LEAF

Lightweight Error Augmentation Framework

Abstract

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

- Efficient delivery of arbitrary error objects to the correct error-handling scope.
- No dynamic memory allocations.
- Compatible with `std::error_code`, `errno` and any other error code type.
- Can be used with or without exception handling.
- Support for multi-thread programming.

LEAF is designed with a strong bias towards the common use case where callers of functions which may fail check for success and forward errors up the call chain but do not handle them. In this case, only a simple success-or-failure discriminant is transported. Actual error objects are delivered directly to the error-handling scope, skipping the intermediate check-only frames altogether.

Five Minute Introduction

We'll implement two versions of the same simple program: one using `result<T>` to handle errors, and one using exception handling.

Using `result<T>`

We'll write a short but complete program that reads a text file in a buffer and prints it to `std::cout`, using LEAF to handle errors without exception handling.

> LEAF works great Using Exception Handling as well.

Let's jump ahead and start with the `main` function: it will try several operations as needed and handle all the errors that occur. Did I say **all** the errors? I did, so we'll use `leaf::try_handle_all`. It has the following signature:

```
template <class TryBlock, class... Handler>
<<deduced>> try_handle_all( TryBlock && try_block, Handler && ... handler );
```

`TryBlock` is a function, almost always a lambda. It is required to return a `result<T>` type — for example, `leaf::result<T>` — that holds a value of type `T` or else it indicates a failure.

The first thing `try_handle_all` does is invoke the `try_block` function. If the returned object `r` indicates success, `try_handle_all` unwraps it, returning the contained `r.value()`; otherwise it calls the first suitable error handling function from the `handler`… list.

We'll see later just what kind of a `TryBlock` will our `main` function pass to `try_handle_all`, but first, let's look at the juicy error-handling part. In case of an error, LEAF will consider each of the `handler`… functions, in order, and call the first suitable match:

```
int main( int argc, char const * argv[] )
{
  return leaf::try_handle_all(

    [&]() -> leaf::result<int>
    {
      // The TryBlock code goes here, we'll see it later
    },

    // Error handlers below:

    [](leaf::match<error_code, open_error>, leaf::match<leaf::e_errno, ENOENT>, leaf
::e_file_name const & fn)
    { ①
      std::cerr << "File not found: " << fn.value << std::endl;
      return 1;
    },
```

```
    [](leaf::match<error_code, open_error>, leaf::e_errno const & errn, leaf
::e_file_name const & fn)
    { ②
      std::cerr << "Failed to open " << fn.value << ", errno=" << errn << std::endl;
      return 2;
    },

    [](leaf::match<error_code, size_error, read_error, eof_error>, leaf::e_errno const
* errn, leaf::e_file_name const & fn)
    { ③
      std::cerr << "Failed to access " << fn.value;
      if( errn )
        std::cerr << ", errno=" << *errn;
      std::cerr << std::endl;
      return 3;
    },

    [](leaf::match<error_code, output_error>, leaf::e_errno const & errn)
    { ④
      std::cerr << "Output error, errno=" << errn << std::endl;
      return 4;
    },

    [](leaf::match<error_code, bad_command_line>)
    { ⑤
      std::cout << "Bad command line argument" << std::endl;
      return 5;
    },

    [](leaf::error_info const & unmatched)
    { ⑥
      std::cerr <<
        "Unknown failure detected" << std::endl <<
        "Cryptic diagnostic information follows" << std::endl <<
        unmatched;
      return 6;
    }
  );
}
```

① This handler will be called if the detected error includes:

- an object of type `enum error_code` equal to the value `open_error`, and
- an object of type `leaf::e_errno` that has `.value` equal to `ENOENT`, and
- an object of type `leaf::e_file_name`.

② This handler will be called if the detected error includes:

- an object of type `enum error_code` equal to `open_error`, and
- an object of type `leaf::e_errno` (regardless of its `.value`), and
- an object of type `leaf::e_file_name`.

③ This handler will be called if the detected error includes:

- an object of type `enum error_code` equal to any of `size_error`, `read_error`, `eof_error`, and
- an optional object of type `leaf::e_errno` (regardless of its `.value`), and
- an object of type `leaf::e_file_name`.

④ This handler will be called if the detected error includes:

- an object of type `enum error_code` equal to `output_error`, and
- an object of type `leaf::e_errno` (regardless of its `.value`),

⑤ This handler will be called if the detected error includes an object of type `enum error_code` equal to `bad_command_line`.

⑥ This last handler is a catch-all for any error, in case no other handler could be selected: it prints diagnostic information to help debug logic errors in the program, since it failed to match an appropriate error handler to the error condition it encountered.

> It is critical to understand that the error handlers are considered in order, rather than by finding a "best match". No error handler is "better" than the others: LEAF will call the first one for which all of the arguments can be supplied using the available error objects. See handler selection procedure for details.

Now, reading and printing a file may not seem like a complex job, but let's split it into several functions, each communicating failures using `leaf::result<T>`:

```
leaf::result<char const *> parse_command_line( int argc, char const * argv[] )
noexcept; ①

leaf::result<std::shared_ptr<FILE>> file_open( char const * file_name ) noexcept; ②

leaf::result<int> file_size( FILE & f ) noexcept; ③

leaf::result<void> file_read( FILE & f, void * buf, int size ) noexcept; ④
```

① Parse the command line, return the file name.

② Open a file for reading.

③ Return the size of the file.

④ Read size bytes from f into buf.

For example, let's look at `file_open`:

```
leaf::result<std::shared_ptr<FILE>> file_open( char const * file_name ) noexcept
{
  if( FILE * f = fopen(file_name,"rb") )
    return std::shared_ptr<FILE>(f,&fclose);
  else
    return leaf::new_error(open_error, leaf::e_errno{errno});
}
```

If fopen succeeds, we return a shared_ptr which will automatically call fclose as needed. If fopen fails, we report an error by calling new_error, which takes any number of error objects to load with the error. In this case we pass the system errno (LEAF defines struct e_errno {int value;}), and our own error code value, open_error.

Here is our complete error code enum:

```
enum error_code
{
  bad_command_line = 1,
  open_error,
  read_error,
  size_error,
  eof_error,
  output_error
};
```

We're now ready to look at the TryBlock we'll pass to try_handle_all. It does all the work, bails out if it encounters an error:

```
int main( int argc, char const * argv[] )
{
  return leaf::try_handle_all(

    [&]() -> leaf::result<int>
    {
      leaf::result<char const *> file_name = parse_command_line(argc,argv);
      if( !file_name )
        return file_name.error();
```

Wait, what's this, if "error" return "error"? There is a better way: we'll use BOOST_LEAF_AUTO. It takes a result<T> and bails out in case of a failure (control leaves the calling function), otherwise defines a local variable to access the T value stored in the result object.

This is what our TryBlock really looks like:

```
int main( int argc, char const * argv[] )
{
  return leaf::try_handle_all(

    [&]() -> leaf::result<int> ①
    {
      BOOST_LEAF_AUTO(file_name, parse_command_line(argc,argv)); ②

      auto load = leaf::on_error( leaf::e_file_name{file_name} ); ③

      BOOST_LEAF_AUTO(f, file_open(file_name)); ④

      BOOST_LEAF_AUTO(s, file_size(*f)); ④

      std::string buffer( 1 + s, '\0' );
      BOOST_LEAF_CHECK(file_read(*f, &buffer[0], buffer.size()-1)); ④

      std::cout << buffer;
      std::cout.flush();
      if( std::cout.fail() )
        return leaf::new_error(output_error, leaf::e_errno{errno});

      return 0;
    },

    .... ⑤

  ); ⑥
}
```

① Our `TryBlock` returns a `result<int>`. In case of success, it will hold `0`, which will be returned from `main` to the OS.

② If `parse_command_line` returns an error, we forward that error to `try_handle_all` (which invoked us) verbatim. Otherwise, `BOOST_LEAF_AUTO` gets us a local variable `file_name` to access the `char const *` result.

③ From now on, all errors escaping this scope will automatically communicate the (now successfully parsed from the command line) file name (LEAF defines `struct e_file_name {std::string value;}`). It's as if every time one of the following functions wants to report an error, `on_error` says "wait, associate this `e_file_name` object with the error, it's important!"

④ Call more functions, forward each failure to the caller.

⑤ List of error handlers goes here. We'll see that later.

⑥ This concludes the `try_handle_all` arguments — as well as our program!

Nice and simple! Writing the `TryBlock`, we focus on the "no errors" code path — if we encounter any error we just return it to `try_handle_all` for processing. Well, that's if we're being good and using RAII for automatic clean-up — which we are, `shared_ptr` will automatically close the file for us.

The complete program from this tutorial is available here. The other version of the same program uses exception handling to report errors (see below).

Using Exception Handling

And now, we'll write the same program that reads a text file in a buffer and prints it to `std::cout`, this time using exceptions to report errors. First, we need to define our exception class hierarchy:

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

We'll split the job into several functions, communicating failures by throwing exceptions:

```
char const * parse_command_line( int argc, char const * argv[] ); ①

std::shared_ptr<FILE> file_open( char const * file_name ); ②

int file_size( FILE & f ); ③

void file_read( FILE & f, void * buf, int size ); ④
```

① Parse the command line, return the file name.

② Open a file for reading.

③ Return the size of the file.

④ Read size bytes from f into buf.

The `main` function brings everything together and handles all the exceptions that are thrown, but instead of using `try` and `catch`, it will use the function template `leaf::try_catch`, which has the following signature:

```
template <class TryBlock, class... Handler>
<<deduced>> try_catch( TryBlock && try_block, Handler && ... handler );
```

`TryBlock` is a function, almost always a lambda; `try_catch` simply returns the value returned by the `try_block`, catching any exception it throws, in which case it calls the first suitable error handling function from the `handler`… list.

Let's first look at the `TryBlock` our `main` function passes to `try_catch`:

```
int main( int argc, char const * argv[] )
{
  return leaf::try_catch(

    [&] ①
    {
      char const * file_name = parse_command_line(argc,argv); ②

      auto load = leaf::on_error( leaf::e_file_name{file_name} ); ③

      std::shared_ptr<FILE> f = file_open( file_name ); ②

      std::string buffer( 1+file_size(*f), '\0' ); ②
      file_read(*f, &buffer[0], buffer.size()-1); ②

      std::cout << buffer;
      std::cout.flush();
      if( std::cout.fail() )
        throw leaf::exception(output_error{}, leaf::e_errno{errno});

      return 0;
    },

    .... ④

  ); ⑤
}
```

① Except if it throws, our `TryBlock` returns `0`, which will be returned from `main` to the OS.

② If any of the functions we call throws, `try_catch` will find an appropriate handler to invoke. We'll look at that later.

③ From now on, all exceptions escaping this scope will automatically communicate the (now successfully parsed from the command line) file name (LEAF defines `struct e_file_name {std::string value;}`). It's as if every time one of the following functions wants to throw an exception, `on_error` says "wait, associate this `e_file_name` object with the exception, it's important!"

④ List of error handlers goes here. We'll see that later.

⑤ This concludes the `try_catch` arguments — as well as our program!

As it is always the case when using exception handling, as long as our `TryBlock` is exception-safe, we can focus on the "no errors" code path. Of course, our `TryBlock` is exception-safe, since `shared_ptr` will automatically close the file for us in case an exception is thrown.

