LEAF

Lightweight Error Augmentation Framework written in C++11 | Emil Dotchevski

Abstract

Boost LEAF is a lightweight error handling library for C++11. Features:

- Portable single-header format, no dependencies.
- Tiny code size, configurable for embedded development.
- No dynamic memory allocations, even with very large payloads.
- Deterministic unbiased efficiency on the "happy" path and the "sad" path.
- Error objects are handled in constant time, independent of call stack depth.
- Can be used with or without exception handling.

Support

- Report issues on GitHub

Distribution

LEAF is distributed under the Boost Software License, Version 1.0.

There are three distribution channels:

- LEAF is included in official Boost releases (starting with Boost 1.75), and therefore available via most package managers.
- The source code is hosted on GitHub.
- For maximum portability, the latest LEAF release is also available in single-header format: leaf.hpp (direct download link).

LEAF does not depend on Boost or other libraries.

Tutorial

Typically, error handling libraries define a variant result type, e.g. `result<T, E>`. In LEAF we drop the `E`, using just `result<T>`.

In case of success, access to the value `T` is immediate and direct, and we can easily bail out in case of a failure.

To handle errors, we use a special syntax, which enables error objects to be transported directly to the error handling scopes that need them. This is more efficient than holding them in result types, and it also allows any given handler to access multiple error objects associated with the same failure.

LEAF is also compatible with exception handling, providing identical functionality but without needing a result type.

Reporting Errors

Let's introduce the `result<T>`-based interface first.

Report errors with `leaf::new_error`:

```
enum class api_error { connect_failed, invalid_request, timeout };

leaf::result<connection> connect()
{
  ....
  if( <<connection-failure-detected>> )
    return leaf::new_error( api_error::connect_failed ); // Pass error objects of any
type

  // Produce and return a connection object.
}
```

result | new_error

Checking for Errors

To bail out on failure, return `result::error()`:

```
leaf::result<data> connect_and_fetch_data()
{
  leaf::result<connection> cr = connect();
  if( !cr )
    return cr.error();
```

```
  connection & c = cr.value();

  // Use c to fetch and return data
}
```

result

Use `BOOST_LEAF_AUTO` to avoid the boilerplate `if` statement:

```
leaf::result<data> connect_and_fetch_data()
{
  BOOST_LEAF_AUTO(c, connect()); // Bail out on error

  // Use c to fetch and return data
}
```

BOOST_LEAF_AUTO

Use `BOOST_LEAF_CHECK` in case of `void` results:

```
leaf::result<void> flush(connection & c);
leaf::result<std::size_t> bytes_sent(connection & c);

leaf::result<std::size_t> flush_and_get_bytes_sent(connection & c)
{
  BOOST_LEAF_CHECK(flush(c)); // Bail out on error
  return bytes_sent(c);
}
```

BOOST_LEAF_CHECK

On implementations that define `__GNUC__` (e.g. GCC/clang), `BOOST_LEAF_CHECK` is compatible with non-`void` results as well:

```
leaf::result<int> count_rows();

float update_average(int n);

leaf::result<float> average_row_count()
{
  return update_average( BOOST_LEAF_CHECK(count_rows()) );
}
```

The following is the portable alternative:

```
leaf::result<float> average_row_count()
{
  BOOST_LEAF_AUTO(n, count_rows());
```

```
  return update_average(n);
}
```

Error Handling

Error handling scopes use a special syntax to indicate that they need to access error objects:

```
leaf::result<data> r = leaf::try_handle_some(

  []() -> leaf::result<data>
  {
    BOOST_LEAF_AUTO(c, connect());
    return fetch(c);
  },

  []( api_error e ) -> leaf::result<data>
  {
    if( e == api_error::connect_failed )
      .... // Handle api_error::connect_failed
    else
      .... // Handle any other api_error value
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

First, `try_handle_some` executes the first function passed to it; it attempts to produce a `result<data>`, but it may fail.

The second lambda is an error handler: it will be called iff the first lambda fails with an error object of type `api_error`. That object is stored on the stack, local to the `try_handle_some` function (LEAF knows to allocate this storage because we gave it an error handler that takes an `api_error`). Error handlers passed to `leaf::try_handle_some` can return a valid `leaf::result` but are allowed to fail.

It is possible for an error handler to declare that it can only handle some specific values of a given error type:

```
leaf::result<data> r = leaf::try_handle_some(

  []() -> leaf::result<data>
  {
    BOOST_LEAF_AUTO(c, connect());
    return fetch(c);
  },

  []( leaf::match<api_error, api_error::connect_failed, api_error::timeout> ) ->
```

```
leaf::result<data>
  {
    // Handle api_error::connect_failed or api_error::timeout
  },

  []( api_error e ) -> leaf::result<data>
  {
    // Handle any other api_error value
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO | match

LEAF considers the provided error handlers in order, and calls the first one for which it is able to supply arguments, based on the error objects currently being communicated. Above:

- The first error handler will be called iff an error object of type `api_error` is available, and its value is either `api_error::connect_failed` or `api_error::timeout`.
- Otherwise the second error handler will be called iff an error object of type `api_error` is available, regardless of its value.
- Otherwise `leaf::try_handle_some` is unable to handle the error.

It is possible for an error handler to conditionally leave the failure unhandled:

```
leaf::result<data> r = leaf::try_handle_some(

  []() -> leaf::result<data>
  {
    BOOST_LEAF_AUTO(c, connect());
    return fetch(c);
  },

  []( api_error e, leaf::error_info const & ei ) -> leaf::result<data>
  {
    if( e == api_error::timeout )
      return cached_data();
    else
      return ei.error();
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO | error_info

Any error handler can take an argument of type `leaf::error_info const &` to get access to generic information about the error being handled; in this case we use the `error` member function, which returns the unique error_id of the current error; we use it to initialize the returned `leaf::result`, effectively propagating the current error out of `try_handle_some`.

If we wanted to signal a new error (rather than propagating the current error), in the `return` statement we would invoke the `leaf::new_error` function.

If we want to ensure that all possible failures are handled, we use `leaf::try_handle_all` instead of `leaf::try_handle_some`:

```
data r = leaf::try_handle_all(

  []() -> leaf::result<data>
  {
    BOOST_LEAF_AUTO(c, connect());
    return fetch(c);
  },

  []( leaf::match<api_error, api_error::connect_failed> ) -> data
  {
    // Handle api_error::connect_failed
  },

  []( api_error e ) -> data
  {
    // Handle any other api_error value
  },

  []() -> data
  {
    // Handle any other failure
  } );
```

try_handle_all

The `leaf::try_handle_all` function enforces at compile time that at least one of the supplied error handlers takes no arguments (and therefore is able to handle any failure). In addition, all error handlers are forced to return a valid object rather than a `leaf::result`, so that `leaf::try_handle_all` is guaranteed to succeed.

Working with Different Error Types

It is of course possible to provide different handlers for different error types:

```
enum class api_error { connect_failed, invalid_request, timeout };
enum class auth_error { unauthorized, forbidden };

....

leaf::result<data> r = leaf::try_handle_some(

  []() -> leaf::result<data>
  {
    BOOST_LEAF_AUTO(c, connect());
```

```
    return fetch(c);
  },

  []( api_error e ) -> leaf::result<data>
  {
    // Handle errors of type `api_error`.
  },

  []( auth_error e ) -> leaf::result<data>
  {
    // Handle errors of type `auth_error`.
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

Error handlers are always considered in order:

- The first error handler will be used if an error object of type `api_error` is available;
- otherwise, the second error handler will be used if an error object of type `auth_error` is available;
- otherwise, `leaf::try_handle_some` fails.

Working with Multiple Error Objects

The `leaf::new_error` function can be invoked with multiple error objects, for example to communicate an error code and the relevant file name:

```
enum class io_error { open_error, read_error, write_error };

struct e_file_name { std::string value; };

leaf::result<File> open_file( char const * name )
{
  ....
  if( open_failed )
    return leaf::new_error(io_error::open_error, e_file_name {name});
  ....
}
```

result | new_error

Similarly, error handlers may take multiple error objects as arguments:

```
leaf::result<T> r = leaf::try_handle_some(

  []() -> leaf::result<T>
```

```
  {
    BOOST_LEAF_AUTO(f, open_file(fn));
    ....
  },

  []( io_error ec, e_file_name fn ) -> leaf::result<T>
  {
    // Handle I/O errors when a file name is also available.
  },

  []( io_error ec ) -> leaf::result<T>
  {
    // Handle I/O errors when no file name is available.
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

Once again, error handlers are considered in order:

- The first error handler will be used if an error object of type `io_error` *and* an error object of type `e_file_name` are available;
- otherwise, the second error handler will be used if an error object of type `io_error` is available;
- otherwise, `leaf::try_handle_some` fails.

An alternative way to write the above is to provide a single error handler that takes the `e_file_name` argument as a pointer:

```
leaf::result<T> r = leaf::try_handle_some(

  []() -> leaf::result<T>
  {
    BOOST_LEAF_AUTO(f, open_file(fn));
    ....
  },

  []( io_error ec, e_file_name const * fn ) -> leaf::result<T>
  {
    if( fn )
      .... // Handle I/O errors when a file name is also available.
    else
      .... // Handle I/O errors when no file name is available.
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

An error handler is never dropped for lack of error objects of types which the handler takes as pointers; in this case LEAF simply passes `nullptr` for these arguments.

When an error handler takes arguments by mutable reference or pointer, changes to their state are preserved when the error is communicated to the caller.

Augmenting Errors

Let's say we have a function `parse_line` which could fail due to an `io_error` or a `parse_error`:

```
enum class io_error { open_error, read_error, write_error };

enum class parse_error { bad_syntax, bad_range };

leaf::result<int> parse_line( FILE * f );
```

The `leaf::on_error` function can be used to automatically associate additional error objects with any failure that is "in flight":

```
struct e_line { int value; };

leaf::result<void> process_file( FILE * f )
{
  for( int current_line = 1; current_line != 10; ++current_line )
  {
    auto load = leaf::on_error( e_line {current_line} );

    BOOST_LEAF_AUTO(v, parse_line(f));

    // use v
  }
}
```

on_error | BOOST_LEAF_AUTO

Because `process_file` does not handle errors, it remains neutral to failures, except to attach the `current_line` if something goes wrong. The object returned by `on_error` holds a copy of `current_line` wrapped in `struct e_line`. If `parse_line` succeeds, the `e_line` object is simply discarded; if it fails, the `e_line` object will be automatically "attached" to the failure.

Such failures can then be handled like so:

```
leaf::result<void> r = leaf::try_handle_some(

  [&]() -> leaf::result<void>
  {
    BOOST_LEAF_CHECK( process_file(f) );
  },

  []( parse_error e, e_line current_line  )
```

```
  {
    std::cerr << "Parse error at line " << current_line.value << std::endl;
  },

  []( io_error e, e_line current_line )
  {
    std::cerr << "I/O error at line " << current_line.value << std::endl;
  },

  []( io_error e )
  {
    std::cerr << "I/O error" << std::endl;
  } );
```

try_handle_some | BOOST_LEAF_CHECK

The following is equivalent, and perhaps simpler:

```
leaf::result<void> r = leaf::try_handle_some(

  []() -> leaf::result<void>
  {
    BOOST_LEAF_CHECK( process_file(f) );
  },

  []( parse_error e, e_line current_line )
  {
    std::cerr << "Parse error at line " << current_line.value << std::endl;
  },

  []( io_error e, e_line const * current_line )
  {
    std::cerr << "I/O error";
    if( current_line )
      std::cerr << " at line " << current_line->value;
    std::cerr << std::endl;
  } );
```

Exception Handling

What happens if an operation throws an exception? Both `try_handle_some` and `try_handle_all` catch exceptions and are able to pass them to any compatible error handler:

```
leaf::result<void> r = leaf::try_handle_some(

  []() -> leaf::result<void>
```

```
  {
    BOOST_LEAF_CHECK( process_file(f) );
  },

  []( std::bad_alloc const & )
  {
    std::cerr << "Out of memory!" << std::endl;
  },

  []( parse_error e, e_line l )
  {
    std::cerr << "Parse error at line " << l.value << std::endl;
  },

  []( io_error e, e_line const * l )
  {
    std::cerr << "I/O error";
    if( l )
      std::cerr << " at line " << l.value;
    std::cerr << std::endl;
  } );
```

try_handle_some | result | BOOST_LEAF_CHECK

Above, we have simply added an error handler that takes a `std::bad_alloc`, and everything "just works" as expected: LEAF will dispatch error handlers correctly no matter if failures are communicated via `leaf::result` or by an exception.

Of course, if we use exception handling exclusively, we do not need `leaf::result` at all. In this case we use `leaf::try_catch`:

```
leaf::try_catch(

  []
  {
    process_file(f);
  },

  []( std::bad_alloc const & )
  {
    std::cerr << "Out of memory!" << std::endl;
  },

  []( parse_error e, e_line l )
  {
    std::cerr << "Parse error at line " << l.value << std::endl;
  },

  []( io_error e, e_line const * l )
  {
```

```
    std::cerr << "I/O error";
    if( l )
      std::cerr << " at line " << l.value;
    std::cerr << std::endl;
  } );
```

try_catch

We did not have to change the error handlers! But how does this work? What kind of exceptions would `process_file` throw?

LEAF enables a novel exception handling technique, which does not require an exception type hierarchy to classify failures and does not carry data in exception objects. Recall that when failures are communicated via `leaf::result`, we call `leaf::new_error` in a `return` statement, passing any number of error objects which are sent directly to the correct error handling scope:

```
enum class api_error { connect_failed, invalid_request, timeout };
enum class auth_error { unauthorized, forbidden };

....

leaf::result<T> f()
{
  ....
  if( error_detected )
    return leaf::new_error(api_error::connect_failed, auth_error::forbidden);

  // Produce and return a T.
}
```

result | new_error

When using exception handling this becomes:

```
enum class api_error { connect_failed, invalid_request, timeout };
enum class auth_error { unauthorized, forbidden };

T f()
{
  if( error_detected )
    leaf::throw_exception(api_error::connect_failed, auth_error::forbidden);

  // Produce and return a T.
}
```

throw_exception

The `leaf::throw_exception` function handles the passed error objects just like `leaf::new_error` does, and then throws an object of a type that derives from `std::exception`.

Using this technique, the exception type is not important: `leaf::try_catch` catches all exceptions, then goes through the usual LEAF error handler selection routine.

If instead we want to use the standard convention of throwing different types to indicate different failures, we simply pass an exception object (that is, an object of a type that derives from `std::exception`) as the first argument to `leaf::throw_exception`:

```
leaf::throw_exception(std::runtime_error("Error!"), api_error::connect_failed,
auth_error::forbidden);
```

In this case the thrown exception object will be of a type that derives from `std::runtime_error`, rather than from `std::exception`.

Finally, `leaf::on_error` "just works" as well. Here is our `process_file` function rewritten to work with exceptions, rather than return a `leaf::result` (see Augmenting Errors):

```
int parse_line( FILE * f ); // Throws

struct e_line { int value; };

void process_file( FILE * f )
{
  for( int current_line = 1; current_line != 10; ++current_line )
  {
    auto load = leaf::on_error( e_line {current_line} );
    int v = parse_line(f);

    // use v
  }
}
```

`on_error`

Using External `result` Types

Static type checking creates difficulties in error handling interoperability in any non-trivial project. Using exception handling alleviates this problem somewhat because in that case error types are not burned into function signatures, so errors easily punch through multiple layers of APIs; but this doesn't help C++ in general because the community is fractured on the issue of exception handling. That debate notwithstanding, the reality is that C++ programs need to handle errors communicated through multiple layers of APIs via a plethora of error codes, `result` types and exceptions.

LEAF enables application developers to shake error objects out of each individual library's `result` type and send them to error handling scopes verbatim. Here is an example:

```
lib1::result<int, lib1::error_code> foo();
```

```
lib2::result<int, lib2::error_code> bar();

int g( int a, int b );

leaf::result<int> f()
{
  auto a = foo();
  if( !a )
    return leaf::new_error( a.error() );

  auto b = bar();
  if( !b )
    return leaf::new_error( b.error() );

  return g( a.value(), b.value() );
}
```

result | new_error

Later we simply call `leaf::try_handle_some`, passing an error handler for each type:

```
leaf::result<int> r = leaf::try_handle_some(

  []() -> leaf::result<int>
  {
    return f();
  },

  []( lib1::error_code ec ) -> leaf::result<int>
  {
    // Handle lib1::error_code
  },

  []( lib2::error_code ec ) -> leaf::result<int>
  {
    // Handle lib2::error_code
  } );
```

try_handle_some | result

A possible complication is that we might not have the option to return `leaf::result<int>` from `f`: a third party API may impose a specific signature on it, forcing it to return a library-specific `result` type. This would be the case when `f` is intended to be used as a callback:

```
void register_callback( std::function<lib3::result<int>()> const & callback );
```

Can we use LEAF in this case? Actually we can, as long as `lib3::result` is able to communicate a `std::error_code`. We just have to let LEAF know, by specializing the `is_result_type` template:

```
namespace boost { namespace leaf {

template <class T>
struct is_result_type<lib3::result<T>>: std::true_type;

} }
```

is_result_type

With this in place, `f` works as before, even though `lib3::result` isn't capable of transporting `lib1` errors or `lib2` errors:

```
lib1::result<int, lib1::error_type> foo();
lib2::result<int, lib2::error_type> bar();

int g( int a, int b );

lib3::result<int> f() // Note: return type is not leaf::result<int>
{
  auto a = foo();
  if( !a )
    return leaf::new_error( a.error() );

  auto b = bar();
  if( !b )
    return leaf::new_error( b.error() );

  return g( a.value(), b.value() );
}
```

new_error

The object returned by `leaf::new_error` converts implicitly to `std::error_code`, using a LEAF-specific `error_category`, which makes `lib3::result` compatible with `leaf::try_handle_some` (and with `leaf::try_handle_all`):

```
lib3::result<int> r = leaf::try_handle_some(

  []() -> lib3::result<int>
  {
    return f();
  },

  []( lib1::error_code ec ) -> lib3::result<int>
  {
    // Handle lib1::error_code
  },

  []( lib2::error_code ec ) -> lib3::result<int>
```

```
  {
    // Handle lib2::error_code
  } );
```

try_handle_some

Interoperability

Ideally, when an error is detected, a program using LEAF would always call new_error, ensuring that each encountered failure is definitely assigned a unique error_id, which then is reliably delivered, by an exception or by a result<T> object, to the appropriate error handling scope.

