Document number:P0824R0
Date: 20171016 (pre-Albuquerque)
Project: Programming Language C++, WG21, SG14, LEWG
Authors: Arthur O'Dwyer, Charley Bay, Odin Holmes, Michael Wong
Email: arthur.j.odwyer@gmail.com, charleyb123@gmail.com, holmes@auto-intern.de, michael@codeplay.com
Reply to: arthur.j.odwyer@gmail.com

Summary of SG14 discussion on <system_error>: towards exception-less error handling

1. Introduction

This paper summarizes a discussion that took place during the SG14 telecon of 2017-10-11. The discussion concerned the facilities of <system_error> (introduced in C++11), which include errc, error_code, error_condition, error_category, and system_error.

The discussion naturally also concerns the current idioms for exceptionless "disappointment handling" (P157R0, Lawrence Crowl), as seen in <filesystem> (C++17) and std::from_chars/to_chars (also C++17); and it concerns *future* idioms for exceptionless disappointment handling, as proposed in status_value<E,T> (P0262R0, Lawrence Crowl), in expected<T,E> (P0323R2, Vicente J. Botet Escribá), and in result<T,E> (Niall Douglas).

In all of those future idioms involving expected<T,E>-style result types, SG14 expects that the E type parameter will default to std::error_code — which is right and good. This means that the coming years will see extremely heavy use of std::error_code at every level of the system. This means that SG14 is very interested in correcting deficiencies in std::error_code sooner, rather than later.

On SG14's 2017-10-11 teleconference call, various people contributed to a miscellaneous "laundry list" of perceived deficiencies with <system_error>. Several "best practices" were also proposed for how we expect <system_error> facilities to be used in the best codebases. (Unfortunately, the Standard Library does not follow *anyone's* proposed "best practice"!) This paper is a summary of that discussion.

Some possible goals of this paper are:

1. Start a conversation about standardizing a non-exception error mechanism that improves on error code, or possibly thers (expected, status_value, outcome)
2. Demonstrate a best practice from three practitioners in different domains
3. Enable wider C++ support across Embedded, games, financial domains for disappointment

2. Description of C++11's <system_error> facilities

<system_error> provides the following patterns for dealing with error codes:

A std::error_code object wraps an integer "error enumerator" and a pointer to an "error category" (that is, a handle to the specific error domain associated with this integer enumerator). The addition of the error domain handle is what allows us to distinguish, say, error=5 "no leader for Kafka partition" (in the rdkafka domain) from error=5 "I/O error" (in the POSIX domain). Two error_code instances compare equal if and only if they represent the same error enumerator *in the same domain*.

Notice that std::error_condition is not the same type as std::error_code!

```
    namespace std {
class error_code {
    int value_;
    std::error_category *cat_;
public:
    error_code() noexcept : error_code(0, std::system_category()) {}
    error_code(int e, const std::error_category& c) noexcept : value_(e), cat_(&c) {}
    template<class E> error_code(E e) noexcept requires(std::is_error_code_enum_v<E>) { *this =
make_error_code(e); }  // intentional ADL
    template<class E> error_code& operator=(E e) noexcept
requires(std::is_error_code_enum_v<E>) { *this = make_error_code(e); }  // intentional ADL
    void assign(int e, const std::error_category& c) noexcept { value_ = e; cat_ = &c; }
    void clear() noexcept { *this = std::error_code(); }

    explicit operator bool() const noexcept { return !!value_; }
    int value() const noexcept { return value_; }
    const std::error_category& category() const noexcept { return cat_; }

    std::string message() const { return cat_->message(value_); }
    std::error_condition default_error_condition() const noexcept { return
cat_->default_error_condition(value_); }
};
bool operator==(const std::error_code& a, const std::error_code& b) noexcept { return a.value() ==
b.value() && &a.category() == &b.category(); }
} // namespace std
```

