryT >>::type
   register_sink_factory(std::string const & sink_name,
                         shared_ptr< FactoryT > const & factory);
```

Description

The function registers a factory for a sink. The factory will be called to create sink instance when the parser discovers the specified sink type in the settings file. The factory must accept a map of parameters [parameter name -> parameter value] that it may use to initialize the sink. The factory must return a non-NULL pointer to the constructed sink instance.

Parameters:	`factory`	Pointer to the custom sink factory. Must not be NULL.
	`sink_name`	The custom sink name

Header <boost/log/utility/setup/from_stream.hpp>

Andrey Semashev

22.03.2008
The header contains definition of facilities that allows to initialize the library from a settings file.

```
namespace boost {
  namespace log {
    template<typename CharT>
      void init_from_stream(std::basic_istream< CharT > &);
  }
}
```

Function template init_from_stream

boost::log::init_from_stream

Synopsis

```
// In header: <boost/log/utility/setup/from_stream.hpp>


template<typename CharT>
  void init_from_stream(std::basic_istream< CharT > & strm);
```

Description

The function initializes the logging library from a stream containing logging settings

Throws: An `std::exception`-based exception if the read data cannot be interpreted as the library settings

Parameters:	`strm`	Stream, that provides library settings

Header <boost/log/utility/setup/settings.hpp>

Andrey Semashev

11.10.2009
The header contains definition of the library settings container.

```
namespace boost {
  namespace log {
    template<typename CharT> class basic_settings;
    template<typename CharT> class basic_settings_section;

    typedef basic_settings< char > settings;  // Convenience typedef for narrow-character logging.
    typedef basic_settings_section< char > settings_section;  // Convenience typedef for narrow-
character logging.
    typedef basic_settings< wchar_t > wsettings;  // Convenience typedef for wide-character log↵
ging.
    typedef basic_settings_section< wchar_t > wsettings_section;  // Convenience typedef for wide-
character logging.
    template<typename CharT>
      void swap(basic_settings_section< CharT > &,
                basic_settings_section< CharT > &);
  }
}
```

Class template basic_settings

boost::log::basic_settings — The class represents settings container.

Synopsis

```
// In header: <boost/log/utility/setup/settings.hpp>

template<typename CharT>
class basic_settings : public boost::log::basic_settings_section< CharT > {
public:
  // types
  typedef basic_settings_section< CharT > section;             // Section type.
  typedef section::property_tree_type     property_tree_type;  // Property tree type.

  // construct/copy/destruct
  basic_settings();
  basic_settings(basic_settings const &);
  basic_settings(this_type &&);
  explicit basic_settings(property_tree_type const &);
  basic_settings & operator=(basic_settings const &);
  basic_settings & operator=(basic_settings &&);
  ~basic_settings();
};
```

Description

All settings are presented as a number of named parameters divided into named sections. The parameters values are stored as strings. Individual parameters may be queried via subscript operators, like this:

```
<preformatted> optional< string > param = settings["Section1"]["Param1"]; // reads parameter
"Param1" in section "Section1" // returns an empty value if no such parameter exists settings["Sec-
tion2"]["Param2"] = 10; // sets the parameter "Param2" in section "Section2" // to value "10"
</preformatted>
```

There are also other methods to work with parameters.

`basic_settings` public construct/copy/destruct

1. ```
   basic_settings();
   ```

   Default constructor. Creates an empty settings container.

2. ```
   basic_settings(basic_settings const & that);
   ```

   Copy constructor.

3. ```
   basic_settings(this_type && that);
   ```

   Move constructor.

4. ```
   explicit basic_settings(property_tree_type const & tree);
   ```

   Initializing constructor. Creates a settings container with the copy of the specified property tree.

5. ```
   basic_settings & operator=(basic_settings const & that);
   ```

   Copy assignment operator.

6. ```
   basic_settings & operator=(basic_settings && that);
   ```

   Move assignment operator.

7. ```
   ~basic_settings();
   ```

   Destructor

Class template basic_settings_section

boost::log::basic_settings_section — The class represents a reference to the settings container section.
   ```

Synopsis

```
// In header: <boost/log/utility/setup/settings.hpp>

template<typename CharT>
class basic_settings_section {
public:
  // types
  typedef CharT                                                 char_type;           // Charac↵
ter type.
  typedef std::basic_string< char_type >                       string_type;         // String ↵
type.
  typedef property_tree::basic_ptree< std::string, string_type > property_tree_type;  // Prop↵
erty tree type.
  typedef property_tree_type::path_type                         path_type;           // Prop↵
erty tree path type.
  typedef implementation_defined                                const_reference;
  typedef implementation_defined                                reference;
  typedef implementation_defined                                const_iterator;
  typedef implementation_defined                                iterator;

  // construct/copy/destruct
  basic_settings_section();
  basic_settings_section(basic_settings_section const &);
  explicit basic_settings_section(property_tree_type *);

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  bool empty() const;
  reference operator[](std::string const &);
  const_reference operator[](std::string const &) const;
  reference operator[](const char *);
  const_reference operator[](const char *) const;
  property_tree_type const & property_tree() const;
  property_tree_type & property_tree();
  bool has_section(string_type const &) const;
  bool has_parameter(string_type const &, string_type const &) const;
  void swap(basic_settings_section &);
};
```

Description

The section refers to a sub-tree of the library settings container. It does not own the referred sub-tree but allows for convenient access to parameters within the subsection.

`basic_settings_section` public types

1. typedef implementation_defined const_reference;

   Constant reference to the parameter value

2. typedef implementation_defined reference;

Mutable reference to the parameter value

3. typedef implementation_defined const_iterator;

   Constant iterator over nested parameters and subsections

4. typedef implementation_defined iterator;

   Mutable iterator over nested parameters and subsections

`basic_settings_section` public construct/copy/destruct

1. ```
   basic_settings_section();
   ```

   Default constructor. Creates an empty settings container.

2. ```
   basic_settings_section(basic_settings_section const & that);
   ```

   Copy constructor.

3. ```
   explicit basic_settings_section(property_tree_type * tree);
   ```

`basic_settings_section` public member functions

1. ```
   explicit operator bool() const noexcept;
   ```

   Checks if the section refers to the container.

2. ```
   bool operator!() const noexcept;
   ```

   Checks if the section refers to the container.

3. ```
   iterator begin();
   ```

   Returns an iterator over the nested subsections and parameters.

4. ```
   iterator end();
   ```

   Returns an iterator over the nested subsections and parameters.

5. ```
   const_iterator begin() const;
   ```

   Returns an iterator over the nested subsections and parameters.

6. ```
   const_iterator end() const;
   ```

   Returns an iterator over the nested subsections and parameters.

7. ```
   reverse_iterator rbegin();
   ```

   Returns a reverse iterator over the nested subsections and parameters.

8.
```
reverse_iterator rend();
```

Returns a reverse iterator over the nested subsections and parameters.

9.
```
const_reverse_iterator rbegin() const;
```

Returns a reverse iterator over the nested subsections and parameters.

10.
```
const_reverse_iterator rend() const;
```

Returns a reverse iterator over the nested subsections and parameters.

11.
```
bool empty() const;
```

Checks if the container is empty (i.e. contains no sections and parameters).

12.
```
reference operator[](std::string const & section_name);
```

Accessor to a single parameter. This operator should be used in conjunction with the subsequent subscript operator that designates the parameter name.

Parameters: `section_name` The name of the section in which the parameter resides
Returns: An unspecified reference type that can be used for parameter name specifying

13.
```
const_reference operator[](std::string const & section_name) const;
```

Accessor to a single parameter. This operator should be used in conjunction with the subsequent subscript operator that designates the parameter name.

Parameters: `section_name` The name of the section in which the parameter resides
Returns: An unspecified reference type that can be used for parameter name specifying

14.
```
reference operator[](const char * section_name);
```

Accessor to a single parameter. This operator should be used in conjunction with the subsequent subscript operator that designates the parameter name.

Parameters: `section_name` The name of the section in which the parameter resides
Returns: An unspecified reference type that can be used for parameter name specifying

15.
```
const_reference operator[](const char * section_name) const;
```

Accessor to a single parameter. This operator should be used in conjunction with the subsequent subscript operator that designates the parameter name.

Parameters: `section_name` The name of the section in which the parameter resides
Returns: An unspecified reference type that can be used for parameter name specifying

16.
```
property_tree_type const & property_tree() const;
```

Accessor for the embedded property tree

17.
```
property_tree_type & property_tree();
```

Accessor for the embedded property tree

18.
```
bool has_section(string_type const & section_name) const;
```

Checks if the specified section is present in the container.

Parameters: `section_name` The name of the section

19.
```
bool has_parameter(string_type const & section_name,
                   string_type const & param_name) const;
```

Checks if the specified parameter is present in the container.

Parameters: `param_name` The name of the parameter
`section_name` The name of the section in which the parameter resides

20.
```
void swap(basic_settings_section & that);
```

Swaps two references to settings sections.

Function template swap

boost::log::swap

Synopsis

```
// In header: <boost/log/utility/setup/settings.hpp>


template<typename CharT>
  void swap(basic_settings_section< CharT > & left,
            basic_settings_section< CharT > & right);
```

Header <boost/log/utility/setup/settings_parser.hpp>

Andrey Semashev

20.07.2012
The header contains definition of a settings parser function.

