Boost Exception

C++ Exception Augmentation Library | Emil Dotchevski

For C++11 or newer, consider using Boost LEAF. It provides similar functionality more efficiently and understands Boost Exception for compatibility; see this overview.

Introduction

The purpose of Boost Exception is to ease the design of exception class hierarchies and to help write exception handling and error reporting code.

It supports transporting of arbitrary data to the catch site, which is otherwise tricky due to the no-throw requirements (15.5.1) for exception types. Data can be added to any exception object, either directly in the throw-expression (15.1), or at a later time as the exception object propagates up the call stack.

The ability to add data to exception objects after they have been passed to throw is important, because often some of the information needed to handle an exception is unavailable in the context where the failure is detected.

Boost Exception also supports N2179-style copying of exception objects, implemented non-intrusively and automatically by the `boost::throw_exception` function.

Tutorial

Transporting of Arbitrary Data to the Catch Site

All exception types that derive from `boost::exception` can be used as type-safe containers of arbitrary data objects, while complying with the no-throw requirements (15.5.1) of the ANSI C++ standard for exception types.

When exceptions derive from `boost::exception`, arbitrary data can be added to exception objects:

- At the point of the throw;
- At a later time as exceptions bubble up the call stack.

Adding of Arbitrary Data at the Point of the Throw

The following example demonstrates how errno can be stored in exception objects using Boost Exception:

```
#include <boost/exception/all.hpp>
#include <iostream>

typedef boost::error_info<struct tag_my_info,int> my_info; //(1)

struct my_error: virtual boost::exception, virtual std::exception { }; //(2)

void
f()
    {
    throw my_error() << my_info(42); //(3)
    }
```

`error_info` | `exception` | `operator<<`

First, we instantiate the `error_info` template using a unique identifier — tag_my_info, and the type of the info it identifies — int. This provides compile-time type safety for the various values stored in exception objects.

Second, we define class my_error, which derives from `boost::exception`.

Finally, (3) illustrates how the typedef from (1) can be used with `operator<<` to store values in exception objects at the point of the throw.

The stored my_info value can be recovered at a later time like this:

```
// ...continued
```

```
void
g()
    {
    try
        {
        f();
        }
    catch(
    my_error & x )
        {
        if( int const * mi=boost::get_error_info<my_info>(x) )
            std::cerr << "My info: " << *mi;
        }
    }
```

get_error_info

The get_error_info function template is instantiated with the typedef from (1), and is passed an exception object of a polymorphic type. If the exception object contains the requested value, err will point to it; otherwise a null pointer is returned.

Adding of Arbitrary Data to Active Exception Objects

Sometimes the throw site does not have all the information that is needed at the catch site to make sense of what went wrong. Let's say we have an exception type file_read_error, which takes a file name in its constructor. Consider the following function:

```
void
file_read( FILE * f, void * buffer, size_t size )
    {
    if( size!=fread(buffer,1,size,f) )
        throw file_read_error(????);
    }
```

How can the file_read function pass a file name to the exception type constructor? All it has is a FILE handle.

Using boost::exception allows us to free the file_read function from the burden of storing the file name in exceptions it throws:

```
#include <boost/exception/all.hpp>
#include <boost/shared_ptr.hpp>
#include <stdio.h>
#include <errno.h>

struct file_read_error: virtual boost::exception { };

void
file_read( FILE * f, void * buffer, size_t size )
```

```
    {
    if( size!=fread(buffer,1,size,f) )
        throw file_read_error() << boost::errinfo_errno(errno);
    }
```

exception | errinfo_errno

If file_read detects a failure, it throws an exception which contains the information that is available at the time, namely the errno. Other relevant information, such as the file name, can be added in a context higher up the call stack, where it is known naturally:

```
#include <boost/exception/all.hpp>
#include <boost/shared_ptr.hpp>
#include <stdio.h>
#include <string>

boost::shared_ptr<FILE> file_open( char const * file_name, char const * mode );
void file_read( FILE * f, void * buffer, size_t size );

void
parse_file( char const * file_name )
    {
    boost::shared_ptr<FILE> f = file_open(file_name,"rb");
    assert(f);
    try
        {
        char buf[1024];
        file_read( f.get(), buf, sizeof(buf) );
        }
    catch(
    boost::exception & e )
        {
        e << boost::errinfo_file_name(file_name);
        throw;
        }
    }
```

exception | errinfo_file_name

The above function is (almost) exception-neutral — if an exception is emitted by any function call within the try block, parse_file does not need to do any real work, but it intercepts any `boost::exception` object, stores the file name, and re-throws using a throw-expression with no operand (15.1.6). The rationale for catching any `boost::exception` object is that the file name is relevant to any failure that occurs in parse_file, *even if the failure is unrelated to file I/O*.

Adding Grouped Data to Exceptions

The code snippet below demonstrates how `boost::http://www.boost.org/libs/tuple/doc/tuple_users_guide.html[tuple]` can be

used to bundle the name of the function that failed, together with the reported errno so that they can be added to exception objects more conveniently together:

```
#include <boost/exception/info_tuple.hpp>
#include <boost/exception/errinfo_file_name.hpp>
#include <boost/exception/errinfo_api_function.hpp>
#include <boost/exception/errinfo_errno.hpp>
#include <boost/shared_ptr.hpp>
#include <stdio.h>
#include <string>
#include <errno.h>

typedef boost::tuple<boost::errinfo_api_function,boost::errinfo_errno> clib_failure;

struct file_open_error: virtual boost::exception { };

boost::shared_ptr<FILE>
file_open( char const * name, char const * mode )
    {
    if( FILE * f=fopen(name,mode) )
        return boost::shared_ptr<FILE>(f,fclose);
    else
        throw file_open_error() <<
            boost::errinfo_file_name(name) <<
            clib_failure("fopen",errno);
    }
```

tuple/operator<< | errinfo_file_name | errinfo_api_function | errinfo_errno

Note that the members of a boost::http://www.boost.org/libs/tuple/doc/tuple_users_guide.html[tuple] are stored separately in exception objects; they can only be retrieved individually, using get_error_info.

Integrating Boost Exception in Existing Exception Class Hierarchies

Some exception hierarchies can not be modified to make boost::exception a base type. In this case, the enable_error_info function template can be used to make exception objects derive from boost::exception anyway. Here is an example:

```
#include <boost/exception/all.hpp>
#include <stdexcept>

typedef boost::error_info<struct tag_std_range_min,size_t> std_range_min;
typedef boost::error_info<struct tag_std_range_max,size_t> std_range_max;
```

```
typedef boost::error_info<struct tag_std_range_index,size_t> std_range_index;

template <class T>
class
my_container
    {
    public:

    size_t size() const;

    T const &
    operator[]( size_t i ) const
        {
        if( i > size() )
            throw boost::enable_error_info(std::range_error("Index out of range")) <<
                std_range_min(0) <<
                std_range_max(size()) <<
                std_range_index(i);
        //....
        }
    };
```

enable_error_info | operator<<

The call to enable_error_info<T> gets us an object of *unspecified type* which is guaranteed to derive from both boost::exception and T. This makes it possible to use operator<< to store additional information in the exception object. The exception can be intercepted as T &, so existing exception handling will not break. It can also be intercepted as boost::exception &, so that more information can be added to the exception at a later time.