Each "error domain" is represented in code by an object of type std::error_category. These are always singletons, effectively, because error_code equality-comparison is implemented in terms of pointer comparison (see above). So two error_category objects located at different memory addresses represent different error domains, *by definition*, as far as the currently standardized scheme is concerned.

```
        namespace std {
   class error_category {
   public:
      constexpr error_category() noexcept = default;
      error_category(const error_category&) = delete;
      virtual ~error_category() {}

      virtual const char *name() const noexcept = 0;

      virtual std::error_condition default_error_condition(int e) const noexcept { return
std::error_condition(e, *this); }
      virtual std::string message(int e) const = 0;

      virtual bool equivalent(int e, const std::error_condition& condition) const noexcept { return
this->default_error_condition(e) == condition; }
      virtual bool equivalent(const std::error_code& code, int e) const noexcept { return code ==
std::error_code(e, *this); }
   };
   bool operator==(const std::error_category& a, const std::error_category& b) noexcept { return &a
== &b; }
   } // namespace std
```

It is intended that the programmer should *inherit publicly from* std::error_category and override its pure virtual methods (and optionally override its non-pure virtual methods) to create new library-specific error domains. An error domain encompasses a set of error enumerators with their associated human meanings (for example, error=5 meaning "no leader for Kafka partition"). So for example we can expect that rdkafka_category().message(5) would return std::string("no leader for partition").

The standard also provides a class std::error_condition which is almost identical in implementation to std::error_code.

```
        namespace std {
   class error_condition {
      int value_;
      std::error_category *cat_;
   public:
      error_condition() noexcept : error_code(0, std::generic_category()) {}
      error_condition(int e, const std::error_category& c) noexcept : value_(e), cat_(&c) {}
      template<class E> error_condition(E e) noexcept requires(std::is_error_condition_enum_v<E>)
{ *this = make_error_condition(e); }  // intentional ADL
      template<class E> error_condition& operator=(E e) noexcept
requires(std::is_error_condition_enum_v<E>) { *this = make_error_condition(e); }  // intentional ADL
      void assign(int e, const std::error_category& c) noexcept { value_ = e; cat_ = &c; }
      void clear() noexcept { *this = std::error_condition(); }

      explicit operator bool() const noexcept { return !!value_; }
      int value() const noexcept { return value_; }
```

```
        const std::error_category& category() const noexcept { return cat_; }

        std::string message() const { return cat_->message(value_); }
    };
    bool operator==(const std::error_condition& a, const std::error_condition& b) noexcept { return
a.value() == b.value() && &a.category() == &b.category(); }
    } // namespace std
```

The best practice for using std::error_condition is the subject of some debate in SG14; see the rest of this paper. However, clearly the vague intent of std::error_condition is to represent in some way a high-level "condition" which a low-level "error code" (thrown up from the bowels of the system) might or might not "match" in some high-level semantic sense. Notice that the error domain of a default-constructed error_code is system_category(), whereas the error domain of a default-constructed error_condition is generic_category().

The standard provides a highly customizable codepath for comparing an error_code against an error_condition with operator==. (Both error_code and error_condition are value types — in fact they are POD types — which means that their own operator==(A,A) are just bitwise comparisons. We are now speaking of operator==(A,B).)

```
          namespace std {
    bool operator==(const std::error_code& a, const std::error_condition& b) noexcept {
        return a.category().equivalent(a.value(), b) or b.category().equivalent(a, b.value());
    }
    bool operator==(const std::error_condition& a, const std::error_code& b) noexcept {
        return a.category().equivalent(a.value(), b) or b.category().equivalent(a, b.value());
    }
    } // namespace std
```

Recall that the base-class implementation of error_category::equivalent() is just to compare for strict equality; but the programmer's own error_category-derived classes can override equivalent() to have different behavior.