```
namespace boost {
  namespace log {
    template<typename CharT>
      basic_settings< CharT > parse_settings(std::basic_istream< CharT > &);
  }
}
```

Function template parse_settings

boost::log::parse_settings

Synopsis

```
// In header: <boost/log/utility/setup/settings_parser.hpp>


template<typename CharT>
  basic_settings< CharT > parse_settings(std::basic_istream< CharT > & strm);
```

Description

The function parses library settings from an input stream

Throws: An `std::exception`-based exception if the read data cannot be interpreted as the library settings

Parameters: `strm` Stream, that provides library settings

Header <boost/log/utility/strictest_lock.hpp>

Andrey Semashev

30.05.2010
The header contains definition of the `strictest_lock` metafunction that allows to select a lock with the strictest access requirements.

```
namespace boost {
  namespace log {
    template<typename... LocksT> struct strictest_lock;
    template<typename LockT> struct thread_access_mode_of;

    template<typename MutexT>
      struct thread_access_mode_of<boost::log::aux::exclusive_lock_guard< MutexT >>;
    template<typename MutexT>
      struct thread_access_mode_of<boost::log::aux::shared_lock_guard< MutexT >>;
    template<typename MutexT>
      struct thread_access_mode_of<lock_guard< MutexT >>;
    template<typename MutexT> struct thread_access_mode_of<no_lock< MutexT >>;
    template<typename MutexT>
      struct thread_access_mode_of<shared_lock< MutexT >>;
    template<typename MutexT>
      struct thread_access_mode_of<unique_lock< MutexT >>;
    template<typename MutexT>
      struct thread_access_mode_of<upgrade_lock< MutexT >>;

    // Access modes for different types of locks.
    enum lock_access_mode { unlocked_access, shared_access, exclusive_access };
  }
}
```

Struct template strictest_lock

boost::log::strictest_lock — The metafunction selects the most strict lock type of the specified.

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename... LocksT>
struct strictest_lock {
  // types
  typedef implementation_defined type;
};
```

Description

The template supports all lock types provided by the Boost.Thread library (except for `upgrade_to_unique_lock`), plus additional pseudo-lock `no_lock` that indicates no locking at all. Exclusive locks are considered the strictest, shared locks are weaker, and `no_lock` is the weakest.

Struct template thread_access_mode_of

boost::log::thread_access_mode_of — The trait allows to select an access mode by the lock type.

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename LockT>
struct thread_access_mode_of {
};
```

Struct template thread_access_mode_of<boost::log::aux::exclusive_lock_guard< MutexT >>

boost::log::thread_access_mode_of<boost::log::aux::exclusive_lock_guard< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<boost::log::aux::exclusive_lock_guard< MutexT >> :
  public mpl::integral_c< lock_access_mode, exclusive_access >
{
};
```

Struct template thread_access_mode_of<boost::log::aux::shared_lock_guard< MutexT >>

boost::log::thread_access_mode_of<boost::log::aux::shared_lock_guard< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<boost::log::aux::shared_lock_guard< MutexT >> :
  public mpl::integral_c< lock_access_mode, shared_access >
{
};
```

Struct template thread_access_mode_of<lock_guard< MutexT >>

boost::log::thread_access_mode_of<lock_guard< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<lock_guard< MutexT >> :
  public mpl::integral_c< lock_access_mode, exclusive_access >
{
};
```

Struct template thread_access_mode_of<no_lock< MutexT >>

boost::log::thread_access_mode_of<no_lock< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<no_lock< MutexT >> :
  public mpl::integral_c< lock_access_mode, unlocked_access >
{
};
```

Struct template thread_access_mode_of<shared_lock< MutexT >>

boost::log::thread_access_mode_of<shared_lock< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<shared_lock< MutexT >> :
  public mpl::integral_c< lock_access_mode, shared_access >
{
};
```

Struct template thread_access_mode_of<unique_lock< MutexT >>

boost::log::thread_access_mode_of<unique_lock< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<unique_lock< MutexT >> :
  public mpl::integral_c< lock_access_mode, exclusive_access >
{
};
```

Struct template thread_access_mode_of<upgrade_lock< MutexT >>

boost::log::thread_access_mode_of<upgrade_lock< MutexT >>

Synopsis

```
// In header: <boost/log/utility/strictest_lock.hpp>

template<typename MutexT>
struct thread_access_mode_of<upgrade_lock< MutexT >> :
  public mpl::integral_c< lock_access_mode, shared_access >
{
};
```

Header <boost/log/utility/string_literal.hpp>

Andrey Semashev

24.06.2007
The header contains implementation of a constant string literal wrapper.

```
namespace boost {
  namespace log {
    template<typename CharT, typename TraitsT> class basic_string_literal;
    template<typename CharT, typename StrmTraitsT, typename LitTraitsT>
      std::basic_ostream< CharT, StrmTraitsT > &
      operator<<(std::basic_ostream< CharT, StrmTraitsT > &,
                 basic_string_literal< CharT, LitTraitsT > const &);
    template<typename CharT, typename TraitsT>
      void swap(basic_string_literal< CharT, TraitsT > &,
                basic_string_literal< CharT, TraitsT > &);
    template<typename T, std::size_t LenV>
      basic_string_literal< T > str_literal(T(&));
  }
}
```

Class template basic_string_literal

boost::log::basic_string_literal — String literal wrapper.

Synopsis

```
// In header: <boost/log/utility/string_literal.hpp>

template<typename CharT, typename TraitsT>
class basic_string_literal {
public:
  // types
  typedef CharT                                          value_type;
  typedef TraitsT                                        traits_type;
  typedef std::size_t                                    size_type;
  typedef std::ptrdiff_t                                 difference_type;
  typedef const value_type *                             const_pointer;
  typedef value_type const &                             const_reference;
  typedef const value_type *                             const_iterator;
  typedef std::reverse_iterator< const_iterator >        const_reverse_iterator;
  typedef std::basic_string< value_type, traits_type > string_type;              // Correspond↵
ing STL string type.

  // construct/copy/destruct
  basic_string_literal() noexcept;
  template<typename T, size_type LenV> basic_string_literal(T(&)) noexcept;
  basic_string_literal(basic_string_literal const &) noexcept;
  this_type & operator=(this_type const &) noexcept;
  template<typename T, size_type LenV> this_type & operator=(T(&)) noexcept;

  // public member functions
  bool operator==(this_type const &) const noexcept;
  bool operator==(const_pointer) const noexcept;
  bool operator==(string_type const &) const;
  bool operator<(this_type const &) const noexcept;
  bool operator<(const_pointer) const noexcept;
  bool operator<(string_type const &) const;
  bool operator>(this_type const &) const noexcept;
  bool operator>(const_pointer) const noexcept;
  bool operator>(string_type const &) const;
  const_reference operator[](size_type) const noexcept;
  const_reference at(size_type) const;
  const_pointer c_str() const noexcept;
  const_pointer data() const noexcept;
  size_type size() const noexcept;
  size_type length() const noexcept;
  bool empty() const noexcept;
  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator rend() const noexcept;
  string_type str() const;
  void clear() noexcept;
  void swap(this_type &) noexcept;
  this_type & assign(this_type const &) noexcept;
  template<typename T, size_type LenV> this_type & assign(T(&)) noexcept;
  size_type copy(value_type *, size_type, size_type = 0) const;
  int compare(size_type, size_type, const_pointer, size_type) const;
  int compare(size_type, size_type, const_pointer) const noexcept;
  int compare(size_type, size_type, this_type const &) const noexcept;
  int compare(const_pointer, size_type) const noexcept;
  int compare(const_pointer) const noexcept;
  int compare(this_type const &) const noexcept;
};
```

Description

The `basic_string_literal` is a thin wrapper around a constant string literal. It provides interface similar to STL strings, but because of read-only nature of string literals, lacks ability to modify string contents. However, `basic_string_literal` objects can be assigned to and cleared.

The main advantage of this class comparing to other string classes is that it doesn't dynamically allocate memory and therefore is fast, thin and exception safe.

`basic_string_literal` public construct/copy/destruct

1. ```
   basic_string_literal() noexcept;
   ```

   Constructor

   Postconditions: `empty() == true`

2. ```
   template<typename T, size_type LenV> basic_string_literal(T(&) p) noexcept;
   ```

   Constructor from a string literal

   Parameters: `p` A zero-terminated constant sequence of characters
   Postconditions: `*this == p`

3. ```
   basic_string_literal(basic_string_literal const & that) noexcept;
   ```

   Copy constructor

   Parameters: `that` Source literal to copy string from
   Postconditions: `*this == that`

4. ```
   this_type & operator=(this_type const & that) noexcept;
   ```

   Assignment operator

   Parameters: `that` Source literal to copy string from
   Postconditions: `*this == that`

5. ```
   template<typename T, size_type LenV> this_type & operator=(T(&) p) noexcept;
   ```

   Assignment from a string literal

   Parameters: `p` A zero-terminated constant sequence of characters
   Postconditions: `*this == p`