Transporting of Exceptions Between Threads

Boost Exception supports transporting of exception objects between threads through cloning. This system is similar to N2179, but because Boost Exception can not rely on language support, the use of enable_current_exception at the time of the throw is required in order to use cloning.

All exceptions emitted by the familiar function boost::throw_exception are guaranteed to derive from boost::exception and to support cloning.

Using enable_current_exception at the Time of the Throw

Here is how cloning can be enabled in a throw-expression (15.1):

```
#include <boost/exception/info.hpp>
#include <boost/exception/errinfo_errno.hpp>
#include <stdio.h>
#include <errno.h>
```

```
struct file_read_error: virtual boost::exception { };

void
file_read( FILE * f, void * buffer, size_t size )
    {
    if( size!=fread(buffer,1,size,f) )
        throw boost::enable_current_exception(file_read_error()) <<
            boost::errinfo_errno(errno);
    }
```

enable_current_exception | errinfo_errno

Of course, enable_current_exception may be used with any exception type; there is no requirement that it should derive from boost::exception.

Cloning and Re-Throwing an Exception

When you catch an exception, you can call current_exception to get an exception_ptr object:

```
#include <boost/exception_ptr.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>

void do_work(); //throws cloning-enabled boost::exceptions

void
worker_thread( boost::exception_ptr & error )
    {
    try
        {
        do_work();
        error = boost::exception_ptr();
        }
    catch(
    ... )
        {
        error = boost::current_exception();
        }
    }
```

current_exception | exception_ptr

In the above example, note that current_exception captures the original type of the exception object. The exception can be thrown again using the rethrow_exception function:

```
// ...continued

void
```

```
work()
    {
    boost::exception_ptr error;
    boost::thread t( boost::bind(worker_thread,boost::ref(error)) );
    t.join();
    if( error )
        boost::rethrow_exception(error);
    }
```

rethrow_exception | exception_ptr

Note that current_exception could fail to copy the original exception object in the following cases:

- if there is not enough memory, in which case the returned exception_ptr points to an instance of std::bad_alloc, or
- if enable_current_exception was not used in the throw-expression passed to the original throw statement and the current implementation does not have the necessary compiler-specific support to copy the exception automatically, in which case the returned exception_ptr points to an instance of unknown_exception.

Regardless, the use of current_exception and rethrow_exception in the above examples is well-formed.

Exception Types as Simple Semantic Tags

Deriving from boost::exception effectively decouples the semantics of a failure from the information that is relevant to each individual instance of reporting a failure with a given semantic.

In other words: with boost::exception, what data a given exception object transports depends primarily on the context in which failures are reported (not on its type.) Since exception types need no members, it becomes very natural to throw exceptions that derive from more than one type to indicate multiple appropriate semantics:

```
struct exception_base: virtual std::exception, virtual boost::exception { };
struct io_error: virtual exception_base { };
struct file_error: virtual io_error { };
struct read_error: virtual io_error { };
struct file_read_error: virtual file_error, virtual read_error { };
```

exception

Using this approach, exception types become a simple tagging system for categorizing errors and selecting failures in exception handlers.

Using Virtual Inheritance in Exception Types

Exception types should use virtual inheritance when deriving from other exception types. This insight is due to Andrew Koenig. Using virtual inheritance prevents ambiguity problems in the exception handler:

```
#include <iostream>
struct my_exc1 : std::exception { char const* what() const throw(); };
struct my_exc2 : std::exception { char const* what() const throw(); };
struct your_exc3 : my_exc1, my_exc2 {};

int
main()
    {
    try { throw your_exc3(); }
    catch(std::exception const& e) {}
    catch(...) { std::cout << "whoops!" << std::endl; }
    }
```

The program above outputs "whoops!" because the conversion to std::exception is ambiguous.

The overhead introduced by virtual inheritance is always negligible in the context of exception handling. Note that virtual bases are initialized directly by the constructor of the most-derived-type (the type passed to the throw statement, in case of exceptions.) However, typically this detail is of no concern when `boost::exception` is used, because it enables exception types to be trivial structs with no members (there's nothing to initialize.) See Exception Types as Simple Semantic Tags.

Diagnostic Information

Boost Exception provides a namespace-scope function `diagnostic_information` which takes a `boost::exception`. The returned string contains:

- the string representation of all data objects added to the `boost::exception` through `operator<<`;
- the output from std::exception::what;
- additional platform-specific diagnostic information.

The returned string is not presentable as a friendly user message, but because it is generated automatically, it is useful for debugging or logging purposes. Here is an example:

```
#include <boost/exception/all.hpp>
#include <iostream>

void f(); //throws unknown types that derive from boost::exception.
```

```
void
g()
    {
    try
        {
        f();
        }
    catch(
    boost::exception & e )
        {
        std::cerr << diagnostic_information(e);
        }
    }
```

`diagnostic_information`

Example:

this is a possible output from the `diagnostic_information` function, as used in *libs/exception/example/example_io.cpp:*

```
example_io.cpp(70): Throw in function class boost::shared_ptr<struct _iobuf> __cdecl
my_fopen(const char *,const char *)
Dynamic exception type: class boost::exception_detail::clone_impl<struct fopen_error>
std::exception::what: example_io error
[struct boost::errinfo_api_function_ *] = fopen
[struct boost::errinfo_errno_ *] = 2, "No such file or directory"
[struct boost::errinfo_file_name_ *] = tmp1.txt
[struct boost::errinfo_file_open_mode_ *] = rb
```

Serialization

Boost Exception provides a serialization API that enables exporting diagnostic information into different formats, such as JSON. This is useful for structured logging, remote debugging, or integrating with monitoring systems.

To serialize diagnostic information, use the following functions:

- `serialize_diagnostic_information_to` — serializes diagnostic information for a given exception.
- `serialize_current_exception_diagnostic_information_to` — serializes diagnostic information for the current exception.

The serialization system uses `output` and `output_at` functions found via ADL:

- `output(enc, x)` serializes value `x` directly to encoder `enc`.
- `output_at(enc, x, name)` serializes value `x` to encoder `enc` as a named field.

Custom Encoders

To support exporting to a specific format, users define an encoder class with associated `output` and `output_at` function templates:

```
struct my_encoder
{
    template <class T>
    friend void output(my_encoder & enc, T const & x)
    {
        // output value x to enc
    }

    template <class T>
    friend void output_at(my_encoder & enc, T const & x, char const * name)
    {
        // output x to enc as a named field
    }
};
```

The `output_at` function typically creates a nested scope (e.g. a JSON object) and then calls `output` to serialize the value.

To enable serialization to a custom encoder type, define a `serialize` function template in the `boost::exception_serialization` namespace:

```
namespace boost { namespace exception_serialization {

template <class Handle, class T>
void serialize(Handle & h, T const & x, char const * name)
{
    h.dispatch([&](my_encoder & enc) {
        output_at(enc, x, name);
    });
}

} }
```

The `serialize` function template takes a handle reference `h` (of unspecified type) that holds an encoder, the object to be serialized, and its type name. Call `h.dispatch` with a single-argument function F to detect the encoder type based on F's argument type; F is called only if the handle contains an encoder of that type.