Now let's look at the second part of the call to `try_catch`, which lists the error handlers:

```
int main( int argc, char const * argv[] )
{
```

```
  return leaf::try_catch(
    [&]
    {
      // The TryBlock code goes here (previous listing)
    },

    // Error handlers below:

    [](open_error &, leaf::match<leaf::e_errno,ENOENT>, leaf::e_file_name const & fn)
    { ①
      std::cerr << "File not found: " << fn.value << std::endl;
      return 1;
    },

    [](open_error &, leaf::e_file_name const & fn, leaf::e_errno const & errn)
    { ②
      std::cerr << "Failed to open " << fn.value << ", errno=" << errn << std::endl;
      return 2;
    },

    [](input_error &, leaf::e_errno const * errn, leaf::e_file_name const & fn)
    { ③
      std::cerr << "Failed to access " << fn.value;
      if( errn )
        std::cerr << ", errno=" << *errn;
      std::cerr << std::endl;
      return 3;
    },

    [](output_error &, leaf::e_errno const & errn)
    { ④
      std::cerr << "Output error, errno=" << errn << std::endl;
      return 4;
    },

    [](bad_command_line &)
    { ⑤
      std::cout << "Bad command line argument" << std::endl;
      return 5;
    },

    [](leaf::error_info const & unmatched)
    { ⑥
      std::cerr <<
        "Unknown failure detected" << std::endl <<
        "Cryptic diagnostic information follows" << std::endl <<
        unmatched;
      return 6;
    } );
}
```

① This handler will be called if:

- an `open_error` exception was caught, with
- an object of type `leaf::e_errno` that has `.value` equal to `ENOENT`, and
- an object of type `leaf::e_file_name`.

② This handler will be called if:

- an `open_error` exception was caught, with
- an object of type `leaf::e_errno` (regardless of its `.value`), and
- an object of type `leaf::e_file_name`.

③ This handler will be called if:

- an `input_error` exception was caught (which is a base type), with
- an optional object of type `leaf::e_errno` (regardless of its `.value`), and
- an object of type `leaf::e_file_name`.

④ This handler will be called if:

- an `output_error` exception was caught, with
- an object of type `leaf::e_errno` (regardless of its `.value`),

⑤ This handler will be called if a `bad_command_line` exception was caught.

⑥ If `try_catch` fails to find an appropriate handler, it will re-throw the exception. But this is the `main` function which should handle all exceptions, so this last handler matches any error and prints diagnostic information, to help debug logic errors.

> It is critical to understand that the error handlers are considered in order, rather than by finding a "best match". No error handler is "better" than the others: LEAF will call the first one for which all of the arguments can be supplied using the available error objects. See handler selection procedure for details.

To conclude this introduction, let's look at one of the error-reporting functions that our `TryBlock` calls, for example `file_open`:

```
std::shared_ptr<FILE> file_open( char const * file_name )
{
  if( FILE * f = fopen(file_name,"rb") )
    return std::shared_ptr<FILE>(f,&fclose);
  else
    throw leaf::exception(open_error{}, leaf::e_errno{errno});
}
```

If `fopen` succeeds, it returns a `shared_ptr` which will automatically call `fclose` as needed. If `fopen` fails, we throw the exception object returned by `leaf::exception`, which in this case is of type that derives from `open_error`; the passed `e_errno` object will be associated with the exception.

> `try_catch` works with any exception, not only exceptions thrown using `leaf::exception`.

The complete program from this tutorial is available here. The other version of the same program does not use exception handling to report errors (see the previous introduction).

Tutorial

Error Communication Model

Using `noexcept` Functionality

The following figure illustrates how error objects are transported when using LEAF without exception handling:

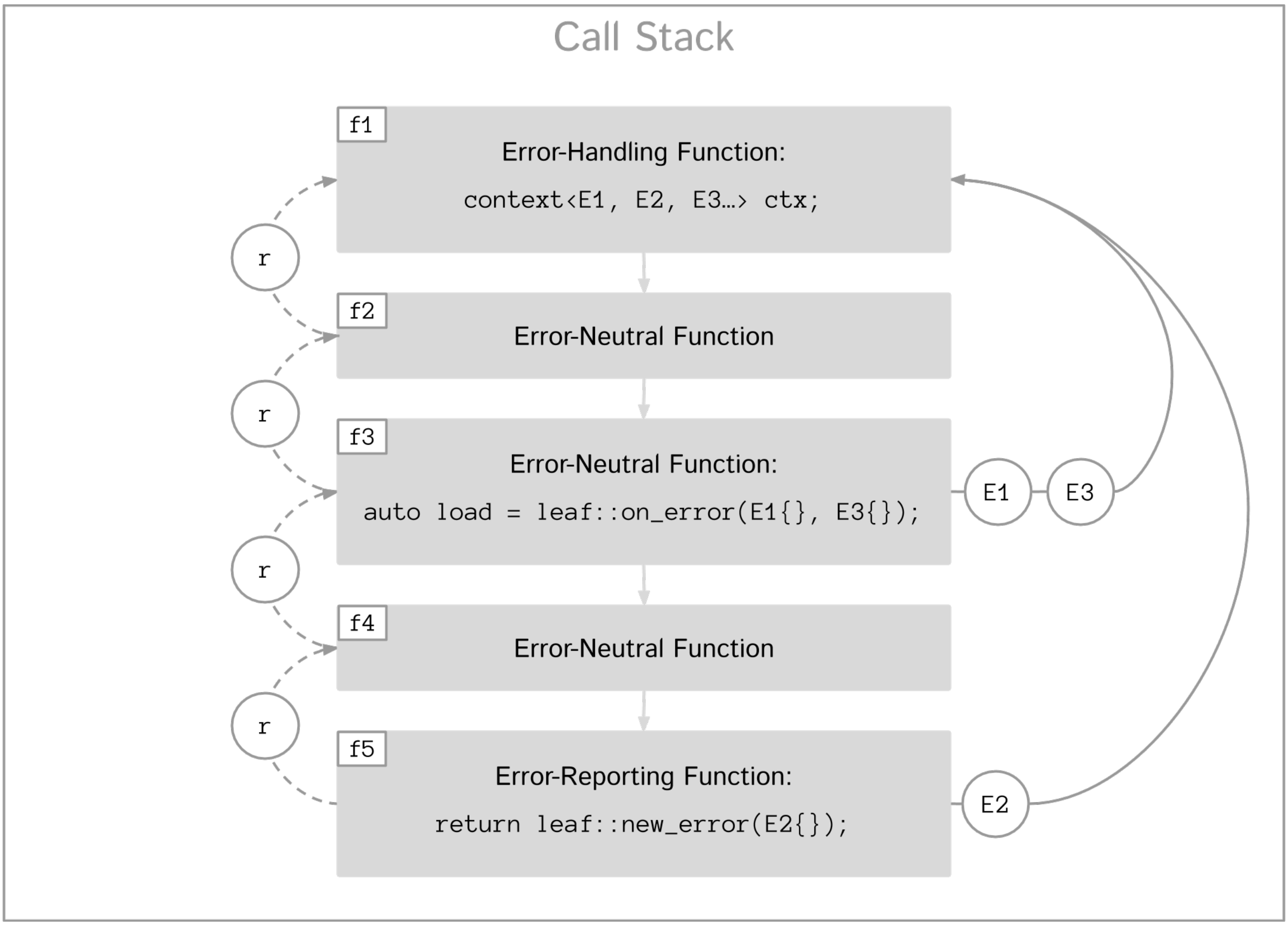

Figure 1. LEAF noexcept Error Communication Model

The arrows pointing down indicate the call stack order for the functions `f1` through `f5`: higher level functions calling lower level functions.

Note the call to `on_error` in `f3`: it caches the passed error objects of types `E1` and `E3` in the returned object `load`, where they stay ready to be communicated in case any function downstream from `f3` reports an error. Presumably these objects are relevant to any such failure, but are conveniently accessible only in this scope.

Figure 1 depicts the condition where `f5` has detected an error. It calls `leaf::new_error` to create a new, unique `error_id`. The passed error object of type `E2` is immediately loaded in the first active `context` object that provides static storage for it, found in any calling scope (in this case `f1`), and is associated with the newly-generated `error_id` (solid arrow);

The `error_id` itself is returned to the immediate caller `f4`, usually stored in a `result<T>` object `r`.

That object takes the path shown by dashed arrows, as each error-neutral function, unable to handle the failure, forwards it to its immediate caller in the returned value — until an error-handling scope is reached.

When the destructor of the `load` object in `f3` executes, it detects that `new_error` was invoked after its initialization, loads the cached objects of types `E1` and `E3` in the first active `context` object that provides static storage for them, found in any calling scope (in this case `f1`), and associates them with the last generated `error_id` (solid arrow).

When the error-handling scope `f1` is reached, it probes `ctx` for any error objects associated with the `error_id` it received from `f2`, and processes a list of user-provided error handlers, in order, until it finds a handler with arguments that can be supplied using the available (in `ctx`) error objects. That handler is called to deal with the failure.

Using Exception Handling

The following figure illustrates the slightly different error communication model used when errors are reported by throwing exceptions:

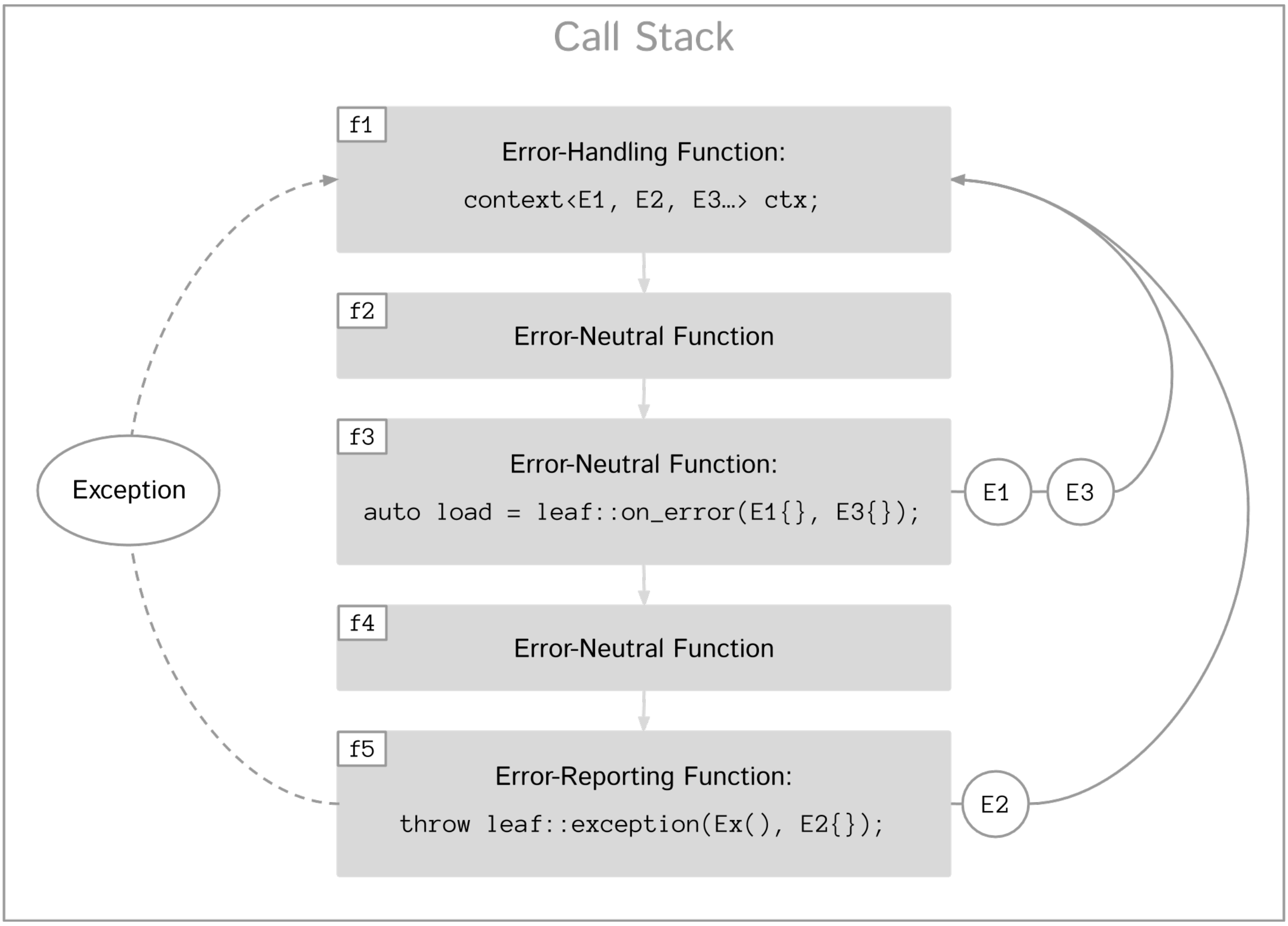

Figure 2. LEAF Error Communication Model Using Exception Handling

The main difference is that the call to `new_error` is implicit in the call to the function template `leaf::exception`, which in this case takes an exception object of type `Ex`, and returns an exception object of unspecified type that derives publicly from `Ex`.