Alas, this is not always possible.

For example, the error may need to be communicated through uncooperative 3rd-party interfaces. To facilitate this transmission, an error ID may be encoded in a std::error_code. As long as a 3rd-party interface is able to transport a std::error_code, it can be compatible with LEAF.

Further, it is sometimes necessary to communicate errors through an interface that does not even use std::error_code. An example of this is when an external low level library throws an exception, which is unlikely to be able to carry an error_id.

To support this tricky use case, LEAF provides the function current_error, which returns the error ID returned by the most recent call (from this thread) to new_error. One possible approach to solving the problem is to use the following logic (implemented by the error_monitor type):

1. Before calling the uncooperative API, call current_error and cache the returned value.
2. Call the API, then call current_error again:
    a. If this returns the same value as before, pass the error objects to new_error to associate them with a new error_id;
    b. else, associate the error objects with the error_id value returned by the second call to current_error.

Note that if the above logic is nested (e.g. one function calling another), new_error will be called only by the inner-most function, because that call guarantees that all calling functions will hit the else branch.

For a detailed tutorial see Using error_monitor to Report Errors from C Callbacks.

Loading of Error Objects

Recall that error objects communicated to LEAF are stored on the stack, local to the try_handle_some, try_handle_all or try_catch function used to handle errors. To *load* an error object means to move it into such storage, if available.

Various LEAF functions take a list of error objects to load. As an example, if a function `copy_file` that takes the name of the input file and the name of the output file as its arguments detects a failure, it could communicate an error code `ec`, plus the two relevant file names using `new_error`:

```
return leaf::new_error(ec, e_input_name{n1}, e_output_name{n2});
```

Alternatively, error objects may be loaded using a `result<T>` that is already communicating an error. This way they become associated with that error, rather than with a new error:

```
leaf::result<int> f() noexcept;

leaf::result<void> g( char const * fn ) noexcept
{
  if( leaf::result<int> r = f() )
  { ①
    ....;
    return { };
  }
  else
  {
    return r.load( e_file_name{fn} ); ②
  }
}
```

`result` | `load`

① Success! Use `r.value()`.

② `f()` has failed; here we associate an additional `e_file_name` with the error. However, this association occurs iff in the call stack leading to `g` there are error handlers that take an `e_file_name` argument. Otherwise, the object passed to `load` is discarded. In other words, the passed objects are loaded iff the program actually uses them to handle errors.

Besides error objects, `load` can take function arguments:

- If we pass a function that takes no arguments, it is invoked, and the returned error object is loaded.

  Consider that if we pass to `load` an error object that is not used by an error handler, it will be discarded. If instead of an error object we pass a function that returns an error object, that function will only be called if the object it returns is needed, that is, if it will not be discarded. This is helpful when the error object is relatively expensive to produce:

  ```
  struct info { .... };

  info compute_info() noexcept;

  leaf::result<void> operation( char const * file_name ) noexcept
  ```

```
{
  if( leaf::result<int> r = try_something() )
  { ①
    ....
    return { };
  }
  else
  {
    return r.load( ②
      [&]
      {
        return compute_info();
      } );
  }
}
```

result | load

① Success! Use `r.value()`.

② `try_something` has failed; `compute_info` will only be called if an error handler exists in the call stack which takes a `info` argument.

- If we pass a function that takes a single argument of some reference type `E &`, LEAF calls the function with the object of type `E` currently loaded in an active `context`, associated with the error. If no such object is available, a new one is default-initialized and then passed to the function.

  For example, if an operation that involves many different files fails, a program may provide for collecting all relevant file names in a `e_relevant_file_names` object:

```
struct e_relevant_file_names
{
  std::vector<std::string> value;
};

leaf::result<void> operation( char const * file_name ) noexcept
{
  if( leaf::result<int> r = try_something() )
  { ①
    ....
    return { };
  }
  else
  {
    return r.load( ②
      [&](e_relevant_file_names & e)
      {
        e.value.push_back(file_name);
      } );
```

```
  }
}
```

result | load

① Success! Use `r.value()`.

② `try_something` has failed — add `file_name` to the `e_relevant_file_names` object, associated with the `error_id` communicated in `r`. Note, however, that the passed function will only be called iff in the call stack there are error handlers that take an `e_relevant_file_names` object.

Using `on_error`

It is not typical for an error reporting function to be able to supply all of the data needed by a suitable error handling function in order to recover from the failure. For example, a function that reports `FILE` failures may not have access to the file name, yet an error handling function needs it in order to print a useful error message.

The file name is typically readily available in the call stack leading to the failed `FILE` operation. Below, while `parse_info` can't report the file name, `parse_file` can and does:

```
leaf::result<info> parse_info( FILE * f ) noexcept; ①

leaf::result<info> parse_file( char const * file_name ) noexcept
{
  auto load = leaf::on_error(leaf::e_file_name{file_name}); ②

  if( FILE * f = fopen(file_name,"r") )
  {
    auto r = parse_info(f);
    fclose(f);
    return r;
  }
  else
    return leaf::new_error( error_enum::file_open_error );
}
```

result | on_error | new_error

① `parse_info` communicates errors using `leaf::result`.

② `on_error` ensures that the file name is included with any error reported out of `parse_file`. When the `load` object expires, if an error is being reported, the passed `e_file_name` value will be automatically associated with it.

`on_error` — like `new_error` — can be passed any number of arguments.

When we invoke `on_error`, we can pass three kinds of arguments:

1. Actual error objects (like in the example above);
2. Functions that take no arguments and return an error object;
3. Functions that take a single error object by mutable reference.

For example, if we want to use `on_error` to capture `errno`, we could use the `e_errno` type, which is a simple struct that wraps an `int`. But, we can't just pass an `e_errno` to `on_error`, because at that time `errno` hasn't been set (yet). Instead, we'd pass a function that returns it:

```
void read_file(FILE * f) {

  auto load = leaf::on_error([]{ return leaf::e_errno{errno}; });

  ....
  size_t nr1=fread(buf1,1,count1,f);
  if( ferror(f) )
    leaf::throw_exception();

  size_t nr2=fread(buf2,1,count2,f);
  if( ferror(f) )
    leaf::throw_exception();

  size_t nr3=fread(buf3,1,count3,f);
  if( ferror(f) )
    leaf::throw_exception();
  ....
}
```

Above, if an exception is thrown, LEAF will invoke the function passed to `on_error` and associate the returned `e_errno` object with the exception.

Finally, if `on_error` is passed a function that takes a single error object by mutable reference, the behavior is similar to how such functions are handled by `load`; see Loading of Error Objects.

Using Predicates to Handle Errors

Usually, the compatibility between error handlers and the available error objects is determined based on the type of the arguments they take. When an error handler takes a predicate type as an argument, the handler selection procedure is able to also take into account the *value* of the available error objects.

Consider this error code enum:

```
enum class validation_error
{
```

```
  empty_field = 1,
  invalid_format,
  out_of_range
};
```

We could handle `validation_error` errors like so:

```
return leaf::try_handle_some(

  []
  {
    return validate_input(); // Returns leaf::result<input>
  },

  []( validation_error e )
  {
    switch(e)
    {
      case validation_error::empty_field:
        ....; // Handle empty_field errors
        break;
      case validation_error::invalid_format:
      case validation_error::out_of_range:
        ....; // Handle invalid_format and out_of_range errors
        break;
      default:
        ....; // Handle unexpected validation_error values
        break;
    }
  } );
```

If a `validation_error` object is available, LEAF will call our error handler. If not, the failure will be forwarded to the caller.

This can be rewritten using the `match` predicate to organize the different cases in different error handlers. The following is equivalent:

```
return leaf::try_handle_some(

  []
  {
    return validate_input(); // Returns leaf::result<input>
  },

  []( leaf::match<validation_error, validation_error::empty_field> m )
  {
    assert(m.matched == validation_error::empty_field);
    ....;
  },
```

```
  []( leaf::match<validation_error, validation_error::invalid_format,
validation_error::out_of_range> m )
  {
    assert(m.matched == validation_error::invalid_format || m.matched ==
validation_error::out_of_range);
    ....;
  },

  []( validation_error e )
  {
    ....;
  } );
```

The first argument to the `match` template generally specifies the type `E` of the error object `e` that must be available for the error handler to be considered at all. Typically, the rest of the arguments are values. The error handler is dropped if `e` does not compare equal to any of them.

In particular, `match` works great with `std::error_code`. The following handler is designed to handle `ENOENT` errors:

```
[]( leaf::match<std::error_code, std::errc::no_such_file_or_directory> )
{
}
```

This, however, requires C++17 or newer. LEAF provides the following workaround, compatible with C++11:

```
[]( leaf::match<leaf::condition<std::errc>, std::errc::no_such_file_or_directory> )
{
}
```

It is also possible to select a handler based on `std::error_category`. The following handler will match any `std::error_code` of the `std::generic_category` (requires C++17 or newer):

```
[]( std::error_code, leaf::category<std::errc> )
{
}
```

See `match` for more examples.

The following predicates are available:

- `match`: as described above.
- `match_value`: where `match<E, V…>` compares the object `e` of type `E` with the values `V…`, `match_value<E, V…>` compare `e.value` with the values `V…`.

- `match_member`: similar to `match_value`, but takes a pointer to the data member to compare; that is, `match_member<&E::value, V…>` is equivalent to `match_value<E, V…>`. Note, however, that `match_member` requires C++17 or newer, while `match_value` does not.
- `catch_<Ex…>`: Similar to `match`, but checks whether the caught `std::exception` object can be `dynamic_cast` to any of the `Ex…` types.
- `if_not` is a special predicate that takes any other predicate `Pred` and requires that an error object of type `E` is available and that `Pred` evaluates to `false`. For example, `if_not<match<E, V…>>` requires that an object `e` of type `E` is available, and that it does not compare equal to any of the specified `V…`.

The predicate system is easily extensible, see Predicates.

See also Working with `std::error_code` and `std::error_condition`.

Reusing Common Error Handlers

Consider this snippet:

```
config cfg = leaf::try_handle_all(

  [&]
  {
    return load_config_file(config_path); // returns leaf::result<config>
  },

  [](api_error e) -> config
  {
    ....
  },

  [](io_error e, e_file_name const & fn) -> config
  {
    ....
  },

  []() -> config
  {
    ....
  });
```

`try_handle_all` | `e_file_name`

If we need to attempt a different set of operations yet use the same handlers, we could repeat the same thing with a different function passed as the `TryBlock` for `try_handle_all`:

```
config cfg = leaf::try_handle_all(

  [&]
  {
    return load_config_file(fallback_path); // returns leaf::result<config>
  },

  [](api_error e) -> config
  {
    ....
  },

  [](io_error e, e_file_name const & fn) -> config
  {
    ....
  },

  []() -> config
  {
    ....
  });
```

That works, but it is also possible to bind the error handlers in a `std::tuple`:

```
auto load_config_error_handlers = std::make_tuple(

  [](api_error e) -> config
  {
    ....
  },

  [](io_error e, e_file_name const & fn) -> config
  {
    ....
  },

  []() -> config
  {
    ....
  });
```

The `load_config_error_handlers` tuple can later be used with any error handling function:

```
config cfg1 = leaf::try_handle_all(

  [&]
  {
    return load_config_file(config_path); ①
```

```
  },

  load_config_error_handlers );

config cfg2 = leaf::try_handle_all(

  [&]
  {
    return load_config_file(fallback_path); ②
  },

  load_config_error_handlers ); ③
```

try_handle_all | error_info

① One set of operations which may fail…

② A different set of operations which may fail…

③ … both using the same `load_config_error_handlers`.

Error handling functions accept a `std::tuple` of error handlers in place of any error handler. The behavior is as if the tuple is unwrapped in-place.

Transporting Errors Between Threads

Like exceptions, LEAF error objects are local to a thread. When using concurrency, sometimes we need to collect error objects in one thread, then use them to handle errors in another thread.

LEAF supports this functionality with or without exception handling. In both cases error objects are captured and transported in a `leaf::result<T>` object.

Transporting Errors Between Threads Without Exception Handling

Let's assume we have a `task` that we want to launch asynchronously, which produces a `task_result` but could also fail:

```
leaf::result<task_result> task();
```

Because the task will run asynchronously, in case of a failure we need to capture any produced error objects but not handle errors. We do this by invoking `try_capture_all`:

```
std::future<leaf::result<task_result>> launch_task() noexcept
{
  return std::async(
    std::launch::async,
    [&]
```

```
    {
      return leaf::try_capture_all(task);
    } );
}
```

result | try_capture_all

In case of a failure, the returned from `try_capture_all` `result<T>` object holds all error objects communicated out of the `task`, at the cost of dynamic allocations. The `result<T>` object can then be stashed away or moved to another thread, and later passed to an error-handling function to unload its content and handle errors:

```
//std::future<leaf::result<task_result>> fut;
fut.wait();

return leaf::try_handle_some(

  [&]() -> leaf::result<void>
  {
    BOOST_LEAF_AUTO(r, fut.get());
    //Success!
    return { };
  },

  [](E1 e1, E2 e2)
  {
    //Deal with E1, E2
    ....
    return { };
  },

  [](E3 e3)
  {
    //Deal with E3
    ....
    return { };
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

Follow this link to see a complete example program: try_capture_all_result.cpp.

Transporting Errors Between Threads With Exception Handling

Let's assume we have an asynchronous `task` which produces a `task_result` but could also throw:

```
task_result task();
```

We use `try_capture_all` to capture all error objects and the `std::current_exception()` in a `result<T>`:

```
std::future<leaf::result<task_result>> launch_task()
{
  return std::async(
    std::launch::async,
    [&]
    {
      return leaf::try_capture_all(task);
    } );
}
```

`try_capture_all`

To handle errors after waiting on the future, we use `try_catch` as usual:

```
//std::future<leaf::result<task_result>> fut;
fut.wait();

return leaf::try_catch(

  [&]
  {
    leaf::result<task_result> r = fut.get();
    task_result v = r.value(); // throws on error
    //Success!
  },

  [](E1 e1, E2 e2)
  {
    //Deal with E1, E2
    ....
  },

  [](E3 e3)
  {
    //Deal with E3
    ....
  } );
```

`try_catch` | `result`

Follow this link to see a complete example program: try_capture_all_exceptions.cpp.

Classification of Failures

It is common for an interface to define an `enum` that lists all possible error codes that the API reports. The benefit of this approach is that the list is complete and usually well documented:

```
enum error_code
{
  ....
  read_error,
  size_error,
  eof_error,
  ....
};
```

The disadvantage of such flat enums is that they do not support handling of a whole class of failures. Consider the following LEAF error handler:

```
....
[](leaf::match<error_code, size_error, read_error, eof_error>, leaf::e_file_name const
& fn)
{
  std::cerr << "Failed to access " << fn.value << std::endl;
},
....
```

`match` | `e_file_name`

It will get called if the value of the `error_code` enum communicated with the failure is one of `size_error`, `read_error` or `eof_error`. In short, the idea is to handle any input error.

But what if later we add support for detecting and reporting a new type of input error, e.g. `permissions_error`? It is easy to add that to our `error_code` enum; but now our input error handler won't recognize this new input error — and we have a bug.

Using exceptions is an improvement because exception types can be organized in a hierarchy in order to classify failures:

```
struct input_error: std::exception { };
struct read_error: input_error { };
struct size_error: input_error { };
struct eof_error: input_error { };
```

In terms of LEAF, our input error exception handler now looks like this:

```
[](input_error &, leaf::e_file_name const & fn)
{
```

```
  std::cerr << "Failed to access " << fn.value << std::endl;
},
```

This is future-proof, but still not ideal, because it is not possible to refine the classification of the failure after the exception object has been thrown.

LEAF supports a novel style of error handling where the classification of failures does not use error code values or exception type hierarchies. Instead of our `error_code` enum, we could define:

```
....
struct input_error { };
struct read_error { };
struct size_error { };
struct eof_error { };
....
```

With this in place, we could define a function `file_read`:

```
leaf::result<void> file_read( FILE & f, void * buf, int size )
{
  int n = fread(buf, 1, size, &f);

  if( ferror(&f) )
    return leaf::new_error(input_error{}, read_error{}, leaf::e_errno{errno}); ①

  if( n!=size )
    return leaf::new_error(input_error{}, eof_error{}); ②

  return { };
}
```

result | new_error | e_errno

① This error is classified as `input_error` and `read_error`.

② This error is classified as `input_error` and `eof_error`.

Or, even better:

```
leaf::result<void> file_read( FILE & f, void * buf, int size )
{
  auto load = leaf::on_error(input_error{}); ①

  int n = fread(buf, 1, size, &f);

  if( ferror(&f) )
    return leaf::new_error(read_error{}, leaf::e_errno{errno}); ②
```

```
  if( n!=size )
    return leaf::new_error(eof_error{}); ③

  return { };
}
```

result | on_error | new_error | e_errno

① Any error escaping this scope will be classified as `input_error`

② In addition, this error is classified as `read_error`.

③ In addition, this error is classified as `eof_error`.

This technique works just as well if we choose to use exception handling, we just call `leaf::throw_exception` instead of `leaf::new_error`:

```
void file_read( FILE & f, void * buf, int size )
{
  auto load = leaf::on_error(input_error{});

  int n = fread(buf, 1, size, &f);

  if( ferror(&f) )
    leaf::throw_exception(read_error{}, leaf::e_errno{errno});

  if( n!=size )
    leaf::throw_exception(eof_error{});
}
```

on_error | throw_exception | e_errno

If the type of the first argument passed to `leaf::throw_exception` derives from `std::exception`, it will be used to initialize the thrown exception object. Here this is not the case, so the function throws a default-initialized `std::exception` object, while the first (and any other) argument is associated with the failure.

Now we can write a future-proof handler for any `input_error`:

```
....
[](input_error, leaf::e_file_name const & fn)
{
  std::cerr << "Failed to access " << fn.value << std::endl;
},
....
```

Remarkably, because the classification of the failure does not depend on error codes or on exception types, this error handler can be used with `try_catch` if we use exception handling, or

with `try_handle_some`/`try_handle_all` if we do not.

Converting Exceptions to `result<T>`

When integrating a library that throws exceptions into code that uses `result<T>`, use `exception_to_result` to convert exceptions at the boundary:

```
struct parse_error: std::exception { };
struct syntax_error: parse_error { };
struct encoding_error: parse_error { };

json_value parse_json(char const * str); // Throws parse_error

leaf::result<json_value> safe_parse_json(char const * str) noexcept
{
  return leaf::exception_to_result<syntax_error, encoding_error>(
    [&]
    {
      return parse_json(str);
    } );
}
```

result | exception_to_result

The template arguments specify which exception types to capture as error objects. All exceptions are caught, and for each type listed, LEAF attempts a `dynamic_cast` and loads a copy of that slice. The `std::current_exception()` is also captured, so unlisted exception types can still be handled.

Errors can then be handled normally:

```
leaf::try_handle_all(

  []() -> leaf::result<void>
  {
    BOOST_LEAF_AUTO(doc, safe_parse_json(input));
    process(doc);
    return { };
  },

  [](syntax_error const &)
  {
    std::cerr << "JSON syntax error" << std::endl;
  },

  [](encoding_error const &)
  {
    std::cerr << "Invalid encoding" << std::endl;
```

```
  },

  []
  {
    std::cerr << "Unknown error" << std::endl;
  } );
```

try_handle_all | result | BOOST_LEAF_AUTO

Handlers that take exception types work the same way whether the object was thrown or loaded via `exception_to_result`.

Using `error_monitor` to Report Errors from C Callbacks

C callbacks have fixed signatures that cannot return C++ types like `leaf::result<T>`. The `error_monitor` class solves this problem.

Consider a C library that invokes a user-provided callback:

```
enum class parse_error { unexpected_token, invalid_syntax };

int on_element(void * ctx, char const * data)
{
  if( <<error-detected>> )
  {
    leaf::new_error(parse_error::unexpected_token);
    return -1;
  }
  return 0;
}
```

new_error

The callback calls `new_error` to associate error objects with the failure, then returns an error code to the C library. The wrapper function uses `error_monitor` to retrieve the `error_id`:

```
leaf::result<document> parse(char const * input)
{
  leaf::error_monitor cur_err;

  if( c_library_parse(input, &on_element, nullptr) != 0 )
    return cur_err.assigned_error_id();
  else
    return make_document();
```

```
}
```

result | error_monitor

If `new_error` was called inside the callback, `assigned_error_id` returns that `error_id`. Otherwise, it calls `new_error` and returns a fresh `error_id`. Either way, the caller can handle the failure normally.

Diagnostic Information

LEAF is able to automatically generate diagnostic messages that include information about all error objects available to error handlers:

```
enum class error_code
{
  read_error,
  write_error
};

....

leaf::try_handle_all(

  []() -> leaf::result<void> ①
  {
    ....
    return leaf::new_error( error_code::write_error, leaf::e_file_name{ "file.txt" }
);
  },

  []( leaf::match<error_code, error_code::read_error> ) ②
  {
    std::cerr << "Read error!" << std::endl;
  },

  []( leaf::diagnostic_details const & info ) ③
  {
    std::cerr << "Unrecognized error detected\n" << info;
  } );
```

① We handle all failures that occur in this try block.

② One or more error handlers that should handle all possible failures.

③ This "catch all" error handler is required by `try_handle_all`. It will be called if LEAF is unable to use another error handler.

The `diagnostic_details` output for the snippet above tells us that we got an `error_code` with

value `1` (`write_error`), and an object of type `e_file_name` with `"file.txt"` stored in its `.value`:

```
Unrecognized error detected
Error with serial #1
Caught:
  error_code: 1
Diagnostic details:
  boost::leaf::e_file_name: file.txt
```

In the `diagnostic_details` output, the section under `Caught:` lists the objects which error handlers take as arguments — these are the objects which are stored on the stack. The section under `Diagnostic details:` lists all other objects that were communicated. These are the objects that would have been discarded if we didn't provide a handler that takes `diagnostic_details`.

To print each error object, LEAF attempts to bind an unqualified call to `operator<<`, passing a `std::ostream` and the error object. If that fails, it will also attempt to bind `operator<<` that takes the `.value` of the error type. If that also does not compile, the error object value will not appear in diagnostic messages, though LEAF will still print its type.

Even with error types that define a printable `.value`, the user may still want to overload `operator<<` for the enclosing `struct`, e.g.:

```
struct e_errno
{
  int value;

  friend std::ostream & operator<<( std::ostream & os, e_errno const & e )
  {
    return os << e.value << ", \"" << strerror(e.value) << '"';
  }
};
```

The `e_errno` type above is designed to hold `errno` values. The defined `operator<<` overload will automatically include the output from `strerror` when `e_errno` values are printed (LEAF defines `e_errno` in `<boost/leaf/common.hpp>`, together with other commonly used error types).

Using `diagnostic_details` comes at a cost. Normally, when the program attempts to communicate error objects of types which are not used in any error handling scope in the current call stack, they are discarded, which saves cycles. However, if an error handler is provided that takes `diagnostic_details` argument, such objects are stored on the heap instead of being discarded.

If handling `diagnostic_details` is considered too costly, use `diagnostic_info` instead:

```
leaf::try_handle_all(
```

```
  []() -> leaf::result<void>
  {
    ....
    return leaf::new_error( error_code::write_error, leaf::e_file_name{ "file.txt" }
);
  },

  []( leaf::match<error_code, error_code::read_error> )
  {
    std::cerr << "Read error!" << std::endl;
  },

  []( leaf::diagnostic_info const & info )
  {
    std::cerr << "Unrecognized error detected\n" << info;
  } );
```

In this case, the output may look like this:

```
Unrecognized error detected
Error serial #1
Caught:
  error_code: 1
```

Notice how we are missing the `Diagnostic details:` section. That's because the `e_file_name` object was discarded by LEAF, since no error handler needed it.

The automatically generated diagnostic messages are developer-friendly, but not user-friendly.

Serialization

LEAF provides a serialization API that enables exporting error information into different formats, such as JSON. This is useful for structured logging, remote debugging, or integrating with monitoring systems. To serialize error information, use the `serialize_to` member function available on the following types:

- `error_info`
- `diagnostic_info`
- `diagnostic_details`
- `result`

LEAF serialization is defined in terms of `output` and `output_at` function calls, found via ADL:

- `output(e, x)` serializes `x` directly to encoder `e` as a value.

- `output_at(e, x, name)` serializes `x` to encoder `e` as a named field.

LEAF provides generic `output` overloads for the following types:

- `error_id`
- `e_source_location`
- `e_errno`
- `std::error_code`
- `std::error_condition`
- `std::exception`
- `std::exception_ptr`
- any type with a `.value` member for which a suitable `output` can be found via ADL

Custom Encoders

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

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

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

The `output_at` function typically creates a nested scope (e.g. a JSON object or XML element) and then makes an unqualified call to `output` to output the value. This will call any compatible overload found via ADL.

The `output` function may need to be defined using SFINAE to avoid ambiguities with the generic `output` overloads provided by LEAF. Custom encoders must also handle types that do not provide ADL `output` overloads, including built-in types like `int` and `std::string`.

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

```
namespace boost { namespace leaf {

namespace serialization {

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

}

} }
```

The `serialize` function template takes a handle reference `h` (of unspecified type) that holds an encoder, the error 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(
  [&](nlohmann_json_encoder & e) { output_at(e, x, name); },
  [&](xml_encoder & xe) { output_at(xe, x, name); }
);
```

JSON Serialization

LEAF provides encoders for JSON serialization with two popular JSON libraries:

- `nlohmann_json_encoder` for nlohmann/json
- `boost_json_encoder` for Boost.JSON

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

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

namespace leaf = boost::leaf;

using nlohmann_json_encoder = leaf::serialization::nlohmann_json_encoder<nlohmann
::json>;

namespace boost { namespace leaf {
```

```
namespace serialization {

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

}

} }
```

With this in place, we can easily (for example) output `diagnostic_details` to JSON:

```
struct e_api_response
{
  int status;
  std::string message;

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

struct e_request_url
{
  std::string value;
};

....

leaf::try_handle_all(
  []() -> leaf::result<void>
  {
    ....
    return leaf::new_error(e_api_response{403, "Access denied"}, e_request_url
{"/api/admin/settings"});
  },
  [](leaf::diagnostic_details const & dd)
  {
    nlohmann::json j;
    nlohmann_json_encoder e(j);
    dd.serialize_to(e);
    std::cout << j.dump(2) << std::endl;
```

```
  }
);
```

Output:

```
{
  "e_api_response": {
    "status": 403,
    "message": "Access denied"
  },
  "e_request_url": "/api/admin/settings"
}
```

diagnostic_details | diagnostic_info | error_info

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

Working with `std::error_code` and `std::error_condition`

Introduction

The relationship between `std::error_code` and `std::error_condition` is not easily understood from reading the standard specification. This section explains how they're supposed to be used, and how LEAF interacts with them.

The idea behind `std::error_code` is to encode both an integer value representing an error code, as well as the domain of that value. The domain is represented by a `std::error_category` reference. Conceptually, a `std::error_code` is like a `pair<std::error_category const &, int>`.

Let's say we have this `enum`:

```
enum class file_error
{
  not_found = 1,
  permission_denied,
  disk_full
};
```

We want to be able to transport `file_error` values in `std::error_code` objects. This erases their

static type, which enables them to travel freely across API boundaries. To this end, we must define a `std::error_category` that represents our `file_error` type:

```
std::error_category const & file_error_category()
{
  struct category: std::error_category
  {
    char const * name() const noexcept override
    {
      return "file_error";
    }

    std::string message(int code) const override
    {
      switch( file_error(code) )
      {
        case file_error::not_found: return "file not found";
        case file_error::permission_denied: return "permission denied";
        case file_error::disk_full: return "disk full";
        default: return "unknown file error";
      }
    }
  };

  static category c;
  return c;
}
```

We also need to inform the standard library that `file_error` is compatible with `std::error_code`, and provide a factory function which can be used to make `std::error_code` objects out of `file_error` values:

```
namespace std
{
  template <>
  struct is_error_code_enum<file_error>: std::true_type
  {
  };
}

std::error_code make_error_code(file_error e)
{
  return std::error_code(int(e), file_error_category());
}
```

With this in place, if we receive a `std::error_code`, we can easily check if it represents some of the `file_error` values we're interested in:

```
std::error_code read_file();

....
auto ec = read_file();
if( ec == file_error::not_found || ec == file_error::permission_denied )
{
  // We got either file_error::not_found or file_error::permission_denied
}
```

This works because the standard library detects that `std::is_error_code_enum<file_error>::value` is `true`, and then uses `make_error_code` to create a `std::error_code` object it actually uses to compare to `ec`.

So far so good, but remember, the standard library defines another type also, `std::error_condition`. The first confusing thing is that in terms of its physical representation, `std::error_condition` is identical to `std::error_code`; that is, it is also like a pair of `std::error_category` reference and an `int`. Why do we need two different types which use identical physical representation?

The key to answering this question is to understand that `std::error_code` objects are designed to be returned from functions to indicate failures. In contrast, `std::error_condition` objects are never supposed to be communicated; their purpose is to interpret the `std::error_code` values being communicated. The idea is that in a given program there may be multiple different "physical" (maybe platform-specific) `std::error_code` values which all indicate the same "logical" `std::error_condition`.

This leads us to the second confusing thing about `std::error_condition`: it uses the same `std::error_category` type, but for a completely different purpose: to specify what `std::error_code` values are equivalent to what `std::error_condition` values.

Let's say that in addition to `file_error`, our program uses a network library which communicates failures in terms of `std::error_code` with a different error category:

```
enum class net_error
{
  connection_refused = 1,
  timeout,
  host_unreachable,
  dns_failed
};

// Boilerplate omitted:
// - net_error_category()
// - specialization of std::is_error_code_enum
// - make_error_code factory function for net_error.
```

We can now use `std::error_condition` to define *logical* error conditions that group related `std::error_code` values from both libraries:

```
enum class resource_condition ①
{
  unavailable = 1,
  access_denied
};

std::error_category const & resource_condition_category() ②
{
  struct category: std::error_category
  {
    char const * name() const noexcept override
    {
      return "resource_condition";
    }

    std::string message(int cond) const override
    {
      switch( resource_condition(cond) )
      {
        case resource_condition::unavailable: return "resource unavailable";
        case resource_condition::access_denied: return "access denied";
        default: return "unknown condition";
      }
    }

    bool equivalent(std::error_code const & code, int cond) const noexcept
    {
      switch( resource_condition(cond) )
      {
        case resource_condition::unavailable: ③
          return
            code == file_error::not_found ||
            code == net_error::host_unreachable ||
            code == net_error::dns_failed;
        case resource_condition::access_denied: ④
          return
            code == file_error::permission_denied ||
            code == net_error::connection_refused;
        default:
          return false;
      }
    }
  };

  static category c;
  return c;
}

namespace std
{
```

```
  template <> ⑤
  struct is_error_condition_enum<resource_condition>: std::true_type
  {
  };
}

std::error_condition make_error_condition(resource_condition e) ⑥
{
  return std::error_condition(int(e), resource_condition_category());
}
```

① Enumeration of the two logical conditions: `unavailable` and `access_denied`.

② Define the `std::error_category` for `std::error_condition` objects that represent a `resource_condition`.

③ Here we specify that `file_error::not_found`, `net_error::host_unreachable`, and `net_error::dns_failed` are all logically equivalent to `resource_condition::unavailable`, and that...

④ ...`file_error::permission_denied` and `net_error::connection_refused` are logically equivalent to `resource_condition::access_denied`.

⑤ This specialization tells the standard library that the `resource_condition` enum is designed to be used with `std::error_condition`.

⑥ The factory function to make `std::error_condition` objects out of `resource_condition` values.

Phew!

Now, if we have a `std::error_code` object `ec`, we can easily check if it is equivalent to `resource_condition::unavailable` like so:

```
if( ec == resource_condition::unavailable )
{
  // The resource is unavailable (file not found, host unreachable, or DNS failed)
}
```

Again, remember that beyond defining the `std::error_category` for `std::error_condition` objects initialized with a `resource_condition` value, we don't need to interact with the actual `std::error_condition` instances: they're created when needed to compare to a `std::error_code`, and that's pretty much all they're good for.

Support in LEAF

The `match` predicate can be used as an argument to a LEAF error handler to match a `std::error_code` with a given error condition. For example, to handle `resource_condition::unavailable` (see above), we could use:

```
leaf::try_handle_some(
```

```
  []
  {
    return open_resource(); // returns leaf::result<resource>
  },

  []( leaf::match<std::error_code, resource_condition::unavailable> m )
  {
    assert(m.matched == resource_condition::unavailable);
    ....
  } );
```

See `match` for more examples.

Boost Exception Integration

Instead of the `boost::get_error_info` API defined by Boost Exception, it is possible to use LEAF error handlers directly. Consider the following use of `boost::get_error_info`:

```
typedef boost::error_info<struct my_info_, int> my_info;

void f(); // Throws using boost::throw_exception

void g()
{
  try
  {
    f();
  },
  catch( boost::exception & e )
  {
    if( int const * x = boost::get_error_info<my_info>(e) )
      std::cerr << "Got my_info with value = " << *x;
  } );
}
```

We can rewrite `g` to access `my_info` using LEAF:

```
#include <boost/leaf/handle_errors.hpp>

void g()
{
  leaf::try_catch(

    []
    {
```

```
      f();
    },

    []( my_info x )
    {
      std::cerr << "Got my_info with value = " << x.value();
    } );
}
```

`try_catch`

Taking `my_info` means that the handler will only be selected if the caught exception object carries `my_info` (which LEAF accesses via `boost::get_error_info`).

The use of `match` is also supported:

```
void g()
{
  leaf::try_catch(

    []
    {
      f();
    },

    []( leaf::match_value<my_info, 42> )
    {
      std::cerr << "Got my_info with value = 42";
    } );
}
```

Above, the handler will be selected if the caught exception object carries `my_info` with `.value()` equal to 42.

Examples

See github.

Synopsis

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

LEAF headers are designed to minimize coupling:

- Headers needed to report or forward but not handle errors are lighter than headers providing error handling functionality.
- Headers that provide exception handling or throwing functionality are separate from headers that provide error handling or reporting but do not use exceptions.

A standalone single-header option is available; please see Distribution.

Error Reporting

common.hpp

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_api_function { char const * value; };

  struct e_file_name { std::string value; };

  struct e_type_info_name { char const * value; };

  struct e_at_line { int value; };

  struct e_errno
  {
    int value;
    explicit e_errno(int value=errno);

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, e_errno
const &);
  };

  namespace windows
  {
    struct e_LastError
    {
      unsigned value;

      explicit e_LastError(unsigned value);
```

```
#if BOOST_LEAF_CFG_WIN32
      e_LastError();

      template <class CharT, class Traits>
      friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
e_LastError const &);
#endif
    };
  }

} }
```