Lastly, the <system_error> header provides an exception class that wraps an error_code (but not an error_condition):

```
          namespace std {
    class system_error : public std::runtime_error {
        std::error_code code_;
        std::string what_;
    public:
        system_error(std::error_code ec) : code_(ec), what_(ec.message()) {}
        system_error(std::error_code ec, const std::string& w) : code_(ec), what_(ec.message() + ": " +
w) {}
        system_error(std::error_code ec, const char *w) : system_error(ec, std::string(w)) {}
        system_error(int e, const std::error_category& cat) : system_error(std::error_code(e, cat)) {}
        system_error(int e, const std::error_category& cat, const std::string& w) :
system_error(std::error_code(e, cat), w) {}
        system_error(int e, const std::error_category& cat, const char *w) :
system_error(std::error_code(e, cat), w) {}
        const std::error_code& code() const noexcept { return code_; }
        const char *what() const noexcept override { return what_.c_str(); }
    };
    } // namespace std
```

3. Proposed best practices for using C++11's <system_error> facilities

In the example that follows, we have an application appA, which calls into library libB, which calls into library libC, which calls into library libD.

Arthur O'Dwyer proposes the following general rules:

1. No enumeration type E should ever satisfy *both* is_error_code_enum_v<E> and is_error_condition_enum_v<E> simultaneously. (To do so would be to represent a low-level error code value and a high-level abstract condition simultaneously, which is impossible.)
2. For any enumeration type E, the ADL function make_error_code(E) should exist if-and-only-if is_error_code_enum_v<E>; and the ADL function make_error_condition(E) should exist if-and-only-if is_error_condition_enum_v<E>.
3. In any enumeration type E satisfying either is_error_code_enum_v<E> or is_error_condition_enum_v<E>, the enumerator value 0 must be set aside as a "success" value, and never allotted to any mode of failure. In fact, the enumeration E should have an enumerator success = 0 or none = 0 to ensure that this invariant is never broken accidentally by a maintainer.
4. Your library should have exactly as many error_category subclasses as it has enumeration types satisfying either is_error_code_enum_v<E> or is_error_condition_enum_v<E>, in a one-to-one correspondence. No error_category subclass should be "shared" between two enumeration types; and no error_category subclass should exist that is not associated with a specific enumeration type.
5. Each error_category subclass should be a singleton; that is, it should have a single instance across the entire program.
6. When your library detects a failure, it should construct an std::error_code representing the failure. This can be done using ADL make_error_code(LibD::ErrCode::failure_mode) or simply using LibD::ErrCode::failure_mode (which works because of std::error_code's implicit constructor from error-code-enum types).
7. This std::error_code is passed up the stack using out-parameters (as <filesystem>) or using Expected<T, std::error_code>.
8. When libB receives a std::error_code code that must be checked for *failure versus success*, it should use if (code) or if (!code).
9. When libB receives a std::error_code code that must be checked for *a particular source-specific error* (such as "rdkafka partition lacks a leader"), it should use if (code == LibD::ErrCode::failure_mode). This performs exact equality, and is useful if you know the exact source of the error you're looking for (such as "rdkafka").
10. When libB receives a std::error_code code that must be checked for *a high-level condition* (such as "file not found"), which might correspond to any of several source-specific errors across different domains, it may use if (code == LibC::ErrCondition::failure_mode), where LibC::ErrCondition is an error-condition-enum type provided by the topmost library (the one whose API we're calling — not any lower-level library). This will perform semantic classification.
11. Your library might perhaps *provide* semantic classification by providing an error-condition-enum type LibB::ErrCondition (and its associated error_category subclass LibB::ErrConditionCategory),

which encodes knowledge about the kinds of error values reported by LibC. Ideally, LibB::ErrConditionCategory::equivalent() should defer to LibC::ErrConditionCategory::equivalent() in any case where LibB is unsure of the meaning of a particular error code (for example, if it comes from an unrecognized error domain).

12. Most likely, std::error_condition and error-condition-enum types should simply *not be used*. libB should not expect its own callers to write if (code == LibB::ErrCondition::oom_failure); instead libB should expect its callers to write if (LibB::is_oom_failure(code)), where bool LibB::is_oom_failure(std::error_code) is a free function provided by libB. This successfully accomplishes semantic classification, and does it without any operator overloading, and therefore does it without the need for the std::error_condition type.