To support multiple output formats, pass multiple functions to `h.dispatch`:

```
h.dispatch(
    [&](json_encoder & enc) { output_at(enc, x, name); },
    [&](xml_encoder & enc) { output_at(enc, x, name); }
```

```
);
```

JSON Serialization

Boost Exception provides two JSON encoders:

- `boost_json_encoder` for Boost.JSON
- `nlohmann_json_encoder` based on ADL calls to `to_json` (compatible with nlohmann/json)

Below is an example using `nlohmann_json_encoder` with nlohmann/json. We just need to define the required `serialize` function template (see Custom Encoders):

```
#include <boost/exception/serialization/nlohmann_json_encoder.hpp>
#include "nlohmann/json.hpp"

using nlohmann_json_encoder = boost::exception_serialization::nlohmann_json_encoder
<nlohmann::json>;

namespace boost { namespace exception_serialization {

template <class Handle, class T>
void serialize(Handle & h, T const & x, char const * name)
{
    h.dispatch([&](nlohmann_json_encoder & enc) {
        output_at(enc, x, name);
    });
}

} }
```

With this in place, we can output diagnostic information to JSON:

```
#include <boost/exception/diagnostic_information.hpp>
#include <boost/throw_exception.hpp>
#include <iostream>

struct api_response
{
    int status;
    std::string message;

    template <class Json>
    friend void to_json(Json & j, api_response const & e)
    {
        j["status"] = e.status;
        j["message"] = e.message;
    }
};
```

```
typedef boost::error_info<struct errinfo_api_response_, api_response>
errinfo_api_response;
typedef boost::error_info<struct errinfo_request_url_, std::string>
errinfo_request_url;

struct api_error : virtual boost::exception, virtual std::exception { };

....

nlohmann::json j;
try
{
    BOOST_THROW_EXCEPTION(api_error()
        << errinfo_api_response({403, "Access denied"})
        << errinfo_request_url("/api/admin/settings"));
}
catch(boost::exception & e)
{
    nlohmann_json_encoder enc{j};
    boost::serialize_diagnostic_information_to(enc, e);
}
std::cout << j.dump(2) << std::endl;
```

Output:

```
{
  "errinfo_api_response_": {
    "status": 403,
    "message": "Access denied"
  },
  "errinfo_request_url_": "/api/admin/settings"
}
```

serialize_diagnostic_information_to | nlohmann_json_encoder

In the example above, `api_response` uses an unqualified call to `to_json` for serialization. This is to demonstrate that `nlohmann_json_encoder` handles third party types with suitable `to_json` overloads automatically. If instead we defined a function `output` compatible with the Boost Exception serialization API, it would make `api_response` compatible with any Boost Exception encoder.

Synopsis

This section lists each public header file, documenting the definitions it provides.

`exception.hpp`

#include <boost/exception/exception.hpp>

```
namespace boost
{
    class exception
    {
    protected:
        exception();
        exception( exception const & x );
        ~exception();
    };

    template <class Tag,class T>
    class error_info;

    typedef error_info<struct throw_function_,char const *> throw_function;
    typedef error_info<struct throw_file_,char const *> throw_file;
    typedef error_info<struct throw_line_,int> throw_line;
}
```

Reference: `exception` | `error_info`

`error_info.hpp`

#include <boost/exception/error_info.hpp>

```
namespace boost
{
    template <class Tag,class T>
    class error_info;
}
```

Reference: `error_info`

`info.hpp`

#include <boost/exception/info.hpp>

```
#include <boost/exception/exception.hpp>
```

```
namespace boost
{
    template <class Tag,class T>
    class error_info
    {
    public:
        typedef T value_type;

        error_info( value_type const & v );

        value_type const & value() const;
        value_type & value();
    };

    template <class E, class Tag, class T>
    E const & operator<<( E const & x, error_info<Tag,T> const & v );
}
```

Reference: error_info | operator<<

info_tuple.hpp

#include <boost/exception/info_tuple.hpp>

```
#include <boost/exception/info.hpp>
#include <boost/tuple/tuple.hpp>

namespace boost
{
    template <class E, class Tag1, class T1, ..., class TagN, class TN>
    E const & operator<<( E const & x,
        tuple<
            error_info<Tag1,T1>,
            ...,
            error_info<TagN,TN> > const & v );
}
```

Reference: tuple/operator<<

enable_error_info.hpp

#include <boost/exception/enable_error_info.hpp>

```
#include <boost/exception/exception.hpp>

namespace boost
```

```
{
    template <class T>
    ---unspecified--- enable_error_info( T const & x );
}
```

Reference: enable_error_info

diagnostic_information.hpp

#include <boost/exception/diagnostic_information.hpp>

```
#include <string>

namespace boost
{
    class exception;

    template <class E>
    std::string diagnostic_information( E const & e, bool verbose=true );

    std::string diagnostic_information( exception_ptr const & p, bool verbose=true
);

    char const * diagnostic_information_what( boost::exception const & e, bool
verbose=true ) throw();

    std::string current_exception_diagnostic_information();

    template <class Encoder, class E>
    void serialize_diagnostic_information_to( Encoder & enc, E const & e );

    template <class Encoder>
    void serialize_current_exception_diagnostic_information_to( Encoder & enc );
}
```

Reference: diagnostic_information | diagnostic_information_what | current_exception_diagnostic_information | serialize_diagnostic_information_to | serialize_current_exception_diagnostic_information_to

current_exception_cast.hpp

#include <boost/exception/current_exception_cast.hpp>

```
namespace boost
{
    template <class E>
```

```
    E * current_exception_cast();
}
```

Reference: current_exception_cast

exception_ptr.hpp

#include <boost/exception_ptr.hpp>

```
#include <boost/exception/exception.hpp>

namespace boost
{
    class unknown_exception:
        public std::exception
        public boost::exception
    {
        ---unspecified---
    };

    typedef error_info<struct tag_original_exception_type,std::type_info const *>
original_exception_type;

    typedef ---unspecified--- exception_ptr;

    template <class T>
    exception_ptr copy_exception( T const & e );

    exception_ptr current_exception();

    void rethrow_exception( exception_ptr const & ep );
}
```

Reference: exception_ptr | unknown_exception | original_exception_type | copy_exception | current_exception | rethrow_exception

enable_current_exception.hpp

#include <boost/exception/enable_current_exception.hpp>

```
#include <boost/exception/exception.hpp>

namespace boost
{
    template <class T>
    ---unspecified--- enable_current_exception( T const & e );
```

```
}
```

Reference: enable_current_exception

throw_exception.hpp

#include <boost/throw_exception.hpp>

```
#if !defined( BOOST_EXCEPTION_DISABLE )
    #include <boost/exception/exception.hpp>
    #include <boost/current_function.hpp>
    #define BOOST_THROW_EXCEPTION(x)\
        ::boost::throw_exception( ::boost::enable_error_info(x) <<\
        ::boost::throw_file(__FILE__) <<\
        ::boost::throw_line((int)__LINE__) )
#else
    #define BOOST_THROW_EXCEPTION(x) ::boost::throw_exception(x)
#endif

namespace boost
{
#ifdef BOOST_NO_EXCEPTIONS
    void throw_exception( std::exception const & e ); // user defined
#else
    template <class E>
    void throw_exception( E const & e );
#endif
}
```