`basic_string_literal` public member functions

1. ```
   bool operator==(this_type const & that) const noexcept;
   ```

   Lexicographical comparison (equality)

   Parameters: `that` Comparand
   Returns: `true` if the comparand string equals to this string, `false` otherwise

2. ```
   bool operator==(const_pointer str) const noexcept;
   ```
   ```

Lexicographical comparison (equality)

Parameters: `str` Comparand. Must point to a zero-terminated sequence of characters, must not be NULL.
Returns: `true` if the comparand string equals to this string, `false` otherwise

3.
```
bool operator==(string_type const & that) const;
```

Lexicographical comparison (equality)

Parameters: `that` Comparand
Returns: `true` if the comparand string equals to this string, `false` otherwise

4.
```
bool operator<(this_type const & that) const noexcept;
```

Lexicographical comparison (less ordering)

Parameters: `that` Comparand
Returns: `true` if this string is less than the comparand, `false` otherwise

5.
```
bool operator<(const_pointer str) const noexcept;
```

Lexicographical comparison (less ordering)

Parameters: `str` Comparand. Must point to a zero-terminated sequence of characters, must not be NULL.
Returns: `true` if this string is less than the comparand, `false` otherwise

6.
```
bool operator<(string_type const & that) const;
```

Lexicographical comparison (less ordering)

Parameters: `that` Comparand
Returns: `true` if this string is less than the comparand, `false` otherwise

7.
```
bool operator>(this_type const & that) const noexcept;
```

Lexicographical comparison (greater ordering)

Parameters: `that` Comparand
Returns: `true` if this string is greater than the comparand, `false` otherwise

8.
```
bool operator>(const_pointer str) const noexcept;
```

Lexicographical comparison (greater ordering)

Parameters: `str` Comparand. Must point to a zero-terminated sequence of characters, must not be NULL.
Returns: `true` if this string is greater than the comparand, `false` otherwise

9.
```
bool operator>(string_type const & that) const;
```

Lexicographical comparison (greater ordering)

Parameters: `that` Comparand
Returns: `true` if this string is greater than the comparand, `false` otherwise

10.
```
const_reference operator[](size_type i) const noexcept;
```

Subscript operator

Parameters: `i` Requested character index
Requires: `i < size()`
Returns: Constant reference to the requested character

11.
```
const_reference at(size_type i) const;
```

Checked subscript

Throws: An `std::exception`-based exception if index *i* is out of string boundaries
Parameters: `i` Requested character index
Returns: Constant reference to the requested character

12.
```
const_pointer c_str() const noexcept;
```

Returns: Pointer to the beginning of the literal

13.
```
const_pointer data() const noexcept;
```

Returns: Pointer to the beginning of the literal

14.
```
size_type size() const noexcept;
```

Returns: Length of the literal

15.
```
size_type length() const noexcept;
```

Returns: Length of the literal

16.
```
bool empty() const noexcept;
```

Returns: `true` if the literal is an empty string, `false` otherwise

17.
```
const_iterator begin() const noexcept;
```

Returns: Iterator that points to the first character of the literal

18.
```
const_iterator end() const noexcept;
```

Returns: Iterator that points after the last character of the literal

19.
```
const_reverse_iterator rbegin() const noexcept;
```

Returns: Reverse iterator that points to the last character of the literal

20.
```
const_reverse_iterator rend() const noexcept;
```

Returns: Reverse iterator that points before the first character of the literal

21.
```
string_type str() const;
```

Returns: STL string constructed from the literal

22.
```
void clear() noexcept;
```

The method clears the literal

Postconditions: `empty() == true`

23.
```
void swap(this_type & that) noexcept;
```

The method swaps two literals

24.
```
this_type & assign(this_type const & that) noexcept;
```

Assignment from another literal

Parameters: `that` Source literal to copy string from
Postconditions: `*this == that`

25.
```
template<typename T, size_type LenV> this_type & assign(T(&) p) noexcept;
```

Assignment from another literal

Parameters: `p` A zero-terminated constant sequence of characters
Postconditions: `*this == p`

26.
```
size_type copy(value_type * str, size_type n, size_type pos = 0) const;
```

The method copies the literal or its portion to an external buffer

Throws: An `std::exception`-based exception if *pos* is out of range.
Parameters: `n` Maximum number of characters to copy
`pos` Starting position to start copying from
`str` Pointer to the external buffer beginning. Must not be NULL. The buffer must have enough capacity to accommodate the requested number of characters.
Requires: `pos <= size()`
Returns: Number of characters copied

27.
```
int compare(size_type pos, size_type n, const_pointer str, size_type len) const;
```

Lexicographically compares the argument string to a part of this string

Throws: An `std::exception`-based exception if *pos* is out of range.
Parameters: `len` Number of characters in the sequence *str*.
`n` Length of the substring of this string to perform comparison to
`pos` Starting position within this string to perform comparison to
`str` Comparand. Must point to a sequence of characters, must not be NULL.
Requires: `pos <= size()`
Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

28.
```
int compare(size_type pos, size_type n, const_pointer str) const noexcept;
```

Lexicographically compares the argument string to a part of this string

Throws: An `std::exception`-based exception if *pos* is out of range.

Parameters:
- `n` Length of the substring of this string to perform comparison to
- `pos` Starting position within this string to perform comparison to
- `str` Comparand. Must point to a zero-terminated sequence of characters, must not be NULL.

Requires: `pos <= size()`

Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

29.
```
int compare(size_type pos, size_type n, this_type const & that) const noexcept;
```

Lexicographically compares the argument string literal to a part of this string

Throws: An `std::exception`-based exception if *pos* is out of range.

Parameters:
- `n` Length of the substring of this string to perform comparison to
- `pos` Starting position within this string to perform comparison to
- `that` Comparand

Requires: `pos <= size()`

Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

30.
```
int compare(const_pointer str, size_type len) const noexcept;
```

Lexicographically compares the argument string to this string

Parameters:
- `len` Number of characters in the sequence *str*.
- `str` Comparand. Must point to a sequence of characters, must not be NULL.

Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

31.
```
int compare(const_pointer str) const noexcept;
```

Lexicographically compares the argument string to this string

Parameters:
- `str` Comparand. Must point to a zero-terminated sequence of characters, must not be NULL.

Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

32.
```
int compare(this_type const & that) const noexcept;
```

Lexicographically compares the argument string to this string

Parameters:
- `that` Comparand

Returns: Zero if the comparand equals this string, a negative value if this string is less than the comparand, a positive value if this string is greater than the comparand.

Function template operator<<

boost::log::operator<< — Output operator.

Synopsis

```
// In header: <boost/log/utility/string_literal.hpp>


template<typename CharT, typename StrmTraitsT, typename LitTraitsT>
  std::basic_ostream< CharT, StrmTraitsT > &
  operator<<(std::basic_ostream< CharT, StrmTraitsT > & strm,
             basic_string_literal< CharT, LitTraitsT > const & lit);
```

Function template swap

boost::log::swap — External swap.

Synopsis

```
// In header: <boost/log/utility/string_literal.hpp>


template<typename CharT, typename TraitsT>
  void swap(basic_string_literal< CharT, TraitsT > & left,
            basic_string_literal< CharT, TraitsT > & right);
```

Function template str_literal

boost::log::str_literal — Creates a string literal wrapper from a constant string literal.

Synopsis

```
// In header: <boost/log/utility/string_literal.hpp>


template<typename T, std::size_t LenV>
  basic_string_literal< T > str_literal(T(&) p);
```

Header <boost/log/utility/string_literal_fwd.hpp>

Andrey Semashev

24.06.2007
The header contains forward declaration of a constant string literal wrapper.

```
namespace boost {
  namespace log {
    typedef basic_string_literal< char > string_literal;  // String literal type for narrow char↵
acters.
    typedef basic_string_literal< wchar_t > wstring_literal;  // String literal type for wide ↵
characters.
  }
}
```

Header <boost/log/utility/type_dispatch/date_time_types.hpp>

Andrey Semashev

13.03.2008
The header contains definition of date and time-related types supported by the library by default.

```
namespace boost {
  namespace log {
    typedef mpl::vector< std::time_t, std::tm > native_date_time_types;
   typedef mpl::vector< posix_time::ptime, local_time::local_date_time > boost_date_time_types;
    typedef mpl::copy< boost_date_time_types, mpl::back_inserter< nat↵
ive_date_time_types >>::type date_time_types;
    typedef native_date_time_types native_date_types;
    typedef mpl::push_back< boost_date_time_types, gregorian::date >::type boost_date_types;
   typedef mpl::copy< boost_date_types, mpl::back_inserter< native_date_types >>::type date_types;
    typedef native_date_time_types native_time_types;
    typedef boost_date_time_types boost_time_types;  // An MPL-sequence of Boost time types.
    typedef date_time_types time_types;
    typedef mpl::vector< double > native_time_duration_types;
    typedef mpl::vector< posix_time::time_duration, gregorian::date_duration > boost_time_dura↵
tion_types;
    typedef mpl::copy< boost_time_duration_types, mpl::back_inserter< native_time_dura↵
tion_types >>::type time_duration_types;
    typedef mpl::vector< posix_time::time_period, local_time::local_time_period, gregori↵
an::date_period > boost_time_period_types;
    typedef boost_time_period_types time_period_types;
  }
}
```

Type definition native_date_time_types

native_date_time_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::vector< std::time_t, std::tm > native_date_time_types;
```

Description

An MPL-sequence of natively supported date and time types of attributes

Type definition boost_date_time_types

boost_date_time_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::vector< posix_time::ptime, local_time::local_date_time > boost_date_time_types;
```

Description

An MPL-sequence of Boost date and time types of attributes

Type definition date_time_types

date_time_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::copy< boost_date_time_types, mpl::back_inserter< nat↵
ive_date_time_types >>::type date_time_types;
```

Description

An MPL-sequence with the complete list of the supported date and time types

Type definition native_date_types

native_date_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef native_date_time_types native_date_types;
```

Description

An MPL-sequence of natively supported date types of attributes

Type definition boost_date_types

boost_date_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::push_back< boost_date_time_types, gregorian::date >::type boost_date_types;
```

Description

An MPL-sequence of Boost date types of attributes

Type definition date_types

date_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>

typedef mpl::copy< boost_date_types, mpl::back_inserter< native_date_types >>::type date_types;
```

Description

An MPL-sequence with the complete list of the supported date types

Type definition native_time_types

native_time_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>

typedef native_date_time_types native_time_types;
```