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, a error ID may be encoded in a `std::error_code`. As long as a 3rd-party interface understands `std::error_code`, it should 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 lower-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 `augment_id` 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.

> To avoid ambiguities, whenever possible, use the `exception` function template when throwing exceptions to ensure that the exception object transports a unique `error_id`; better yet, use the `BOOST_LEAF_THROW_EXCEPTION` macro, which in addition will capture `__FILE__` and `__LINE__`.

Error Object Types

LEAF allows users to efficiently associate with a failure any number of relevant error values. These values may be of any no-throw movable type. This of course includes simple enums:

```
enum class my_error_code
{
  ok,
  failure_a,
  failure_b,
  failure_c
};
```

Error handlers recognize error objects associated with a failure by their static type. For example:

```
leaf::result<void> f() noexcept
{
  ....
  if( err )
    return leaf::new_error(my_error_code::failure_a);
}

leaf::result<void> g() noexcept
{
  return leaf::try_handle_some(
    []() -> leaf::result<void>
    {
      BOOST_LEAF_CHECK(f());
    },
    [](my_error_code ec) ①
    {
      ....
    } );
}
```

result | new_error | try_handle_all | BOOST_LEAF_CHECK

① This handler is selected based on the static type of `ec`. If an error is reported that does not have a `my_error_code` associated with it, it will be returned to the caller (because this is the only provided error handler).

If `f` communicates failures by throwing, the above becomes:

```
void f()
{
  ....
  if( err )
    throw leaf::exception(my_error_code::failure_a);
}

void g()
{
  leaf::try_catch(
    [
    {
      f();
    },
    [](my_error_code ec) ①
    {
      ....
    } );
}
```

result | exception | try_catch | BOOST_LEAF_CHECK

① This handler is selected based on the static type of `ec`, after catching `std::exception` (which `try_catch` always does).

Because error handlers are selected based on the static type of their arguments, when we need to communicate objects of generic types (e.g. `int` or `std::string`), they should be enclosed in a C-`struct` that acts as their compile-time identifier and gives them semantic meaning. Examples:

```
struct e_input_name  { std::string value; };
struct e_output_name { std::string value; };

struct e_minimum_temperature { float value; };
struct e_maximum_temperature { float value; };
```

By convention, the enclosing C-`struct` names use the `e_` prefix, and define a data member called `value`.

Loading of Error Objects

To load an error object is to move it into an active `context`, usually local to a `try_handle_some`, a `try_handle_all` or a `try_catch` scope in the calling thread, where it becomes uniquely associated with a specific `error_id` — or discarded if storage is not 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`.

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 be passed functions:

- If we pass a function that takes no arguments, it is called, and the returned error object is loaded.
- If we pass a function that takes a single argument of 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`.

② `try_something` has failed — add `file_name` to the `e_relevant_file_names` object, associated with the `error_id` communicated in `r`.

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` operation failures may not have access to the file name, yet an error handling function needs it in order to print a useful error message.

Of course the file name is typically readily available in the call stack leading to the failed `FILE` operation. In the example 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 parses f, communicating errors using result<info>.

② Using on_error ensures that the file name is included with any error reported out of parse_file. All we need to do is hold on to the returned object load: when it expires, if an error is being reported, the passed e_file_name value will be automatically associated with it.

> on_error — like load — 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 an error object by mutable reference.

Consider for example if we want to use on_error to capture errno. We can't just pass e_errno to it, because at that time it 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 e_errno{errno}; });

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

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

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

Above, if a `throw` statement is reached, LEAF will invoke the function passed to `on_error` and associate the returned `e_errno` object with the exception.

The final type of arguments that can be passed to `on_error` is a function that takes a single mutable error object reference. In this case, `on_error` uses it similarly to how such functios are used by `load`; see Loading of Error Objects.

Binding Error Handlers in a `std::tuple`

Consider this snippet:

```
leaf::try_handle_all(

  [&]
  {
    // Operations which may fail
  },

  [](my_error_enum x)
  {
    ...
  },

  [](read_file_error_enum y, e_file_name const & fn)
  {
    ...
  },

  []
  {
    ...
  });
```

try_handle_all | e_file_name

Looks pretty simple, but what 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 `TryBlock` for `try_handle_all`:

```
leaf::try_handle_all(

  [&]
  {
    // Different operations which may fail
  },

  [](my_error_enum x)
  {
    ...
  },

  [](read_file_error_enum y, e_file_name const & fn)
  {
    ...
  },

  []
  {
    ...
  });
```

That works, but it is better to bind our error handlers in a `std::tuple`:

```
auto error_handlers = std::make_tuple(
  [](my_error_enum x)
  {
    ...
  },

  [](read_file_error_enum y, e_file_name const & fn)
  {
    ...
  },

  []
  {
    ...
  });
```

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

```
leaf::try_handle_all(
  [&]
  {
    // Operations which may fail ①
  },

  error_handlers );

leaf::try_handle_all(
  [&]
  {
    // Different operations which may fail ②
  },

  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 `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 Error Objects Between Threads

Error objects are stored on the stack in an instance of the `context` class template in the scope of e.g. `try_handle_some`, `try_handle_all` or `try_catch` functions. When using concurrency, we need a mechanism to collect error objects in one thread, then use them to handle errors in another thread.

LEAF offers two interfaces for this purpose, one using `result<T>`, and another designed for programs that use exception handling.

Using `result<T>`

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 it to capture the relevant error objects but not handle errors. To this end, in the main thread we bind our error handlers in a `std::tuple`, which we will later use to handle errors from each completed asynchronous task (see tutorial):

```
auto error_handlers = std::make_tuple(
  [](E1 e1, E2 e2)
  {
    //Deal with E1, E2
    ....
    return { };
  },

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

Why did we start with this step? Because we need to create a `context` object to collect the error objects we need. We could just instantiate the `context` template with `E1`, `E2` and `E3`, but that would be prone to errors, since it could get out of sync with the handlers we use. Thankfully LEAF can deduce the types we need automatically, we just need to show it our `error_handlers`:

```
std::shared_ptr<leaf::polymorphic_context> ctx = leaf::make_shared_context
(error_handlers);
```

The `polymorphic_context` type is an abstract base class that has the same members as any instance of the `context` class template, allowing us to erase its exact type. In this case what we're holding in `ctx` is a `context<E1, E2, E3>`, where `E1`, `E2` and `E3` were deduced automatically from the `error_handlers` tuple we passed to `make_shared_context`.

We're now ready to launch our asynchronous task:

```
std::future<leaf::result<task_result>> launch_task() noexcept
{
  return std::async(
    std::launch::async,
    [&]
    {
      std::shared_ptr<leaf::polymorphic_context> ctx = leaf::make_shared_context
(error_handlers);
      return leaf::capture(ctx, &task);
    } );
}
```

`result` | `make_shared_context` | `capture`

That's it! Later when we `get` the `std::future`, we can process the returned `result<task_result>` in a call to try_handle_some, using the `error_handlers` tuple we created earlier:

```
//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 { }
  },

  error_handlers );
```

try_handle_some | result | BOOST_LEAF_AUTO

The reason this works is that in case it communicates a failure, `leaf::result<T>` is able to hold a `shared_ptr<polymorphic_context>` object. That is why earlier instead of calling `task()` directly, we called `leaf::capture`: it calls the passed function, and in case it fails it stores the `shared_ptr<polymorphic_context>` we created in the returned `result<T>`, which now doesn't just communicate the fact that an error has occurred, but also holds the `context` object that `try_handle_some` needs in order to supply a suitable handler with arguments.

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

Using Exception Handling

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

```
task_result task();
```

Just like we saw in Using `result<T>`, first we will bind our error hondlers in a `std::tuple`:

```
auto handle_errors = std::make_tuple(
{
  [](E1 e1, E2 e2)
  {
    //Deal with E1, E2
    ....
    return { };
  },

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

Launching the task looks the same as before, except that we don't use `result<T>`:

```
std::future<task_result> launch_task()
{
  return std::async(
    std::launch::async,
    [&]
    {
      std::shared_ptr<leaf::polymorphic_context> ctx = leaf::make_shared_context(
&handle_error);
      return leaf::capture(ctx, &task);
    } );
}
```

make_shared_context | capture

That's it! Later when we `get` the `std::future`, we can process the returned `task_result` in a call to `try_catch`, using the `error_handlers` we saved earlier, as if it was generated locally:

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

return leaf::try_catch(

  [&]
  {
    task_result r = fut.get(); // Throws on error
    //Success!
  },

  error_handlers );
```

This works similarly to using `result<T>`, except that the `std::shared_ptr<polymorphic_context>` is transported in an exception object (of unspecified type which `try_catch` recognizes and then automatically unwraps the original exception).

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

Classification of Failures

It is common for any given 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 contains comments, so we know where to go for reference.

The disadvantage of such flat enums is that they do not support handling a whole class of failures. Consider this error handler from the introduction section:

```
....
[](leaf::match<error_code, size_error, read_error, eof_error>, leaf::e_errno const *
errn, leaf::e_file_name const & fn)
{
  std::cerr << "Failed to access " << fn.value;
  if( errn )
    std::cerr << ", errno=" << *errn;
  std::cerr << std::endl;
  return 3;
},
....
```

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.