Reference: throw_exception | BOOST_THROW_EXCEPTION

errinfo_api_function.hpp

#include <boost/exception/errinfo_api_function.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    typedef error_info<struct errinfo_api_function_,char const *>
errinfo_api_function;
}
```

Reference: errinfo_api_function

errinfo_at_line.hpp

#include <boost/exception/errinfo_at_line.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    typedef error_info<struct errinfo_at_line_,int> errinfo_at_line;
}
```

Reference: errinfo_at_line

errinfo_errno.hpp

#include <boost/exception/errinfo_errno.hpp>

```
#include <boost/exception/error_info.hpp>
#include <errno.h>

namespace boost
{
    typedef error_info<struct errinfo_errno_,int> errinfo_errno;
}
```

Reference: errinfo_errno

errinfo_file_handle.hpp

#include <boost/exception/errinfo_file_handle.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    template <class> class weak_ptr;
    typedef error_info<struct errinfo_file_handle_,weak_ptr<FILE> >
errinfo_file_handle;
}
```

Reference: errinfo_file_handle

errinfo_file_name.hpp

#include <boost/exception/errinfo_file_name.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_file_name_,std::string> errinfo_file_name;
}
```

Reference: errinfo_file_name

errinfo_file_open_mode.hpp

#include <boost/exception/errinfo_file_open_mode.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_file_open_mode_,std::string>
errinfo_file_open_mode;
}
```

Reference: errinfo_file_open_mode

errinfo_nested_exception.hpp

#include <boost/exception/errinfo_nested_exception.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    typedef ---unspecified--- exception_ptr;
    typedef error_info<struct errinfo_nested_exception_,exception_ptr>
errinfo_nested_exception;
}
```

Reference: errinfo_nested_exception

`errinfo_type_info_name.hpp`

#include <boost/exception/errinfo_type_info_name.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_type_info_name_,std::string>
errinfo_type_info_name;
}
```

Reference: `errinfo_type_info_name`

`all.hpp`

#include <boost/exception/all.hpp>

```
#include <boost/exception/diagnostic_information.hpp>
#include <boost/exception/error_info.hpp>
#include <boost/exception/exception.hpp>
#include <boost/exception/get_error_info.hpp>
#include <boost/exception/info.hpp>
#include <boost/exception/info_tuple.hpp>
#include <boost/exception/errinfo_api_function.hpp>
#include <boost/exception/errinfo_at_line.hpp>
#include <boost/exception/errinfo_errno.hpp>
#include <boost/exception/errinfo_file_handle.hpp>
#include <boost/exception/errinfo_file_name.hpp>
#include <boost/exception/errinfo_file_open_mode.hpp>
#include <boost/exception/errinfo_type_info_name.hpp>
#ifndef BOOST_NO_EXCEPTIONS
#include <boost/exception/errinfo_nested_exception.hpp>
#include <boost/exception_ptr.hpp>
#endif
```

This header includes all Boost Exception headers except `boost/exception_ptr.hpp` (unless BOOST_NO_EXCEPTIONS is defined.)

Serialization

`boost_json_encoder.hpp`

#include <boost/exception/serialization/boost_json_encoder.hpp>

```
namespace boost { namespace exception_serialization {

struct boost_json_encoder
{
    boost::json::value & v_;

    // Uses unspecified SFINAE expression designed to make the
    // overload selected only if no other compatible overload is found.
    // Implemented in terms of boost::json::value_from.
    template <class T>
    friend void output( boost_json_encoder &, T const & x );

    template <class T>
    friend void output_at( boost_json_encoder &, T const &, char const * name );
};

} }
```

Reference: boost_json_encoder

nlohmann_json_encoder.hpp

#include <boost/exception/serialization/nlohmann_json_encoder.hpp>

```
namespace boost { namespace exception_serialization {

template <class Json>
struct nlohmann_json_encoder
{
    Json & j_;

    // Uses unspecified SFINAE expression designed to make the
    // overload selected only if no other compatible overload is found.
    // Implemented in terms of to_json.
    template <class T>
    friend void output( nlohmann_json_encoder &, T const & x );

    template <class T>
    friend void output_at( nlohmann_json_encoder &, T const &, char const * name
);
};

} }
```

Reference: nlohmann_json_encoder

Reference

The contents of each Reference section are organized alphabetically.

Types

`boost_json_encoder`

#include <boost/exception/serialization/boost_json_encoder.hpp>

```
namespace boost { namespace exception_serialization {

struct boost_json_encoder
{
    boost::json::value & v_;

    // Uses unspecified SFINAE expression designed to make the
    // overload selected only if no other compatible overload is found.
    // Implemented in terms of boost::json::value_from.
    template <class T>
    friend void output( boost_json_encoder &, T const & x );

    template <class T>
    friend void output_at( boost_json_encoder &, T const &, char const * name );
};

} }
```

The `boost_json_encoder` type serializes objects to JSON format using Boost.JSON. The `output` function is implemented in terms of `boost::json::value_from`.

See Serialization.

`error_info`

#include <boost/exception/info.hpp>

```
namespace boost
{
    template <class Tag,class T>
    class error_info
    {
    public:

        typedef T value_type;

        error_info( value_type const & v );
```

```
        value_type const & value() const;
        value_type & value();
    };
}
```

Requirements:

T must have accessible copy constructor and must not be a reference (there is no requirement that T's copy constructor does not throw.)

This class template is used to associate a Tag type with a value type T. Objects of type `error_info<Tag,T>` can be passed to `operator<<` to be stored in objects of type `boost::exception`.

The header `<boost/exception/error_info.hpp>` provides a declaration of the `error_info` template, which is sufficient for the purpose of typedefing an instance for specific Tag and T, for example:

```
#include <boost/exception/error_info.hpp>

struct tag_errno;
typedef boost::error_info<tag_errno,int> errno_info;
```

Or, the shorter equivalent:

```
#include <boost/exception/error_info.hpp>

typedef boost::error_info<struct tag_errno,int> errno_info;
```

This errno_info typedef can be passed to `operator<<` (#include `<boost/exception/info.hpp>` first) to store an int named tag_errno in exceptions of types that derive from `boost::exception`:

```
throw file_read_error() << errno_info(errno);
```

It can also be passed to `get_error_info` (#include `<boost/exception/get_error_info.hpp>` first) to retrieve the tag_errno int from a `boost::exception`:

```
catch( boost::exception & x )
    {
    if( int const * e=boost::get_error_info<errno_info>(x) )
        ....
    }
```

For convenience and uniformity, Boost Exception defines the following commonly used `error_info` typedefs, ready for use with `operator<<`:

- errinfo_api_function
- errinfo_at_line
- errinfo_errno
- errinfo_file_handle
- errinfo_file_name
- errinfo_file_open_mode
- errinfo_nested_exception
- errinfo_type_info_name

error_info::error_info

```
error_info( value_type const & v );
```

Effects:

Stores a copy of v.

Throws:

Whatever T's copy constructor throws.

error_info::value_type

```
typedef T value_type;
```

This type is the same as the `error_info` T parameter.

error_info::value

```
value_type const & value() const;
value_type & value();
```

Returns:

A reference to the copy of the value passed to the `error_info` constructor.

Throws:

Nothing.

---

errinfo_api_function

#include <boost/exception/errinfo_api_function.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
```

```
{
    typedef error_info<struct errinfo_api_function_,char const *>
errinfo_api_function;
}
```

This type is designed to be used as a standard `error_info` instance for transporting the name of a failed API function in exceptions deriving from `boost::exception`.

