P1028R4: SG14 `status_code` and standard `error` object

Document #:	P1028R4
Date:	2022-10-28
Project:	Programming Language C++ Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A proposal for the replacement, in new code, of the system header `<system_error>` with a substantially refactored and lighter weight design, which meets modern C++ design and implementation. This paper received the following vote at the May 2018 meeting of SG14: 8/2/1/0/0 (SF/WF/N/WA/SA).

A C++ 11 reference implementation of the proposed replacement can be found at `https://github.com/ned14/status-code`. Support for the proposed objects has been wired into Boost.Outcome [1] which has been shipping with the Boost C++ Libraries for three years.

The reference implementation has been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64. It has been quite popular with the C++ userbase, indeed there are two known complete re-implementations, one of which was described by [P2170] *Feedback on implementing the proposed std::error type*. This proposed design has shipped on every recent copy of Microsoft Windows and Apple iOS, and a fair chunk of Android devices. I believe it is amongst the best tested designs proposed for library standardisation in recent years.

This proposal is a much richer and more powerful framework than `<system_error>`, whilst remaining fully backwards compatible with it. Indeed, it can almost completely replace the dynamic exception mechanism with a fully deterministic alternative, and it has been proposed as the `std::error` implementation for [P0709] *Zero overhead deterministic exceptions* in [P1095] *Zero overhead deterministic failure*.

My apologies for the two years which have elapsed since R3. 'Real life' has intervened on my standards paper writing, and it is only due to my current client MayStreet London Stock Exchange Group allowing me a few work hours per month to work on standards papers that you see this R4 now and not yet more months from now. I hope that normal service will resume in 2023. I will not be at the upcoming Kona meeting, but I currently expect to be at the Issaquah meeting in early 2023.

Main change since R3:

- Added a new section reviewing design feedback from reimplementers.
- Many formerly non-`constexpr` member functions became `constexpr` thanks to C++ 20. Thanks to whomever suggested this for the reference implementation about a year ago.
- The only major design fix in the past two years was to add `status_code_domain::payload_info()`. This describes the alignment of erased payloads, so the correct bytes are copied when de-erasing a payload. New constructors have been added to erased status codes allowing construction from a compile-time unknown status code type, these will fail at runtime if the construction is incompatible.
- Added `quick_status_code_from_enum` which was suggested by a reimplementer.
- Semantics comparisons can now compare to a wider set of types than previously (i.e. we allow for implicit conversions).
- Added `status_error<void>::code()` to obtain access to the unknown status code from an unknown status code exception type.
- Added `posix_code::current()` and `win32_code::current()` to wrap the system's API for fetching the current platform error state.
- Added missing mixins for `std_error_code`, `posix_code`, `win32_code` etc.
- Added `http_status_code`.
- Reworked addendum about `result<T>` now that Boost.System has reimplemented the same facility.

Contents

1 Introduction

The `<system_error>` header entered the C++ standard in the C++ 11 standard, the idea for which having been split off from the Filesystem TS proposal into its own [N2066] proposal back in 2006. Despite its relative lack of direct usage by the C++ userbase, according to [2], `<system_error>` has become one of the most common internal dependencies for all other standard header files, frequently constituting up to 20% of all the tokens brought into the compiler by other standard header files e.g. `<array>`, `<complex>` or `<optional>`. In this sense, it is amongst the most popular system headers in the C++ standard library.

So why would anyone want to replace it? It unfortunately suffers from a number of design problems

only now apparent after twelve years of hindsight, which makes it low hanging fruit in the achievement of the 'reduce compile time' and 'alternatives to complicated and/or error-prone features' goals listed in [P2000] *Direction for ISO C++*. We, from Study Group 14 (the GameDev & Low Latency WG21 working group), listed many of these problems in [P0824], and after an extensive period of consultation with other stakeholders including the Boost C++ Libraries, we thence designed and implemented an improved substitute which does not have those problems. It is this improved, fully backwards compatible, design that we propose now.

This proposed library may be useful as the standardised implementation of the lightweight throwable `error` object as proposed by [P0709] *Zero-overhead deterministic exceptions: Throwing values*. It is [P0829] *Freestanding C++* compatible i.e. without dependency on any STL or language facility not usable on embedded systems.