If we can use exceptions, the situation is better 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_errno const * errn, leaf::e_file_name const & fn)
{
  std::cerr << "Failed to access " << fn.value;
  if( errn )
    std::cerr << ", errno=" << *errn;
  std::cerr << std::endl;
  return 3;
},
```

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. If we go back to the introduction section, instead of defining:

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

We could define:

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

With this in place, `file_read` from the print_file_result.cpp example can be rewritten like this:

```
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:

```
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) )
    throw leaf::exception(read_error{}, leaf::e_errno{errno});

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

on_error | exception | e_errno

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

And now we can write a future-proof handler that can handle any `input_error`:

```
....
[](input_error, leaf::e_errno const * errn, leaf::e_file_name const & fn)
{
  std::cerr << "Failed to access " << fn.value;
  if( errn )
    std::cerr << ", errno=" << *errn;
  std::cerr << std::endl;
  return 3;
},
....
```

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. Here is the complete example from the introduction section, rewritten to use this novel technique:

- print_file_result_error_tags.cpp (using `leaf::result<T>`).
- print_file_eh_error_tags.cpp (using exception handling).

Working with Disparate Error Types

Because most libraries define their own mechanism for reporting errors, programmers often need to use multiple incompatible error-reporting interfaces in the same program. If error objects must be communicated via function return values, this naturally leads to attempts to design a one-size-fits-all error type, e.g. `std::error_code`.

Yet `std::error_code` is not universally used. The net effect is that we now have one more error type that our programs needs to support. Did I say one more? Really it is two more, if we consider the existence of `boost::system::error_code`. It has almost identical interface, yet both types are sometimes used in the same program. The typical solution to this problem is to express one with the other; indeed, `boost::system::error_code` is capable of encoding a `std::error_code` without any loss of information.

LEAF provides an alternative option. Its design recognizes the reality that no matter what, there will be many different error object types that programs must be able to work with. It frees return values from the burden of transporting error objects, which means that all the different error types can be communicated verbatim, without any prone-to-bugs translation.

Using LEAF, functions are able to easily communicate any number of different error types. Here is a function which forwards either `std::error_code` or `boost::system::error_code` objects reported by lower level functions:

```
std::error_code f1() noexcept;
boost::system::error_code f2() noexcept;

leaf::result<void> g() noexcept
{
  if( auto ec = f1() )
    return leaf::new_error(ec);

  if( auto ec = f2() )
    return leaf::new_error(ec);

  return {};
}
```

result | new_error

A scope that is able to handle either `std::error_code` or `boost::system::error_code` communicated out of `g()` would look like this:

```
return try_handle_some(

  []() -> leaf::result<void> ①
  {
    BOOST_LEAF_CHECK(g()); ②
  },

  [](std::error_code const & e)
  {
    ....; ③
  },

  [](boost::system::error_code const & e)
  {
    ....; ④
  } );
```

try_handle_some | result

① Errors are communicated via `result<void>`.

② Call `g()`, communicate errors back to `try_handle_some`.

③ Handle `std::error_code` errors.

④ Handle `boost::system::error_code` errors.

Converting Exceptions to `result<T>`

It is sometimes necessary to catch exceptions thrown by a lower-level library function, and report the error through different means, to a higher-level library which may not use exception handling.

Suppose we have an exception type hierarchy and a function `compute_answer_throws`:

```
class error_base: public std::exception { };
class error_a: public error_base { };
class error_b: public error_base { };
class error_c: public error_base { };

int compute_answer_throws()
{
  switch( rand()%4 )
  {
    default: return 42;
    case 1: throw error_a();
    case 2: throw error_b();
    case 3: throw error_c();
  }
}
```

We can write a simple wrapper using `exception_to_result`, which calls `compute_answer_throws` and switches to `result<int>` for error handling:

```
leaf::result<int> compute_answer() noexcept
{
  return leaf::exception_to_result<error_a, error_b>(
    []
    {
      return compute_answer_throws();
    } );
}
```

[result](#) | [exception_to_result](#)

(As a demonstration, `compute_answer` specifically converts exceptions of type `error_a` or `error_b`, while it leaves `error_c` to be captured by `std::exception_ptr`).

Here is a simple function which prints successfully computed answers, forwarding any error (originally reported by throwing an exception) to its caller:

```
leaf::result<void> print_answer() noexcept
{
  BOOST_LEAF_AUTO(answer, compute_answer());
  std::cout << "Answer: " << answer << std::endl;
  return { };
}
```

[result](#) | [BOOST_LEAF_AUTO](#)

Finally, here is a scope that handles the errors (which used to be exception objects):

```
leaf::try_handle_all(

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

  [](error_a const & e)
  {
    std::cerr << "Error A!" << std::endl;
  },

  [](error_b const & e)
  {
    std::cerr << "Error B!" << std::endl;
  },

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

`try_handle_all` | `result` | `BOOST_LEAF_CHECK`

The complete program illustrating this technique is available here.

Using `augment_id` to Report Arbitrary Errors from C-callbacks

Communicating information pertaining to a failure detected in a C callback is tricky, because C callbacks are limited to a specific static signature, which may not use C++ types.

LEAF makes this easy. As an example, we'll write a program that uses Lua and reports a failure from a C++ function registered as a C callback, called from a Lua program. The failure will be propagated from C++, through the Lua interpreter (written in C), back to the C++ function which called it.

C/C++ functions designed to be invoked from a Lua program must use the following signature:

```
int do_work( lua_State * L ) ;
```

Arguments are passed on the Lua stack (which is accessible through `L`). Results too are pushed onto the Lua stack.

First, let's initialize the Lua interpreter and register `do_work` as a C callback, available for Lua programs to call:

```
std::shared_ptr<lua_State> init_lua_state() noexcept
{
  std::shared_ptr<lua_State> L(lua_open(), &lua_close); ①

  lua_register(&*L, "do_work", &do_work); ②

  luaL_dostring(&*L, "\ ③
\n      function call_do_work()\
\n          return do_work()\
\n      end");

  return L;
}
```

① Create a new `lua_State`. We'll use `std::shared_ptr` for automatic cleanup.

② Register the `do_work` C++ function as a C callback, under the global name `do_work`. With this, calls from Lua programs to `do_work` will land in the `do_work` C++ function.

③ Pass some Lua code as a `C` string literal to Lua. This creates a global Lua function called `call_do_work`, which we will later ask Lua to execute.

Next, let's define our `enum` used to communicate `do_work` failures:

```
enum do_work_error_code
{
  ec1=1,
  ec2
};
```

We're now ready to define the `do_work` callback function:

```
int do_work( lua_State * L ) noexcept
{
  bool success = rand()%2; ①
  if( success )
  {
    lua_pushnumber(L, 42); ②
    return 1;
  }
  else
  {
    leaf::new_error(ec1); ③
    return luaL_error(L, "do_work_error"); ④
  }
}
```

① "Sometimes" `do_work` fails.

② In case of success, push the result on the Lua stack, return back to Lua.

③ Generate a new `error_id` and associate a `do_work_error_code` with it. Normally, we'd return this in a `leaf::result<T>`, but the `do_work` function signature (required by Lua) does not permit this.

④ Tell the Lua interpreter to abort the Lua program.

Now we'll write the function that calls the Lua interpreter to execute the Lua function `call_do_work`, which in turn calls `do_work`. We'll return `result<int>`, so that our caller can get the answer in case of success, or an error:

```
leaf::result<int> call_lua( lua_State * L )
{
  lua_getfield(L, LUA_GLOBALSINDEX, "call_do_work");

  augment_id augment;
  if( int err=lua_pcall(L, 0, 1, 0) ) ①
  {
    auto load = leaf::on_error(e_lua_error_message{lua_tostring(L,1)}); ②
    lua_pop(L,1);

    return augment.get_error(e_lua_pcall_error{err}); ③
  }
  else
  {
    int answer = lua_tonumber(L, -1); ④
    lua_pop(L, 1);
    return answer;
  }
}
```

result | on_error | augment_id

① Ask the Lua interpreter to call the global Lua function `call_do_work`.

② `on_error` works as usual.

③ `get_error` will return the `error_id` generated in our Lua callback. This is the same `error_id` the `on_error` uses as well.

④ Success! Just return the `int` answer.

Finally, here is the `main` function which exercises `call_lua`, each time handling any failure:

```
int main() noexcept
{
  std::shared_ptr<lua_State> L=init_lua_state();

  for( int i=0; i!=10; ++i )
  {
    leaf::try_handle_all(

      [&]() -> leaf::result<void>
      {
        BOOST_LEAF_AUTO(answer, call_lua(&*L));
        std::cout << "do_work succeeded, answer=" << answer << '\n'; ①
        return { };
      },

      [](do_work_error_code e) ②
      {
        std::cout << "Got do_work_error_code = " << e <<  "!\n";
      },

      [](e_lua_pcall_error const & err, e_lua_error_message const & msg) ③
      {
        std::cout << "Got e_lua_pcall_error, Lua error code = " << err.value << ", "
<< msg.value << "\n";
      },

      [](leaf::error_info const & unmatched)
      {
        std::cerr <<
          "Unknown failure detected" << std::endl <<
          "Cryptic diagnostic information follows" << std::endl <<
          unmatched;
      } );
  }
```

try_handle_all | result | BOOST_LEAF_AUTO | error_info

① If the call to `call_lua` succeeded, just print the answer.

② Handle `do_work` failures.

③ Handle all other `lua_pcall` failures.

> Follow this link to see the complete program: lua_callback_result.cpp.
>
> Remarkably, the Lua interpreter is C++ exception-safe, even though it is written in C. Here is the same program, this time using a C++ exception to report failures from `do_work`: lua_callback_eh.cpp.

Diagnostic Information

LEAF is able to automatically generate diagnostic messages that include information about all error objects available to error handlers. For this purpose, it needs to be able to print objects of user-defined error types.

To do this, 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 doesn't 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 << "errno = " << 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).

> The automatically-generated diagnostic messages are developer-friendly, but not user-friendly. Therefore, `operator<<` overloads for error types should only print technical information in English, and should not attempt to localize strings or to format a user-friendly message; this should be done in error-handling functions specifically designed for that purpose.

Examples

See github.

Synopsis

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

LEAF headers are organized as to minimize coupling:

- Headers needed to report 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.

There is also a reference section split in four parts, the contents of each part organized alphabetically:

Reference: Functions | Reference: Types | Reference: Macros | Reference: Traits

---

Error Reporting

LEAF supports error-reporting via a `result<T>` type or by throwing exceptions. Functions that throw exceptions or use exception handling are defined in separate headers, so that client code that does not use exceptions is not coupled with them.

`error.hpp`

The header `<boost/leaf/error.hpp>` contains definitions that are sufficient for a translation unit to report errors, if it does not throw exceptions.

#include <boost/leaf/error.hpp>

```
namespace boost { namespace leaf {

  class error_id
  {
  public:

    error_id() 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 noexept;

    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;

    friend std::ostream & operator<<( std::ostream & os, error_id x );
  };

  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;

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

  class polymorphic_context
  {
  protected:

    polymorphic_context() noexcept = default;
    ~polymorphic_context() noexcept = default;

  public:

    virtual void activate() noexcept = 0;
    virtual void deactivate() noexcept = 0;
    virtual bool is_active() const noexcept = 0;

    virtual void propagate() noexcept = 0;
```

```
    virtual void print( std::ostream & ) const = 0;
  };

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

  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;

#define BOOST_LEAF_NEW_ERROR <<unspecified>>
#define BOOST_LEAF_AUTO <<unspecified>>
#define BOOST_LEAF_CHECK <<unspecified>>
```

error_id | is_error_id | new_error | current_error | polymorphic_context | context_activator | activate_context | BOOST_LEAF_NEW_ERROR | BOOST_LEAF_AUTO | BOOST_LEAF_CHECK

common.hpp

This header contains definitions of commonly-used error types.

#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;
    friend std::ostream & operator<<( std::ostream &, e_errno const & );
  };

  namespace windows
  {
    struct e_LastError
    {
      unsigned value;
      friend std::ostream & operator<<( std::ostream &, e_LastError const & );
    };
  }

} }
```

`e_api_function` | `e_file_name` | `e_at_line` | `e_type_info_name` | `e_source_location` | `e_errno` | `e_LastError`

`result.hpp`

This header defines a lightweight `result<T>` template. Note that LEAF error-handling functions can work with any external type for which the `is_result_type` template is specialized, that has value-or-error variant semantics similar to `leaf::result<T>`.

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  class result
  {
  public:

    result() noexcept;
    result( T && v ) noexcept;
    result( T const & v );

    result( error_id err ) noexcept;
    result( std::error_code const & ec ) noexcept;
    result( std::shared_ptr<polymorphic_context> && ctx ) noexcept;

    result( result && r ) noexcept;

    template <class U>
    result( result<U> && r ) noexcept;

    result & operator=( result && r ) noexcept;

    template <class U>
    result & operator=( result<U> && r ) noexcept;

    explicit operator bool() const noexcept;

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

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

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

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

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

  struct bad_result: std::exception { };

} }
```

result

`on_error.hpp`

This header defines the `on_error` function, which is used for automatic inclusion of error objects with any error exiting the scope in which it is invoked. See Using `on_error` and Classification of Failures.

The `augment_id` type is used internally by `on_error` to determine the correct `error_id` to associate error objects with.