Description

An MPL-sequence of natively supported time types

Type definition time_types

time_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>

typedef date_time_types time_types;
```

Description

An MPL-sequence with the complete list of the supported time types

Type definition native_time_duration_types

native_time_duration_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>

typedef mpl::vector< double > native_time_duration_types;
```

Description

An MPL-sequence of natively supported time duration types of attributes

Type definition boost_time_duration_types

boost_time_duration_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::vector< posix_time::time_duration, gregorian::date_duration > boost_time_dura↵
tion_types;
```

Description

An MPL-sequence of Boost time duration types of attributes

Type definition time_duration_types

time_duration_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::copy< boost_time_duration_types, mpl::back_inserter< native_time_dura↵
tion_types >>::type time_duration_types;
```

Description

An MPL-sequence with the complete list of the supported time duration types

Type definition boost_time_period_types

boost_time_period_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef mpl::vector< posix_time::time_period, local_time::local_time_period, gregorian::date_peri↵
od > boost_time_period_types;
```

Description

An MPL-sequence of Boost time duration types of attributes

Type definition time_period_types

time_period_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/date_time_types.hpp>


typedef boost_time_period_types time_period_types;
```

Description

An MPL-sequence with the complete list of the supported time period types

Header <boost/log/utility/type_dispatch/dynamic_type_dispatcher.hpp>

Andrey Semashev

15.04.2007
The header contains implementation of the run-time type dispatcher.

```
namespace boost {
  namespace log {
    class dynamic_type_dispatcher;
  }
}
```

Class dynamic_type_dispatcher

boost::log::dynamic_type_dispatcher — A dynamic type dispatcher.

Synopsis

```
// In header: <boost/log/utility/type_dispatch/dynamic_type_dispatcher.hpp>


class dynamic_type_dispatcher : public type_dispatcher {
public:
  // construct/copy/destruct
  dynamic_type_dispatcher();
  dynamic_type_dispatcher(dynamic_type_dispatcher const &);
  dynamic_type_dispatcher & operator=(dynamic_type_dispatcher const &);

  // public member functions
  template<typename T, typename VisitorT> void register_type(VisitorT const &);
  dispatching_map::size_type registered_types_count() const;
};
```

Description

The type dispatcher can be used to pass objects of arbitrary types from one component to another. With regard to the library, the type dispatcher can be used to extract attribute values.

The dynamic type dispatcher can be initialized in run time and, therefore, can support different types, depending on runtime conditions. Each supported type is associated with a functional object that will be called when an object of the type is dispatched.

`dynamic_type_dispatcher` public construct/copy/destruct

1.
```
dynamic_type_dispatcher();
```

Default constructor

2.
```
dynamic_type_dispatcher(dynamic_type_dispatcher const & that);
```

Copy constructor

3.
```
dynamic_type_dispatcher & operator=(dynamic_type_dispatcher const & that);
```

Copy assignment

`dynamic_type_dispatcher` public member functions

1.
```
template<typename T, typename VisitorT>
  void register_type(VisitorT const & visitor);
```

The method registers a new type

Parameters: `visitor` Function object that will be associated with the type `T`

2.
```
dispatching_map::size_type registered_types_count() const;
```

The method returns the number of registered types

Header <boost/log/utility/type_dispatch/standard_types.hpp>

Andrey Semashev

19.05.2007
The header contains definition of standard types supported by the library by default.

```
namespace boost {
  namespace log {
    typedef mpl::vector< bool, char, wchar_t, signed char, unsigned char, short, un↵
signed short, int, unsigned int, long, unsigned long > integral_types;
    typedef mpl::vector< float, double, long double > floating_point_types;
    typedef mpl::copy< floating_point_types, mpl::back_inserter< integral_types >>::type numer↵
ic_types;
    typedef mpl::vector< std::string, string_literal, std::wstring, wstring_literal > string_types;
    typedef mpl::copy< string_types, mpl::back_inserter< numeric_types >>::type default_attrib↵
ute_types;
  }
}
```

Type definition integral_types

integral_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/standard_types.hpp>


typedef mpl::vector< bool, char, wchar_t, signed char, unsigned char, short, un↵
signed short, int, unsigned int, long, unsigned long > integral_types;
```

Description

An MPL-sequence of integral types of attributes, supported by default

Type definition floating_point_types

floating_point_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/standard_types.hpp>


typedef mpl::vector< float, double, long double > floating_point_types;
```

Description

An MPL-sequence of FP types of attributes, supported by default

Type definition numeric_types

numeric_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/standard_types.hpp>


typedef mpl::copy< floating_point_types, mpl::back_inserter< integral_types >>::type numeric_types;
```

Description

An MPL-sequence of all numeric types of attributes, supported by default

Type definition string_types

string_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/standard_types.hpp>


typedef mpl::vector< std::string, string_literal, std::wstring, wstring_literal > string_types;
```

Description

An MPL-sequence of string types of attributes, supported by default

Type definition default_attribute_types

default_attribute_types

Synopsis

```
// In header: <boost/log/utility/type_dispatch/standard_types.hpp>


typedef mpl::copy< string_types, mpl::back_inserter< numeric_types >>::type default_attrib↵
ute_types;
```

Description

An MPL-sequence of all attribute value types that are supported by the library by default.

Header <boost/log/utility/type_dispatch/static_type_dispatcher.hpp>

Andrey Semashev

15.04.2007
The header contains implementation of a compile-time type dispatcher.

```
namespace boost {
  namespace log {
    template<typename T> class static_type_dispatcher;
  }
}
```

Class template static_type_dispatcher

boost::log::static_type_dispatcher — A static type dispatcher class.

Synopsis

```
// In header: <boost/log/utility/type_dispatch/static_type_dispatcher.hpp>

template<typename T>
class static_type_dispatcher {
public:
  // construct/copy/destruct
  template<typename ReceiverT> explicit static_type_dispatcher(ReceiverT &);
  static_type_dispatcher(static_type_dispatcher const &) = delete;
  static_type_dispatcher & operator=(static_type_dispatcher const &) = delete;
};
```

Description

The type dispatcher can be used to pass objects of arbitrary types from one component to another. With regard to the library, the type dispatcher can be used to extract attribute values.

Static type dispatchers allow to specify one or several supported types at compile time.

`static_type_dispatcher` public construct/copy/destruct

1.
```
template<typename ReceiverT>
  explicit static_type_dispatcher(ReceiverT & receiver);
```

Constructor. Initializes the dispatcher internals.

The *receiver* object is not copied inside the dispatcher, but references to it may be kept by the dispatcher after construction. The receiver object must remain valid until the dispatcher is destroyed.

Parameters: `receiver` Unary function object that will be called on a dispatched value. The receiver must be callable with an argument of any of the supported types of the dispatcher.

2.
```
static_type_dispatcher(static_type_dispatcher const &) = delete;
```

3.
```
static_type_dispatcher & operator=(static_type_dispatcher const &) = delete;
```

Header <boost/log/utility/type_dispatch/type_dispatcher.hpp>

Andrey Semashev

15.04.2007
The header contains definition of generic type dispatcher interfaces.

```
namespace boost {
  namespace log {
    class type_dispatcher;
  }
}
```

Class type_dispatcher

boost::log::type_dispatcher — A type dispatcher interface.

Synopsis

```
// In header: <boost/log/utility/type_dispatch/type_dispatcher.hpp>


class type_dispatcher {
public:
  // member classes/structs/unions
  template<typename T>
  class callback {
  public:

    // public member functions
    void operator()(T const &) const;
    explicit operator bool() const noexcept;
    bool operator!() const noexcept;
  };
  // construct/copy/destruct
  explicit type_dispatcher(get_callback_impl_type) noexcept;
  type_dispatcher(type_dispatcher const &) = default;
  type_dispatcher & operator=(type_dispatcher const &) = default;
  ~type_dispatcher();

  // public member functions
  template<typename T> callback< T > get_callback();
};
```

Description

All type dispatchers support this interface. It is used to acquire the visitor interface for the requested type.

`type_dispatcher` public construct/copy/destruct

1. ```
   explicit type_dispatcher(get_callback_impl_type get_callback_impl) noexcept;
   ```

   Initializing constructor

2. ```
   type_dispatcher(type_dispatcher const & that) = default;
   ```

3. ```
   type_dispatcher & operator=(type_dispatcher const & that) = default;
   ```

4. ```
   ~type_dispatcher();
   ```

`type_dispatcher` public member functions

1. ```
   template<typename T> callback< T > get_callback();
   ```

   The method requests a callback for the value of type `T`

   Returns: The type-specific callback or an empty value, if the type is not supported
   ```

Class template callback

boost::log::type_dispatcher::callback

Synopsis

```
// In header: <boost/log/utility/type_dispatch/type_dispatcher.hpp>


template<typename T>
class callback {
public:

  // public member functions
  void operator()(T const &) const;
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
};
```

Description

This interface is used by type dispatchers to consume the dispatched value.

`callback` public member functions

1. ```
   void operator()(T const & value) const;
   ```

   The operator invokes the visitor-specific logic with the given value

   Parameters: `value` The dispatched value

2. ```
   explicit operator bool() const noexcept;
   ```

   The operator checks if the visitor is attached to a receiver

3. ```
   bool operator!() const noexcept;
   ```

   The operator checks if the visitor is not attached to a receiver

Header <boost/log/utility/type_info_wrapper.hpp>

Andrey Semashev

15.04.2007
The header contains implementation of a type information wrapper.