An example of use:

```
std::system_code sc;  // default constructs to empty
native_handle_type h = open_file(path, sc);
// Is the code a failure?
if(sc.failure())
{
  // Do semantic comparison to test if this was a file not found failure
  // This will match any system-specific error codes meaning a file not found
  if(sc != std::errc::no_such_file_or_directory)
  {
    std::cerr << "FATAL: " << sc.message().c_str() << std::endl;
    std::terminate();
  }
}
```

The above is 100% portable code. Meanwhile, the implementation of `open_file()` might be these:

```
// POSIX implementation
using native_handle_type = int;
native_handle_type open_file(const char *path,
  std::system_code &sc) noexcept
{
  sc.clear();  // clears to empty
  native_handle_type h = ::open(path, O_RDONLY);
  if(-1 == h)
  {
    // posix_code type erases into system_code
    sc = std::posix_code(errno);
  }
  return h;
}
```

```
// Microsoft Windows implementation
using native_handle_type = HANDLE;
native_handle_type open_file(const wchar_t *path,
  std::system_code &sc) noexcept
{
  sc.clear();  // clears to empty
  native_handle_type h = CreateFile(path,
    GENERIC_READ,
    FILE_SHARE_READ|FILE_SHARE_WRITE|
        FILE_SHARE_DELETE,
    nullptr,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    nullptr
  );
  if(INVALID_HANDLE_VALUE == h)
  {
    // win32_code type erases into system_code
    sc = std::win32_code(GetLastError());
  }
  return h;
}
```

2 Impact on the Standard

The proposed library is a pure-library solution.

It would be great if the object could be bit relocatable e.g. via [P1029] *move* = *bitcopies* or equivalent, but I suspect that support for those won't be done by EWG in time. Some compilers e.g. clang have proprietary attributes which implement bitcopying moves.

3 Feedback from reimplementors of the proposed design

3.1 Jesse Towner

Quoted with permission from Jesse Towner:

> I'd like to give you some feedback on how we've been using `status_code` and `error` in a few large projects over the past couple of years. We've had a lot of success with the library, but there have been a few rough edges and a major pain point, but more on that below as well as my solution to it.
>
> Now, just for some background, I've implemented our own independent version of the status-code library directly from P1028, and did so for a couple of reasons. First, for better integration into our own proprietary low-level C++ extensions library and second because we're currently using C++17 as a baseline and that naturally allows for a much more concise implementation. Furthermore, doing so allowed me to vet your design and give you all of that feedback a while ago as well, so there's also that, hehe. Anyway, you might be happy to know that we're using this in Microsoft Solitaire Collection, which of course comes pre-installed on Windows 10 these days and has wide adoption in both the Apple and Google Stores. You can probably imagine the size of the install base. So if anyone is still doubting that the status-code library as proposed isn't in wide use, you can always give them that, hehe.
>
> So jumping right into things, we started off by using type-erased error objects for propagating errors in an exceptionless manner across threads with our own multi-platform concurrency library that has futures & promises, senders & receivers and executors. We opted to use `std::expected` to package return value and errors together, since at the time it seemed fairly mature in its design. All of `std::expected`s copy constructors and copy-assignment operators automatically get excluded from the overload set when using a move-only Error object, and then all that needed to be done was to specialize `std::bad_expected_access` for both `status_code<erased<ErasedType>>` and error so that it calls `.clone()` to store a copy of the underlying Error. If we were to do it all over again, we might consider using `result<T>` but we will probably hold off on changing anything until LEWG makes further decisions on P1028 and P0709. So if they don't like neither P0709 nor `result<T>`, in case you haven't thought of it already – although I bet you probably have – you could as a last resort try pitching `std::expected` with move-only `status_code`s using the above technique to allow P1028 to go through, perhaps

just making `std::result<T>` be a template alias for `std::expected<T, std::error>`. But I digress.

Moving on, I think the core part of the status-code library is a little complex and you certainly have already heard complaints along those lines. But after implementing it myself, I think that complexity is fully justified given what the library does for us. It really enables a rich and powerful way of working with various different types of error codes in a more-or-less orthogonal and extensible manner. It address all of the issues with `<system_error>`. For all of the builtin system-level error codes, the library does a great job. And for users of the library, things are more or less straightforward and easy to comprehend when working directly with a `status_code` or error object, especially once they understand how semantic equivalence testing works and that you can have equivalence classes of status codes in the same way that you can with exceptions.

The big problem, and this was the major point of pain for us I was talking about earlier, is that's really really difficult to teach everyone how to effectively write concrete `status_code_domain`s for custom enum-based status codes. The reasons for this I imagine are kind of related to the same reasons its often hard to get everyone to write exception-safe code all of the time, which is why we originally chose not to use exception handling in the first place. The intricacies of error handling sadly is not an area of interest for most people, and when you're on a deadline and needing to finish off implementing end-user facing features for the next milestone, things like setting up and utilizing `status_code_domain`s properly or writing exception-safe code all end up low on the priority list.