Charley Bay proposes (something like) the following general rules (paraphrased here by Arthur O'Dwyer):

1. No enumeration type E should ever satisfy *both* is_error_code_enum_v<E> and is_error_condition_enum_v<E> simultaneously. (To do so would be to represent a low-level error code value and a high-level abstract condition simultaneously, which is impossible.)
2. For any enumeration type E, the ADL function make_error_code(E) should exist if-and-only-if is_error_code_enum_v<E>; and the ADL function make_error_condition(E) should exist if-and-only-if is_error_condition_enum_v<E>.
3. In any enumeration type E satisfying either is_error_code_enum_v<E> or is_error_condition_enum_v<E>, the enumerator value 0 must be set aside as a "success" value, and never allotted to any mode of failure. In fact, the enumeration E should have an enumerator success = 0 or none = 0 to ensure that this invariant is never broken accidentally by a maintainer.
4. error_category subclasses are not necessarily singletons. It is conceivable that multiple instances of the same error_category subclass type could exist within the same program.
5. When your library detects a failure, it should construct an std::error_code representing the failure. This can be done using ADL make_error_code(LibD::ErrCode::failure_mode) or simply using LibD::ErrCode::failure_mode (which works because of std::error_code's implicit constructor from error-code-enum types).
6. This std::error_code is passed up the stack using out-parameters (as <filesystem>) or using Expected<T, std::error_code>.
7. When libB receives a std::error_code code that must be checked for *failure versus success*, it should use if (code) or if (!code).
8. When libB receives a std::error_code code, it must *never* be checked for *a particular source-specific error* (such as "rdkafka partition lacks a leader"). Every test should be done on the basis of semantic classification — whether at the coarse granularity of "failure versus success" or at the fine granularity of "partition lacks a leader." It is always conceivable that libC might change out its implementation so that it no longer uses libD; therefore, all testing of error codes returned by a libC API must be expressed in terms of the specific set of abstract failure modes exposed by that same libC API.
9. As explained in the preceding point, exact-equality comparisons should never be used. But with the standard syntax, there is a significant risk that the programmer will accidentally write if (code == LibB::ErrCode::oom_failure) (exact-equality comparison) instead of the intended if (code == make_error_condition(LibB::ErrCode::oom_failure)) (semantic classification). Therefore std::error_condition and error-condition-enum types should *not be used*. libB should expect its callers to write if (LibB::is_oom_failure(code)), where bool LibB::is_oom_failure(std::error_code) is a free function provided by libB. This successfully accomplishes semantic classification, and does

it without any operator overloading, and therefore does it without the need for the std::error_condition type.

Odin Holmes proposes the following rules with the following caveat and background from an embedded system point of view:

Odin rule 1 code that is expect to be executed in constexpr context should refrain from dynamic payloads (also required non virtual destructor on error catagory and non std::string solution as arther and charley have noted)

Odin rule 2 library code should refrain from touching dynamic payloads, after all the use case as I see it is to marschall error message contents up the call chain, the actuall use of this is only in toplevel user code.

Odin rule 3 there should be a customization point at the generation site of dynamic payloads which allows the user to make them empty

Background / use case: for the sake of arguement I am a microcontroller in a cows ear tag which periodically logs health data to a micro SD card and can send one message a day over sigfox to the rancher. If I encounter a filesystem error what should I do? I don't have a screen to push the error message to, my sigfox payload is like 10 bytes, if I try and log onto the SD Card after an error I run the rist of corrupting it. I have no use for the text in other words. My control flow would depend entirely on the error condition, maybe I get a file not found because its a fresh SD card, well I know how to handle that, if its anything else I probably just want to stop logging and send out an 'ear tag down' message over sigfox.

The current methodology for these category of devices is usually just a pile of hacks, it would be wonderful if we could make the standard library work on them. The reason we can't just dynamically allocate is that RAM uses power and we can't just use a bigger battery. In fact typical deep sleep persistent storage in these controllers is often less than 1K, while you have upwards of 64k or something when not asleep. Jutter or in other words non deterministic timing behavior is also a real issue as while your sigfox window approaches you need to start monitoring things and must be ready to send when your turn is up, if you miss it you may have to wait until tomorrow to send again. Also because of the very low amount of persistent RAM you often cannot use a classic scheduler (have one heap per thread) which brings its own problems, bottom line you probably don't have a heap. If we could just flip a switch and the dynamic payloads get a size of zero everything should still work as long as the library internals only look at the error_condition/code/domain.