```
namespace boost {
  namespace log {
    class type_info_wrapper;
    bool operator!=(type_info_wrapper const &, type_info_wrapper const &);
    bool operator<=(type_info_wrapper const &, type_info_wrapper const &);
    bool operator>(type_info_wrapper const &, type_info_wrapper const &);
    bool operator>=(type_info_wrapper const &, type_info_wrapper const &);
    void swap(type_info_wrapper &, type_info_wrapper &);
    std::string to_string(type_info_wrapper const &);
  }
}
```
```

Class type_info_wrapper

boost::log::type_info_wrapper — A simple `std::type_info` wrapper that implements value semantic for type information objects.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>


class type_info_wrapper {
public:
  // construct/copy/destruct
  type_info_wrapper() noexcept;
  type_info_wrapper(type_info_wrapper const &) noexcept;
  type_info_wrapper(std::type_info const &) noexcept;

  // public member functions
  explicit operator bool() const noexcept;
  std::type_info const & get() const noexcept;
  void swap(type_info_wrapper &) noexcept;
  std::string pretty_name() const;
  bool operator!() const noexcept;
  bool operator==(type_info_wrapper const &) const noexcept;
  bool operator<(type_info_wrapper const &) const noexcept;
};
```

Description

The type info wrapper is very useful for storing type information objects in containers, as a key or value. It also provides a number of useful features, such as default construction and assignment support, an empty state and extended support for human-friendly type names.

`type_info_wrapper` public construct/copy/destruct

1. ```
   type_info_wrapper() noexcept;
   ```

   Default constructor

   Postconditions: `!*this == true`

2. ```
   type_info_wrapper(type_info_wrapper const & that) noexcept;
   ```

   Copy constructor

   Parameters: `that` Source type info wrapper to copy from
   Postconditions: `*this == that`

3. ```
   type_info_wrapper(std::type_info const & that) noexcept;
   ```

   Conversion constructor

   Parameters: `that` Type info object to be wrapped
   Postconditions: `*this == that && !!*this`

`type_info_wrapper` public member functions

1. ```
   explicit operator bool() const noexcept;
   ```

Returns: `true` if the type info wrapper was initialized with a particular type, `false` if the wrapper was default-constructed and not yet initialized

2.
```
std::type_info const & get() const noexcept;
```

Stored type info getter

Requires: `!!*this`
Returns: Constant reference to the wrapped type info object

3.
```
void swap(type_info_wrapper & that) noexcept;
```

Swaps two instances of the wrapper

4.
```
std::string pretty_name() const;
```

The method returns the contained type name string in a possibly more readable format than `get().name()`

Requires: `!!*this`
Returns: Type name string

5.
```
bool operator!() const noexcept;
```

Returns: `false` if the type info wrapper was initialized with a particular type, `true` if the wrapper was default-constructed and not yet initialized

6.
```
bool operator==(type_info_wrapper const & that) const noexcept;
```

Equality comparison

Parameters: `that` Comparand
Returns: If either this object or comparand is in empty state and the other is not, the result is `false`. If both arguments are empty, the result is `true`. If both arguments are not empty, the result is `true` if this object wraps the same type as the comparand and `false` otherwise.

7.
```
bool operator<(type_info_wrapper const & that) const noexcept;
```

Ordering operator

Note

The results of this operator are only consistent within a single run of application. The result may change for the same types after rebuilding or even restarting the application.

Parameters: `that` Comparand
Requires: `!!*this && !!that`
Returns: `true` if this object wraps type info object that is ordered before the type info object in the comparand, `false` otherwise

Function operator!=

boost::log::operator!= — Inequality operator.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>


bool operator!=(type_info_wrapper const & left,
                type_info_wrapper const & right);
```

Function operator<=

boost::log::operator<= — Ordering operator.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>


bool operator<=(type_info_wrapper const & left,
                type_info_wrapper const & right);
```

Function operator>

boost::log::operator> — Ordering operator.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>


bool operator>(type_info_wrapper const & left,
               type_info_wrapper const & right);
```

Function operator>=

boost::log::operator>= — Ordering operator.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>


bool operator>=(type_info_wrapper const & left,
                type_info_wrapper const & right);
```

Function swap

boost::log::swap — Free swap for type info wrapper.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>

void swap(type_info_wrapper & left, type_info_wrapper & right);
```

Function to_string

boost::log::to_string — The function for exception serialization to string.

Synopsis

```
// In header: <boost/log/utility/type_info_wrapper.hpp>

std::string to_string(type_info_wrapper const & ti);
```

Header <boost/log/utility/unique_identifier_name.hpp>

Andrey Semashev

30.04.2008
The header contains `BOOST_LOG_UNIQUE_IDENTIFIER_NAME` macro definition.

```
BOOST_LOG_UNIQUE_IDENTIFIER_NAME(prefix)
```

Macro BOOST_LOG_UNIQUE_IDENTIFIER_NAME

BOOST_LOG_UNIQUE_IDENTIFIER_NAME

Synopsis

```
// In header: <boost/log/utility/unique_identifier_name.hpp>

BOOST_LOG_UNIQUE_IDENTIFIER_NAME(prefix)
```

Description

Constructs a unique (in the current file scope) token that can be used as a variable name. The name will contain a prefix passed in the *prefix* argument. This allows to use the macro multiple times on a single line.

Header <boost/log/utility/unused_variable.hpp>

Andrey Semashev

10.05.2008
The header contains definition of a macro to suppress compiler warnings about unused variables.

```
BOOST_LOG_UNUSED_VARIABLE(type, var, initializer)
```

Macro BOOST_LOG_UNUSED_VARIABLE

BOOST_LOG_UNUSED_VARIABLE — The macro suppresses compiler warnings for `var` being unused.

Synopsis

```
// In header: <boost/log/utility/unused_variable.hpp>

BOOST_LOG_UNUSED_VARIABLE(type, var, initializer)
```

Header <boost/log/utility/value_ref.hpp>

Andrey Semashev

27.07.2012
The header contains implementation of a value reference wrapper.

```
namespace boost {
  namespace log {
    template<typename T, typename TagT> class value_ref;
    template<typename T, typename TagT>
      void swap(value_ref< T, TagT > &, value_ref< T, TagT > &);
    template<typename CharT, typename TraitsT, typename T, typename TagT>
      std::basic_ostream< CharT, TraitsT > &
      operator<<(std::basic_ostream< CharT, TraitsT > &,
                 value_ref< T, TagT > const &);
    template<typename CharT, typename TraitsT, typename AllocatorT,
             typename T, typename TagT>
      basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
      operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > &,
                 value_ref< T, TagT > const &);
    template<typename T, typename TagT, typename U>
      bool operator==(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator==(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator==(value_ref< T1, TagT1 > const &,
                      value_ref< T2, TagT2 > const &);
    template<typename T, typename TagT, typename U>
      bool operator!=(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator!=(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator!=(value_ref< T1, TagT1 > const &,
                      value_ref< T2, TagT2 > const &);
    template<typename T, typename TagT, typename U>
      bool operator<(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator<(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator<(value_ref< T1, TagT1 > const &,
                     value_ref< T2, TagT2 > const &);
    template<typename T, typename TagT, typename U>
      bool operator>(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator>(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator>(value_ref< T1, TagT1 > const &,
                     value_ref< T2, TagT2 > const &);
    template<typename T, typename TagT, typename U>
      bool operator<=(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator<=(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator<=(value_ref< T1, TagT1 > const &,
                      value_ref< T2, TagT2 > const &);
    template<typename T, typename TagT, typename U>
      bool operator>=(value_ref< T, TagT > const &, U const &);
    template<typename U, typename T, typename TagT>
      bool operator>=(U const &, value_ref< T, TagT > const &);
    template<typename T1, typename TagT1, typename T2, typename TagT2>
      bool operator>=(value_ref< T1, TagT1 > const &,
                      value_ref< T2, TagT2 > const &);
  }
}
```

Class template value_ref

boost::log::value_ref — Reference wrapper for a stored attribute value.

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>

template<typename T, typename TagT>
class value_ref {
public:
  // construct/copy/destruct
  value_ref() = default;
  value_ref(value_ref const &) = default;
  template<typename U>
    explicit value_ref(U const &,
                       typename enable_if< typename base_type::template is_compat↵
ible< U >, int >::type = 0) noexcept;

  // public member functions
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;
  bool empty() const noexcept;
  void swap(value_ref &) noexcept;
};
```

Description

The `value_ref` class template provides access to the stored attribute value. It is not a traditional reference wrapper since it may be empty (i.e. refer to no value at all) and it can also refer to values of different types. Therefore its interface and behavior combines features of Boost.Ref, Boost.Optional and Boost.Variant, depending on the use case.

The template parameter `T` can be a single type or an MPL sequence of possible types being referred. The reference wrapper will act as either an optional reference or an optional variant of references to the specified types. In any case, the referred values will not be modifiable (i.e. `value_ref` always models a const reference).

Template parameter `TagT` is optional. It can be used for customizing the operations on this reference wrapper, such as putting the referred value to log.

`value_ref` public construct/copy/destruct

1. ```
   value_ref() = default;
   ```

   Default constructor. Creates a reference wrapper that does not refer to a value.

2. ```
   value_ref(value_ref const & that) = default;
   ```

   Copy constructor.

3. ```
   template<typename U>
     explicit value_ref(U const & val,
                        typename enable_if< typename base_type::template is_compatible< U >, int
    >::type = 0) noexcept;
   ```

   Initializing constructor. Creates a reference wrapper that refers to the specified value.

`value_ref` public member functions

1. ```
   explicit operator bool() const noexcept;
   ```

   The operator verifies if the wrapper refers to a value.

2. ```
   bool operator!() const noexcept;
   ```

   The operator verifies if the wrapper does not refer to a value.