Having to write one to two hundred lines of code to create a `status_code_domain` each time you want to allow an enum to be used with the status-code system just doesn't scale well. On the other hand, when you look at exceptions in C++, the one nice thing about exceptions is that it's a really simple process to create a new exception class. You merely derive from `std::exception` or another child class of `std::exception` that already exists and you're off to the races. If you need to give the exception some state and a custom error message, you add some member variables and override `what()`. Easy. It would be really nice if creating custom `status_code_domain`s for enums, or indeed any trivial or move-relocatable class type, would be just as simple. I believe status code domains need to be invisible most of the time, and yet something that you can get at when needed and have it all just work.

Now initially, I myself played this off as not a big issue. Mostly because at the time I couldn't think of a better way of doing things. But as the complaints continued to roll in and as the situation worsened, I realized something needed to be done about it or we'd be in trouble. I won't go into the precise details, but even even for the C++ experts, having to set up a new `status_code_domain` was becoming an exercise in writing boilerplate, when we'd rather be working on more important things. When you have multiple dozens of custom enum-based status codes all throughout your code base for your various application level components and services, it's a real chore to maintain all of that.

[... Jesse goes on to propose a solution which ended up becoming proposed

`quick_status_code_from_enum` below ...]

My thanks to Jesse for taking the time to send such a detailed piece of design feedback and suggestions for improvement, which were incorporated since R3 of this paper.

An example of `quick_status_code_from_enum` in use:

```
namespace another_namespace
{
  // This is some custom enum code
  enum class AnotherCode : size_t
  {
    success1,
    goaway,
    success2,
    error2
  };
}  // namespace another_namespace


/* Rather than copy and paste the extensive boilerplate to declare a custom status code domain,
have the compiler generate it via metaprogramming. We simply supply the bits to poke into the
boilerplate generation.

'quick_status_code_from_enum_defaults' sets sensible defaults so you only need to override
the specific bits you want to customise.
*/
template <>
struct quick_status_code_from_enum<another_namespace::AnotherCode>
  : quick_status_code_from_enum_defaults<another_namespace::AnotherCode>
{
  // Text name of the enum
  static constexpr const auto domain_name = "Another Code";

  // Unique UUID for the enum. PLEASE use https://www.random.org/cgi-bin/randbyte?nbytes=16&format=h
  static constexpr const auto domain_uuid = "{be201f65-3962-dd0e-1266-a72e63776a42}";

  // Map of each enum value to its text string, and list of semantically equivalent errc's
  static const std::initializer_list<mapping> &value_mappings()
  {
    static const std::initializer_list<mapping> v = {

    // Format is: { enum value, "string representation", { list of errc mappings ... } }
    {another_namespace::AnotherCode::success1, "Success 1", {errc::success}},        //
    {another_namespace::AnotherCode::goaway, "Go away", {errc::permission_denied}},  //
    {another_namespace::AnotherCode::success2, "Success 2", {errc::success}},        //
    {another_namespace::AnotherCode::error2, "Error 2", {}},                         //

    };
    return v;
  }

  // Completely optional definition of mixin for the status code synthesised from 'Enum'.
  // It can be omitted.
  template <class Base> struct mixin : Base
  {
```

```
    using Base::Base;
    constexpr int custom_method() const { return int(this->value()); }
  };
};

namespace another_namespace
{
  // ADL discovered, must be in same namespace as AnotherCode
  constexpr inline
  quick_status_code_from_enum_code<another_namespace::AnotherCode>
  status_code(AnotherCode c)
  {
    // Because we customised quick_status_code_from_enum above, status_code
    // knows now how to implicitly construct from a AnotherCode
    return c;
  }
}  // namespace another_namespace


// status_code<erased<intptr_t>> will now construct from AnotherCode
system_code v1(another_namespace::AnotherCode::error2);

// ADL discovered convenience factory function is another option
// Note the mixed in 'custom_method()' to the status code type
constexpr auto v2 = status_code(another_namespace::AnotherCode::error2);
assert(v2.value() == another_namespace::AnotherCode::error2);
assert(v2.custom_method() == 3);
```

3.2 [P2170] *Feedback on implementing the proposed std::error type*

P2170 made a number of alternative design suggestions:

1. Proposed `std::error` ought to be copy constructible as well as move constructible, thus asking the domain to perform a copy per `std::error` copy.
2. `std::error_code` ought to be convertible into `std::error`.
3. Proposed `std::generic_code` ought to represent a different enum to `std::errc`.
4. Remove everything outside `std::error` and `std::error_domain` (with four domains supplied out of the box) as being superfluous to need.
5. Semantic comparison ought to be simpler than that proposed in this paper.
6. The unique id ought to be 128 bit, not 64 bit.