---

errinfo_at_line

#include <boost/exception/errinfo_at_line.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    typedef error_info<struct errinfo_at_line_,int> errinfo_at_line;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a line number in exceptions deriving from `boost::exception`.

---

errinfo_errno

#include <boost/exception/errinfo_errno.hpp>

```
#include <boost/exception/error_info.hpp>
#include <errno.h>

namespace boost
{
    typedef error_info<struct errinfo_errno_,int> errinfo_errno;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a relevant errno value in exceptions deriving from `boost::exception`.

Example:

```
#include <boost/exception/errinfo_api_function.hpp>
#include <boost/exception/errinfo_at_line.hpp>
#include <boost/exception/errinfo_errno.hpp>
#include <boost/exception/errinfo_file_handle.hpp>
#include <boost/exception/errinfo_file_name.hpp>
#include <boost/exception/errinfo_file_open_mode.hpp>
#include <boost/exception/info.hpp>
#include <boost/throw_exception.hpp>
```

```
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
#include <stdio.h>
#include <errno.h>
#include <exception>

struct error : virtual std::exception, virtual boost::exception { };
struct file_error : virtual error { };
struct file_open_error: virtual file_error { };
struct file_read_error: virtual file_error { };

boost::shared_ptr<FILE>
open_file( char const * file, char const * mode )
    {
    if( FILE * f=fopen(file,mode) )
        return boost::shared_ptr<FILE>(f,fclose);
    else
        BOOST_THROW_EXCEPTION(
            file_open_error() <<
            boost::errinfo_api_function("fopen") <<
            boost::errinfo_errno(errno) <<
            boost::errinfo_file_name(file) <<
            boost::errinfo_file_open_mode(mode) );
    }

size_t
read_file( boost::shared_ptr<FILE> const & f, void * buf, size_t size )
    {
    size_t nr=fread(buf,1,size,f.get());
    if( ferror(f.get()) )
        BOOST_THROW_EXCEPTION(
            file_read_error() <<
            boost::errinfo_api_function("fread") <<
            boost::errinfo_errno(errno) <<
            boost::errinfo_file_handle(f) );
    return nr;
    }
```

errinfo_file_handle

#include <boost/exception/errinfo_file_handle.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    template <class> class weak_ptr;
    typedef error_info<struct errinfo_file_handle_,weak_ptr<FILE> >
errinfo_file_handle;
```

```
}
```

This type is designed to be used as a standard `error_info` instance for transporting a FILE pointer in exceptions deriving from `boost::exception`.

`errinfo_file_name`

#include <boost/exception/errinfo_file_name.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_file_name_,std::string> errinfo_file_name;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a file name in exceptions deriving from `boost::exception`.

`errinfo_file_open_mode`

#include <boost/exception/errinfo_file_open_mode.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_file_open_mode_,std::string>
errinfo_file_open_mode;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a fopen file open mode in exceptions deriving from `boost::exception`.

`errinfo_nested_exception`

#include <boost/exception/errinfo_nested_exception.hpp>

```
#include <boost/exception/error_info.hpp>

namespace boost
{
    typedef ---unspecified--- exception_ptr;
```

```
    typedef error_info<struct errinfo_nested_exception_,exception_ptr>
errinfo_nested_exception;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a nested exception in exceptions deriving from `boost::exception`.

---

errinfo_type_info_name

#include <boost/exception/errinfo_type_info_name.hpp>

```
#include <boost/exception/error_info.hpp>
#include <string>

namespace boost
{
    typedef error_info<struct errinfo_type_info_name_,std::string>
errinfo_type_info_name;
}
```

This type is designed to be used as a standard `error_info` instance for transporting a type name in exceptions deriving from `boost::exception`.

---

exception

#include <boost/exception/exception.hpp>

```
namespace boost
{
    class exception
    {
    protected:

        exception();
        exception( exception const & x );
        ~exception();
    };
}
```

Class `boost::exception` is designed to be used as a universal base for user-defined exception types.

An object of any type deriving from `boost::exception` can store data of arbitrary types, using the `error_info` wrapper and `operator<<`.

To retrieve data from a `boost::exception` object, use the `get_error_info` function template.

`exception::exception`

```
exception();
exception( exception const & x );
```

Effects:

- Default constructor: initializes an empty `boost::exception` object.
- Copy constructor: initializes a `boost::exception` object which shares with x the pointers to all currently stored data. Subsequently, data can be added to or retrieved from both exception objects interchangeably, however doing so concurrently from multiple threads is undefined behavior.

Throws:

Nothing.

`exception::~exception`

```
~exception();
```

Effects:

Releases all resources associated with the `boost::exception` object.

Throws:

Nothing.

`exception_ptr`

#include <boost/exception_ptr.hpp>

```
namespace boost
{
    typedef ---unspecified--- exception_ptr;
}
```

The `exception_ptr` type can be used to refer to a copy of an exception object. It is Default Constructible, Copy Constructible, Assignable and Equality Comparable; `exception_ptr's operations do not throw.

The referenced object remains valid at least as long as there is an `exception_ptr` object that refers to it.

Two instances of `exception_ptr` are equivalent and compare equal if and only if they refer to the same exception.

The default constructor of `exception_ptr` produces the null value of the type. The null value is equivalent only to itself.

Thread safety:

The `exception_ptr` type is "as thread-safe as built-in types":

- An `exception_ptr` instance can be "read" simultaneously by multiple threads
- Different `exception_ptr` instances can be "written to" simultaneously by multiple threads, even when these instances refer to the same exception object

All other simultaneous accesses result in undefined behavior.

Nesting of exceptions:

An `exception_ptr` can be added as `error_info` to any `boost::exception`. This is a convenient way to nest exceptions. There is no limit on the depth of the nesting, however cyclic references result in undefined behavior.

`nlohmann_json_encoder`

#include <boost/exception/serialization/nlohmann_json_encoder.hpp>

```
namespace boost { namespace exception_serialization {

template <class Json>
struct nlohmann_json_encoder
{
    Json & j_;

    // Uses unspecified SFINAE expression designed to make the
    // overload selected only if no other compatible overload is found.
    // Implemented in terms of to_json.
    template <class T>
    friend void output( nlohmann_json_encoder &, T const & x );

    template <class T>
    friend void output_at( nlohmann_json_encoder &, T const &, char const * name );
};

} }
```

The `nlohmann_json_encoder` type serializes objects to JSON format based on ADL calls to `to_json`. This is compatible with nlohmann/json.

See Serialization.

`original_exception_type`

#include <boost/exception_ptr.hpp>

```
namespace boost
```

```
{
    typedef error_info<struct tag_original_exception_type,std::type_info const *>
original_exception_type;
}
```

This type is used by the `exception_ptr` support in Boost Exception. Please see `current_exception`.

---

unknown_exception

#include <boost/exception_ptr.hpp>