3. ```
   bool empty() const noexcept;
   ```

   Returns: `true` if the wrapper does not refer to a value.

4. ```
   void swap(value_ref & that) noexcept;
   ```

   Swaps two reference wrappers

Function template swap

boost::log::swap — Free swap function.

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT>
  void swap(value_ref< T, TagT > & left, value_ref< T, TagT > & right);
```

Function template operator<<

boost::log::operator<< — Stream output operator.

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename CharT, typename TraitsT, typename T, typename TagT>
  std::basic_ostream< CharT, TraitsT > &
  operator<<(std::basic_ostream< CharT, TraitsT > & strm,
             value_ref< T, TagT > const & val);
```

Function template operator<<

boost::log::operator<< — Log formatting operator.

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename CharT, typename TraitsT, typename AllocatorT, typename T,
         typename TagT>
  basic_formatting_ostream< CharT, TraitsT, AllocatorT > &
  operator<<(basic_formatting_ostream< CharT, TraitsT, AllocatorT > & strm,
             value_ref< T, TagT > const & val);
```
```

Function template operator==

boost::log::operator==

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator==(value_ref< T, TagT > const & left, U const & right);
```

Function template operator==

boost::log::operator==

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator==(U const & left, value_ref< T, TagT > const & right);
```

Function template operator==

boost::log::operator==

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator==(value_ref< T1, TagT1 > const & left,
                  value_ref< T2, TagT2 > const & right);
```

Function template operator!=

boost::log::operator!=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator!=(value_ref< T, TagT > const & left, U const & right);
```

Function template operator!=

boost::log::operator!=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator!=(U const & left, value_ref< T, TagT > const & right);
```

Function template operator!=

boost::log::operator!=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator!=(value_ref< T1, TagT1 > const & left,
                  value_ref< T2, TagT2 > const & right);
```

Function template operator<

boost::log::operator<

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator<(value_ref< T, TagT > const & left, U const & right);
```

Function template operator<

boost::log::operator<

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator<(U const & left, value_ref< T, TagT > const & right);
```

Function template operator<

boost::log::operator<

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator<(value_ref< T1, TagT1 > const & left,
                 value_ref< T2, TagT2 > const & right);
```

Function template operator>

boost::log::operator>

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator>(value_ref< T, TagT > const & left, U const & right);
```

Function template operator>

boost::log::operator>

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator>(U const & left, value_ref< T, TagT > const & right);
```

Function template operator>

boost::log::operator>

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator>(value_ref< T1, TagT1 > const & left,
                 value_ref< T2, TagT2 > const & right);
```

Function template operator<=

boost::log::operator<=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator<=(value_ref< T, TagT > const & left, U const & right);
```

Function template operator<=

boost::log::operator<=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator<=(U const & left, value_ref< T, TagT > const & right);
```

Function template operator<=

boost::log::operator<=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator<=(value_ref< T1, TagT1 > const & left,
                  value_ref< T2, TagT2 > const & right);
```

Function template operator>=

boost::log::operator>=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T, typename TagT, typename U>
  bool operator>=(value_ref< T, TagT > const & left, U const & right);
```

Function template operator>=

boost::log::operator>=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename U, typename T, typename TagT>
  bool operator>=(U const & left, value_ref< T, TagT > const & right);
```

Function template operator>=

boost::log::operator>=

Synopsis

```
// In header: <boost/log/utility/value_ref.hpp>


template<typename T1, typename TagT1, typename T2, typename TagT2>
  bool operator>=(value_ref< T1, TagT1 > const & left,
                  value_ref< T2, TagT2 > const & right);
```

Header <boost/log/utility/value_ref_fwd.hpp>

Andrey Semashev

27.07.2012
The header contains forward declaration of a value reference wrapper.

Other libraries support layer

Header <boost/log/support/date_time.hpp>

Andrey Semashev

07.11.2012
This header enables Boost.DateTime support for Boost.Log.

Header <boost/log/support/exception.hpp>

Andrey Semashev

18.07.2009
This header enables Boost.Exception support for Boost.Log.

```
namespace boost {
  namespace log {
    typedef error_info< struct attribute_name_info_tag, attribute_name > attribute_name_info;
    typedef error_info< struct type_info_info_tag, type_info_wrapper > type_info_info;
    typedef error_info< struct position_info_tag, unsigned int > position_info;
    typedef error_info< struct current_scope_info_tag, attributes::named_scope_list > cur↵
rent_scope_info;
    current_scope_info current_scope();
  }
}
```

Type definition attribute_name_info

attribute_name_info

Synopsis

```
// In header: <boost/log/support/exception.hpp>

typedef error_info< struct attribute_name_info_tag, attribute_name > attribute_name_info;
```

Description

Attribute name exception information

Type definition type_info_info

type_info_info

Synopsis

```
// In header: <boost/log/support/exception.hpp>

typedef error_info< struct type_info_info_tag, type_info_wrapper > type_info_info;
```

Description

Type info exception information

Type definition position_info

position_info

Synopsis

```
// In header: <boost/log/support/exception.hpp>

typedef error_info< struct position_info_tag, unsigned int > position_info;
```

Description

Parse position exception information

Type definition current_scope_info

current_scope_info

Synopsis

```
// In header: <boost/log/support/exception.hpp>


typedef error_info< struct current_scope_info_tag, attributes::named_scope_list > cur↵
rent_scope_info;
```

Description

Current scope exception information

Function current_scope

boost::log::current_scope

Synopsis