#include <boost/leaf/on_error.hpp>

```
namespace boost { namespace leaf {

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

  class augment_id
  {
  public:

    augment_id() noexcept;

    error_id check_error() const noexcept;

    template <class... E>
    error_id get_error( E && ... e ) const noexcept;
  };

} }
```

`on_error` | `augment_id`

`exception.hpp`

This header provides support for throwing exceptions.

#include <boost/leaf/exception.hpp>

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

namespace boost { namespace leaf {

  template <class Ex, class... E> ①
  <<unspecified>> exception( Ex &&, E && ... ) noexcept;

  template <class E1, class... E> ②
  <<unspecified>> exception( E1 &&, E && ... ) noexcept;

  <<unspecified>> exception() noexcept;

} }

#define BOOST_LEAF_EXCEPTION(...) ....

#define BOOST_LEAF_THROW_EXCEPTION(...) ....
```

exception | BOOST_LEAF_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.

capture.hpp

This header is used when transporting error objects between threads, or to convert exceptions to result<T>.

#include <boost/leaf/capture_exception.hpp>

```
namespace boost { namespace leaf {

  template <class F, class... A>
  decltype(std::declval<F>()(std::forward<A>(std::declval<A>())...))
  capture(std::shared_ptr<polymorphic_context> && ctx, F && f, A... a);

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

} }
```

capture | exception_to_result

Error Handling

Headers providing error-handling functionality are designed to minimize coupling:

- Translation units that work with context objects but do not handle errors should #include <boost/leaf/context.hpp>;
- Translation units that handle errors but **do not** catch exceptions should #include <boost/leaf/handle_error.hpp>;
- Translation units that **do** catch exceptions should #include <boost/leaf/handle_exception.hpp>.

Namespace-scope error-handling functions use the try_ prefix in their name:

- try_catch always catches and handles exceptions, but it can not work with a result<T> type.
- try_handle_some and try_handle_all work with a result<T> type (see is_result_type). They also handle exceptions iff at least one of the user-supplied handlers takes an argument of type that derives from std::exception, or an instance of the catch_ template.

These error-handling functions:

1. Create an internal context<E…> object ctx, deducing the E… types automatically from the arguments of the supplied handlers;
2. Attempt the set of operations contained in the passed TryBlock function;
3. If that fails, they invoke the first of the specified error handlers that LEAF is able to supply with arguments using the available (in ctx) error objects.

In addition, the context template provides a lower-level error handling member function, handle_error, which selects an error handler based on available error objects in *this, associated with a supplied error_id. This function is designed to be called after the caller has detected a failure; it does not use a result type and can not deal with exceptions. Use one of the try_ functions (above) for these cases.

context.hpp

This header defines the context template, which is used in error-handling scopes to provide storage for the error objects needed by user-defined error-handling functions.

#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 propagate () noexcept;

    void print( std::ostream & os ) const;

    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 inline context_type_from_handlers<H...> make_context()
noexcept;

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

  template <class...  H>
  inline context_ptr make_shared_context() noexcept;

  template <class...  H>
  inline context_ptr make_shared_context( H && ... ) noexcept;

} }
```

context | context_type_from_handlers | make_context | make_shared_context

handle_error.hpp

This header defines functions and types that can be used to handle errors but not catch exceptions.

#include <boost/leaf/handle_error.hpp>

```
#include <boost/leaf/context.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 Enum>
  class match;

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

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

  class error_info
  {
    //Constructors unspecified

  public:

    error_id error() const noexcept;

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

    friend std::ostream & operator<<( std::ostream & os, error_info const & x );
  };

  class diagnostic_info: public error_info
  {
    //Constructors unspecified

    friend std::ostream & operator<<( std::ostream & os, diagnostic_info const & x
);
  };

  class verbose_diagnostic_info: public error_info
  {
    //Constructors unspecified
```

```
    friend std::ostream & operator<<( std::ostream & os, diagnostic_info const & x
);
  };

} }
```

context.hpp | try_handle_all | try_handle_some | match | condition | error_info | diagnostic_info | verbose_diagnostic_info

handle_exception.hpp

This header:

- Defines namespace-scope functions and types that can be used to catch exceptions.
- Enables all functions using the `_some` or `_all` suffix (defined in handle_error.hpp) to handle exceptions, not only failures communicated by `result<T>`.

#include <boost/leaf/handle_exception.hpp>

```
#include <boost/leaf/handle_error.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 );

  template <class... Ex>
  struct catch_;

} }
```

handle_error.hpp | try_catch | catch_

Reference: Traits

`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 `std::shared_ptr<polymorphic_context>`.

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

capture

#include <boost/leaf/capture_result.hpp>

```
namespace boost { namespace leaf {

  template <class F, class... A>
  decltype(std::declval<F>()(std::forward<A>(std::declval<A>())...))
  capture(std::shared_ptr<polymorphic_context> && ctx, F && f, A... a);

} }
```

polymorphic_context

This function can be used to capture error objects stored in a `context` in one thread and transport them to a different thread for handling, either in a `result<T>` object or in an exception.

Returns:

The same type returned by `F`.

Effects:

Uses an internal `context_activator` to `activate` `*ctx`, then invokes `std::forward<F>(f)(std::forward<A>(a)…)`. Then:

- If the returned value `r` is not a `result<T>` type (see `is_result_type`), it is forwarded to the caller.

- Otherwise:
  - If `!r`, the return value of `capture` is initialized with `ctx`;

    An object of type `leaf::result<T>` can be initialized with a `std::shared_ptr<leaf::polymorphic_context>`.

  - otherwise, it is initialized with `r`.

In case `f` throws, `capture` catches the exception in a `std::exception_ptr`, and throws a different exception of unspecified type that transports both the `std::exception_ptr` as well as `ctx`. This exception type is recognized by `try_catch`, which automatically unpacks the original exception and propagates the contents of `*ctx` (presumably, in a different thread).

See also Transporting Error Objects Between Threads from the Tutorial.

`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 Usage:

```
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;
```

`error_info` | `diagnostic_info`

In the example above, `ctx` will be of type `context<e_this, e_that>`, deduced automatically from the handler list in `handle_error`. This guarantees that `ctx` provides storage for all error types that are required by `handle_error` in order to handle errors.

Alternatively, a suitable context may be created by calling make_context, or allocated dynamically by calling make_shared_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

#include <boost/leaf/exception.hpp>

```
namespace boost { namespace leaf {

  template <class Ex, class... E> ①
  <<unspecified>> exception( Ex && ex, E && ... e ) noexcept;

  template <class E1, class... E> ②
  <<unspecified>> exception( E1 && e1, E && ... e ) noexcept;

  <<unspecified>> exception() noexcept;

} }
```

The exception function is overloaded: it can be invoked with no arguments, or else there are two alternatives, selected using std::enable_if based on the type of the first argument:

① Selected if the first argument is an exception object, that is, iff Ex derives publicly from std::exception. In this case the return value 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 otherwise. In this case the return value 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)…).

To automatically capture __FILE__, __LINE__ and __FUNCTION__ with the returned object, use BOOST_LEAF_EXCEPTION instead of leaf::exception.

---

exception_to_result

#include <boost/leaf/capture.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>(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 { };
  }

} }
```

`context_type_from_handlers`

`make_shared_context`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class...  H>
  context_ptr make_shared_context() noexcept
  {
    return std::make_shared<leaf_detail::polymorphic_context_impl
<context_type_from_handlers<H...>>>();
  }

  template <class...  H>
  context_ptr make_shared_context( H && ... ) noexcept
  {
    return std::make_shared<leaf_detail::polymorphic_context_impl
<context_type_from_handlers<H...>>>();
  }

} }
```

`context_type_from_handlers`

See also Transporting Error Objects Between Threads from the tutorial.

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:

How each `item` is handled depends on its type:

- If it is a function that takes no arguments, the function is invoked, and the returned object is loaded into an active `context`.
- If it is a function that takes a single argument of some 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.
- Otherwise, the `item` itself is loaded into an active `context`.

`new_error` discards error objects which are not used in any 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.

Returns:

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

Ensures:

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

`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 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_exception` returns `true`, `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.

`try_catch`

#include <boost/leaf/handle_exception.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 also Five Minute Introduction Using Exception Handling.

`try_handle_all`

#include <boost/leaf/handle_error.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` return 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 also Five Minute Introduction Using `result<T>`.

`try_handle_some`

#include <boost/leaf/handle_error.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:
    - may take any error objects, by value, by (`const`) reference, or as pointer (to `const`);
    - may take arguments, by value, of the predicate type `match<E, V…>`, where `E` is enumerator or an instance of the `condition` class template.
    - may take arguments, by value, of the predicate type `catch_<Ex…>`, where each of the `Ex` types derives from `std::exception` (in this case, please also `#include <boost/leaf/handle_exception.hpp>`);
    - may take an `error_info` argument by `const &`;
    - may take a `diagnostic_info` argument by `const &`;
    - may take a `verbose_diagnostic_info` argument by `const &`;
    - may not take any other types of arguments.
    - 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 `{}`.

Effects:

- Creates a local `context<E…>` object `ctx`, where the `E…` types are automatically deduced from the types of arguments taken by each `h…`, which guarantees that it is able to store all of the types required to handle errors.

- Invokes the `try_block`:
  - if the returned object `r` indicates success, it 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 user-supplied handlers 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 stored 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, `Aᵢ const &` or `Aᵢ &`:
  - If an error object of type A_i, associated with `err`, is currently stored in `ctx`, a_i is initialized with a reference to the stored object; otherwise
  - If A_i derives from `std::exception`, and the `try_block` throws a type that derives from `std::exception` which can be converted, using `dynamic_cast`, to A_i, a_i is initialized with a reference to the exception object;
  - Otherwise the handler is dropped.

> Handling of types that derive from `std::exception` requires `#include <boost/leaf/handle_exception.hpp>`.

Example:

```
....
auto r = leaf::try_handle_some(
  []
  {
    return f(); // returns leaf::result<int>
  },

  [](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` (the first lambda) 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, a_i is initialized with `0`.

Example:

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

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

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 the predicate type `match<E,V…>`, if an object of type `E`, associated with `err`, is currently stored in `ctx`, a_i is initialized with a reference to the stored object; otherwise the handler is dropped. The handler is also dropped if the expression a_i`()` evaluates to `false` (see match<E,V…>).

Example:

```
enum class errors
{
  ec1=1,
  ec2,
  ec3
};

....

try_handle_some(
  []
  {
    return f();
  },

  [](leaf::match<errors, errors::ec1>) ①
  {
    ....
  },

  [](errors ec) ②
  {
    ....
  } );
}
```

result | match

① This handler is selected if the error includes an object of type `errors` with value `ec1`.

② This handler is selected if the error includes an object of type `errors` regardless of its value.

In particular, the `E` type used to instantiate the `match` template may be an instance of the `condition` class template, which is used to match a `std::error_condition` enumerated value:

Example:

```
enum class cond_x { x00, x11, x22, x33 };

namespace std
{
  template <> struct is_error_condition_enum<cond_x>: true_type { };
};

....

try_handle_some(
  []
  {
    return f();
  },

  [&c](leaf::match<leaf::condition<cond_x>, cond_x::x11>) ①
  {
    ....
  },

  [](std::error_code const & ec) ②
  {
    ....
  } );
}
```

result | match | condition

① This handler is selected if the error includes an object of type `std::error_code` equivalent to the error condition `cond_x::x11`.

② This handler is selected if the error includes any object of type `std::error_code`.

- If `Aᵢ` is of the predicate type catch_<Ex…>, and the `try_block` throws, `aᵢ` is initialized with the current `std::exception`. The handler is dropped if the expression `aᵢ()` evaluates to `false` (see catch_<Ex…>).

Example:

```
struct exception1: std::exception { };
struct exception2: std::exception { };
struct exception3: std::exception { };

....