Reference: e_api_function | e_file_name | e_at_line | e_type_info_name | e_source_location | e_errno | e_LastError

error.hpp

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  class error_id
  {
  public:

    error_id() noexcept;

    template <class Enum>
    error_id( Enum e, typename std::enable_if<std::is_error_code_enum<Enum>::
value, Enum>::type * = 0 ) noexcept;

    error_id( std::error_code const & ec ) noexcept;

    int value() const noexcept;
    explicit operator bool() const noexcept;

    std::error_code to_error_code() const noexcept;

    friend bool operator==( error_id a, error_id b ) noexcept;
    friend bool operator!=( error_id a, error_id b ) noexcept;
    friend bool operator<( error_id a, error_id b ) noexcept;

    template <class... Item>
    error_id load( Item && ... item ) const noexcept;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
error_id );
  };
```

```
  bool is_error_id( std::error_code const & ec ) noexcept;

  template <class... Item>
  error_id new_error( Item && ... item ) noexcept;

  error_id current_error() noexcept;

  //////////////////////////////////////

  template <class Ctx>
  class context_activator
  {
    context_activator( context_activator const & ) = delete;
    context_activator & operator=( context_activator const & ) = delete;

  public:

    explicit context_activator( Ctx & ctx ) noexcept;
    context_activator( context_activator && ) noexcept;
    ~context_activator() noexcept;
  };

  template <class Ctx>
  context_activator<Ctx> activate_context( Ctx & ctx ) noexcept;

  template <class R>
  struct is_result_type: std::false_type
  {
  };

  template <class R>
  struct is_result_type<R const>: is_result_type<R>
  {
  };

} }

#define BOOST_LEAF_ASSIGN(v, r)\
  auto && <<temp>> = r;\
  if( !<<temp>> )\
    return <<temp>>.error();\
  v = std::forward<decltype(<<temp>>)>(<<temp>>).value()

#define BOOST_LEAF_AUTO(v, r)\
  BOOST_LEAF_ASSIGN(auto v, r)

#if BOOST_LEAF_CFG_GNUC_STMTEXPR

#define BOOST_LEAF_CHECK(r)\
  ({\
```

```
    auto && <<temp>> = (r);\
    if( !<<temp>> )\
        return <<temp>>.error();\
    std::move(<<temp>>);\
  }).value()

#else

#define BOOST_LEAF_CHECK(r)\
  {\
    auto && <<temp>> = r;\
    if( !<<temp>> )\
      return <<temp>>.error()\
  }

#endif

#define BOOST_LEAF_NEW_ERROR <<exact-definition-unspecified>>
```

Reference: error_id | is_error_id | new_error | current_error | context_activator | activate_context | is_result_type | BOOST_LEAF_ASSIGN | BOOST_LEAF_AUTO | BOOST_LEAF_CHECK | BOOST_LEAF_NEW_ERROR

exception.hpp

#include <boost/leaf/exception.hpp>

```
namespace boost { namespace leaf {

  template <class Ex, class... E> ①
  [[noreturn]] void throw_exception( Ex &&, E && ... );

  template <class E1, class... E> ②
  [[noreturn]] void throw_exception( E1 &&, E && ... );

  [[noreturn]] void throw_exception();

  template <class Ex, class... E> ①
  [[noreturn]] void throw_exception( error_id id, Ex &&, E && ... );

  template <class E1, class... E> ②
  [[noreturn]] void throw_exception( error_id id, E1 &&, E && ... );

  [[noreturn]] void throw_exception( error_id id );

  template <class... Ex, class F>
  <<result<T>-deduced>> exception_to_result( F && f ) noexcept;

} }
```

```
#define BOOST_LEAF_THROW_EXCEPTION <<exact-definition-unspecified>>
```

Reference: throw_exception | BOOST_LEAF_THROW_EXCEPTION

① Only enabled if std::is_base_of<std::exception, Ex>::value.

② Only enabled if !std::is_base_of<std::exception, E1>::value.

on_error.hpp

#include <boost/leaf/on_error.hpp>

```
namespace boost { namespace leaf {

  template <class... Item>
  <<unspecified-type>> on_error( Item && ... e ) noexcept;

  class error_monitor
  {
  public:

    error_monitor() noexcept;

    error_id check() const noexcept;
    error_id assigned_error_id() const noexcept;
  };

} }
```

Reference: on_error | error_monitor

result.hpp

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  class result
  {
  public:

    using value_type = T;

    // NOTE: Copy constructor implicitly deleted.
    result( result && r ) noexcept;
```

```
    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result( result<U> && r ) noexcept;

    result() noexcept;

    result( T && v ) noexcept;

    result( T const & v );

    result( error_id err ) noexcept;

    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result( U && u );

#if BOOST_LEAF_CFG_STD_SYSTEM_ERROR

    result( std::error_code const & ec ) noexcept;

    template <class Enum, class = typename std::enable_if<std::is_error_code_enum
<Enum>::value, int>::type>
    result( Enum e ) noexcept;

#endif

    // NOTE: Assignment operator implicitly deleted.
    result & operator=( result && r ) noexcept;

    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result & operator=( result<U> && r ) noexcept;

    bool has_value() const noexcept;
    bool has_error() const noexcept;
    explicit operator bool() const noexcept;

    T const & value() const &;
    T & value() &;
    T const && value() const &&;
    T && value() &&;

    T const * operator->() const noexcept;
    T * operator->() noexcept;

    T const & operator*() const & noexcept;
    T & operator*() & noexcept;
    T const && operator*() const && noexcept;
    T && operator*() && noexcept;

    <<unspecified-type>> error() noexcept;
```

```
    template <class... Item>
    error_id load( Item && ... item ) noexcept;

    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, result
const & );
  };

  template <>
  class result<void>
  {
  public:

    using value_type = void;

    // NOTE: Copy constructor implicitly deleted.
    result( result && r ) noexcept;

    result() noexcept;

    result( error_id err ) noexcept;

#if BOOST_LEAF_CFG_STD_SYSTEM_ERROR

    result( std::error_code const & ec ) noexcept;

    template <class Enum, typename std::enable_if<std::is_error_code_enum<Enum>
::value, Enum>::type>
    result( Enum e ) noexcept;

#endif

    // NOTE: Assignment operator implicitly deleted.
    result & operator=( result && r ) noexcept;

    explicit operator bool() const noexcept;

    void value() const;

    void const * operator->() const noexcept;
    void * operator->() noexcept;

    void operator*() const noexcept;

    <<unspecified-type>> error() noexcept;

    template <class... Item>
```

```
    error_id load( Item && ... item ) noexcept;

    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, result
const &);
  };

  class bad_result: std::exception { };

  template <class T>
  struct is_result_type<result<T>>: std::true_type
  {
  };

} }
```

Reference: result | is_result_type

Error Handling

context.hpp

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  class context
  {
    context( context const & ) = delete;
    context & operator=( context const & ) = delete;

  public:

    context() noexcept;
    context( context && x ) noexcept;
    ~context() noexcept;

    void activate() noexcept;
    void deactivate() noexcept;
    bool is_active() const noexcept;

    void unload( error_id ) noexcept;
```

```
    template <class R, class... H>
    R handle_error( R &, H && ... ) const;
  };

  template <class... H>
  using context_type_from_handlers = typename <<unspecified>>::type;

  template <class...  H>
  BOOST_LEAF_CONSTEXPR context_type_from_handlers<H...> make_context() noexcept;

  template <class...  H>
  BOOST_LEAF_CONSTEXPR context_type_from_handlers<H...> make_context( H && ... )
noexcept;

} }
```

Reference: context | context_type_from_handlers | make_context

diagnostics.hpp

#include <boost/leaf/diagnostics.hpp>

```
namespace boost { namespace leaf {

  class diagnostic_info: public error_info
  {
    //No public constructors

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
diagnostic_info const & );
  };

  class diagnostic_details: public error_info
  {
    //No public constructors

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
diagnostic_info const & );
  };

} }
```

Reference: diagnostic_info | diagnostic_details

handle_errors.hpp

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()().value())>::type
  try_handle_all( TryBlock && try_block, H && ... h );

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()())>::type
  try_handle_some( TryBlock && try_block, H && ... h );

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()())>::type
  try_catch( TryBlock && try_block, H && ... h );

#if BOOST_LEAF_CFG_CAPTURE
  template <class TryBlock>
  result<T> // T deduced depending on TryBlock return type
  try_capture_all( TryBlock && try_block );
#endif

  class error_info
  {
    //No public constructors

  public:

    error_id error() const noexcept;

    bool exception_caught() const noexcept;
    std::exception const * exception() const noexcept;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
error_info const & );
  };

} }
```

Reference: try_handle_all | try_handle_some | try_catch | try_capture_all | error_info

pred.hpp

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  struct is_predicate: std::false_type
  {
  };

  template <class E, auto... V>
  struct match
  {
    E matched;

    // Other members not specified
  };

  template <class E, auto... V>
  struct is_predicate<match<E, V...>>: std::true_type
  {
  };

  template <class E, auto... V>
  struct match_value
  {
    E matched;

    // Other members not specified
  };

  template <class E, auto... V>
  struct is_predicate<match_value<E, V...>>: std::true_type
  {
  };

  template <auto, auto...>
  struct match_member;

  template <class E, class T, T E::* P, auto... V>
  struct match_member<P, V...>
  {
    E matched;

    // Other members not specified
  };

  template <auto P, auto... V>
  struct is_predicate<match_member<P, V...>>: std::true_type
  {
  };
```

```
  template <class... Ex>
  struct catch_
  {
    std::exception const & matched;

    // Other members not specified
  };

  template <class Ex>
  struct catch_<Ex>
  {
    Ex const & matched;

    // Other members not specified
  };

  template <class... Ex>
  struct is_predicate<catch_<Ex...>>: std::true_type
  {
  };

  template <class Pred>
  struct if_not
  {
    E matched;

    // Other members not specified
  };

  template <class Pred>
  struct is_predicate<if_not<Pred>>: std::true_type
  {
  };

  template <class ErrorCodeEnum>
  bool category( std::error_code const & ec ) noexcept;

  template <class Enum, class EnumType = Enum>
  struct condition;

} }
```

Reference: match | match_value | match_member | catch_ | if_not | category | condition

to_variant.hpp

#include <boost/leaf/to_variant.hpp>

```
namespace boost { namespace leaf {

  // Requires at least C++17
  template <class... E, class TryBlock>
  std::variant<
    typename std::decay<decltype(std::declval<TryBlock>()().value())>::type,
    std::tuple<
      std::optional<E>...>>
  to_variant( TryBlock && try_block );

} }
```

Reference: to_variant

Serialization

boost_json_encoder.hpp

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

```
namespace boost { namespace leaf {

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

    // Enabled if x is assignable to boost::json::value, or
    // if tag_invoke is defined for boost::json::value_from_tag.
    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/leaf/serialization/nlohmann_json_encoder.hpp>

```
namespace boost { namespace leaf {

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

    // Enabled if to_json is available for Json and T.
    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: Functions

The contents of each Reference section are organized alphabetically.

`activate_context`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  template <class Ctx>
  context_activator<Ctx> activate_context( Ctx & ctx ) noexcept
  {
    return context_activator<Ctx>(ctx);
  }

} }
```

`context_activator`

Example:

```
leaf::context<E1, E2, E3> ctx;

{
  auto active_context = ctx.raii_activate(); ①
} ②
```

① Activate `ctx`.

② Automatically deactivate `ctx`.

`context_type_from_handlers`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... H>
  using context_type_from_handlers = typename <<unspecified>>::type;

} }
```

Example:

```
auto error_handlers = std::make_tuple(
```

```
  [](e_this const & a, e_that const & b)
  {
    ....
  },

  [](leaf::diagnostic_info const & info)
  {
    ....
  },
  .... );

leaf::context_type_from_handlers<decltype(error_handlers)> ctx; ①
```

① `ctx` will be of type `context<e_this, e_that>`, deduced automatically from the specified error handlers.

Alternatively, a suitable context may be created by calling `make_context`.

---

current_error

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  error_id current_error() noexcept;

} }
```

Returns:

The `error_id` value returned the last time `new_error` was invoked from the calling thread.

See also `on_error`.

---

exception_to_result

#include <boost/leaf/exception.hpp>

```
namespace boost { namespace leaf {

  template <class... Ex, class F>
  <<result<T>-deduced>> exception_to_result( F && f ) noexcept;

} }
```

This function can be used to catch exceptions from a lower-level library and convert them to `result<T>`.

Returns:

Where `f` returns a type `T`, `exception_to_result` returns `leaf::result<T>`.

Effects:

1. Catches all exceptions, then captures `std::current_exception` in a `std::exception_ptr` object, which is loaded with the returned `result<T>`.
2. Attempts to convert the caught exception, using `dynamic_cast`, to each type Ex_i in Ex.... If the cast to Ex_i succeeds, the Ex_i slice of the caught exception is loaded with the returned `result<T>`.

An error handler that takes an argument of an exception type (that is, of a type that derives from `std::exception`) will work correctly whether the object is thrown as an exception or communicated via `new_error` (or converted using `exception_to_result`).

Example:

```
int compute_answer_throws();

//Call compute_answer, convert exceptions to result<int>
leaf::result<int> compute_answer()
{
  return leaf::exception_to_result<ex_type1, ex_type2>(
    []
    {
      return compute_answer_throws();
    } );
}
```

At a later time we can invoke `try_handle_some` / `try_handle_all` as usual, passing handlers that take `ex_type1` or `ex_type2`, for example by reference:

```
return leaf::try_handle_some(

  []() -> leaf::result<void>
  {
    BOOST_LEAF_AUTO(answer, compute_answer());
    //Use answer
    ....
    return { };
  },

  [](ex_type1 & ex1)
  {
    //Handle ex_type1
```

```
    ....
    return { };
  },

  [](ex_type2 & ex2)
  {
    //Handle ex_type2
    ....
    return { };
  },

  [](std::exception_ptr const & p)
  {
    //Handle any other exception from compute_answer.
    ....
    return { };
  } );
```

try_handle_some | result | BOOST_LEAF_AUTO

When a handler takes an argument of an exception type (that is, a type that derives from `std::exception`), if the object is thrown, the argument will be matched dynamically (using `dynamic_cast`); otherwise (e.g. after being converted by `exception_to_result`) it will be matched based on its static type only (which is the same behavior used for types that do not derive from `std::exception`).

See also Converting Exceptions to `result<T>` from the tutorial.

`make_context`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class...  H>
  context_type_from_handlers<H...> make_context() noexcept
  {
    return { };
  }

  template <class...  H>
  context_type_from_handlers<H...> make_context( H && ... ) noexcept
  {
    return { };
  }

} }
```

Example:

```
auto ctx = leaf::make_context( ①
  []( e_this ) { .... },
  []( e_that ) { .... } );
```

① `decltype(ctx)` is `leaf::context<e_this, e_that>`.

---

new_error

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  template <class... Item>
  error_id new_error(Item && ... item) noexcept;

} }
```

Requires:

Each of the `Item`... types must be no-throw movable.

Effects:

As if:

```
error_id id = <<generate-new-unique-id>>;
return id.load(std::forward<Item>(item)...);
```

Returns:

A new `error_id` value, which is unique across the entire program.

Ensures:

`id.value()!=0`, where `id` is the returned `error_id`.

`new_error` discards error objects which are not used in any active error handling calling scope.

When loaded into a `context`, an error object of a type `E` will overwrite the previously loaded object of type `E`, if any.

---

on_error

#include <boost/leaf/on_error.hpp>

```
namespace boost { namespace leaf {

  template <class... Item>
  <<unspecified-type>> on_error(Item && ... item) noexcept;

} }
```

Requires:

Each of the `Item`... types must be no-throw movable.

Effects:

All `item`... objects are forwarded and stored, together with the value returned from `std::unhandled_exceptions`, into the returned object of unspecified type, which should be captured by `auto` and kept alive in the calling scope. When that object is destroyed, if an error has occurred since `on_error` was invoked, LEAF will process the stored items to obtain error objects to be associated with the failure.

On error, LEAF first needs to deduce an `error_id` value `err` to associate error objects with. This is done using the following logic:

- If `new_error` was invoked (by the calling thread) since the object returned by `on_error` was created, `err` is initialized with the value returned by `current_error`;
- Otherwise, if `std::unhandled_exceptions` returns a greater value than it returned during initialization, `err` is initialized with the value returned by `new_error`;
- Otherwise, the stored `item`... objects are discarded and no further action is taken (no error has occurred).

Next, LEAF proceeds similarly to:

```
err.load(std::forward<Item>(item)...);
```

The difference is that unlike `load`, `on_error` will not overwrite any error objects already associated with `err`.

See Using `on_error` from the Tutorial.

serialize

```
namespace boost { namespace leaf {

namespace serialization
```

```
{
  template <class Handle, class E>
  void serialize( Handle &, E const &, char const * name );
}

} }
```

The `serialize` function template is a user-defined customization point. If provided, it is called by the serialization system to output error objects; see [custom-writers].

`throw_exception`

#include <boost/leaf/exception.hpp>

```
namespace boost { namespace leaf {

  template <class Ex, class... E> ①
  [[noreturn]] void throw_exception( Ex && ex, E && ... e );

  template <class E1, class... E> ②
  [[noreturn]] void throw_exception( E1 && e1, E && ... e );

  [[noreturn]] void throw_exception(); ③

  template <class Ex, class... E> ④
  [[noreturn]] void throw_exception( error_id id, Ex && ex, E && ... e );

  template <class E1, class... E> ⑤
  [[noreturn]] void throw_exception( error_id id, E1 && e1, E && ... e );

  [[noreturn]] void throw_exception( error_id id ); ⑥

} }
```

The `throw_exception` function is overloaded: it can be invoked with no arguments, or else there are several alternatives, selected using `std::enable_if` based on the type of the passed arguments. All overloads throw an exception:

① Selected if the first argument is not of type `error_id` and is an exception object, that is, iff `Ex` derives publicly from `std::exception`. In this case the thrown exception is of unspecified type which derives publicly from `Ex` **and** from class `error_id`, such that:

- its `Ex` subobject is initialized by `std::forward<Ex>(ex)`;
- its `error_id` subobject is initialized by `new_error(std::forward<E>(e)...)`.