```
// In header: <boost/log/support/exception.hpp>


current_scope_info current_scope();
```

Description

The function returns an error information object that contains current stack of scopes. This information can then be attached to an exception and extracted at the catch site. The extracted scope list won't be affected by any scope changes that may happen during the exception propagation.

Note

See the `named_scope` attribute documentation on how to maintain scope list.

Header <boost/log/support/regex.hpp>

Andrey Semashev

18.07.2009
This header enables Boost.Regex support for Boost.Log.

Header <boost/log/support/spirit_classic.hpp>

Andrey Semashev

19.07.2009
This header enables Boost.Spirit (classic) support for Boost.Log.

Header <boost/log/support/spirit_qi.hpp>

Andrey Semashev

19.07.2009
This header enables Boost.Spirit.Qi support for Boost.Log.

Header <boost/log/support/std_regex.hpp>

Andrey Semashev

19.03.2014
This header enables `std::regex` support for Boost.Log.

Header <boost/log/support/xpressive.hpp>

Andrey Semashev

18.07.2009
This header enables Boost.Xpressive support for Boost.Log.

Changelog

2.3, Boost 1.56

General changes:

- For Windows targets, the library now compiles for Windows XP by default.
- Added indexing operators with `attribute_name` arguments to `record` and `record_view`. The operators behave the same way as the similar operators of `attribute_value_set` (i.e. return an `attribute_value` identified by the name).
- Added operators for non-const object output to `basic_formatting_ostream`. (#9389)
- Added new format flags "%c", "%C" and "%F" to the named scope formatter. The new flags allow putting function names and source file names of named scopes into the formatted strings. (#9263)
- Added support for incomplete and empty markers to the named scope formatter. The markers can be used to customize the scope list presentation when the list is empty or limited by the `depth` named parameter. (#9123)
- The default presentation for incomplete named scope lists has changed. In previous releases the incomplete marker included a trailing delimiter, for example "scope1<-scope2<-...". From this release the trailing delimiter is omitted, so in the same case the formatting result would be: "scope1<-scope2..." (note the missing trailing "<-").
- Added a support header for `std::regex`. If `boost/log/support/std_regex.hpp` is included, one can use `std::regex` expressions with string matching filters.
- By default Boost.Log uses Boost.Regex internally as the regular expressions backend. Boost.Xpressive was used in previous releases. This backend is used to implement string matching filters parsed from strings or settings. Using Boost.Regex by default results in smaller executables and also better runtime performance.
- Added build configuration macros for regex backend selection. By defining `BOOST_LOG_USE_STD_REGEX`, `BOOST_LOG_USE_BOOST_REGEX` or `BOOST_LOG_USE_BOOST_XPRESSIVE` at Boost.Log build time the user can select which regex implementation will be used by the library internally for the string matching filters parsed from strings and settings. Note that this selection does not affect string matching filters in expressions.

Bug fixes:

- Fixed `dump` manipulator output on AVX2-enabled CPUs (e.g. Intel Haswell).
- Fixed compilation of `get_attribute` method of loggers.
- Fixed a possible race in `locked_backend()` function implementation of synchronous and asynchronous sink frontends.
- Fixed a possible infinite block of the logging threads in the asynchronous sink enqueue methods when `block_on_overflow` strategy was used.
- Added a workaround for ticket #9363.
- Added a workaround for MSVC bug that caused the `add_value` manipulator produce garbage attribute values when using with immediate integer constants. (#9320)

2.2, Boost 1.55

General changes:

- Added a new configuration macro `BOOST_LOG_WITHOUT_DEFAULT_FACTORIES`. By defining this macro the user can disable compilation of the default filter and formatter factories used by settings parsers. This can substantially reduce binary sizes while still retaining support for settings parsers. Note that when this macro is defined the user will have to register all attributes in the library.

- Rewritten some of the parsers to reduce the compiled binary size. The rewritten parsers are more robust in detecting ambiguous and incorrect input.
- The header `boost/log/utility/intrusive_ref_counter.hpp` is deprecated and will be removed in future releases. Its contents have been reworked and moved to Boost.SmartPtr, as `boost/smart_ptr/intrusive_ref_counter.hpp`.
- The header `boost/log/utility/explicit_operator_bool.hpp` is deprecated and will be removed in future releases. Its contents have been moved to Boost.Utility, as `boost/utility/explicit_operator_bool.hpp`.
- The header `boost/log/utility/empty_deleter.hpp` is deprecated and will be removed in future releases. Its contents have been moved to Boost.Utility, as `boost/utility/empty_deleter.hpp`.

Bug fixes:

- Fixed `timer` attribute generating incorrect time readings on Windows on heavy thread contention when `QueryPerformanceCounter` API was used.
- Fixed a bug in the filter parser that prevented using parsed filters with some attributes. For example, parsed filters didn't work with a string-typed attribute value, if the value was compared to a numeric operand.
- Fixed thread id formatting discrepancies between the default sink and formatters.
- Fixed parsing RotationTimePoint parameter values from settings, when the time point only included time of day (i.e. daily rotation).
- Closed tickets: #8815, #8819, #8915, #8917, #9139, #9140, #9153, #9155.

2.1, Boost 1.54

Breaking changes:

- `basic_formatting_ostream` no longer derives from `std::basic_ostream`, but rather reimplements its and its base classes interface closely. This solves problems with overloading `operator<<` for `basic_formatting_ostream` and user-defined types. This will break user's code if it relied on the inheritance from the standard stream types (such as passing `basic_formatting_ostream` object as an argument to a function receiving `std::basic_ostream`). Please, use the `stream()` member function to access the standard stream. This change will **not** break the code that outputs user-defined types to a `basic_formatting_ostream` stream while there are only `operator<<` overloads for the standard stream types - the code will compile and use the standard operator overloads, as before.

General changes:

- Removed the use of deprecated macros of Boost.Config.
- Build system improved. On Windows, presence of Message Compiler is now detected automatically, and support for event log is only enabled when the tool is available.
- Fixed compilation when `BOOST_LOG_USE_COMPILER_TLS` configuration macro is defined.
- Fixed compilation of some uses of the `add_value` manipulator with MSVC.
- Added a new `dump` output manipulator for printing binary data.

2.0, 13 April 2013

General changes:

- The library is now compatible with Boost 1.53 or newer. Boost.Filesystem v2 no longer supported.
- The library now does not introduce separate logging cores for different character types. A lot of other library components also became character type agnostic. The application can now use loggers of different character types with the common logging core.

The library performs character code conversion as needed. Boost.Locale can be used to construct locale objects for proper encoding conversion.

- The `BOOST_LOG_NO_COMPILER_TLS` configuration macro has been replaced with `BOOST_LOG_USE_COMPILER_TLS` with the opposite meaning. The support for compiler intrinsics for TLS is now disabled by default.
- Added configuration macros `BOOST_LOG_WITHOUT_DEBUG_OUTPUT`, `BOOST_LOG_WITHOUT_EVENT_LOG` and `BOOST_LOG_WITHOUT_SYSLOG`. `BOOST_LOG_NO_SETTINGS_PARSERS_SUPPORT` macro renamed to `BOOST_LOG_WITHOUT_SETTINGS_PARSERS`. The new macros allow to selectively disable support for the corresponding sink backends.
- The library now uses Boost.Xpressive instead of Boost.Regex internally which makes it unnecessary to build the latter in order to use the library. Boost.Regex is still supported on the user's side.
- Made some internal code to detect Windows NT6 API availability at run time, if not explicitly enabled by the `BOOST_LOG_USE_WINNT6_API` macro. The code compiled without the macro defined will still be able run on NT5, but when run on NT6 it will be more efficient. With the macro defined the resulting code will not run on NT5, but will be a little more efficient on NT6 than without the macro.
- Added a concept of a default sink. The default sink is used when there are no sinks configured in the logging core. The sink is synchronous and thread-safe, it requires no configuration and is overridden by any sinks configured in the core by user. The default sink will write log messages to the console, prepending with a timestamp, thread id and severity level.
- Trivial logging no longer implicitly initializes the library. Instead, the default sink is used to display log messages, unless the library is configured otherwise. It is now possible to use both trivial and advanced logging.
- Attribute values can now be added to log records after filtering. Such values do not participate in filtering but can be used by formatters and sinks. Log record message is now one of such attribute values, it is no longer directly accessible from the log record interface.
- Formatters and sinks no longer operate on log records but rather on `record_view`s. Records are now moved from when pushed to the core for further processing. This is done in order to eliminate the possibility of unsafe record modification after pushing to the core. As a consequence, log records can no longer be copied, only moving is allowed. Record views can be copied and moved; copying is a shallow operation.
- The implementation now provides several stream manipulators. Notably, the `to_log` manipulator allows to customize formatting for particular types and attributes without changing the regular streaming operator. Also, the `add_value` manipulator can be used in logging expressions to attach attribute values to the record.
- Made a lot of improvements to speedup code compilation.

Attributes:

- Changed the interface and the way of handling attribute values. The value is now a pimpl wrapper around the value holder. The `attribute_value` class in various components of the library is no longer pointed to with `shared_ptr`s but instead is handled by value. This allowed to simplify attribute value handling in simple cases.
- Similarly to attribute values, the interface of attributes has been reworked in the pimpl fashion. All attributes now derive from the `attribute` base class, which holds the reference to the implementation. All attributes now have to be created by value rather than wrapped into `shared_ptr` by user, which makes the code more concise.
- Added support for casting attributes from the base class `attribute` to the actual attribute type. This can be useful when the concrete attribute factory provides additional interfaces.
- The attribute value no longer has the `get` method. Use the `extract` function as a replacement.
- The key type of attribute sets and attribute values set has been changed. The new key type is called `attribute_name`. It is constructible from strings, so in most cases users won't need to change the code. See here for more information.

- Attribute values view have been renamed to attribute value set. The container now supports adding more attribute values after being constructed.
- Attribute sets and attribute value sets no longer maintain order of elements. Although it wasn't stated explicitly, the containers used to be ordered associative containers. Now the order of elements is unspecified. The implementation has been reworked to speed up insertion/removal of attributes, as well as attribute lookup and values set construction. The drawback is that memory footprint may get increased in some cases.
- Attribute sets now use small memory pools to speed up element insertion/removal.
- The header `scoped_attribute.hpp` moved from `utility` to the `attributes` directory. The header `attribute_value_extractor.hpp` in `utility` has been replaced with headers `boost/log/attributes/value_extraction.hpp` and `boost/log/attributes/value_visitation.hpp` in the `attributes` directory. The two new headers define the revised API of attribute value extraction and visitation, respectively. See here for more details.
- Scoped attribute macros simplified. The attribute constructor arguments are specified next to the attribute type and tag type is no longer required.
- The `current_thread_id` attribute no longer uses `boost::thread::id` type for thread identification. An internal type is used instead, the type is accessible as `current_thread_id::value_type`. The new thread ids are taken from the underlying OS API and thus more closely correlate to what may be displayed by debuggers and system diagnostic tools.
- Added `current_process_name` attribute. The attribute generates a string with the executable name of the current process.
- The `functor` attribute has been renamed to `function`. The generator function has been renamed from `make_functor_attr` to `make_function`. The header has been renamed from `functor.hpp` to `function.hpp`.

Logging sources:

- Fixed compilation problems with exception handling logger feature.
- Global logger storage made more friendly to the setups in which hidden visibility is set by default.
- Added the macros for separated global logger declaration and definition. Old macros have been renamed to better reflect their effect (`BOOST_LOG_DECLARE_GLOBAL_LOGGER_INIT` to `BOOST_LOG_INLINE_GLOBAL_LOGGER_INIT`, `BOOST_LOG_DECLARE_GLOBAL_LOGGER` to `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT`, `BOOST_LOG_DECLARE_GLOBAL_LOGGER_CTOR_ARGS` to `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS`). Also, the macros no longer define the `get_logger` free function for logger acquisition. Use `logger::get` instead. See here for more information.
- The channel logger now supports changing the channel name after construction. The channel name can be set either by calling the modifier method or by specifying the name in the logging statement. Added `BOOST_LOG_STREAM_CHANNEL` and `BOOST_LOG_STREAM_CHANNEL_SEV` (as well as their shorthands `BOOST_LOG_CHANNEL` and `BOOST_LOG_CHANNEL_SEV`) macros that allow to specify channel name for the log record.