```
namespace boost
{
    class unknown_exception:
        public std::exception
        public boost::exception
    {
        ---unspecified---
    };
}
```

This type is used by the `exception_ptr` support in Boost Exception. Please see `current_exception`.

Functions

copy_exception

#include <boost/exception_ptr.hpp>

```
namespace boost
{
    template <class T>
    exception_ptr copy_exception( T const & e );
}
```

Effects:

As if

```
try
    {
    throw enable_current_exception(e);
    }
catch(...)
```

```
    {
    return current_exception();
    }
```

`current_exception`

#include <boost/exception_ptr.hpp>

```
namespace boost
{
    exception_ptr current_exception();
}
```

Requirements:

The `current_exception` function must not be called outside of a catch block.

In addition, to safely copy an exception from one thread to another, if the exception object is copied by `current_exception` or `copy_exception`, the two copies must not have shared state. Exceptions that have value-type semantics (as well as the `boost::exception` type itself) satisfy this requirement.

Returns:

- An `exception_ptr` that refers to the currently handled exception or a copy of the currently handled exception.
- If the function needs to allocate memory and the attempt fails, it returns an `exception_ptr` that refers to an instance of std::bad_alloc.

Throws:

Nothing.

Notes:

- It is unspecified whether the return values of two successive calls to `current_exception` refer to the same exception object.
- Correct implementation of `current_exception` may require compiler support (e.g. C++11 std::current_exception() is used when available, as specified by Boost.Config BOOST_NO_CXX11_HDR_EXCEPTION), unless `enable_current_exception` was used at the time the currently handled exception object was passed to throw. Whenever `current_exception` fails to properly copy the current exception object, it returns an `exception_ptr` to an object of type that is as close as possible to the original exception type, using `unknown_exception` as a final fallback. All such types derive from `boost::exception`, and:
  - if the original exception object derives from `boost::exception`, then the `boost::exception` sub-object of the object referred to by the returned `exception_ptr` is initialized by the `boost::exception` copy constructor;
  - if available, the exception contains the std::type_info of the original exception object,

accessible through `get_error_info`<`original_exception_type`>.

`current_exception_cast`

#include <boost/exception/current_exception_cast.hpp>

```
namespace boost
{
    template <class E>
    E * current_exception_cast();
}
```

Requirements:

This function must not be called outside of a catch block.

Returns:

A pointer of type E to the current exception object, or null if the current exception object can not be converted to E *.

Throws:

Nothing.

`current_exception_diagnostic_information`

#include <boost/exception/diagnostic_information.hpp>

```
namespace boost
{
    std::string current_exception_diagnostic_information();
}
```

Requirements:

This function must not be called outside of a catch block.

Returns:

If the current exception object can be converted to `boost::exception` or std::exception, this function returns the same string value returned by `diagnostic_information` for the current exception object. Otherwise, an unspecified non-empty string is returned.

Typical use is to call `current_exception_diagnostic_information` from a top-level function to output diagnostic information about unhandled exceptions:

```
int
main()
    {
```

```
    try
        {
        run_program();
        }
    catch(
    error & e )
        {
        //handle error
        }
    catch(
    ...)
        {
        std::cerr << "Unhandled exception!" << std::endl <<
            boost::current_exception_diagnostic_information();
        }
    }
```

diagnostic_information

#include <boost/exception/diagnostic_information.hpp>

```
namespace boost
{
    template <class E>
    std::string diagnostic_information( E const & e, bool verbose=true );

    std::string diagnostic_information( exception_ptr const & p, bool verbose=true );
}
```

Returns:

A string value that contains varying amount of diagnostic information about the passed object:

- If E can be statically converted to either `boost::exception` or to std::exception, dynamic_cast is used to access both the `boost::exception` and std::exception subobjects of e; otherwise, the `boost::diagnostic_information` template is not available.
- The returned value contains the string representations of all `error_info` objects stored in the `boost::exception` subobject through `operator<<`.
- In addition, if verbose is true, it contains other diagnostic information relevant to the exception, including the string returned by std::exception::what().

The string representation of each `error_info` object is deduced by an unqualified call to to_string(x), where x is of type `error_info<Tag,T>`, for which Boost Exception defines a generic overload. It converts x.`value`() to string, attempting to bind (at the time the `error_info<Tag,T>` template is instantiated) the following functions in order:

1. Unqualified call to to_string(x.`value`()) (the return value is expected to be of type std::string.)

2. Unqualified call to s << x.value(), where s is a std::ostringstream.

The first successfully bound function is used at the time `diagnostic_information` is called; if both overload resolutions are unsuccessful, the system is unable to convert the `error_info` object to string, and *an unspecified stub string value is used without issuing a compile error.*

The `exception_ptr` overload of `diagnostic_information` is equivalent to:

```
if( p )
    try
        {
        rethrow_exception(p);
        }
    catch(...)
        {
        return current_exception_diagnostic_information(verbose);
        }
else return <unspecified-string-value>;
```

Example:

this is a possible output from the `diagnostic_information` function, as used in *libs/exception/example/example_io.cpp:*

```
example_io.cpp(70): Throw in function class boost::shared_ptr<struct _iobuf> __cdecl
my_fopen(const char *,const char *)
Dynamic exception type: class boost::exception_detail::clone_impl<struct fopen_error>
std::exception::what: example_io error
[struct boost::errinfo_api_function_ *] = fopen
[struct boost::errinfo_errno_ *] = 2, "No such file or directory"
[struct boost::errinfo_file_name_ *] = tmp1.txt
[struct boost::errinfo_file_open_mode_ *] = rb
```

`diagnostic_information_what`

#include <boost/exception/diagnostic_information.hpp>

```
namespace boost
{
    char const * diagnostic_information_what( boost::exception const & e, bool
verbose=true ) throw();
}
```

The `diagnostic_information_what` function is intended to be called from a user-defined std::exception::what() override. This allows diagnostic information to be returned as the what() string.

Returns:

A pointer to a zero-terminated buffer that contains a string similar to the std::string returned by the `diagnostic_information` function, or null to indicate a failure.

Throws:

Nothing.

Note:

The returned pointer becomes invalid if any `error_info` is modified or added to the exception object, or if another diagnostic information function is called.

`enable_current_exception`

#include <boost/exception/enable_current_exception.hpp>

```
namespace boost
{
    template <class T>
    ---unspecified--- enable_current_exception( T const & e );
}
```

Requirements:

- T must be a class with an accessible no-throw copy constructor.
- If T has any virtual base types, those types must have an accessible default constructor.

Returns:

An object of *unspecified* type which derives publicly from T. That is, the returned object can be intercepted by a catch(T &).

This function is designed to be used directly in a throw-expression to enable the `exception_ptr` support in Boost Exception. For example:

```
class
my_exception:
    public std::exception
    {
    };

....
throw boost::enable_current_exception(my_exception());
```

Unless `enable_current_exception` is called at the time an exception object is used in a throw-expression, an attempt to copy it using `current_exception` may return an `exception_ptr` which refers to an instance of `unknown_exception`. See `current_exception` for details.

Note:

Instead of using the throw keyword directly, it is preferable to call `boost::throw_exception`. This is guaranteed to throw an exception that derives from `boost::exception` and supports the `exception_ptr` functionality.

`enable_error_info`

#include <boost/exception/enable_error_info.hpp>