② Selected if the first argument is not of type `error_id` and is not an exception object. In this case the thrown exception is of unspecified type which derives publicly from `std::exception` **and** from class `error_id`, such that:

- its `std::exception` subobject is default-initialized;
- its `error_id` subobject is initialized by `new_error(std::forward<E1>(e1), std::forward<E>(e)…)`.

③ If the function is invoked without arguments, the thrown exception is of unspecified type which derives publicly from `std::exception` **and** from class `error_id`, such that:

- its `std::exception` subobject is default-initialized;
- its `error_id` subobject is initialized by `new_error()`.

④ Selected if the first argument is of type `error_id` and the second argument is an exception object, that is, iff `Ex` derives publicly from `std::exception`. In this case the thrown exception is of unspecified type which derives publicly from `Ex` **and** from class `error_id`, such that:

- its `Ex` subobject is initialized by `std::forward<Ex>(ex)`;
- its `error_id` subobject is initialized by `id.load(std::forward<E>(e)…)`.

⑤ Selected if the first argument is of type `error_id` and the second argument is not an exception object. In this case the thrown exception is of unspecified type which derives publicly from `std::exception` **and** from class `error_id`, such that:

- its `std::exception` subobject is default-initialized;
- its `error_id` subobject is initialized by `id.load(std::forward<E1>(e1), std::forward<E>(e)…)`.

⑥ If `throw_exception` is invoked with just an `error_id` object, the thrown exception is of unspecified type which derives publicly from `std::exception` **and** from class `error_id`, such that:

- its `std::exception` subobject is default-initialized;
- its `error_id` subobject is initialized by copying from `id`.

The first three overloads throw an exception object that is associated with a new `error_id`. The second three overloads throw an exception object that is associated with the specified `error_id`.

Example 1:

```
struct my_exception: std::exception { };

leaf::throw_exception(my_exception{}); ①
```

① Throws an exception of a type that derives from `error_id` and from `my_exception` (because `my_exception` derives from `std::exception`).

Example 2:

```
enum class my_error { e1=1, e2, e3 }; ①

leaf::throw_exception(my_error::e1);
```

① Throws an exception of a type that derives from `error_id` and from `std::exception` (because `my_error` does not derive from `std::exception`).

To automatically capture `__FILE__`, `__LINE__` and `__FUNCTION__` with the returned object, use `BOOST_LEAF_THROW_EXCEPTION` instead of `leaf::throw_exception`.

`to_variant`

#include <boost/leaf/to_variant.hpp>

```
namespace boost { namespace leaf {

  template <class... E, class TryBlock>
  std::variant<
    typename std::decay<decltype(std::declval<TryBlock>()().value())>::type,
    std::tuple<
      std::optional<E>...>>
  to_variant( TryBlock && try_block );

} }
```

Requires:

- This function is only available under C++-17 or newer.
- The `try_block` function may not take any arguments.
- The type returned by the `try_block` function must be a `result<T>` type (see `is_result_type`). It is valid for the `try_block` to return `leaf::result<T>`, however this is not a requirement.

The `to_variant` function uses `try_handle_all` internally to invoke the `try_block` and capture the result in a `std::variant`. On success, the variant contains the `T` object from the produced `result<T>`. Otherwise, the variant contains a `std::tuple` where each `std::optional` element contains an object of type E_i from the user-supplied sequence `E...`, or is empty if the failure did not produce an error object of that type.

Example:

```
enum class E1 { e11, e12, e13 };
enum class E2 { e21, e22, e23 };
enum class E3 { e31, e32, e33 };

....

auto v = leaf::to_variant<E1, E2, E3>(
  []() -> leaf::result<int>
  {
    return leaf::new_error( E1::e12, E3::e33 );
```

```
  } );

assert(v.index() == 1); ①
auto t = std::get<1>(v); ②

assert(std::get<0>(t).value() == E1::e12); ③
assert(!std::get<1>(t).has_value()); ④
assert(std::get<2>(t).value() == E3::e33); ③
```

① We report a failure, so the variant must contain the error object tuple, rather than an `int`.

② Grab the error tuple.

③ We communicated an `E1` and an `E3` error object...

④ ...but not an `E2` error object.

`try_capture_all`

#include <boost/leaf/handle_errors.hpp>

```
#if BOOST_LEAF_CFG_CAPTURE

namespace boost { namespace leaf {

  template <class TryBlock>
  result<T> // T deduced depending on TryBlock return type
  try_capture_all( TryBlock && try_block ) noexcept;

} }

#endif
```

Return type:

An instance of `leaf::result<T>`, where T is deduced depending on the return type `R` of the `TryBlock`:

- If `R` is a some type `Result<T>` for which `is_result_type` is true, `try_capture_all` returns `leaf::result<T>`.
- Otherwise it is assumed that the `TryBlock` reports errors by throwing exceptions, and the return value of `try_capture_all` is deduced as `leaf::result<R>`.

Effects:

`try_capture_all` executes `try_block`, catching and capturing all exceptions and all communicated error objects in the returned `leaf::result` object. The error objects are allocated dynamically.

Calls to `try_capture_all` must not be nested in `try_handle_all`/`try_handle_some`/`try_catch` or in another `try_capture_all`.

Under `BOOST_LEAF_CFG_CAPTURE=0`, `try_capture_all` is unavailable.

See also:

Transporting Errors Between Threads.

try_catch

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()())>::type
  try_catch( TryBlock && try_block, H && ... h );

} }
```

The `try_catch` function works similarly to `try_handle_some`, except that it does not use or understand the semantics of `result<T>` types; instead:

- It assumes that the `try_block` throws to indicate a failure, in which case `try_catch` will attempt to find a suitable handler among `h...`;
- If a suitable handler isn't found, the original exception is re-thrown using `throw;`.

See Exception Handling.

try_handle_all

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()().value())>::type
  try_handle_all( TryBlock && try_block, H && ... h );

} }
```

The `try_handle_all` function works similarly to `try_handle_some`, except:

- In addition, it requires that at least one of `h...` can be used to handle any error (this requirement is enforced at compile time);
- If the `try_block` returns some `result<T>` type, it must be possible to initialize a value of type `T` with the value returned by each of `h...`, and

- Because it is required to handle all errors, `try_handle_all` unwraps the `result<T>` object `r` returned by the `try_block`, returning `r.value()` instead of `r`.

See Error Handling.

`try_handle_some`

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  template <class TryBlock, class... H>
  typename std::decay<decltype(std::declval<TryBlock>()())>::type
  try_handle_some( TryBlock && try_block, H && ... h );

} }
```

Requires:

- The `try_block` function may not take any arguments.
- The type `R` returned by the `try_block` function must be a `result<T>` type (see `is_result_type`). It is valid for the `try_block` to return `leaf::result<T>`, however this is not a requirement.
- Each of the `h...` functions:
  - must return a type that can be used to initialize an object of the type `R`; in case R is a `result<void>` (that is, in case of success it does not communicate a value), handlers that return `void` are permitted. If such a handler is selected, the `try_handle_some` return value is initialized by `{}`;
  - may take any error objects, by value, by (`const`) reference, or as pointer (to `const`);
  - may take arguments, by value, of any predicate type: `catch_`, `match`, `match_value`, `match_member`, `if_not`, or of any user-defined predicate type `Pred` for which `is_predicate<Pred>::value` is `true`;
  - may take an `error_info` argument by `const &`;
  - may take a `diagnostic_info` argument by `const &`;
  - may take a `diagnostic_details` argument by `const &`.

Effects:

- Creates a local `context<E...>` object `ctx`, where the `E...` types are automatically deduced from the types of arguments taken by each of `h...`, which guarantees that `ctx` is able to store all of the types required to handle errors.
- Invokes the `try_block`:
  - if the returned object `r` indicates success and the `try_block` did not throw, `r` is forwarded to the caller.

- otherwise, LEAF considers each of the `h...` handlers, in order, until it finds one that it can supply with arguments using the error objects currently stored in `ctx`, associated with `r.error()`. The first such handler is invoked and its return value is used to initialize the return value of `try_handle_some`, which can indicate success if the handler was able to handle the error, or failure if it was not.
- if `try_handle_some` is unable to find a suitable handler, it returns `r`.

`try_handle_some` is exception-neutral: it does not throw exceptions, however the `try_block` and any of `h...` are permitted to throw.

Handler Selection Procedure:

A handler `h` is suitable to handle the failure reported by `r` iff `try_handle_some` is able to produce values to pass as its arguments, using the error objects currently available in `ctx`, associated with the error ID obtained by calling `r.error()`. As soon as it is determined that an argument value can not be produced, the current handler is dropped and the selection process continues with the next handler, if any.

The return value of `r.error()` must be implicitly convertible to `error_id`. Naturally, the `leaf::result` template satisfies this requirement. If an external `result` type is used instead, usually `r.error()` would return a `std::error_code`, which is able to communicate LEAF error IDs; see Interoperability.

If `err` is the `error_id` obtained from `r.error()`, each argument a_i taken by the handler currently under consideration is produced as follows:

- If a_i is of type A_i, A_i `const&` or A_i`&`:
  - If an error object of type A_i, associated with `err`, is currently available in `ctx`, a_i is initialized with a reference to that object; otherwise
  - If A_i derives from `std::exception`, and the `try_block` throws an object `ex` of type that derives from `std::exception`, LEAF obtains A_i`* p = dynamic_cast<`A_i`*>(&ex)`. The handler is dropped if `p` is null, otherwise a_i is initialized with `*p`.
  - Otherwise the handler is dropped.

Example:

```
....
auto r = leaf::try_handle_some(

  []() -> leaf::result<int>
  {
    return f();
  },

  [](leaf::e_file_name const & fn) ①
  {
    std::cerr << "File Name: \"" << fn.value << '"' << std::endl; ②
```

```
    return 1;
  } );
```

result | e_file_name

① In case the `try_block` indicates a failure, this handler will be selected if `ctx` stores an `e_file_name` associated with the error. Because this is the only supplied handler, if an `e_file_name` is not available, `try_handle_some` will return the `leaf::result<int>` returned by `f`.

② Print the file name, handle the error.

- If a_i is of type A_i `const*` or A_i`*`, `try_handle_some` is always able to produce it: first it attempts to produce it as if it is taken by reference; if that fails, rather than dropping the handler, a_i is initialized with `0`.

  Example:

```
....
try_handle_some(

  []() -> leaf::result<int>
  {
    return f();
  },

  [](leaf::e_file_name const * fn) ①
  {
    if( fn ) ②
      std::cerr << "File Name: \"" << fn->value << '"' << std::endl;

    return 1;
  } );
```

result | e_file_name

① This handler can be selected to handle any error, because it takes `e_file_name` as a `const *` (and nothing else).

② If an `e_file_name` is available with the current error, print it.

- If a_i is of a predicate type `Pred` (for which `is_predicate<Pred>::value` is `true`), `E` is deduced as `typename Pred::error_type`, and then:
  - If `E` is not `void`, and an error object `e` of type `E`, associated with `err`, is not currently stored in `ctx`, the handler is dropped; otherwise the handler is dropped if the expression `Pred::evaluate(e)` returns `false`.
  - if `E` is `void`, and a `std::exception` was not caught, the handler is dropped; otherwise the handler is dropped if the expression `Pred::evaluate(e)`, where `e` is of type `std::exception const &`, returns `false`.

◦ To invoke the handler, the `Pred` argument a_i is initialized with `Pred{e}`.

See also: Predicates.

- If a_i is of type `error_info const &`, `try_handle_some` is always able to produce it.

  Example:

```
....
try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  [](leaf::error_info const & info) ①
  {
    std::cerr << "leaf::error_info:\n" << info; ②
    return info.error(); ③
  } );
```

`result` | `error_info`

① This handler matches any error.

② Print error information.

③ Return the original error, which will be returned out of `try_handle_some`.

- If a_i is of type `diagnostic_info const &`, `try_handle_some` is always able to produce it.

  Example:

```
....
try_handle_some(

  []
  {
    return f(); // throws
  },

  [](leaf::diagnostic_info const & info) ①
  {
    std::cerr << "leaf::diagnostic_information:\n" << info; ②
    return info.error(); ③
  } );
```

`result` | `diagnostic_info`

① This handler matches any error.

② Print diagnostic information, including limited information about dropped error objects.

③ Return the original error, which will be returned out of `try_handle_some`.

- If a_i is of type `diagnostic_details const &`, `try_handle_some` is always able to produce it.

  Example:

```
....
try_handle_some(

  []
  {
    return f(); // throws
  },

  [](leaf::diagnostic_details const & info) ①
  {
    std::cerr << "leaf::diagnostic_details\n" << info; ②
    return info.error(); ③
  } );
```

`result` | `diagnostic_details`

① This handler matches any error.

② Print verbose diagnostic information, including values of dropped error objects.

③ Return the original error, which will be returned out of `try_handle_some`.

Reference: Types

The contents of each Reference section are organized alphabetically.

boost_json_encoder

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

```
namespace boost { namespace leaf {

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

    // Enabled if x is assignable to boost::json::value, or
    // if tag_invoke is defined for boost::json::value_from_tag.
    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 error objects to JSON format using Boost.JSON. The `output` function is enabled for:

- Types directly assignable to `boost::json::value`
- Types for which a `tag_invoke` overload for `value_from_tag` can be found via ADL

See Serialization.

context

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  class context
  {
    context( context const & ) = delete;
```

```
    context & operator=( context const & ) = delete;

  public:

    context() noexcept;
    context( context && x ) noexcept;
    ~context() noexcept;

    void activate() noexcept;
    void deactivate() noexcept;
    bool is_active() const noexcept;

    void unload( error_id ) noexcept;

    template <class R, class... H>
    R handle_error( error_id, H && ... ) const;

  };

  template <class... H>
  using context_type_from_handlers = typename <<unspecified>>::type;

} }
```

Constructors | activate | deactivate | is_active | unload | handle_error | context_type_from_handlers

The `context` class template provides storage for each of the specified `E...` types. Typically, `context` objects are not used directly; they're created internally when the `try_handle_some`, `try_handle_all` or `try_catch` functions are invoked, instantiated with types that are automatically deduced from the types of the arguments of the passed handlers.

Independently, users can create `context` objects if they need to capture error objects and then transport them, by moving the `context` object itself.

Even in that case it is recommended that users do not instantiate the `context` template by explicitly listing the `E...` types they want it to be able to store. Instead, use `context_type_from_handlers` or call the `make_context` function template, which deduce the correct `E...` types from a captured list of handler function objects.

To be able to load up error objects in a `context` object, it must be activated. Activating a `context` object `ctx` binds it to the calling thread, setting thread-local pointers of the stored `E...` types to point to the corresponding storage within `ctx`. It is possible, even likely, to have more than one active `context` in any given thread. In this case, activation/deactivation must happen in a LIFO manner. For this reason, it is best to use a `context_activator`, which relies on RAII to activate and deactivate a `context`.

When a `context` is deactivated, it detaches from the calling thread, restoring the thread-local pointers to their pre-`activate` values. Typically, at this point the stored error objects, if any, are either discarded (by default) or moved to corresponding storage in other `context` objects active in

the calling thread (if available), by calling `unload`.

While error handling typically uses `try_handle_some`, `try_handle_all` or `try_catch`, it is also possible to handle errors by calling the member function `handle_error`. It takes an `error_id`, and attempts to select an error handler based on the error objects stored in `*this`, associated with the passed `error_id`.

`context` objects can be moved, as long as they aren't active.

Moving an active `context` results in undefined behavior.

Constructors

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  context<E...>::context() noexcept;

  template <class... E>
  context<E...>::context( context && x ) noexcept;

} }
```

The default constructor initializes an empty `context` object: it provides storage for, but does not contain any error objects.

The move constructor moves the stored error objects from one `context` to the other.

Moving an active `context` object results in undefined behavior.

`activate`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  void context<E...>::activate() noexcept;

} }
```

Requires:

`!is_active()`.

Effects:

Associates `*this` with the calling thread.

Ensures:

`is_active()`.

When a context is associated with a thread, thread-local pointers are set to point each `E...` type in its store, while the previous value of each such pointer is preserved in the `context` object, so that the effect of `activate` can be undone by calling `deactivate`.

When an error object is loaded, it is moved in the last activated (in the calling thread) `context` object that provides storage for its type (note that this may or may not be the last activated `context` object). If no such storage is available, the error object is discarded.

`deactivate`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  void context<E...>::deactivate() noexcept;

} }
```

Requires:

- `is_active()`;
- `*this` must be the last activated `context` object in the calling thread.

Effects:

Un-associates `*this` with the calling thread.

Ensures:

`!is_active()`.

When a context is deactivated, the thread-local pointers that currently point to each individual error object storage in it are restored to their original value prior to calling `activate`.

`handle_error`

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  template <class R, class... H>
  R context<E...>::handle_error( error_id err, H && ... h ) const;
```

```
} }
```

This function works similarly to `try_handle_all`, but rather than calling a `try_block` and obtaining the `error_id` from a returned `result` type, it matches error objects (stored in `*this`, associated with `err`) with a suitable error handler from the `h...` pack.

The caller is required to specify the return type `R`. This is because in general the supplied handlers may return different types (which must all be convertible to `R`).

`is_active`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  bool context<E...>::is_active() const noexcept;

} }
```

Returns:

`true` if the `*this` is active in any thread, `false` otherwise.

`unload`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  void context<E...>::unload( error_id id ) noexcept;

} }
```

Requires:

`!is_active()`.

Effects:

Each stored error object of some type `E` is moved into another `context` object active in the call stack that provides storage for objects of type `E`, if any, or discarded. Target objects are not overwritten if they are associated with the specified `id`, except if `id.value() == 0`.

---

context_activator

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  template <class Ctx>
  class context_activator
  {
    context_activator( context_activator const & ) = delete;
    context_activator & operator=( context_activator const & ) = delete;

  public:

    explicit context_activator( Ctx & ctx ) noexcept;
    context_activator( context_activator && ) noexcept;
    ~context_activator() noexcept;
  };

} }
```

`context_activator` is a simple class that activates and deactivates a `context` using RAII:

If `ctx.is_active()` is `true` at the time the `context_activator` is initialized, the constructor and the destructor have no effects. Otherwise:

- The constructor stores a reference to `ctx` in `*this` and calls `ctx.activate()`.
- The destructor:
  - Has no effects if `ctx.is_active()` is `false` (that is, it is valid to call `deactivate` manually, before the `context_activator` object expires);
  - Otherwise, calls `ctx.deactivate()`.