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

  [](leaf::catch_<exception1, exception2>) ①
  {
    ....
  },

  [](leaf::error_info const & info) ②
  {
    ....
  } );
```

result | catch_ | error_info

① This handler is selected if the current exception is either of type exception1 or exception2.

② This handler matches any error. Use info.exception_caught() to check if try_handle_some has caught an exception, in which case you can call info.exception() to access it as std::exception.

Using catch_ with try_handle_some requires #include <boost/leaf/handle_exception.hpp>.

If you want to catch_ a single exception type Ex that derives from std::exception, using catch_ is not required — simply take an argument of type Ex const & or Ex &. The use of catch_ is required only if you want to match any one of a list of exception types, or if Ex does not derive from std::exception.

- 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:" << std::endl << 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:" << std::endl << 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 `verbose_diagnostic_info const &`, `try_handle_some` is always able to produce it.

Example:

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

  [](leaf::verbose_diagnostic_info const & info) ①
  {
    std::cerr << "leaf::verbose_diagnostic_information:" << std::endl << info;
②
    return info.error(); ③
  } );
```

[`result`](#) | [`verbose_diagnostic_info`](#)

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.

augment_id

```
namespace boost { namespace leaf {

  class augment_id
  {
  public:

    augment_id() noexcept;

    error_id check_error() const noexcept;

    template <class... E>
    error_id get_error( 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::error<int> augmenter() noexcept
{
  leaf::augment_id augment; ②

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

  if( failure(ec) )
    return augment.get_error(e1, e2, ...); ③
  else
    return val; ④
}
```

① Uncooperative third-party API that does not use LEAF, but results 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 `augment_id` object.

③ The call to `compute_value` has failed:

- If `new_error` was invoked (by the calling thread) after the `augment` object was initialized, `get_error` 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, `get_error` invokes `new_error` and returns that `error_id`.

④ The call was successful, return the computed value.

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

See Using `augment_id` to Report Arbitrary Errors from C-callbacks.

catch_

```
namespace boost { namespace leaf {

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

    explicit catch_( std::exception const & ex ) noexcept;

    bool operator()() const noexcept;
  };

} }
```

The `catch_` template is useful only as an argument to a handler function passed to a LEAF error-handling function, such as `try_handle_all`, `try_handle_some` or `try_catch`.

Effects:

The `catch_` constructor initializes the `value` reference with `ex`.

The `catch_` template is a predicate function type: `operator()` returns `true` iff for at least one of Ex_i in `Ex...`, the expression `dynamic_cast<Exᵢ const *>(&value) != 0` is `true`.

Example:

```
struct exception1: std::exception { };
struct exception2: std::exception { };
struct exception3: std::exception { };

exception2 x;

catch_<exception1> c1(x);
assert(!c1());

catch_<exception2> c2(x);
assert(c2());

catch_<exception1,exception2> c3(x);
assert(c3());

catch_<exception1,exception3> c4(x);
assert(!c4());
```

`condition`

```
namespace boost { namespace leaf {

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

} }
```

> The `condition` template is useful only as argument to the `match` template, to match a specific `std::error_condition`.

Example:

```
enum class cond_x { x00, x11, x22, x33 };

namespace std
{
  template <> struct is_error_condition_enum<cond_x>: true_type { };
};

std::error_code ec;
match<condition<cond_x, cond_x::x11>> m(ec);

// m() evaluates to true if ec is equivalent to the error condition cond_x::x11.
```

See also match.

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 propagate() noexcept;

    void print( std::ostream & os ) const;

    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 | propagate | print | 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 `propagate`.

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_error.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.

`print`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  void context<E...>::print( std::ostream & os ) const;

} }
```

Effects:

Prints all error objects currently stored in `*this`, together with the unique error ID each individual error object is associated with.

---

propagate

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  template <class... E>
  void context<E...>::propagate() 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.

---

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`() and, if there are new uncaught exceptions since the constructor was called, the destructor calls `ctx.propagate`().

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

diagnostic_info

```
namespace boost { namespace leaf {

  class diagnostic_info: public error_info
  {
    //Constructors unspecified

    friend std::ostream & operator<<( std::ostream & os, diagnostic_info const & x );
  };

} }
```

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 behavior of `diagnostic_info` (and verbose_diagnostic_info) is affected by the value of the macro `BOOST_LEAF_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;

    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;

    friend std::ostream & operator<<( std::ostream & os, error_id x );
  };

  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;

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

} }
```

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

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` has 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` (or `match`<`condition`>).

`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 `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, than function is called to obtain an error object, which is associated with `*this`.
  - Otherwise, the `item` itself is assumed to be an error object, which is associated with `*this`.

Returns:

`*this`.

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` ocurred 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.

e_api_function

```
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

```
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

```
namespace boost { namespace leaf {

  struct e_errno
  {
    int value;
    friend std::ostream & operator<<( std::ostream & os, e_errno const & err );
  };

} }
```

To capture errno, use e_errno. When printed in automatically-generated diagnostic messages, e_errno objects use strerror to convert the errno code to string.

e_file_name

```
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

```
namespace boost { namespace leaf {

  namespace windows
  {
    struct e_LastError
    {
      unsigned value;
      friend std::ostream & operator<<( std::ostream & os, e_LastError const & err );
    };
  }

} }
```

`e_LastError` is designed to communicate `GetLastError()` values on Windows.

`e_source_location`

```
namespace boost { namespace leaf {

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

    friend std::ostream & operator<<( std::ostream & os, e_source_location const & x
);
  };

} }
```

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

`e_type_info_name`

```
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

```
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;

    friend std::ostream & operator<<( std::ostream & os, error_info const & x );
  };

} }
```

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()`.

match

```
namespace boost { namespace leaf {

  template <class E, typename deduced-type<E>::type... V>
  class match
  {
  public:

    using error_type = <<deduced>>;
    using value_type = <<deduced>>;

    explicit match( error_type const * value ) noexcept;

    explicit bool operator()() const noexcept;

    value_type value() const noexcept;
  };

} }
```

Effects:

- If `E` is an instance of the `condition` template:
  - `match<E>::error_type` is deduced as `std::error_code`;
  - `match<E>::value_type` is deduced as the type of the error condition enum used with `condition`;
  - The type of the parameter pack `V...` is deduced as `value_type`;
  - The boolean conversion operator evaluates to `true` iff the `std::error_code` pointer passed to the constructor is not `0` and the object it points is equivalent to one of the error condition enum values used with `condition`.
- Otherwise, if `E` defines an accessible data member `value`:
  - `match<E>::error_type` is deduced as `E`;
  - `match<E>::value_type` is deduced as `decltype(std::declval<E>().value)`;
  - The type of the parameter pack `V...` is deduced as `value_type`;
  - The boolean conversion operator evaluates to `true` iff the `value` pointer passed to the constructor is not `0` and `*value` is equal to one of `V...`.
- Otherwise, if `E` defines an accessible member function `value()`:
  - `match<E>::error_type` is deduced as `E`;
  - `match<E>::value_type` is deduced as `decltype(std::declval<E>().value())`;
  - The type of the parameter pack `V...` is deduced as `value_type`;
  - The boolean conversion operator evaluates to `true` iff the `value` pointer passed to the constructor is not `0` and `*value` is equal to one of `V...`.
- Otherwise:

- `match<E>::error_type` is deduced as `E`;
- `match<E>::value_type` is deduced as `E`;
- The type of the parameter pack `V…` is deduced as `value_type`;
- The boolean conversion operator evaluates to `true` iff the `value` pointer passed to the constructor is not `0` and `*value` is equal to one of `V…`.

> The examples below demonstrate how `match` works in isolation, but it is designed to be used as argument to a handler function passed to an error-handling function such as `try_handle_some`, `try_handle_all`, `try_catch`. See Five Minute Introduction Using `result<T>` for a more practical example.

Example 1:

```
struct my_error_code { int value; };

my_error_code ec = {42};

match<my_error_code, 1> m1(&ec);
assert(!m1());

match<my_error_code, 42> m2(0);
assert(!m2());

match<my_error_code, 42> m2(&ec);
assert(m2());

match<my_error_code, 1, 5, 42, 7> m3(&ec);
assert(m3());

match<my_error_code, 1, 3, -42> m4(&ec);
assert(!m4());
```

Example 2:

```
enum my_error_code { e1=1, e2, e3 };

my_error_code ec = e2;

match<my_error_code, e1> m1(&ec);
assert(!m1());

match<my_error_code, e2> m2(0);
assert(m2());

match<my_error_code, e2> m2(&ec);
assert(m2());

match<my_error_code, e1, e2> m3(&ec);
assert(m3());

match<my_error_code, e1, e3> m4(&ec);
assert(!m4());
```

`polymorphic_context`

#include <boost/leaf/context.hpp>

```
namespace boost { namespace leaf {

  class polymorphic_context
  {
  protected:

    polymorphic_context() noexcept;
    ~polymorphic_context() noexcept;

  public:

    virtual void activate() noexcept = 0;
    virtual void deactivate() noexcept = 0;
    virtual bool is_active() const noexcept = 0;

    virtual void propagate() noexcept = 0;

    virtual void print( std::ostream & ) const = 0;
  };

} }
```

The `polymorphic_context` class is an abstract base type which can be used to erase the type of the exact instantiation of the `context` class template used. See `make_shared_context`.

`result`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  class result
  {
  public:

    result() noexcept;
    result( T && v ) noexcept;
    result( T const & v );

    result( error_id err ) noexcept;
    result( std::error_code const & ec ) noexcept;
    result( std::shared_ptr<polymorphic_context> && ctx ) noexcept;

    result( result && r ) noexcept;

    template <class U>
    result( result<U> && r ) noexcept;

    result & operator=( result && r ) noexcept;

    template <class U>
    result & operator=( result<U> && r ) noexcept;

    explicit operator bool() const noexcept;

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

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

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

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

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

  struct bad_result: std::exception { };

} }
```

Constructors | operator= | operator bool | value / operator* / operator-> | error | load

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`/`operator*`/`operator->` throws `leaf::bad_result`.
- Error-capture state, which is the same as the Error state, but in addition to the error ID, it holds a `std::shared_ptr<polymorphic_context>`.

`result<T>` objects are nothrow-moveable but are not copyable.

Constructors

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  result<T>::result() noexcept;

  template <class T>
  result<T>::result( T && v ) noexcept;

  template <class T>
  result<T>::result( leaf::error_id err ) noexcept;

  template <class T>
  result<T>::result( std::error_code const & ec ) noexcept;

  template <class T>
  result<T>::result( std::shared_ptr<polymorphic_context> && ctx ) noexcept;

  template <class T>
  result<T>::result( result && ) noexcept;

  template <class T>
  template <class U>
  result<T>::result( result<U> && ) noexcept;

} }
```

Requires:

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

Effects:

Establishes the `result<T>` invariant:

- 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 or with a `std::error_code`.
- To get a `result<T>` in Error-capture state, initialize it with a `std::shared_ptr<polymorphic_context>` (which can be obtained by calling e.g. `make_shared_context`).

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.

`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=`

#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 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`.

`value` / `operator*` / `operator->`

#include <boost/leaf/result.hpp>

```
namespace boost { namespace leaf {

  template <class T>
  T const & result<T>::value() const;

  template <class T>
  T & result<T>::value();

  template <class T>
  T const & result<T>::operator*() const;

  template <class T>
  T & result<T>::operator*();

  template <class T>
  T const * result<T>::operator->() const;

  template <class T>
  T * result<T>::operator->();

  struct bad_result: std::exception { };

} }
```

Effects:

If `*this` is in value state, returns a reference (or pointer) to the stored value, otherwise throws `bad_result`.

verbose_diagnostic_info

```
namespace boost { namespace leaf {

  class verbose_diagnostic_info: public error_info
  {
    //Constructors unspecified

    friend std::ostream & operator<<( std::ostream & os, verbose_diagnostic_info const
& x );
  };

} }
```

Handlers passed to error-handling functions such as `try_handle_some`, `try_handle_all` or `try_catch` may take an argument of type `verbose_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 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.