Logging sinks:

- Types for integral constants for syslog and event log were renamed to drop the `_t` suffix.
- Formatting functionality moved to sink frontends. Sink backends that support record formatting derive from the `basic_formatting_sink_backend` class template, which indicates to the frontend that record formatting is required. This breaks user-side API of the library: the formatter and locale has to be set to the frontend rather than backend.
- Formatting support no longer makes frontend thread synchronization mandatory. Formatting is done prior to locking for processing the record in the backend and can be performed concurrently in multiple threads.
- Added support for flushing sinks. A sink backend that supports flushing has to define public method with the following signature: `void flush()`.
- Asynchronous sink frontend reworked, ordering asynchronous sink removed. The `asynchronous_sink` class template now allows to specify record queueing strategy. Several strategies provided, including `unbounded_fifo_queue` (the default) and `unboun-`

`ded_ordering_queue` which cover the functionality of asynchronous sink frontends in 1.x releases. See the asynchronous sink frontend docs for more details.

- Lock-free FIFO record queueing in asynchronous sinks reworked to reduce log record processing stalls.
- Added `Append` configuration file parameter for text file sinks. If this parameter is set to `true`, the sink will append log records to the existing log file instead of overwriting it.
- Added bounded variants of asynchronous sink frontends. Implemented two strategies to handle queue overflows: either log records are dropped or logging threads are blocked until there is space in the queue.

Filters and formatters:

- As a result of character type unification, filters no longer depend on the character type.
- Two new types were introduced to dynamically store filters and formatters: `filter` and `basic_formatter`. Both new types implement type erasure and provide function calling operators to invoke the stored filter or formatter.
- Filters and formatters were rewritten. The new implementation is based on Boost.Phoenix and resides in the `expressions` namespace. Attribute placeholders are now interoperable with other template expressions based on Boost.Phoenix. All template expression headers now reside in the `expressions` subdirectory.
- The library now supports defining keywords for attributes (see `BOOST_LOG_ATTRIBUTE_KEYWORD` macro). Keywords can be used in template expressions instead of attribute placeholders and also as a key in container lookups.
- Filters and formatters do not throw exceptions by default when an attribute value cannot be used to complete the function (e.g. when the value is missing or has inappropriate type). The offending filter subexpression will return `false` in such cases, the formatter will result in empty string instead of the value. The behavior can be changed by calling `or_default` or `or_throw` member functions on the attribute value placeholder in the filtering/formatting expression.
- Date and time formatter implementation is not based on Boost.DateTime IO facets anymore. The new implementation improves formatting performance. The formatter has been renamed to `format_date_time`.
- Named scope formatter now supports scope format specification. The scope format can include the scope name, as well as file name and line number. The formatter has been renamed to `format_named_scope`.
- Character decorators were renamed to `c_decor`, `c_ascii_decor`, `xml_decor` and `csv_decor`. The generic character decorator is named `char_decor` now.
- Added a new channel severity filter. The filter allows to setup severity thresholds for different channels. The filter checks log record severity level against the threshold corresponding to the channel the record belongs to.

Documentation changes:

- Most code examples from the docs have been extracted into compilable standalone examples, which can be used for testing and experimenting with the library.
- Added a lot of cross-references to the documentation, which should simplify navigation.

Miscellaneous:

- Fixed a bug: the logging core could enter an infinite loop inside `push_record` if a sink throws and the exception is suppressed by the exception handler set in the core.
- Changed the type dispatching implementation to reduce the usage of virtual functions. This greatly reduced the library size.
- Type dispatchers made more friendly to the setups in which hidden visibility is set by default.
- The interface of type dispatchers changed. The dispatcher now returns `type_visitor` instance by value, and the visitor is no longer a base for the actual receiver of the dispatched value. Instead, the visitor now refers to the receiver, if one is capable to

consume the value. The `visit` method has been renamed to `operator ()`. The static type dispatcher now requires a reference to the receiver on construction, it doesn't imply that the receiver derives from the dispatcher anymore.

- The `slim_string` utility has been removed. There is no replacement.
- The library now uses many features from the latest C++ standard (aka C++11). For instance, many library components now support move semantics. Boost.Move is used for move emulation on C++03-compatible compilers.

1.1, 02 December 2011

This release mostly fixes bugs in the code and documentation.

- Added support for Boost.Filesystem v3.
- A number of bugs fixed.
- Corrections in the documentation.

1.0, 09 May 2010

This release mostly fixes bugs in the code and documentation. The next major release (2.0) will contain breaking changes and feature additions. The 1.0 branch will not receive any feature updates.

- Added some optimization for thread local storage. In Windows setups, if you dynamically load Boost.Log binaries during the application run time, this optimization may lead to crashes. In that case, you may disable it by defining `BOOST_LOG_NO_COMPILER_TLS` during the library build process. The macro also affects other platforms, which may be useful if your compiler does not support TLS.
- Added a few public accessors and convenience constructors to severity and channel loggers.
- Added ability to rotate log files at the specified time points. The `rotation_interval` keyword is no longer available. The same functionality is achieved with the new `time_based_rotation` keyword and the `rotation_at_time_interval` predicate. See here for more details.
- Improved support for MinGW and Cygwin.
- A number of bugs fixed. Added workarounds to compile on GCC 4.2.
- Lots of corrections in the documentation.

Release Candidate 4, 08 Jan 2010

- Substantial documentation improvement. The tutorial section has been reorganized.
- Library headers have been reorganized. Some other Boost libraries that were previously included by headers have been made optional. Such dependencies have been extracted into separate headers in the `support` directory. Top level library headers now mostly include nested headers.
- Keywords have moved into a dedicated `keywords` namespace. There are no longer nested `keywords` namespaces in `sinks`, `attributes`, etc. All keywords have been extracted into separate headers in the `keywords` directory.
- Removed rotating file stream. As a replacement, a new file sink has been added, which allows to achieve the same results ans adds a few more features.
- Added a new multifile sink backend.
- Added a new ordering asynchronous sink frontend.
- The syslog sink backend is now supported on Windows, too. The sink no longer requires native support for POSIX API for syslog, but is able to send syslog packets to a remote server over UDP.

- Loggers implementation has been improved. Feature composition mechanism has been cleaned up.
- Added support for scoped logging. There is now a distinct log record entity, which is returned by the core as a result of filtering. It is possible to fill in the record message in any way the user wants, not necessarily with a streaming expression. The record object is now processed by sinks and formatters.
- Added support for exception control. User can register exception handlers at one of the three layers: for a particular sink, at the core layer, and for a particular logger (given that it has the appropriate feature). Sinks and core will not suppress exceptions by default. Filter and formatters will throw if the requested attribute value is not found.
- Added a few new formatters, called character decorators. These can be useful to post-process the formatted output before passing it on to the sink.
- Added attributes for thread and process identifiers. These identifiers are automatically added after the call to `add_common_attributes`.
- Helper initialization functions, such as `init_log_to_file` now accept more customization options as named arguments.
- A new initialization interface has been exposed. One can fill a settings container and use it to initialize the library.
- The library setup support code has been extracted into a separate binary. Further on, this binary will be made optional to build.
- Added a new mode of logging, called trivial logging. In this mode the library requires no initialization at all, however it does not offer many ways of customization.
- A number of bugs fixed.
- A few optimizations added to improve multithreaded applications performance.
- Removed some bug workarounds for older Boost releases. The library now requires Boost 1.39 or newer.

Release Candidate 3, 08 Feb 2009

- Substantial documentation improvement.
- Added several Windows-specific sinks: Event Log (simplified and advanced), Windows debugger and experimental Event Trace for Windows Vista and later.
- Loggers now consist of a number of independent features that can be composed the way the user needs. User-defined features can be developed and injected into the mix.
- Attribute value extractors improved. With the new extract function attribute values can be extracted from the attribute values view by employing lambda functors.
- Some files and classes were moved or renamed to improve code clarity and shorten names.
- A number of bugs fixed.
- Added tests.

Release Candidate 2

No one really remembers these dark ages...

TODO in future releases

Points in this section are not necessarily going to be implemented. These are mainly some thoughts on further improvements of the library.

- Optimize single-threaded configuration. In many places dynamic memory allocation can be avoided if multithreading support is disabled.
- SNMP support. The idea is to implement a sink backend that would emit SNMP traps as a result of processing log records. This needs quite an amount of research and thinking over.
- Provide a compile-time option to remove all logging from the application (the compiled binary should contain no traces of logging internally). There are two reasons for this request: attempting to achieve maximum performance and concealing internal information, such as function names and internal messages, to prevent reverse engineering in no-logging builds. Effectively, this would require not only all library macros to be redefined to emptiness, but also to provide dummy implementations of many library components. Needs more consideration. Perhaps, suppressing only macros would be sufficient.
- Provide a macro, like `BOOST_LOG_FUNCTION`, but with ability to automatically log all function arguments.
- Think over a header-only configuration. Perhaps, with a reduced functionality.
- Update syslog support to RFC 5424.
- Provide some kind of shared formatters. The idea is that several sinks may use the same formatter. If a log record passes filtering to multiple such sinks, the formatting is done just once for all sinks that share the formatter. Maybe, it will require refactoring the sinks architecture, transforming them into pipelines with formatter and backends being just steps in log record processing.
- Allow to change the locale for the file stream in the text file backend. The locale can alter the character code conversion in wide-character logging.
- Improve file collection in the file sink. Make it possible to (i) rename collected files and (ii) collect files in a dedicated thread.
- Provide headers with forward declarations of the library components.
- Make it possible to update library configuration after loading settings from a file. Probably, this will require a new configuration entity that will be able to detect and apply changes between settings.
- Develop a statistics gathering framework. The basic idea is to provide a specific log source and a pin. The user can pin his data or explicitly indicate events by invoking the log source. The source would automatically collect the data from the pinned variables. This source should have a better integration with filters to be able which pins should be collected and which should not.
- Allow to specify a process ID in the file name pattern for file-based sinks.
- Improve support for `format` formatter, implement placeholder format flags.

Acknowledgments

- Vladimir Prus managed the library review in Boost and actually reviewed it in the process.
- Adam Wulkiewicz created the logo used on the GitHub project page. The logo was taken from his collection of Boost logos.
- Luca Rigini wrote the initial implementation of the NT event log sink and made a lot of suggestions on how to improve the library with regard to writing user-defined sinks.
- Jean-Daniel Michaud, Michael Lacher and all others who took part in the discussion of the requirements to the library on Wiki.
- John Torjo, Gennadiy Rozental and others for their discussion on John's logging library on the Boost developers list. It helped a lot to learn the requirements and possible solutions for the library.
- All authors of the great Boost libraries that were involved in this library (notably, Boost.SmartPtr, Boost.Thread, Boost.DateTime, Boost.Filesystem, Boost.Intrusive, Boost.Spirit2 and others) and Boost.Quickbook authors for a simple yet powerful documenting tool.
- All the reviewers and the users who made suggestions and offered their feedback on the library. Most notably, Steven Watanabe for his in-depth studying the docs and the code, with a lot of fruitful comments on both.