For automatic deduction of `Ctx`, use `activate_context`.

---

diagnostic_details

#include <boost/leaf/diagnostics.hpp>

```
namespace boost { namespace leaf {

  class diagnostic_details: public diagnostic_info
  {
    //Constructors unspecified

  public:

    template <class Encoder>
    void serialize_to( Encoder & e ) const;
```

```
    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
diagnostic_details const & );
  };

} }
```

Handlers passed to error handling functions such as `try_handle_some`, `try_handle_all` or `try_catch` may take an argument of type `diagnostic_details const &` if they need to print diagnostic information about the error.

The message printed by `operator<<` includes the message printed by `error_info`, followed by information about error objects that were communicated to LEAF (to be associated with the error) for which there was no storage available in any active `context` (these error objects were discarded by LEAF, because no handler needed them).

The additional information includes the types and the values of all such error objects (but see `show_in_diagnostics`).

The `serialize_to` member function is used with the serialization system; see Serialization.

The behavior of `diagnostic_details` (and `diagnostic_info`) is affected by the value of the macro `BOOST_LEAF_CFG_DIAGNOSTICS`:

- If it is 1 (the default), LEAF produces `diagnostic_details` but only if an active error handling context on the call stack takes an argument of type `diagnostic_details`;
- If it is 0, the `diagnostic_details` functionality is stubbed out even for error handling contexts that take an argument of type `diagnostic_details`. This could save some cycles on the error path in some programs (but is probably not worth it).

Using `diagnostic_details` may allocate memory dynamically, but only if an active error handler takes an argument of type `diagnostic_details`.

`diagnostic_info`

#include <boost/leaf/diagnostics.hpp>

```
namespace boost { namespace leaf {

  class diagnostic_info: public error_info
  {
    //Constructors unspecified

  public:
```

```
    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
diagnostic_info const & );
  };

} }
```

Handlers passed to `try_handle_some`, `try_handle_all` or `try_catch` may take an argument of type `diagnostic_info const &` if they need to print diagnostic information about the error.

The message printed by `operator<<` includes the message printed by `error_info`, followed by basic information about error objects that were communicated to LEAF (to be associated with the error) for which there was no storage available in any active `context` (these error objects were discarded by LEAF, because no handler needed them).

The additional information is limited to the type name of the first such error object, as well as their total count.

The `serialize_to` member function is used with the serialization system; see Serialization.

The behavior of `diagnostic_info` (and `diagnostic_details`) is affected by the value of the macro `BOOST_LEAF_CFG_DIAGNOSTICS`:

- If it is 1 (the default), LEAF produces `diagnostic_info` but only if an active error handling context on the call stack takes an argument of type `diagnostic_info`;
- If it is 0, the `diagnostic_info` functionality is stubbed out even for error handling contexts that take an argument of type `diagnostic_info`. This could shave a few cycles off the error path in some programs (but it is probably not worth it).

error_id

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  class error_id
  {
  public:

    error_id() noexcept;

    template <class Enum>
    error_id( Enum e, typename std::enable_if<std::is_error_code_enum<Enum>::value,
```

```
Enum>::type * = 0 ) noexcept;

    error_id( std::error_code const & ec ) noexcept;

    int value() const noexcept;
    explicit operator bool() const noexcept;

    std::error_code to_error_code() const noexcept;

    friend bool operator==( error_id a, error_id b ) noexcept;
    friend bool operator!=( error_id a, error_id b ) noexcept;
    friend bool operator<( error_id a, error_id b ) noexcept;

    template <class... Item>
    error_id load( Item && ... item ) const noexcept;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, error_id );
  };

  bool is_error_id( std::error_code const & ec ) noexcept;

  template <class... E>
  error_id new_error( E && ... e ) noexcept;

  error_id current_error() noexcept;

} }
```

Constructors | value | operator bool | to_error_code | operator==, !=, < | load | is_error_id | new_error | current_error

Values of type `error_id` identify a specific occurrence of a failure across the entire program. They can be copied, moved, assigned to, and compared to other `error_id` objects. They're as efficient as an `int`.

Constructors

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  error_id::error_id() noexcept = default;

  template <class Enum>
  error_id::error_id( Enum e, typename std::enable_if<std::is_error_code_enum<Enum
>::value, Enum>::type * = 0 ) noexcept;

  error_id::error_id( std::error_code const & ec ) noexcept;
```

```
} }
```

A default-initialized `error_id` object does not represent a specific failure. It compares equal to any other default-initialized `error_id` object. All other `error_id` objects identify a specific occurrence of a failure.

When using an object of type `error_id` to initialize a `result<T>` object, it will be initialized in error state, even when passing a default-initialized `error_id` value.

Converting an `error_id` object to `std::error_code` uses an unspecified `std::error_category` which LEAF recognizes. This allows an `error_id` to be transported through interfaces that work with `std::error_code`. The `std::error_code` constructor allows the original `error_id` to be restored.

To check if a given `std::error_code` is actually carrying an `error_id`, use `is_error_id`.

Typically, users create new `error_id` objects by invoking `new_error`. The constructor that takes `std::error_code`, and the one that takes a type `Enum` for which `std::is_error_code_enum<Enum>::value` is `true`, have the following effects:

- If `ec.value()` is `0`, the effect is the same as using the default constructor.
- Otherwise, if `is_error_id(ec)` is `true`, the original `error_id` value is used to initialize `*this`;
- Otherwise, `*this` is initialized by the value returned by `new_error`, while `ec` is passed to `load`, which enables handlers used with `try_handle_some`, `try_handle_all` or `try_catch` to receive it as an argument of type `std::error_code`.

`is_error_id`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  bool is_error_id( std::error_code const & ec ) noexcept;

} }
```

Returns:

`true` if `ec` uses the LEAF-specific `std::error_category` that identifies it as carrying an error ID rather than another error code; otherwise returns `false`.

`load`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  template <class... Item>
  error_id error_id::load( Item && ... item ) const noexcept;

} }
```

Requires:

Each of the `Item`... types must be no-throw movable.

Effects:

- If `this->value()==0`, all of `item`... are discarded and no further action is taken.
- Otherwise, what happens with each `item` depends on its type:
  - If it is a function that takes a single argument of some type `E &`, that function is called with the object of type `E` currently associated with `*this`. If no such object exists, a default-initialized object is associated with `*this` and then passed to the function.
  - If it is a function that takes no arguments, that function is called to obtain an error object which is associated with `*this`, except in the special case of a `void` function, in which case it is invoked and no error object is obtained/loaded.
  - Otherwise, the `item` itself is assumed to be an error object, which is associated with `*this`.

Returns:

`*this`.

`load` discards error objects which are not used in any active error handling calling scope.

When loaded into a `context`, an error object of a type `E` will overwrite the previously loaded object of type `E`, if any.

See also:

Loading of Error Objects.

`operator==, !=, <`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  friend bool operator==( error_id a, error_id b ) noexcept;
  friend bool operator!=( error_id a, error_id b ) noexcept;
```

```
  friend bool operator<( error_id a, error_id b ) noexcept;

} }
```

These functions have the usual semantics, comparing `a.value()` and `b.value()`.

The exact strict weak ordering implemented by `operator<` is not specified. In particular, if for two `error_id` objects `a` and `b`, `a < b` is true, it does not follow that the failure identified by `a` occurred earlier than the one identified by `b`.

`operator bool`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

    explicit error_id::operator bool() const noexcept;

} }
```

Effects:

As if `return value()!=0`.

`to_error_code`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

    std::error_code error_id::to_error_code() const noexcept;

} }
```

Effects:

Returns a `std::error_code` with the same `value()` as `*this`, using an unspecified `std::error_category`.

The returned object can be used to initialize an `error_id`, in which case the original `error_id` value will be restored.

Use `is_error_id` to check if a given `std::error_code` carries an `error_id`.

`value`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

    int error_id::value() const noexcept;

} }
```

Effects:

- If `*this` was initialized using the default constructor, returns 0.
- Otherwise returns an `int` that is guaranteed to not be 0: a program-wide unique identifier of the failure.

`error_monitor`

#include <boost/leaf/on_error.hpp>

```
namespace boost { namespace leaf {

  class error_monitor
  {
  public:

    error_monitor() noexcept;

    error_id check() const noexcept;

    error_id assigned_error_id( E && ... e ) const noexcept;
  };

} }
```

This class helps obtain an `error_id` to associate error objects with, when augmenting failures communicated using LEAF through uncooperative APIs that do not use LEAF to report errors (and therefore do not return an `error_id` on error).

The common usage of this class is as follows:

```
error_code compute_value( int * out_value ) noexcept; ①

leaf::result<int> augmenter() noexcept
{
  leaf::error_monitor cur_err; ②

  int val;
```

```
  auto ec = compute_value(&val);

  if( failure(ec) )
    return cur_err.assigned_error_id().load(e1, e2, ...); ③
  else
    return val; ④
}
```

① Uncooperative third-party API that does not use LEAF, but may result in calling a user callback that does use LEAF. In case our callback reports a failure, we'll augment it with error objects available in the calling scope, even though `compute_value` can not communicate an `error_id`.

② Initialize an `error_monitor` object.

③ The call to `compute_value` has failed:

- If `new_error` was invoked (by the calling thread) after the `augment` object was initialized, `assigned_error_id` returns the last `error_id` returned by `new_error`. This would be the case if the failure originates in our callback (invoked internally by `compute_value`).
- Else, `assigned_error_id` invokes `new_error` and returns that `error_id`.

④ The call was successful, return the computed value.

The `check` function works similarly, but instead of invoking `new_error` it returns a default-initialized `error_id`.

See Using `error_monitor` to Report Errors from C Callbacks.

`e_api_function`

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_api_function {char const * value;};

} }
```

The `e_api_function` type is designed to capture the name of the API function that failed. For example, if you're reporting an error from `fread`, you could use `leaf::e_api_function {"fread"}`.

The passed value is stored as a C string (`char const *`), so `value` should only be initialized with a string literal.

e_at_line

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_at_line { int value; };

} }
```

`e_at_line` can be used to communicate the line number when reporting errors (for example parse errors) about a text file.

---

e_errno

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_errno
  {
    int value;
    explicit e_errno(int value=errno);

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, e_errno
const & );
  };

} }
```

By default, the constructor initializes `value` with `errno`, but the caller can pass a specific error code instead. When printed in automatically-generated diagnostic messages, `e_errno` objects use `strerror` to convert the error code to string.

---

e_file_name

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_file_name { std::string value; };

} }
```

When a file operation fails, you could use `e_file_name` to store the name of the file.

It is probably better to define your own file name wrappers to avoid clashes if different modules all use `leaf::e_file_name`. It is best to use a descriptive name that clarifies what kind of file name it is (e.g. `e_source_file_name`, `e_destination_file_name`), or at least define `e_file_name` in a given module's namespace.

e_LastError

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  namespace windows
  {
    struct e_LastError
    {
      unsigned value;

      explicit e_LastError(unsigned value);

#if BOOST_LEAF_CFG_WIN32
      e_LastError();

      template <class CharT, class Traits>
      friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
e_LastError const & );
#endif
    };
  }

} }
```

`e_LastError` is designed to communicate `GetLastError()` values on Windows. The default constructor initializes `value` via `GetLastError()`. See Configuration.

e_source_location

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  struct e_source_location
  {
    char const * file;
    int line;
    char const * function;
```

```
    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &,
e_source_location const & );
  };

} }
```

The `BOOST_LEAF_NEW_ERROR` and `BOOST_LEAF_THROW_EXCEPTION` macros capture `__FILE__`, `__LINE__` and `__FUNCTION__` into a `e_source_location` object.

e_type_info_name

#include <boost/leaf/common.hpp>

```
namespace boost { namespace leaf {

  struct e_type_info_name { char const * value; };

} }
```

`e_type_info_name` is designed to store the return value of `std::type_info::name`.

error_info

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

  class error_info
  {
    //Constructors unspecified

  public:

    error_id error() const noexcept;

    bool exception_caught() const noexcept;
    std::exception const * exception() const noexcept;

    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, error_info
const & );
  };
```

```
} }
```

Handlers passed to error handling functions such as `try_handle_some`, `try_handle_all` or `try_catch` may take an argument of type `error_info const &` to receive generic information about the error being handled.

The `error` member function returns the program-wide unique `error_id` of the error.

The `exception_caught` member function returns `true` if the handler that received `*this` is being invoked to handle an exception, `false` otherwise.

If handling an exception, the `exception` member function returns a pointer to the `std::exception` subobject of the caught exception, or `0` if that exception could not be converted to `std::exception`.

It is illegal to call the `exception` member function unless `exception_caught()` is `true`.

The `operator<<` overload prints diagnostic information about each error object currently stored in the `context` local to the `try_handle_some`, `try_handle_all` or `try_catch` scope that invoked the handler, but only if it is associated with the `error_id` returned by `error()`.

The `serialize_to` member function is used with the serialization system; see Serialization.

`nlohmann_json_encoder`

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

```
namespace boost { namespace leaf {

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

    // Enabled if to_json is available for Json and T.
    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 error objects to JSON format using unqualified calls to `to_json`. This is compatible with nlohmann/json; See Serialization.

result

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  class result
  {
  public:

    using value_type = T;

    // NOTE: Copy constructor implicitly deleted.
    result( result && r ) noexcept;

    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result( result<U> && r ) noexcept;

    result() noexcept;

    result( T && v ) noexcept;

    result( T const & v );

    result( error_id err ) noexcept;

    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result( U && u );

#if BOOST_LEAF_CFG_STD_SYSTEM_ERROR

    result( std::error_code const & ec ) noexcept;

    template <class Enum, class = typename std::enable_if<std::is_error_code_enum<
Enum>::value, int>::type>
    result( Enum e ) noexcept;

#endif

    // NOTE: Assignment operator implicitly deleted.
    result & operator=( result && r ) noexcept;

    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
```

```
::value>::type>
    result & operator=( result<U> && r ) noexcept;

    bool has_value() const noexcept;
    bool has_error() const noexcept;
    explicit operator bool() const noexcept;

    T const & value() const &;
    T & value() &;
    T const && value() const &&;
    T && value() &&;

    T const * operator->() const noexcept;
    T * operator->() noexcept;

    T const & operator*() const & noexcept;
    T & operator*() & noexcept;
    T const && operator*() const && noexcept;
    T && operator*() && noexcept;

    <<unspecified-type>> error() noexcept;

    template <class... Item>
    error_id load( Item && ... item ) noexcept;

    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, result
const & );
  };

  template <>
  class result<void>
  {
  public:

    using value_type = void;

    // NOTE: Copy constructor implicitly deleted.
    result( result && r ) noexcept;

    result() noexcept;

    result( error_id err ) noexcept;

#if BOOST_LEAF_CFG_STD_SYSTEM_ERROR

    result( std::error_code const & ec ) noexcept;
```

```
    template <class Enum, typename std::enable_if<std::is_error_code_enum<Enum>::
value, Enum>::type>
    result( Enum e ) noexcept;

#endif

    // NOTE: Assignment operator implicitly deleted.
    result & operator=( result && r ) noexcept;

    explicit operator bool() const noexcept;

    void value() const;

    void const * operator->() const noexcept;
    void * operator->() noexcept;

    void operator*() const noexcept;

    <<unspecified-type>> error() noexcept;

    template <class... Item>
    error_id load( Item && ... item ) noexcept;

    template <class Encoder>
    void serialize_to( Encoder & e ) const;

    template <class CharT, class Traits>
    friend std::ostream & operator<<( std::basic_ostream<CharT, Traits> &, result
const &);
  };

  class bad_result: std::exception { };

  template <class T>
  struct is_result_type<result<T>>: std::true_type
  {
  };

} }
```

Constructors | operator= | has_value | has_error | operator bool | value | operator-> | operator* | error | load | serialize_to

The `result<T>` type can be returned by functions which produce a value of type `T` but may fail doing so.

Requires:

`T` must be movable, and its move constructor may not throw.

Invariant:

A `result<T>` object is in one of three states:

- Value state, in which case it contains an object of type `T`, and `value` / `operator*` / `operator->` can be used to access the contained value.
- Error state, in which case it contains an error ID, and calling `value` throws `leaf::bad_result`.
- Dynamic capture state, which is the same as the Error state, but in addition to the error ID, it holds a list of dynamically captured error objects; see `try_capture_all`.

`result<T>` objects are nothrow-moveable but are not copyable.

Constructors

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

    // NOTE: Copy constructor implicitly deleted.

    template <class T>
    result<T>::result( result && r ) noexcept;

    template <class T>
    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result<T>::result( result<U> && r ) noexcept;

    template <class T>
    result<T>::result() noexcept;

    template <class T>
    result<T>::result( T && v ) noexcept;

    template <class T>
    result<T>::result( T const & v );

    template <class T>
    result<T>::result( error_id err ) noexcept;

    template <class T>
    template <class U, class = typename std::enable_if<std::is_convertible<U, T>
::value>::type>
    result<T>::result( U && u );

#if BOOST_LEAF_CFG_STD_SYSTEM_ERROR

    template <class T>
```

```
    result<T>::result( std::error_code const & ec ) noexcept;

    template <class T>
    template <class Enum, class = typename std::enable_if<std::is_error_code_enum<
Enum>::value, int>::type>
    result<T>::result( Enum e ) noexcept;

#endif

} }
```

Requires:

`T` must be movable, and its move constructor may not throw; or `void`.

Effects:

Establishes the `result<T>` invariants:

- To get a `result<T>` in Value state, initialize it with an object of type `T` or use the default constructor.
- To get a `result<T>` in Error state, initialize it with:
  - an `error_id` object.