The behavior of `verbose_diagnostic_info` (and `diagnostic_info`) is affected by the value of the macro `BOOST_LEAF_DIAGNOSTICS`:

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

Using `verbose_diagnostic_info` will likely allocate memory dynamically.

Reference: Macros

The contents of each Reference section are organized alphabetically.

BOOST_LEAF_AUTO

#include <boost/leaf/result.hpp>

```
#define BOOST_LEAF_AUTO(v,r)\
  auto && _r_##v = r;\
  if( !_r_##v )\
    return _r_##v.error();\
  auto & v = _r_##v.value()
```

`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:

Compute two int values, return their sum as a float, using BOOST_LEAF_AUTO:

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

leaf::result<float> add_values()
{
  BOOST_LEAF_AUTO(v1, compute_value());
  BOOST_LEAF_AUTO(v2, compute_value());
  return v1 + v2;
}
```

`result`

Of course, we could write `add_value` without using `BOOST_LEAF_AUTO`. This is equivalent:

Compute two int values, return their sum as a float, without BOOST_LEAF_AUTO:

```
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();
}
```

BOOST_LEAF_CHECK

#include <boost/leaf/result.hpp>

```
#define BOOST_LEAF_CHECK(r)\
  {\
    auto && _r = r;\
    if(!_r)\
      return _r.error();\
  }
```

`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:

Try to send a message, then compute a value, report errors using BOOST_LEAF_CHECK:

```
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!"));
  return compute_value();
}
```

`result`

Equivalent implementation without `BOOST_LEAF_CHECK`:

Try to send a message, then compute a value, report errors without BOOST_LEAF_CHECK:

```
leaf::result<float> add_values()
{
  auto r = send_message("Hello!");
  if( !r )
    return r.error();

  return compute_value();
}
```

BOOST_LEAF_NEW_ERROR

#include <boost/leaf/result.hpp>

```
#define BOOST_LEAF_NEW_ERROR <<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).

BOOST_LEAF_EXCEPTION

#include <boost/leaf/exception.hpp>

```
#define BOOST_LEAF_EXCEPTION <<unspecified>>
```

Effects:

`BOOST_LEAF_EXCEPTION(e…)` is equivalent to `leaf::exception(e…)`, except the current source location is automatically passed, in a `e_source_location` object (in addition to all `e…` objects).

BOOST_LEAF_THROW_EXCEPTION

#include <boost/leaf/exception.hpp>

```
#define BOOST_LEAF_THROW_EXCEPTION throw BOOST_LEAF_EXCEPTION
```

Effects:

Throws the exception object returned by `BOOST_LEAF_EXCEPTION`.

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 <u>all</u> 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 simply provides the space needed to store these types (in the form of a `std::tuple`, using automatic storage duration) until they are passed to a suitable handler.

At the time this space is reserved in the scope of an error-handling function, `thread_local` pointers of the required error types are set to point to the corresponding objects within it. Later on, *error-initiating* or *error-neutral* functions wanting to communicate an error object of a given type `E` use the corresponding `thread_local` pointer to detect if there is currently storage available for this type:

- If the pointer is not null, storage is available and the object is moved into the pointed storage, exactly once — regardless of how many levels of function calls must unwind before an *error-handling* function is reached.
- If the pointer is null, storage is not available and the error object is discarded, since no error-handling function makes any use of it in this program — saving resources.

This almost works, except we need to make sure that *error-handling* functions are protected from accessing stale error objects stored in response to previous failures, which would be a serious logic error. To this end, each occurrence of an error is assigned a unique `error_id`. Each of the `E…` objects stored in error-handling scopes is assigned an `error_id` as well, permanently associating it with a particular failure.

Thus, to handle a failure we simply match the available error objects (associated with its unique `error_id`) with the argument types required by each user-provided error-handling function. In terms of C++ exception handling, it is as if we could write something like:

```
try
{
  auto r = process_file();

  //Success, use r:
  ....
}

catch(file_read_error &, e_file_name const & fn, e_errno const & err)
{
  std::cerr <<
    "Could not read " << fn << ", errno=" << err << std::endl;
}

catch(file_read_error &, e_errno const & err)
{
  std::cerr <<
    "File read error, errno=" << err << std::endl;
}

catch(file_read_error &)
{
  std::cerr << "File read error!" << std::endl;
}
```

Of course this syntax is not valid, so LEAF uses lambda functions to express the same idea:

```
leaf::try_catch(

  []
  {
    auto r = process_file(); //Throws in case of failure, error objects stored inside
the try_catch scope

    //Success, use r:
    ....
  }

  [](file_read_error &, e_file_name const & fn, e_errno const & err)
  {
    std::cerr <<
      "Could not read " << fn << ", errno=" << err << std::endl;
  },

  [](file_read_error &, e_errno const & err)
  {
    std::cerr <<
      "File read error, errno=" << err << std::endl;
  },

  [](file_read_error &)
  {
    std::cerr << "File read error!" << std::endl;
  } );
```

try_catch | e_file_name | e_errno

Similar syntax works without exception handling as well. Below is the same snippet, written using result<T>:

```
return leaf::try_handle_some(

  []() -> leaf::result<void>
  {
    BOOST_LEAF_AUTO(r, process_file()); //In case of errors, error objects are stored
inside the try_handle_some scope

    //Success, use r:
    ....

    return { };
  }

  [](leaf::match<error_enum, file_read_error>, e_file_name const & fn, e_errno const &
err)
  {
    std::cerr <<
      "Could not read " << fn << ", errno=" << err << std::endl;
  },

  [](leaf::match<error_enum, file_read_error>, e_errno const & err)
  {
    std::cerr <<
      "File read error, errno=" << err << std::endl;
  },

  [](leaf::match<error_enum, file_read_error>)
  {
    std::cerr << "File read error!" << std::endl;
  } );
```

`result` | `try_handle_some` | `match` | `e_file_name` | `e_errno`

> Please post questions and feedback on the Boost Developers Mailing List (LEAF is not part of Boost).

Critique 1: Error Types Do Not Participate in Function Signatures

A knee-jerk critique of the LEAF design is that it does not statically enforce that each possible error condition is recognized and handled by the program. One idea I've heard from multiple sources is to add `E…` parameter pack to `result<T>`, essentially turning it into `expected<T,E…>`, so we could write something along these lines:

```
expected<T, E1, E2, E3> f() noexcept; ①

expected<T, E1, E3> g() noexcept ②
{
  if( expected<T, E1, E2, E3> r = f() )
  {
    return r; //Success, return the T
  }
  else
  {
    return r.handle_error<E2>( [] ( .... ) ③
      {
        ....
      } );
  }
}
```

① `f` may only return error objects of type `E1`, `E2`, `E3`.

② `g` narrows that to only `E1` and `E3`.

③ Because `g` may only return error objects of type `E1` and `E3`, it uses `handle_error` to deal with `E2`. In case `r` contains `E1` or `E3`, `handle_error` simply returns `r`, narrowing the error type parameter pack from `E1, E2, E3` down to `E1, E3`. If `r` contains an `E2`, `handle_error` calls the supplied lambda, which is required to return one of `E1`, `E3` (or a valid `T`).

The motivation here is to help avoid bugs in functions that handle errors that pop out of `g`: as long as the programmer deals with `E1` and `E3`, he can rest assured that no error is left unhandled.

Congratulations, we've just discovered exception specifications. The difference is that exception specifications, before being removed from C++, were enforced dynamically, while this idea is equivalent to statically-enforced exception specifications, like they are in Java.

Why not use the equivalent of exception specifications, even if they are enforced statically?

> The short answer is that nobody knows how to fix exception specifications in any language, because the dynamic enforcement C++ chose has only different (not greater or fewer) problems than the static enforcement Java chose. … When you go down the Java path, people love exception specifications until they find themselves all too often encouraged, or even forced, to add `throws Exception`, which immediately renders the exception specification entirely meaningless. (Example: Imagine writing a Java generic that manipulates an arbitrary type `T`).[1]
>
> — Herb Sutter

Consider again the example above: assuming we don't want important error-related information to be lost, values of type `E1` and/or `E3` must be able to encode any `E2` value dynamically. But like Sutter points out, in generic contexts we don't know what errors may result in calling a user-supplied function. The only way around that is to specify a single type (e.g. `std::error_code`) that can communicate any and all errors, which ultimately defeats the idea of using static type checking to enforce correct error handling.

That said, in every program there are certain *error-handling* functions (e.g. `main`) which are required to handle any error, and it is highly desirable to be able to enforce this requirement at compile-time. In LEAF, the `try_handle_all` function implements this idea: if the user fails to supply at least one handler that will match any error, the result is a compile error. This guarantees that the scope invoking `try_handle_all` is prepared to recover from any failure.

Critique 2: LEAF Does Not Facilitate Mapping Between Different Error Types

Most C++ programs use multiple C and C++ libraries, and each library may provide its own system of error codes. But because it is difficult to define static interfaces that can communicate arbitrary error code types, a popular idea is to map each library-specific error code to a common program-wide enum.

For example, if we have —

```
namespace lib_a
{
  enum error
  {
    ok,
    ec1,
    ec2,
    ....
  };
}
```

```
namespace lib_b
{
  enum error
  {
    ok,
    ec1,
    ec2,
    ....
  };
}
```

— we could define:

```
namespace program
{
  enum error
  {
    ok,
    lib_a_ec1,
    lib_a_ec2,
    ....
    lib_b_ec1,
    lib_b_ec2,
    ....
  };
}
```

An error-handling library could provide conversion API that uses the C++ static type system to automate the mapping between the different error enums. For example, it may define a class template `result<T,E>` with value-or-error variant semantics, so that:

- `lib_a` errors are transported in `result<T,lib_a::error>`,
- `lib_b` errors are transported in `result<T,lib_b::error>`,
- then both are automatically mapped to `result<T,program::error>` once control reaches the appropriate scope.

There are several problems with this idea:

- It is prone to errors, both during the initial implementation as well as under maintenance.
- It does not compose well. For example, if both of `lib_a` and `lib_b` use `lib_c`, errors that originate in `lib_c` would be obfuscated by the different APIs exposed by each of `lib_a` and `lib_b`.
- It presumes that all errors in the program can be specified by exactly one error code, which is false.

To elaborate on the last point, consider a program that attempts to read a configuration file from three different locations: in case all of the attempts fail, it should communicate each of the failures. In theory `result<T,E>` handles this case well:

```
struct attempted_location
{
  std::string path;
  error ec;
};

struct config_error
{
  attempted_location current_dir, user_dir, app_dir;
};

result<config,config_error> read_config();
```

This looks nice, until we realize what the `config_error` type means for the automatic mapping API we wanted to define: an `enum` can not represent a `struct`. It is a fact that we can not assume that all error conditions can be fully specified by an `enum`; an error handling library must be able to transport arbitrary static types efficiently.

Critique 3: LEAF Does Not Treat Low Level Error Types as Implementation Details

This critique is a combination of Critique 1 and Critique 2, but it deserves special attention. Let's consider this example using LEAF:

```
leaf::result<std::string> read_line( reader & r );

leaf::result<parsed_line> parse_line( std::string const & line );

leaf::result<parsed_line> read_and_parse_line( reader & r )
{
  BOOST_LEAF_AUTO(line, read_line(r)); ①
  BOOST_LEAF_AUTO(parsed, parse_line(line)); ②
  return parsed;
}
```

`result` | `BOOST_LEAF_AUTO`

① Read a line, forward errors to the caller.

② Parse the line, forward errors to the caller.