4. Issues with the <system_error> facilities

On the 2017-10-11 teleconference, the following issues were discussed.

4.1. Use of std::string

std::error_category's method virtual std::string message(int) const converts an error-code-enumerator into a human-readable message. This functionality is apparently useful only for presentation to humans; i.e., we do not expect that anyone should be treating the result of message() as a unique key, nor scanning into its elements with strstr. However, as a pure virtual method, this method *must* be implemented by each error_category subclass.

Its return type is std::string, i.e., it introduces a hard-coded dependency on std::allocator<char>. This seems to imply that if you are on a computer system without new and delete, without std::allocator, without std::string, then you cannot implement your own error_category subclasses, which effectively means that you cannot use std::error_code to deal with disappointment. SG14 sees any hard-coded dependency on std::allocator as unfortunate. For a supposedly "fundamental" library like <system_error> it is *extremely* unfortunate.

During the call, Charley Bay commented that returning ownership of a dynamically allocated string allows the error_category subclass to return a message that differs based on *the current locale*. However, nobody on the call claimed that this functionality was important to them. Furthermore, if locale-awareness were desirable, then branching on the *current (global) locale* would be the wrong way to go about it, because that mechanism would not be usable by multi-threaded programs. The right way to introduce locale-awareness into error_category would be to provide a virtual method std::string message(int, const std::locale&).

SG14 seems to agree that eliminating the dependency on std::allocator would be nice.

SG14 seems to agree that dynamically allocated message strings are not an important feature.

Two ways of removing std::string were proposed: return const char*, or return std::string_view. Arthur O'Dwyer commented that he strongly prefers const char* for simplicity (no new library dependencies) and for consistency with std::error_category::name() and std::exception::what(). Niall Douglas commented that he prefers std::string_view over raw null-terminated const char* whenever possible.

Both ways of removing std::string alter the return type of a pure virtual method and thus inevitably break every subclass of std::error_category ever. SG14 has no way out of this dilemma other than to suggest "wait for std2 and do it correctly there."

4.2. Proliferation of "two-API" libraries

<filesystem> is the poster child for this issue. Every function and method in <filesystem> comes in two flavors: throwing and non-throwing. The throwing version gets the "natural" signature (as is right and expected in C++); and the non-throwing version gets a signature with an extra out-parameter of type std::error_code&. The expectation is apparently that <system_error> users will be willing to write "C-style" code:

```
    namespace fs = std::filesystem;
void truncate_if_large(const fs::path& p) noexcept
{
    std::error_code ec;  // declare an uninitialized variable
    uintmax_t oldsize = fs::file_size(p, ec);
    if (ec) { report_error(ec); return; }
    if (oldsize > 1000) {
        fs::resize_file(p, 1000, ec);
        if (ec) { report_error(ec); return; }
    }
}
```

It would be nicer if the non-throwing API had exactly the same signatures as the throwing API, except that it should return expected<T> or result<T> instead of T. On the telecon, Arthur O'Dwyer commented that this can't easily be done because you cannot have two functions with the same name and the same signature, differing only in return type.

It is possible to segregate the *free functions* into a separate namespace, say namespace std::filesystem::nothrow, so that the above code could be written as

```
    namespace fs = std::filesystem::nothrow;  // hypothetical
void truncate_if_large(const fs::path& p) noexcept
{
    auto oldsize = fs::file_size(p);
    if (!oldsize.has_value()) { report_error(oldsize.error()); return; }
    if (oldsize.value() > 1000) {
        auto failure = fs::resize_file(p, 1000);
        if (failure) { report_error(failure.error()); return; }
    }
}
```

However, this doesn't help with member functions, such as directory_entry::is_symlink().