    

    Initializing a `result<T>` with a default-initialized `error_id` object (for which `.value()` returns `0`) will still result in Error state!

  - a `std::error_code` object.
  - an object of type `Enum` for which `std::is_error_code_enum<Enum>::value` is `true`.
- To get a `result<T>` in dynamic capture state, call `try_capture_all`.

When a `result` object is initialized with a `std::error_code` object, it is used to initialize an `error_id` object, then the behavior is the same as if initialized with `error_id`.

Throws:

- Initializing the `result<T>` in Value state may throw, depending on which constructor of `T` is invoked;
- Other constructors do not throw.

A `result` that is in value state converts to `true` in boolean contexts. A `result` that is not in value state converts to `false` in boolean contexts.

`result<T>` objects are nothrow-moveable but are not copyable.

`error`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  <<unspecified-type>> result<T>::error() noexcept;

} }
```

Returns: A proxy object of unspecified type, implicitly convertible to any instance of the `result` class template, as well as to `error_id`.

- If the proxy object is converted to some `result<U>`:
  - If `*this` is in Value state, returns `result<U>(error_id())`.
  - Otherwise the state of `*this` is moved into the returned `result<U>`.
- If the proxy object is converted to an `error_id`:
  - If `*this` is in Value state, returns a default-initialized `error_id` object.
  - If `*this` is in Error capture state, all captured error objects are loaded in the calling thread, and the captured `error_id` value is returned.
  - If `*this` is in Error state, returns the stored `error_id`.
- If the proxy object is not used, the state of `*this` is not modified.

The returned proxy object refers to `*this`; avoid holding on to it.

`has_error`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  bool result<T>::has_error() const noexcept;

} }
```

Returns:

If `*this` is in value state, returns `false`, otherwise returns `true`.

`has_value`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {
```

```
  template <class T>
  bool result<T>::has_value() const noexcept;

} }
```

Returns:

If `*this` is in value state, returns `true`, otherwise returns `false`.

`load`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  template <class... Item>
  error_id result<T>::load( Item && ... item ) noexcept;

} }
```

This member function is designed for use in `return` statements in functions that return `result<T>` to forward additional error objects to the caller.

Effects:

As if `error_id(this->error()).load(std::forward<Item>(item)…)`.

Returns:

`*this`.

`operator bool`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  result<T>::operator bool() const noexcept;

} }
```

Returns:

If `*this` is in value state, returns `true`, otherwise returns `false`.

---

operator*

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  T const & result<T>::operator*() const noexcept;

  template <class T>
  T & result<T>::operator*() noexcept;

} }
```

Requires:

`*this` must be in value state.

Returns

a reference to the stored value.

---

operator=

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  result<T> & result<T>::operator=( result && ) noexcept;

  template <class T>
  template <class U>
  result<T> & result<T>::operator=( result<U> && ) noexcept;

} }
```

Effects:

Destroys `*this`, then re-initializes it as if using the appropriate `result<T>` constructor. Basic exception-safety guarantee.

---

operator->

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  T const * result<T>::operator->() const noexcept;
```

```
  template <class T>
  T * result<T>::operator->() noexcept;

} }
```

Returns

If `*this` is in value state, returns a pointer to the stored value; otherwise returns `nullptr`.

`value`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  void result<void>::value() const;

  template <class T>
  T const & result<T>::value() const;

  template <class T>
  T & result<T>::value();

  class bad_result: std::exception { };

} }
```

Effects:

- If `*this` is in value state, returns a reference to the stored value.
- If `*this` is in dynamic capture state, the captured error objects are unloaded, and:
  - If `*this` contains a captured exception object `ex`, the behavior is equivalent to `throw_exception(ex)`.
  - Otherwise, the behavior is equivalent to `throw_exception(bad_result{})`.
- If `*this` is in any other state, the behavior is equivalent to `throw_exception(bad_result{})`.

`value_type`

A member type of `result<T>`, defined as a synonym for `T`.

`serialize_to`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  template <class Encoder>
  void result<T>::serialize_to( Encoder & e ) const;

} }
```

The `serialize_to` member function is used with the serialization system; see Serialization.

If the result is in value state, outputs the value. If it is in error state, outputs the `error_id`. If the result holds captured error objects, outputs them as well.

Result objects carry error objects only when in capture state. Otherwise, to output error objects, use `serialize_to` on `diagnostic_details` in an error handling scope.

`show_in_diagnostics`

#include <boost/leaf/handle_errors.hpp>

```
namespace boost { namespace leaf {

template <class E>
struct show_in_diagnostics: std::true_type
{
};

} }
```

This template can be specialized to prevent error objects of sensitive types from appearing in automatically generated diagnostic messages. Example:

```
struct e_user_name
{
  std::string value;
};

namespace boost { namespace leaf {

  template <>
  struct show_in_diagnostics<e_user_name>: std::false_type
  {
  };
```

```
} }
```

Reference: Predicates

The contents of each Reference section are organized alphabetically.

A predicate is a special type of error handler argument which enables the handler selection procedure to consider the *value* of available error objects, not only their type; see Using Predicates to Handle Errors.

The following predicates are available:

- `match`
- `match_value`
- `match_member`
- `catch_`
- `if_not`

In addition, any user-defined type `Pred` for which `is_predicate<Pred>::value` is `true` is treated as a predicate. In this case, it is required that:

- `Pred` defines an accessible member type `error_type` to specify the error object type it requires;
- `Pred` defines an accessible static member function `evaluate`, which returns a boolean type, and can be invoked with an object of type `error_type const &`;
- A `Pred` instance can be initialized with an object of type `error_type`.

When an error handler takes an argument of a predicate type `Pred`, the handler selection procedure drops the handler if an error object `e` of type `Pred::error_type` is not available. Otherwise, the handler is dropped if `Pred::evaluate(e)` returns `false`. If the handler is invoked, the `Pred` argument is initialized with `Pred{e}`.

Predicates are evaluated before the error handler is invoked, and so they may not access dynamic state (of course the error handler itself can access dynamic state, e.g. by means of lambda expression captures).

Example 1:

```
enum class my_error { e1 = 1, e2, e3 };

struct my_pred
{
  using error_type = my_error; ①

  static bool evaluate(my_error) noexcept; ②

  my_error matched; ③
};
```

```
namespace boost { namespace leaf {

  template <>
  struct is_predicate<my_pred>: std::true_type
  {
  };

} }
```

① This predicate requires an error object of type `my_error`.

② The handler selection procedure will call this function with an object `e` of type `my_error` to evaluate the predicate…

③ …and if successful, initialize the `my_pred` error handler argument with `my_pred{e}`.

Example 2:

```
struct my_pred
{
  using error_type = leaf::e_errno; ①

  static bool evaluate(leaf::e_errno const &) noexcept; ②

  leaf::e_errno const & matched; ③
};

namespace boost { namespace leaf {

  template <>
  struct is_predicate<my_pred>: std::true_type
  {
  };

} }
```

① This predicate requires an error object of type `e_errno`.

② The handler selection procedure will call this function with an object `e` of type `e_errno` to evaluate the predicate…

③ …and if successful, initialize the `my_pred` error handler argument with `my_pred{e}`.

`catch_`

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class... Ex>
```

```
  struct catch_
  {
    std::exception const & matched;

    // Other members not specified
  };

  template <class Ex>
  struct catch_<Ex>
  {
    Ex const & matched;

    // Other members not specified
  };

  template <class... Ex>
  struct is_predicate<catch_<Ex...>>: std::true_type
  {
  };

} }
```

is_predicate

When an error handler takes an argument of type that is an instance of the `catch_` template, the handler selection procedure first checks if a `std::exception` was caught. If not, the handler is dropped. Otherwise, the handler is dropped if the caught `std::exception` can not be `dynamic_cast` to any of the specified types `Ex`....

If the error handler is invoked, the `matched` member can be used to access the exception object.

See also: Using Predicates to Handle Errors.

While `catch_` requires that the caught exception object is of type that derives from `std::exception`, it is not required that the `Ex`… types derive from `std::exception`.

Example 1:

```
struct ex1: std::exception { };
struct ex2: std::exception { };

leaf::try_catch(

  []
  {
    return f(); // throws
  },

  [](leaf::catch_<ex1, ex2> c)
```

```
  { ①
    assert(dynamic_cast<ex1 const *>(&c.matched) || dynamic_cast<ex2 const *>(&c
.matched));
    ....
  } );
```

① The handler is selected if `f` throws an exception of type `ex1` or `ex2`.

Example 2:

```
struct ex1: std::exception { };

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  [](ex1 & e)
  { ①
    ....
  } );
```

① The handler is selected if `f` throws an exception of type `ex1`. Notice that if we're interested in only one exception type, as long as that type derives from `std::exception`, the use of `catch_` is not required.

`if_not`

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class P>
  struct if_not
  {
    <<deduced>> matched;

    // Other members not specified
  };

  template <class P>
  struct is_predicate<if_not<P>>: std::true_type
  {
  };

} }
```

When an error handler takes an argument of type `if_not<P>`, where `P` is another predicate type, the handler selection procedure first checks if an error object of the type `E` required by `P` is available. If not, the handler is dropped. Otherwise, the handler is dropped if `P` evaluates to `true`.

If the error handler is invoked, `matched` can be used to access the matched object `E`.

See also Using Predicates to Handle Errors.

Example:

```
enum class my_enum { e1, e2, e3 };

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::if_not<leaf::match<my_enum, my_enum::e1, my_enum::e2>> )
  { ①
    ....
  } );
```

try_handle_some | match

① The handler is selected if an object of type `my_enum`, which **does not** compare equal to `e1` or to `e2`, **is** associated with the detected error.

`match`

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class E, auto... V>
  struct match
  {
    <<deduced>> matched;

    // Other members not specified
  };

  template <class E, auto... V>
  struct is_predicate<match<E, V...>>: std::true_type
  {
  };
```

```
} }
```

is_predicate

When an error handler takes an argument of type `match<E, V…>`, the handler selection procedure first checks if an error object `e` of type `E` is available. If it is not available, the handler is dropped. Otherwise, the handler is dropped if the following condition is not met:

$$p_1 \text{ || } p_2 \text{ || } \dots \ p_n.$$

Where p_i is equivalent to `e == V`$_i$, except if V_i is pointer to a function

$$\texttt{bool (*}V_i\texttt{)(T x)}.$$

In this case it is required that `V`$_i$ `!= 0` and that `x` can be initialized with `E const &`, and then p_i is equivalent to:

$$V_i(e).$$

In particular, it is valid to pass pointer to the function `leaf::category<Enum>` for any V_i, where:

```
std::is_error_code_enum<Enum>::value ||
std::is_error_condition_enum<Enum>::value.
```

In this case, p_i is equivalent to:

```
&e.category() == &std::error_code(Enum{}).category().
```

If the error handler is invoked, `matched` can be used to access `e`.

See also Using Predicates to Handle Errors.

Example 1: Handling of a subset of enum values.

```
enum class my_enum { e1, e2, e3 };

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match<my_enum, my_enum::e1, my_enum::e2> m )
  { ①
    static_assert(std::is_same<my_enum, decltype(m.matched)>::value);
    assert(m.matched == my_enum::e1 || m.matched == my_enum::e2);
    ....
  } );
```

① The handler is selected if an object of type `my_enum`, which compares equal to `e1` or to `e2`, is associated with the detected error.

Example 2: Handling of a subset of std::error_code enum values (requires at least C++17, see Example 4 for a C++11-compatible workaround).

```
enum class my_enum { e1=1, e2, e3 };

namespace std
{
  template <> struct is_error_code_enum<my_enum>: std::true_type { };
}

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match<std::error_code, my_enum::e1, my_enum::e2> m )
  { ①
    static_assert(std::is_same<std::error_code const &, decltype(m.matched)>::value);
    assert(m.matched == my_enum::e1 || m.matched == my_enum::e2);
    ....
  } );
```

① The handler is selected if an object of type `std::error_code`, which compares equal to `e1` or to `e2`, is associated with the detected error.

Example 3: Handling of a specific std::error_code::category (requires at least C++17).

```
enum class enum_a { a1=1, a2, a3 };
enum class enum_b { b1=1, b2, b3 };

namespace std
{
  template <> struct is_error_code_enum<enum_a>: std::true_type { };
  template <> struct is_error_code_enum<enum_b>: std::true_type { };
}

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match<std::error_code, leaf::category<enum_a>, enum_b::b2> m )
  { ①
    static_assert(std::is_same<std::error_code const &, decltype(m.matched)>::value);
```

```
    assert(&m.matched.category() == &std::error_code(enum_a{}).category() || m.matched
== enum_b::b2);
    ....
  } );
```

① The handler is selected if an object of type `std::error_code`, which either has the same `std::error_category` as that of `enum_a` or compares equal to `enum_b::b2`, is associated with the detected error.

The use of the `leaf::category` template requires automatic deduction of the type of each V_i, which in turn requires C++17 or newer. The same applies to the use of `std::error_code` as `E`, but LEAF provides a compatible C++11 workaround for this case, using the template `condition`. The following is equivalent to Example 2:

Example 4: Handling of a subset of std::error_code enum values using the C++11-compatible API.

```
enum class my_enum { e1=1, e2, e3 };

namespace std
{
  template <> struct is_error_code_enum<my_enum>: std::true_type { };
}

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match<leaf::condition<my_enum>, my_enum::e1, my_enum::e2> m )
  {
    static_assert(std::is_same<std::error_code const &, decltype(m.matched)>::value);
    assert(m.matched == my_enum::e1 || m.matched == my_enum::e2);
    ....
  } );
```

Instead of a set of values, the `match` template can be given pointers to functions that implement a custom comparison. In the following example, we define a handler which will be selected to handle any error that communicates an object of the user-defined type `severity` with value greater than 4:

Example 5: Handling of failures with severity::value greater than a specified threshold (requires at least C++17).

```
struct severity { int value; };

template <int S>
constexpr bool severity_greater_than( severity const & e ) noexcept
{
```

```
  return e.value > S;
}

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match<severity, severity_greater_than<4>> m )
  {
    static_assert(std::is_same<severity const &, decltype(m.matched)>::value);
    assert(m.matched.value > 4);
    ....
  } );
```

match_member

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <auto, auto... V>
  struct match_member;

  template <class E, class T, T E::* P, auto... V>
  struct match_member<P, V...>
  {
    E const & matched;

    // Other members not specified
  };

  template <auto P, auto... V>
  struct is_predicate<match_member<P, V...>>: std::true_type
  {
  };

} }
```

is_predicate

This predicate is similar to match_value, but able to bind any accessible data member of E; e.g. match_member<&E::value, V…> is equivalent to match_value<E, V…>.

See also Using Predicates to Handle Errors.

`match_member` requires at least C++17, whereas `match_value` does not.

`match_value`

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class E, auto... V>
  struct match_value
  {
    E const & matched;

    // Other members not specified
  };

  template <class E, auto... V>
  struct is_predicate<match_value<E, V...>>: std::true_type
  {
  };

} }
```

`is_predicate`

This predicate is similar to `match`, but where `match` compares the available error object `e` of type `E` to the specified values `V...`, `match_value` works with `e.value`.

See also Using Predicates to Handle Errors.

Example:

```
struct e_errno { int value; }

leaf::try_handle_some(

  []
  {
    return f(); // returns leaf::result<T>
  },

  []( leaf::match_value<e_errno, ENOENT> m )
  { ①
    static_assert(std::is_same<e_errno const &, decltype(m.matched)>::value);
    assert(m.matched.value == ENOENT);
    ....
  } );
```

① The handler is selected if an object of type `e_errno`, with `.value` equal to `ENOENT`, is associated with the detected error.

Reference: Traits

The contents of each Reference section are organized alphabetically.

`is_predicate`

#include <boost/leaf/pred.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  struct is_predicate: std::false_type
  {
  };

} }
```

The `is_predicate` template is used by the handler selection procedure to detect predicate types. See Using Predicates to Handle Errors.