The objection is that LEAF will forward verbatim the errors that are detected in `read_line` or `parse_line` to the caller of `read_and_parse_line`. The premise of this objection is that such low-level errors are implementation details and should be treated as such. Under this premise, `read_and_parse_line` should act as a translator of sorts, in both directions:

- When called, it should translate its own arguments to call `read_line` and `parse_line`;

- If an error is detected, it should translate the errors from the error types returned by `read_line` and `parse_line` to a higher-level type.

The motivation is to isolate the caller of `read_and_parse_line` from its implementation details `read_line` and `parse_line`.

There are two possible ways to implement this translation:

1) `read_and_parse_line` understands the semantics of **all possible failures** that may be reported by both `read_line` and `parse_line`, implementing a non-trivial mapping which both *erases* information that is considered not relevant to its caller, as well as encodes *different* semantics in the error it reports. In this case `read_and_parse_line` assumes full responsibility for describing precisely what went wrong, using its own type specifically designed for the job.

2) `read_and_parse_line` returns an error object that essentially indicates which of the two inner functions failed, and also transports the original error object without understanding its semantics and without any loss of information, wrapping it in a new error type.

The problem with **1)** is that typically the caller of `read_and_parse_line` is not going to handle the error, but it does need to forward it to its caller. In our attempt to protect the **one** error-handling function from "implementation details", we've coupled the interface of **all** intermediate error-neutral functions with the static types of errors they do not understand and do not handle.

Consider the case where `read_line` communicates `errno` in its errors. What is `read_and_parse_line` supposed to do with e.g. `EACCESS`? Turn it into `READ_AND_PARSE_LINE_EACCESS`? To what end, other than to obfuscate the original (already complex and platform-specific) semantics of `errno`?

And what if the call to `read` is polymorphic, which is also typical? What if it involves a user-supplied function object? What kinds of errors does it return and why should `read_and_parse_line` care?

Therefore, we're left with **2)**. There's almost nothing wrong with this option, since it passes any and all error-related information from lower level functions without any loss. However, using a wrapper type to grant (presumably dynamic) access to any lower-level error type it may be transporting is cumbersome and (like Niall Douglas explains) in general probably requires dynamic allocations. It is better to use independent error types that communicate the additional information not available in the original error object, while error handlers rely on LEAF to provide efficient access to any and all low-level error types, as needed.

[1] https://herbsutter.com/2007/01/24/questions-about-exception-specifications/

Alternatives to LEAF

- Boost Exception
- Boost Outcome
- `tl::expected`

Below we offer a comparison of LEAF to Boost Exception and to Boost Outcome.

Comparison to Boost Exception

While LEAF can be used without exception handling, in the use case when errors are communicated by throwing exceptions, it can be viewed as a better, more efficient alternative to Boost Exception. LEAF has the following advantages over Boost Exception:

- LEAF does not allocate memory dynamically;
- LEAF does not waste system resources communicating error objects not used by specific error handling functions;
- LEAF does not store the error objects in the exception object, and therefore it is able to augment exceptions thrown by external libraries (Boost Exception can only augment exceptions of types that derive from `boost::exception`).

The following tables outline the differences between the two libraries which should be considered when code that uses Boost Exception is refactored to use LEAF instead:

Table 1. Defining a custom type for transporting values of type T

Boost Exception	LEAF
`typedef error_info<struct my_info_,T> my_info;` `boost::error_info`	`struct my_info { T value; };`

Table 2. Passing arbitrary info at the point of the throw

Boost Exception	LEAF
`throw my_exception() << my_info(x) << my_info(y);` `operator<<`	`throw leaf::exception( my_exception(), my_info{x}, my_info{y} );` `exception`

Table 3. Augmenting exceptions in error-neutral contexts

Boost Exception	LEAF
`try { f(); } catch( boost::exception & e ) { e << my_info(x); throw; }`	`auto load = leaf::on_error( my_info{x} ); f();`
boost::exception | operator<<	on_error

Table 4. Obtaining arbitrary info at the point of the catch

Boost Exception

```
try
{
  f();
}
catch( my_exception & e )
{
  if( T * v = get_error_info<my_info>(e) )
  {
    //my_info is available in e.
  }
}
```

boost::get_error_info

LEAF

```
leaf::try_catch(
  []
  {
    f();
  }
  [](my_exception &, my_info const & x)
  {
    //my_info is available with
    //the caught exception.
  } );
```

try_catch

Table 5. Transporting of error objects

Boost Exception	LEAF
All supplied boost::error_info objects are allocated dynamically and stored in the boost::exception subobject of exception objects.	User-defined error objects are stored statically in the scope of try_catch, but only if their types are needed to handle errors; otherwise they are discarded.

Table 6. Transporting of error objects across thread boundaries

Boost Exception	LEAF
`boost::exception_ptr` automatically captures `boost::error_info` objects stored in a `boost::exception` and can transport them across thread boundaries.	Transporting error objects across thread boundaries requires the use of `capture`.

Table 7. Printing of error objects in automatically-generated diagnostic information messages

Boost Exception	LEAF
`boost::error_info` types may define conversion to `std::string` by providing `to_string` overloads **or** by overloading `operator<<` for `std::ostream`.	LEAF does not use `to_string`. Error types may define `operator<<` overloads for `std::ostream`.

The fact that Boost Exception stores all supplied `boost::error_info` objects — while LEAF discards them if they aren't needed — affects the completeness of the message we get when we print `leaf::diagnostic_info` objects, compared to the string returned by `boost::diagnostic_information`.

If the user requires a complete diagnostic message, the solution is to use `leaf::verbose_diagnostic_info`. In this case, before unused error objects are discarded by LEAF, they are converted to string and printed. Note that this allocates memory dynamically.

Comparison to Boost Outcome

Design Differences

Like LEAF, the Boost Outcome library is designed to work in low latency environments. It provides two class templates, `result<>` and `outcome<>`:

- `result<T,EC,NVP>` can be used as the return type in `noexcept` functions which may fail, where `T` specifies the type of the return value in case of success, while `EC` is an "error code" type. Semantically, `result<T,EC>` is similar to `std::variant<T,EC>`. Naturally, `EC` defaults to `std::error_code`.
- `outcome<T,EC,EP,NVP>` is similar to `result<>`, but in case of failure, in addition to the "error code" type `EC` it can hold a "pointer" object of type `EP`, which defaults to `std::exception_ptr`.

`NVP` is a policy type used to customize the behavior of `.value()` when the `result<>` or the `outcome<>` object contains an error.

The idea is to use `result<>` to communicate failures which can be fully specified by an "error code", and `outcome<>` to communicate failures that require additional information.

Another way to describe this design is that `result<>` is used when it suffices to return an error object of some static type `EC`, while `outcome<>` can also transport a polymorphic error object, using the pointer type `EP`.

> In the default configuration of `outcome<T>` the additional information — or the additional polymorphic object — is an exception object held by `std::exception_ptr`. This targets the use case when an exception thrown by a lower-level library function needs to be transported through some intermediate contexts that are not exception-safe, to a higher-level context able to handle it. LEAF directly supports this use as well, see `exception_to_result`.

Similar reasoning drives the design of LEAF as well. The difference is that while both libraries recognize the need to transport "something else" in addition to an "error code", LEAF provides an efficient solution to this problem, while Outcome shifts this burden to the user.

The `leaf::result<>` template deletes both `EC` and `EP`, which decouples it from the type of the error objects that are transported in case of a failure. This enables lower-level functions to freely communicate anything and everything they "know" about the failure: error code, even multiple error codes, file names, URLs, port numbers, etc. At the same time, the higher-level error-handling functions control which of this information is needed in a specific client program and which is not. This is ideal, because:

- Authors of lower-level library functions lack context to determine which of the information that is both relevant to the error *and* naturally available to them needs to be communicated in order for a particular client program to recover from that error;
- Authors of higher-level error-handling functions can easily and confidently make this determination, which they communicate naturally to LEAF, by simply writing the different error handlers. LEAF will transport the needed error objects while discarding the ones handlers don't care to use, saving resources.

> The LEAF examples include an adaptation of the program from the Boost Outcome `result<>` tutorial. You can view it on GitHub.

> Programs using LEAF for error-handling are not required to use `leaf::result<T>`; for example, it is possible to use `outcome::result<T>` with LEAF.

The Interoperability Problem

The Boost Outcome documentation discusses the important problem of bringing together multiple libraries — each using its own error reporting mechanism — and incorporating them in a robust error handling infrastructure in a client program.

Users are advised that whenever possible they should use a common error handling system throughout their entire codebase, but because this is not practical, both the `result<>` and the `outcome<>` templates can carry user-defined "payloads".

The following analysis is from the Boost Outcome documentation:

> 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.

The analysis above (emphasis added) is clear and precise, but LEAF and Boost Outcome tackle the interoperability problem differently:

- The Boost Outcome design asserts that the "cleanest" solution based on type-erasure is suboptimal ("almost certainly requires the use of `malloc`"), and instead provides a system for injecting custom converters into the `outcome::convert` namespace, used to translate between library-specific and program-wide error types, even though this approach "may lose the original failure detail".
- The LEAF design asserts that coupling the signatures of error-neutral functions with the static types of the error objects they need to forward to the caller does not scale, and instead transports error objects directly to error-handling scopes where they are stored statically, effectively implementing the third choice outlined above (without the use of `malloc`).

Further, consider that Outcome aims to hopefully become *the* one error-handling API all libraries would use, and in theory everyone would benefit from uniformity and standardization. But the reality is that this is wishful thinking. In fact, that reality is reflected in the design of `outcome::result<>`, in its lack of commitment to using `std::error_code` for its intended purpose: to be *the* standard type for transporting error codes. The fact is that `std::error_code` became *yet another* error code type programmers need to understand and support.

In contrast, the design of LEAF acknowledges that C++ programmers don't even agree on what a string is. If your project uses 10 different libraries, this probably means 15 different ways to report errors, sometimes across uncooperative interfaces (e.g. C APIs). LEAF helps you get the job done.

Benchmark

This benchmark compares the performance of LEAF, Boost Outcome and `tl::expected`.

Distribution

The source code is available on GitHub.

Portability

LEAF requires a C++11 compiler.

See unit test matrix at Travis-CI and AppVeyor.

Building

LEAF is a header-only library and it requires no building. It does not depend on Boost or on any other library.

The unit tests can be run with Boost Build or with Meson Build. To run the unit tests:

1. If using Boost Build:
   a. Clone LEAF under your `boost/libs` directory.
   b. Execute:

   ```
   cd leaf/test
   ../../../b2
   ```

2. If using Meson Build:
   a. Clone LEAF into any local directory.
   b. Execute:

   ```
   cd leaf
   meson bld/debug
   cd bld/debug
   meson test
   ```

Configuration Macros

The following configuration macros are recognized:

- `BOOST_LEAF_DIAGNOSTICS`: Defining this macro to `0` stubs out both `diagnostic_info` and `verbose_diagnostic_info`, which could improve the performance of the error path in some programs (if the macro is left undefined, LEAF defines it as `1`).
- `BOOST_LEAF_NO_EXCEPTIONS`: Disables all exception handling support. If left undefined, LEAF defines it based on the compiler configuration (e.g. `-fno-exceptions`).
- `BOOST_LEAF_NO_THREADS`: Disable all multi-thread support.

Support

The following support options are available:

- cpplang on Slack (use the `#boost` channel)
- Boost Users Mailing List
- Boost Developers Mailing List

(LEAF is not part of Boost).

Acknowledgements

Special thanks to Peter Dimov and Sorin Fetche.

Ivo Belchev, Sean Palmer, Jason King, Vinnie Falco, Glen Fernandes, Nir Friedman, Augustín Bergé — thanks for the valuable feedback.

Documentation rendered by Asciidoctor with these customizations.