Having two APIs (throwing and non-throwing) side by side in the same namespace has another disadvantage. There is a significant risk that the programmer might accidentally leave off the out-parameter that signifies "non-throwing-ness", resulting in a call to the throwing version when a call to the non-throwing version was intended.

```
    namespace fs = std::filesystem;
void truncate_if_large(const fs::path& p) noexcept
{
    std::error_code ec;  // declare an uninitialized variable
    uintmax_t oldsize = fs::file_size(p, ec);
    if (ec) { report_error(ec); return; }
    if (oldsize > 1000) {
        fs::resize_file(p, 1000);  // Oops! Bug goes undetected by all major vendors.
        if (ec) { report_error(ec); return; }
    }
}
```

Therefore, segregating throwing from non-throwing functions is desirable. But we don't know how to make segregation work for member functions. Therefore perhaps the *best* outcome would be to stick with a single (non-throwing) API for each library.

If we had a single (non-throwing) API that returned something like Expected<T>, and if Expected<T> had a member function T or_throw() that returned the ex.value() if possible or else threw a system_error initialized from ex.error(), then we could write exception-throwing code fluently as follows:

```
    namespace fs = std::filesystem::nothrow;
void truncate_if_large(const fs::path& p) noexcept
{
    uintmax_t oldsize = fs::file_size(p).or_throw();
    if (oldsize > 1000) {
        fs::resize_file(p, 1000).or_throw();
    }
}
```

Here we assume that the template class Expected<void> is marked with the standard [[nodiscard]] attribute, so that if the programmer accidentally leaves off the final or_throw() the compiler will emit a warning.

SG14 seems not to have a great answer for how to avoid "two-API" libraries such as <filesystem> going forward; but we believe that "two-API" libraries should be avoided. The Networking TS seems to be shaping up to be another "two-API" library. We believe this is unfortunate.

4.3. No wording sets aside the 0 enumerator

The current Standard strongly implies the best-practice mentioned above: that every error-code-enumerator and every error-condition-enumerator should set aside success = 0 as a special case.

```
    enum class TroublesomeCode { out_of_memory, out_of_files };
struct TroublesomeCategory : public std::error_category {
    const char *name() const noexcept override { return ""; }
    std::string message(int e) const override {
        switch (e) {
            case TroublesomeCode::out_of_memory: return "out of memory";
            case TroublesomeCode::out_of_files: return "out of files";
            default: __builtin_unreachable();
        }
    }
};
const std::error_category& troublesome_category() {
    static const TroublesomeCategory instance;
    return instance;
}

template<> struct std::is_error_code_enum<TroublesomeCode> : std::true_type {};
std::error_code make_error_code(TroublesomeCode e) {
    return std::error_code((int)e, troublesome_category());
}

int main() {
    std::error_code ec = TroublesomeCode::out_of_memory;
    if (ec) {
        puts("This line will not be printed.");
    }
}
```

SG14 would like to see some explicit acknowledgment in the Standard that error-code enumerators with value 0 are "special," i.e., they will not be treated as "errors" by any of the machinery in the Standard. Error codes with value 0 are effectively reserved for the "success" case, and programmers should not attempt to use them for any other purpose.

Vice versa, programmers should be aware that using a non-zero integer value to represent "success" will not work as expected. Consider an HTTP library that naively attempts to use ok = 200 as its "success" code, and then provides an ADL make_error_code like this:

```
    enum class HTTPStatusCode { ok = 200, created = 201, /* ... */ };

template<> struct std::is_error_code_enum<HTTPStatusCode> : std::true_type {};
std::error_code make_error_code(HTTPStatusCode e) {
  return std::error_code((e == ok) ? 0 : (int)e, http_status_category());
}

std::string HTTPStatusCategory::message(int e) const {
  switch (e) {
    case 0: return "200 OK";
    case 201: return "201 Created";
    // ...
  }
}
```

The programmer may head far down this "garden path" under the assumption that his goal of a non-zero "ok" code is attainable; but we on the Committee know that it is *not* attainable. We should save the programmer some time and some headaches, by explicitly reserving error-code 0 in the standard.