`is_result_type`

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  template <class R>
  struct is_result_type: std::false_type
  {
  };

} }
```

The error handling functionality provided by `try_handle_some` and `try_handle_all` — including the ability to load error objects of arbitrary types — is compatible with any external `result<T>` type R, as long as for a given object `r` of type `R`:

- If `bool(r)` is `true`, `r` indicates success, in which case it is valid to call `r.value()` to recover the `T` value.
- Otherwise `r` indicates a failure, in which case it is valid to call `r.error()`. The returned value is used to initialize an `error_id` (note: `error_id` can be initialized by `std::error_code`).

To use an external `result<T>` type R, you must specialize the `is_result_type` template so that `is_result_type<R>::value` evaluates to `true`.

Naturally, the provided `leaf::result<T>` class template satisfies these requirements. In addition,

it allows error objects to be transported across thread boundaries, using a `try_capture_all`.

Reference: Macros

The contents of each Reference section are organized alphabetically.

BOOST_LEAF_ASSIGN

#include <boost/leaf/error.hpp>

```
#define BOOST_LEAF_ASSIGN(v, r)\
  auto && <<temp>> = r;\
  if( !<<temp>> )\
    return <<temp>>.error();\
  v = std::forward<decltype(<<temp>>)>(<<temp>>).value()
```

`BOOST_LEAF_ASSIGN` is useful when calling a function that returns `result<T>` (other than `result<void>`), if the desired behavior is to forward any errors to the caller verbatim.

In case of success, the result `value()` of type `T` is assigned to the specified variable `v`, which must have been declared prior to invoking `BOOST_LEAF_ASSIGN`. However, it is possible to use `BOOST_LEAF_ASSIGN` to declare a new variable, by passing in `v` its type together with its name, e.g. `BOOST_LEAF_ASSIGN(auto && x, f())` calls `f`, forwards errors to the caller, while capturing successful values in `x`.

See also BOOST_LEAF_AUTO.

BOOST_LEAF_AUTO

#include <boost/leaf/error.hpp>

```
#define BOOST_LEAF_AUTO(v, r)\
  BOOST_LEAF_ASSIGN(auto v, r)
```

BOOST_LEAF_ASSIGN

`BOOST_LEAF_AUTO` is useful when calling a function that returns `result<T>` (other than `result<void>`), if the desired behavior is to forward any errors to the caller verbatim.

Example:

```
leaf::result<int> compute_value();

leaf::result<float> add_values()
{
  BOOST_LEAF_AUTO(v1, compute_value()); ①
  BOOST_LEAF_AUTO(v2, compute_value()); ②
```

```
  return v1 + v2;
}
```

① Call `compute_value`, bail out on failure, define a local variable `v1` on success.

② Call `compute_value` again, bail out on failure, define a local variable `v2` on success.

Of course, we could write `add_value` without using `BOOST_LEAF_AUTO`. This is equivalent:

```
leaf::result<float> add_values()
{
  auto v1 = compute_value();
  if( !v1 )
    return v1.error();

  auto v2 = compute_value();
  if( !v2 )
    return v2.error();

  return v1.value() + v2.value();
}
```

See also `BOOST_LEAF_ASSIGN`.

BOOST_LEAF_CHECK

#include <boost/leaf/error.hpp>

```
#if BOOST_LEAF_CFG_GNUC_STMTEXPR

#define BOOST_LEAF_CHECK(r)\
  ({\
    auto && <<temp>> = (r);\
    if( !<<temp>> )\
      return <<temp>>.error();\
    std::move(<<temp>>);\
  }).value()

#else

#define BOOST_LEAF_CHECK(r)\
  {\
    auto && <<temp>> = (r);\
    if( !<<temp>> )\
      return <<temp>>.error();\
  }
```

```
#endif
```

`BOOST_LEAF_CHECK` is useful when calling a function that returns `result<void>`, if the desired behavior is to forward any errors to the caller verbatim.

Example:

```
leaf::result<void> send_message( char const * msg );

leaf::result<int> compute_value();

leaf::result<int> say_hello_and_compute_value()
{
  BOOST_LEAF_CHECK(send_message("Hello!")); ①
  return compute_value();
}
```

① Try to send a message, then compute a value, report errors using BOOST_LEAF_CHECK.

Equivalent implementation without `BOOST_LEAF_CHECK`:

```
leaf::result<float> add_values()
{
  auto r = send_message("Hello!");
  if( !r )
    return r.error();

  return compute_value();
}
```

If `BOOST_LEAF_CFG_GNUC_STMTEXPR` is `1` (which is the default under `__GNUC__`), `BOOST_LEAF_CHECK` expands to a GNU C statement expression, which allows its use with non-`void` result types in any expression; see Checking for Errors.

BOOST_LEAF_THROW_EXCEPTION

#include <boost/leaf/exception.hpp>

```
#define BOOST_LEAF_THROW_EXCEPTION <<exact-dedfinition-unspecified>>
```

Effects:

`BOOST_LEAF_THROW_EXCEPTION(e…)` is equivalent to `leaf::throw_exception(e…)`, except the current source location is automatically communicated with the thrown exception, in a `e_source_location` object (in addition to all `e…` objects).

BOOST_LEAF_NEW_ERROR

#include <boost/leaf/error.hpp>

```
#define BOOST_LEAF_NEW_ERROR <<exact-definition-unspecified>>
```

Effects:

`BOOST_LEAF_NEW_ERROR(e…)` is equivalent to `leaf::new_error(e…)`, except the current source location is automatically passed, in a `e_source_location` object (in addition to all `e…` objects).

Configuration

The following configuration macros are recognized:

- `BOOST_LEAF_CFG_DIAGNOSTICS`: Defining this macro as `0` stubs out both `diagnostic_info` and `diagnostic_details` (if the macro is left undefined, LEAF defines it as `1`).
- `BOOST_LEAF_CFG_STD_SYSTEM_ERROR`: Defining this macro as `0` disables the `std::error_code` / `std::error_condition` integration. In this case LEAF does not `#include <system_error>`, which may be too heavy for embedded platforms (if the macro is left undefined, LEAF defines it as `1`).
- `BOOST_LEAF_CFG_STD_STRING`: Defining this macro as `0` disables all use of `std::string` (this requires `BOOST_LEAF_CFG_DIAGNOSTICS=0` as well). In this case LEAF does not `#include <string>` which may be too heavy for embedded platforms (if the macro is left undefined, LEAF defines it as `1`).
- `BOOST_LEAF_CFG_CAPTURE`: Defining this macro as `0` disables `try_capture_all`, which (only if used) allocates memory dynamically (if the macro is left undefined, LEAF defines it as `1`).
- `BOOST_LEAF_CFG_GNUC_STMTEXPR`: This macro controls whether or not `BOOST_LEAF_CHECK` is defined in terms of a GNU C statement expression, which enables its use to check for errors similarly to how the questionmark operator works in some languages (see Checking for Errors). By default the macro is defined as `1` under `__GNUC__`, otherwise as `0`.
- `BOOST_LEAF_CFG_WIN32`: This macro controls the use of Win32 APIs. If left undefined, LEAF defines it as `0` (even on Windows, since including `windows.h` is generally not desirable). The possible values are:
  - `0`: Disables all Win32-specific features.
  - `1`: Includes `windows.h` and enables `e_LastError` support, which is otherwise stubbed out.
  - `2`: In addition, switches LEAF to using the Win32 TLS API instead of C++11 `thread_local`, enabling error objects to be used across DLL boundaries.
- `BOOST_LEAF_NO_EXCEPTIONS`: Disables all exception handling support. If left undefined, LEAF defines it automatically based on the compiler configuration (e.g. `-fno-exceptions`).
- `BOOST_LEAF_NO_THREADS`: Disables all thread safety in LEAF.

Configuring TLS Access

LEAF requires support for thread-local `void` pointers. The available TLS implementations are:

- C++11 `thread_local` keyword (the default).
- Win32 TLS API: selected by defining `BOOST_LEAF_CFG_WIN32=2`. This enables error objects to be used across DLL boundaries.
- Custom TLS array: selected by defining `BOOST_LEAF_USE_TLS_ARRAY`. This is intended for embedded platforms where the C++11 `thread_local` keyword is not available or not suitable.

When using `BOOST_LEAF_USE_TLS_ARRAY`, the user is required to define the following two functions to implement the required TLS access:

```
namespace boost { namespace leaf {

namespace tls
{
    void * read_void_ptr( int tls_index ) noexcept;
    void write_void_ptr( int tls_index, void * p ) noexcept;
}

} }
```

For efficiency, `read_void_ptr` and `write_void_ptr` should be defined `inline`.

Under `BOOST_LEAF_USE_TLS_ARRAY` the following additional configuration macros are recognized:

- `BOOST_LEAF_CFG_TLS_ARRAY_START_INDEX` specifies the start TLS array index available to LEAF (if the macro is left undefined, LEAF defines it as `0`).
- `BOOST_LEAF_CFG_TLS_ARRAY_SIZE` may be defined to specify the size of the TLS array. In this case TLS indices are validated via `BOOST_LEAF_ASSERT` before being passed to `read_void_ptr` / `write_void_ptr`.
- `BOOST_LEAF_CFG_TLS_INDEX_TYPE` may be defined to specify the integral type used to store assigned TLS indices (if the macro is left undefined, LEAF defines it as `unsigned char`).

TLS slots are allocated only for types used in error handlers. The minimum size of the TLS pointer array required by LEAF is the total number of different types used as arguments to error handlers (in the entire program), plus one.

Beware of `read_void_ptr`/`write_void_ptr` accessing thread local pointers beyond the static boundaries of the thread local pointer array; this will likely result in undefined behavior.

Embedded Platforms

Defining `BOOST_LEAF_EMBEDDED` is equivalent to the following:

```
#ifndef BOOST_LEAF_CFG_DIAGNOSTICS
#   define BOOST_LEAF_CFG_DIAGNOSTICS 0
#endif

#ifndef BOOST_LEAF_CFG_STD_SYSTEM_ERROR
#   define BOOST_LEAF_CFG_STD_SYSTEM_ERROR 0
#endif

#ifndef BOOST_LEAF_CFG_STD_STRING
#   define BOOST_LEAF_CFG_STD_STRING 0
#endif
```

```
#ifndef BOOST_LEAF_CFG_CAPTURE
#   define BOOST_LEAF_CFG_CAPTURE 0
#endif
```

LEAF supports FreeRTOS out of the box, define `BOOST_LEAF_TLS_FREERTOS` (in which case LEAF automatically defines `BOOST_LEAF_EMBEDDED`, if it is not defined already).

For other embedded platforms, define `BOOST_LEAF_USE_TLS_ARRAY`, see Configuring TLS Access.

If your program does not use concurrency at all, simply define `BOOST_LEAF_NO_THREADS`, which requires no TLS support at all (but is NOT thread-safe).

Contrary to popular belief, exception handling works great on embedded platforms. In this talk Khalil Estell demonstrates that using exceptions to handle errors leads to a significant reduction in firmware code size (of course LEAF works with or without exception handling).

Portability

The source code is compatible with C++11 or newer.

Running the Unit Tests

The unit tests can be run with Meson Build or with Boost Build. To run the unit tests:

Meson Build

Clone LEAF into any local directory and execute:

```
cd leaf
meson setup _bld/debug
cd _bld/debug
meson test
```

See `meson_options.txt` found in the root directory for available build options.

Boost Build

Assuming the current working directory is `<boostroot>/libs/leaf`:

```
../../b2 test
```

Design Rationale

Definition:

Objects that carry information about error conditions are called error objects. For example, objects of type `std::error_code` are error objects.

The following reasoning is independent of the mechanism used to transport error objects, whether it is exception handling or anything else.

Definition:

Depending on their interaction with error objects, functions can be classified as follows:

- **Error initiating**: functions that initiate error conditions by creating new error objects.
- **Error neutral**: functions that forward to the caller error objects communicated by lower-level functions they call.
- **Error handling**: functions that dispose of error objects they have received, recovering normal program operation.

A crucial observation is that *error initiating* functions are typically low level functions that lack any context and can not determine, much less dictate, the correct program behavior in response to the errors they may initiate. Error conditions which (correctly) lead to termination in some programs may (correctly) be ignored in others; yet other programs may recover from them and resume normal operation.

The same reasoning applies to *error neutral* functions, but in this case there is the additional issue that the errors they need to communicate, in general, are initiated by functions multiple levels removed from them in the call chain, functions which usually are — and should be treated as — implementation details. An *error neutral* function should not be coupled with error object types communicated by *error initiating* functions, for the same reason it should not be coupled with any other aspect of their interface.

Finally, *error handling* functions, by definition, have the full context they need to deal with at least some, if not all, failures. In their scope it is an absolute necessity that the author knows exactly what information must be communicated by lower level functions in order to recover from each error condition. Specifically, none of this necessary information can be treated as implementation details; in this case, the coupling which is to be avoided in *error neutral* functions is in fact desirable.

We're now ready to define our

Design goals:

- **Error initiating** functions should be able to communicate all information available to them that is relevant to the failure being reported.
- **Error neutral** functions should not be coupled with error types communicated by lower-level *error initiating* functions. They should be able to augment any failure with additional relevant information available to them.

- **Error handling** functions should be able to access all the information communicated by *error initiating* or *error neutral* functions that is needed in order to deal with failures.

The design goal that *error neutral* functions are not coupled with the static type of error objects that pass through them seems to require dynamic polymorphism and therefore dynamic memory allocations (the Boost Exception library meets this design goal at the cost of dynamic memory allocation).

As it turns out, dynamic memory allocation is not necessary due to the following

Fact:

- **Error handling** functions "know" which of the information *error initiating* and *error neutral* functions are able to communicate is actually needed in order to deal with failures in a particular program. Ideally, no resources should be ~~used~~ wasted storing or communicating information which is not currently needed to handle errors, even if it is relevant to the failure.

For example, if a library function is able to communicate an error code but the program does not need to know the exact error code, then that information may be ignored at the time the library function attempts to communicate it. On the other hand, if an *error handling* function needs that information, the memory needed to store it can be reserved statically in its scope.

The LEAF functions `try_handle_some`, `try_handle_all` and `try_catch` implement this idea. Users provide error handling lambda functions, each taking arguments of the types it needs in order to recover from a particular error condition. LEAF reserves storage for these types (using automatic storage duration) until they are passed to a suitable handler.

When an error is reported, LEAF checks if storage is available for each error object type:

- If storage is available, the object is stored directly in the error handling scope, bypassing all *error neutral* functions in between.
- If storage is not available, the error object is discarded, since no error handling function makes any use of it — saving resources.

Each error occurrence is assigned a unique `error_id`, which prevents handlers from accessing stale error objects from previous failures. To handle a failure, LEAF matches the available error objects (by their `error_id`) with the argument types required by each handler.

Dynamic Linking

POSIX

When compiling with `-fvisibility=hidden`, error types must be declared with `default` visibility. It is recommended to use `BOOST_LEAF_SYMBOL_VISIBLE`, which expands to the appropriate attribute on POSIX and to nothing on Windows:

```
struct BOOST_LEAF_SYMBOL_VISIBLE my_error_info
{
    int value;
};
```

LEAF declares the following types with `BOOST_LEAF_SYMBOL_VISIBLE`: `error_id`, `error_info`, `diagnostic_info`, `diagnostic_details`, `e_api_function`, `e_file_name`, `e_errno`, `e_type_info_name`, `e_at_line`, `e_source_location`, `bad_result`, `result<T>`.

Windows

To use error objects across DLL boundaries, define `BOOST_LEAF_CFG_WIN32=2`. This switches LEAF to using the Win32 TLS API instead of C++11 `thread_local`. Dynamic unloading of DLLs is not supported.

Modules linked against different CRT configurations (static or dynamic) can share LEAF error objects. However, for two modules to share error objects, they must be compiled with the same compiler (not necessarily the same compiler version). For example, if one module is compiled with MSVC and another with GCC, error objects will not be shared between them.

Alternatives to LEAF

- `std::expected` (C++23)
- Boost Outcome
- Boost Exception

Below we offer a comparison of Boost LEAF to these alternatives.

Comparison to `std::expected`

`std::expected<T, E>`, introduced in C++23, is a vocabulary type for functions that return either a value of type `T` or an error of type `E`. It is similar to `leaf::result<T>` in purpose but differs in design:

- `std::expected<T, E>` encodes the error type `E` in the function signature. `leaf::result<T>` does not specify an error type; error objects are transported separately.
- With `std::expected`, a function can only return a single error object of type `E`. LEAF allows associating multiple error objects of different types with a single failure.
- LEAF's `on_error` allows intermediate functions to attach additional context as errors propagate, without modifying function signatures.

The choice between them depends on whether encoding error types in function signatures is desired, and whether associating multiple error objects with a single failure is needed.

Comparison to Boost Outcome

Like LEAF, Boost Outcome is designed for low latency environments. It provides `result<T,EC,NVP>` and `outcome<T,EC,EP,NVP>`, where `EC` is an error code type (defaulting to `std::error_code`) and `EP` is a pointer type for additional error information (defaulting to `std::exception_ptr`).

The key design difference is in how error information is transported:

- Outcome encodes error types in function signatures via the `EC` and `EP` template parameters, giving callers explicit visibility of possible error types.
- LEAF decouples `leaf::result<T>` from error types entirely. Error objects are transported separately to error handling scopes, where the handlers determine which types are needed.

The LEAF examples include an adaptation of the program from the Boost Outcome `result<>` tutorial. You can view it on GitHub.

LEAF is compatible with `outcome::result<T>`; you don't have to use `leaf::result<T>`.

The Interoperability Problem

The Boost Outcome documentation discusses the challenge of integrating multiple libraries with different error types:

> If library A uses `result<T, libraryA::failure_info>`, and library B uses `result<T, libraryB::error_info>` and so on, there becomes a problem for the application writer who is bringing in these third party dependencies and tying them together into an application. As a general rule, each third party library author will not have built in explicit interoperation support for unknown other third party libraries. The problem therefore lands with the application writer.
>
> The application writer has one of three choices:
>
> 1. In the application, the form of result used is `result<T, std::variant<E1, E2, …>>` where `E1, E2 …` are the failure types for every third party library in use in the application. This has the advantage of preserving the original information exactly, but comes with a certain amount of use inconvenience and maybe excessive coupling between high level layers and implementation detail.
> 2. One can translate/map the third party's failure type into the application's failure type at the point of the failure exiting the third party library and entering the application. One might do this, say, with a C preprocessor macro wrapping every invocation of the third party API from the application. This approach may lose the original failure detail, or mis-map under certain circumstances if the mapping between the two systems is not one-one.
> 3. One can type erase the third party's failure type into some application failure type, which can later be reconstituted if necessary. **This is the cleanest solution with the least coupling issues and no problems with mis-mapping**, but it almost certainly requires the use of `malloc` which the previous two did not.

Boost Outcome addresses this by providing converters in the `outcome::convert` namespace to translate between error types (option 2).

LEAF takes a different approach: error objects are transported directly to error handling scopes where storage is allocated statically, implementing option 3 without dynamic allocation.

Comparison to Boost Exception

Both LEAF and Boost Exception allow transporting arbitrary error information with exceptions. The key design differences:

- Boost Exception stores error objects in the exception object itself (requiring dynamic allocation). LEAF stores error objects in the error handling scope, discarding those not needed by handlers.
- Boost Exception can only augment exceptions deriving from `boost::exception`. LEAF can augment any exception type.

- Boost Exception automatically captures error objects across thread boundaries via `boost::exception_ptr`. LEAF requires explicit use of [capture].

LEAF can access Boost Exception error information using its error handling interface. See Boost Exception Integration.

Acknowledgements

Special thanks to Peter Dimov and Sorin Fetche.

Ivo Belchev, Sean Palmer, Jason King, Vinnie Falco, Glen Fernandes, Augustín Bergé — thanks for the valuable feedback.

Documentation rendered by Asciidoctor with these customizations.