```
namespace boost
{
    template <class T>
    ---unspecified--- enable_error_info( T const & x );
}
```

Requirements:

T must be a class with an accessible no-throw copy constructor as per (15.5.1).

Returns:

- If T derives from `boost::exception`, the returned object is of type T and is a copy of x.
- Otherwise, the returned object is of an unspecified type that derives publicly from both T and `boost::exception`. The T sub-object is initialized from x by the T copy constructor.

Throws:

Nothing.

`get_error_info`

#include <boost/exception/get_error_info.hpp>

```
namespace boost
{
    template <class ErrorInfo,class E>
    typename ErrorInfo::value_type const * get_error_info( E const & x );

    template <class ErrorInfo,class E>
    typename ErrorInfo::value_type * get_error_info( E & x );
}
```

Requirements:

- ErrorInfo must be an instance of the `error_info` template.
- E must be polymorphic.

Returns:

- If dynamic_cast<boost::exception const *>(&x) is 0, or if x does not store an object of type

ErrorInfo, the returned value is null.

- Otherwise, the returned pointer points to the stored value (use `operator<<` to store values in exception objects.) When x is destroyed, any pointers returned by `get_error_info` become invalid.

Throws:

Nothing.

Note:

The interface of `get_error_info` may be affected by the build configuration macros.

`operator<<`

#include <boost/exception/info.hpp>

```
namespace boost
{
    template <class E, class Tag, class T>
    E const & operator<<( E const & x, error_info<Tag,T> const & v );
}
```

Requirements:

E must be `boost::exception`, or a type that derives (indirectly) from `boost::exception`.

Postcondition:

A copy of v is stored into x. If x already contains data of type `error_info<Tag,T>`, that data is overwritten. Basic exception safety guarantee.

Returns:

x.

Throws:

std::bad_alloc, or any exception emitted by the T copy constructor.

`rethrow_exception`

#include <boost/exception_ptr.hpp>

```
namespace boost
{
    void rethrow_exception( exception_ptr const & ep );
}
```

Precondition:

ep shall not be null.

Throws:

The exception to which ep refers.

`serialize`

```
namespace boost { namespace exception_serialization {

template <class Handle, class E>
void serialize( Handle & h, E const & x, char const * name );

} }
```

The `serialize` function template is a user-defined customization point. If provided, it is called by the serialization system to output objects to an encoder; see Custom Encoders.

`serialize_current_exception_diagnostic_information_to`

#include <boost/exception/diagnostic_information.hpp>

```
namespace boost
{
    template <class Encoder>
    void serialize_current_exception_diagnostic_information_to( Encoder & enc );
}
```

Requirements:

This function must not be called outside of a catch block.

Effects:

Serializes diagnostic information for the current exception to the encoder `enc`. Equivalent to calling `serialize_diagnostic_information_to` with the current exception object.

See Serialization.

`serialize_diagnostic_information_to`

#include <boost/exception/diagnostic_information.hpp>

```
namespace boost
{
    template <class Encoder, class E>
    void serialize_diagnostic_information_to( Encoder & enc, E const & e );
}
```

Serializes diagnostic information for the exception `e` to the encoder `enc`.

Requirements:

E must be convertible to either `boost::exception const &` or `std::exception const &`.

Effects:

Outputs diagnostic information about the exception to the encoder, including:

- Throw location (file, line, function) if available
- Dynamic exception type (if RTTI is enabled)
- `std::exception::what()` if available
- All `error_info` objects stored in the exception

See Serialization.

`throw_exception`

Please see Boost.ThrowException.

`tuple/operator<<`

#include <boost/exception/info_tuple.hpp>

```
namespace boost
{
    template <class E, class Tag1, class T1, ..., class TagN, class TN>
    E const & operator<<( E const & x,
        tuple<
            error_info<Tag1,T1>,
            ...,
            error_info<TagN,TN> > const & v );
}
```

Requirements:

E must be `boost::exception`, or a type that derives (indirectly) from `boost::exception`.

Effects:

Equivalent to x << v.http://www.boost.org/libs/tuple/doc/tuple_users_guide.html#accessing_elements[get]<0>() << ... << v.http://www.boost.org/libs/tuple/doc/tuple_users_guide.html#accessing_elements[get]<N>().

Returns:

x.

Throws:

std::bad_alloc, or any exception emitted by T1..TN copy constructor.

Macros

`BOOST_THROW_EXCEPTION`

Please see Boost.ThrowException.

Configuration Macros

Boost Exception responds to the following configuration macros:

BOOST_NO_RTTI
BOOST_NO_TYPEID (both defined automatically by boost/config.hpp)

The first macro prevents Boost Exception from using dynamic_cast and dynamic typeid. If the second macro is also defined, Boost Exception does not use static typeid either. There are no observable degrading effects on the library functionality, except for the following: by default, the `get_error_info` function template can be called with any exception type; if BOOST_NO_RTTI is defined, `get_error_info` can be used only with objects of type `boost::exception`.

Note:

The library needs RTTI functionality. Disabling the language RTTI support enables an internal RTTI system, which may have more or less overhead depending on the platform.

Note that on some non-conformant compilers, for example MSVC 7.0 and older, as well as BCC, BOOST_EXCEPTION_DISABLE is implicitly defined in `boost/throw_exception.hpp`.

BOOST_NO_EXCEPTIONS (defined automatically by boost/config.hpp)

This macro disables exception handling in Boost, forwarding all exceptions to a user-defined non-template version of `boost::throw_exception`. However, unless BOOST_EXCEPTION_DISABLE is also defined, users can still examine the exception object for any data added at the point of the throw, or use `boost::diagnostic_information` (of course under BOOST_NO_EXCEPTIONS, the user-defined boost::throw_exception is not allowed to return to the caller.)

In addition, the following user-defined macros are recognized:

BOOST_EXCEPTION_DISABLE (user-defined)

By default, `enable_current_exception` and `enable_error_info` are integrated directly in the `throw_exception` function. Defining BOOST_EXCEPTION_DISABLE disables this integration.

Design Rationale

Traditionally, when using exceptions to report failures, the throw site:

- creates an exception object of the appropriate type, and
- stuffs it with data relevant to the detected error.

A higher context in the program contains a catch statement which:

- selects failures based on exception types, and
- inspects exception objects for data required to deal with the problem.

The main issue with this "traditional" approach is that often, the data available at the point of the throw is insufficient for the catch site to handle the failure.

Here is an example of a catch statement:

```
catch( file_read_error & e )
    {
    std::cerr << e.file_name();
    }
```

And here is a possible matching throw:

```
void
read_file( FILE * f )
    {
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
        throw file_read_error(???);
    ....
    }
```

Clearly, the problem is that the handler requires a file name but the read_file function does not have a file name to put in the exception object; all it has is a FILE pointer!

In an attempt to deal with this problem, we could modify read_file to accept a file name:

```
void
read_file( FILE * f, char const * name )
    {
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
        throw file_read_error(name);
    ....
```

```
    }
```

This is not a real solution: it simply shifts the burden of supplying a file name to the immediate caller of the read_file function.

In general, the data required to handle a given library-emitted exception depends on the program that links to it. Many contexts between the throw and the catch may have relevant information which must be transported to the exception handler.

Exception wrapping

The idea of exception wrapping is to catch an exception from a lower level function (such as the read_file function above), and throw a new exception object that contains the original exception (and also carries a file name.) This method seems to be particularly popular with C++ programmers with Java background.

Exception wrapping leads to the following problems:

- To wrap an exception object it must be copied, which may result in slicing.
- Wrapping is practically impossible to use in generic contexts.

The second point is actually special case of violating the exception neutrality principle. Most contexts in a program can not handle exceptions; such contexts should not interfere with the process of exception handling.

The boost::exception solution

- Simply derive your exception types from `boost::exception`.
- Confidently limit the throw site to provide only data that is available naturally.
- Use exception-neutral contexts between the throw and the catch to augment exceptions with more relevant data as they bubble up.

For example, in the throw statement below we only add the errno code, since this is the only failure-relevant information available in this context:

```
struct exception_base: virtual std::exception, virtual boost::exception { };
struct io_error: virtual exception_base { };
struct file_read_error: virtual io_error { };

typedef boost::error_info<struct tag_errno_code,int> errno_code;

void
read_file( FILE * f )
    {
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
```

```
        throw file_read_error() << errno_code(errno);
    ....
    }
```

exception | error_info | operator<<

In a higher exception-neutral context, we add the file name to *any* exception that derives from `boost::exception`:

```
typedef boost::error_info<struct tag_file_name,std::string> file_name;

....
try
    {
    if( FILE * fp=fopen("foo.txt","rt") )
        {
        shared_ptr<FILE> f(fp,fclose);
        ....
        read_file(fp); //throws types deriving from boost::exception
        do_something();
        ....
        }
    else
        throw file_open_error() << errno_code(errno);
    }
catch( boost::exception & e )
    {
    e << file_name("foo.txt");
    throw;
    }
```

exception | error_info | operator<<

Finally here is how the handler retrieves data from exceptions that derive from `boost::exception`:

```
catch( io_error & e )
    {
    std::cerr << "I/O Error!\n";

    if( std::string const * fn=get_error_info<file_name>(e) )
        std::cerr << "File name: " << *fn << "\n";

    if( int const * c=get_error_info<errno_code>(e) )
        std::cerr << "OS says: " << strerror(*c) << "\n";
    }
```

get_error_info

In addition, `boost::diagnostic_information` can be used to compose an automatic (if not user-friendly) message that contains all of the `error_info` objects added to a `boost::exception`. This is useful for inclusion in logs and other diagnostic objects.

Frequently Asked Questions

What is the cost of calling boost::throw_exception?

The cost is that `boost::exception` is added as a base of the exception emitted by `boost::throw_exception` (unless the passed type already derives from `boost::exception`.)

Calling `boost::throw_exception` does not cause dynamic memory allocations.

What is the cost of BOOST_THROW_EXCEPTION?

In addition to calling `boost::throw_exception`, `BOOST_THROW_EXCEPTION` invokes __FILE__ and __LINE__ macros. The space required to store the information is already included in sizeof(boost::exception).

Calling `BOOST_THROW_EXCEPTION` does not cause dynamic memory allocations.

Should I use boost::throw_exception or BOOST_THROW_EXCEPTION or just throw?

The benefit of calling `boost::throw_exception` instead of using throw directly is that it ensures that the emitted exception derives from `boost::exception` and that it is compatible with `boost::current_exception`.

The `BOOST_THROW_EXCEPTION` macro also results in a call to `boost::throw_exception`, but in addition it records in the exception object the __FILE__ and __LINE__ of the throw, as well as the pretty name of the function that throws. This enables `boost::diagnostic_information` to compose a more useful, if not user-friendly message.

Typical use of `boost::diagnostic_information` is:

```
catch(...)
    {
    std::cerr <<
        "Unexpected exception, diagnostic information follows:\n" <<
        current_exception_diagnostic_information();
    }
```

This is a possible message it may display — the information in the first line is only available if `BOOST_THROW_EXCEPTION` was used to throw:

```
example_io.cpp(70): Throw in function class boost::shared_ptr<struct _iobuf> __cdecl
my_fopen(const char *,const char *)
Dynamic exception type: class boost::exception_detail::clone_impl<class fopen_error>
std::exception::what: example_io error
[struct boost::errinfo_api_function_ *] = fopen
```

```
[struct boost::errinfo_errno_ *] = 2, "No such file or directory"
[struct boost::errinfo_file_name_ *] = tmp1.txt
[struct boost::errinfo_file_open_mode_ *] = rb
```

In some development environments, the first line in that message can be clicked to show the location of the throw in the debugger, so it's easy to set a break point and run again to see the unexpected throw in the context of its call stack.

Why doesn't boost::exception derive from std::exception?

Despite that virtual inheritance should be used in deriving from base exception types, quite often exception types (including the ones defined in the standard library) don't derive from std::exception virtually.

If `boost::exception` derives from std::exception, using the `enable_error_info` function with such user-defined types would introduce dangerous ambiguity which would break all catch(std::exception &) statements.

Of course, `boost::exception` should not be used to replace std::exception as a base type in exception type hierarchies. Instead, it should be included as a virtual base, in addition to std::exception (which should probably also be derived virtually.)

Why is boost::exception abstract?

To prevent exception-neutral contexts from erroneously erasing the type of the original exception when adding `error_info` to an active exception object:

```
catch( boost::exception & e )
    {
    e << foo_info(foo);
    throw e; //Compile error: boost::exception is abstract
    }
```

The correct code is:

```
catch( boost::exception & e )
    {
    e << foo_info(foo);
    throw; //Okay, re-throwing the original exception object.
    }
```

Why use operator<< overload for adding info to exceptions?

Before throwing an object of type that derives from `boost::exception`, it is often desirable to add one or more `error_info` objects in it. The syntactic sugar provided by `operator<<` allows this to be done directly in a throw expression:

```
throw error() << foo_info(foo) << bar_info(bar);
```

Why is operator<< allowed to throw?

This question is referring to the following issue. Consider this throw statement example:

```
throw file_open_error() << file_name(fn);
```

The intention here is to throw a file_open_error, however if `operator<<` fails to copy the std::string contained in the file_name `error_info` wrapper, a std::bad_alloc could propagate instead. This behavior seems undesirable to some programmers.

> "Throwing an exception requires an object to throw. A C++ implementation is required to have enough spare memory to be able to throw bad_alloc in case of memory exhaustion. However, it is possible that throwing some other exception will cause memory exhaustion."
>
> — Bjarne Stroustrup, The C++ Programming Language 3rd Edition p. 371

Therefore, the language itself does not guarantee that an attempt to throw an exception is guaranteed to throw an object of the specified type; propagating a std::bad_alloc seems to be a possibility even outside of the scope of Boost Exception.

Acknowledgements

Thanks to Peter Dimov for his continuing help. Also thanks to Tobias Schwinger, Tom Brinkman, Pavel Vozenilek and everyone who participated in the review process.

Copyright 2006-2026 Emil Dotchevski and Reverge Studios, Inc.

Distributed under the Boost Software License, Version 1.0.