My reply to those design points:

1. P1028 provides `.clone()` if you want to copy a status code. It does not produce good codegen, because it requires asking the domain to make the copy (and it may throw an exception refusing or rejecting the request). Insisting on move-only semantics for `std::error` makes for *much* better codegen because the compiler can collapse inlined TRY operations, you can see an example of this at https://godbolt.org/z/qdnY78cxn. I would also point out that if you

really want a copy constructible `error`, it's trivial to build your own on top of `status_code`. Or, just use `std::error_code`, it's not going anywhere.

2. Proposed `std::error` already implicitly constructs from `std::error_code`, with full preservation of the error code's `std::error_category` i.e. to be clear here, the **original** `std::error_category` is used for semantic comparisons to status codes if the input error category does not map onto a status code domain (i.e. is a custom third party defined error category).

3. I would suggest that the enum values in `std::errc` can be extended with new values at any time by the committee.

4. It is tempting to think that one just needs a `std::error` and the minimum possible additional. However, in my opinion, as soon as your needs exceed bare minimum, you'll then find yourself straightjacketed by the limited design. The design proposed by this paper has stood up very well over the past three years of production use by multiple people not just this author, with only a few minor design issues discovered and fixed in that time (alignment of erased codes was the big one).

5. One of the specific reasons that semantic comparisons are more complex than would be otherwise necessary is to preserve semantic comparison compatibility with `std::error_category` such that status codes can be compared to error codes using an arbitrary third party supplied error category instance. I think this backwards compatibility worth preserving, even if semantic comparison logic is then slightly more complex.

6. I would refer to [P0824] *Summary of SG14 discussion on* <*system_error*> where the statistical chance of collision in a properly random 64 bit unsigned integer is considered to be very low. As an example, you would need 190 unique status code domains in a process to have the same chance of collision as a bit flipping on your entire hard drive.

4 Proposed Design

4.1 `status_code_domain`

```
/*! The main workhorse of the system_error2 library, can be typed
('status_code<DomainType>'), erased-immutable ('status_code<void>') or
erased-mutable ('status_code<erased<T>>').

Be careful of placing these into containers! Equality and inequality operators are
*semantic* not exact. Therefore two distinct items will test true! To help prevent
surprise on this, 'operator<' and 'std::hash<>' are NOT implemented in order to
trap potential incorrectness. Define your own custom comparison functions for your
container which perform exact comparisons.
*/
template <class DomainType> class status_code;

class _generic_code_domain;

//! The generic code is a status code with the generic code domain, which is that of 'errc' (POSIX).
using generic_code = status_code<_generic_code_domain>;
```

```
/*! Abstract base class for a coding domain of a status code.
*/
class status_code_domain
{
  template <class DomainType> friend class status_code;
  template <class StatusCode> friend class indirecting_domain;

public:
  //! Type of the unique id for this domain.
  using unique_id_type = unsigned long long;

  /*! (Potentially thread safe) Reference to a message string.

  Be aware that you cannot add payload to implementations of this class.
  You get exactly the `void *[3]` array to keep state, this is usually
  sufficient for a `std::shared_ptr<>` or a `std::string`.

  You can install a handler to be called when this object is copied,
  moved and destructed. This takes the form of a C function pointer.
  */
  class string_ref
  {
  public:
    //! The value type
    using value_type = const char;
    //! The size type
    using size_type = size_t;
    //! The pointer type
    using pointer = const char *;
    //! The const pointer type
    using const_pointer = const char *;
    //! The iterator type
    using iterator = const char *;
    //! The const iterator type
    using const_iterator = const char *;

  protected:
    //! The operation occurring
    enum class _thunk_op
    {
      copy,
      move,
      destruct
    };
    //! The prototype of the handler function. Copies can throw, moves and destructs cannot.
    using _thunk_spec = void (*)(string_ref *dest, const string_ref *src, _thunk_op op);

    //! Pointers to beginning and end of character range
    pointer _begin{}, _end{};

    //! Three `void*` of state
    void *_state[3]{};  // at least the size of a shared_ptr

    //! Handler for when operations occur
    const _thunk_spec _thunk{nullptr};
```

```
56
57    constexpr explicit string_ref(_thunk_spec thunk) noexcept;
58
59  public:
60    //! Construct from a C string literal
61    constexpr explicit string_ref(const char *str, size_type len = static_cast<size_type>(-1),
62                                  void *state0 = nullptr, void *state1 = nullptr,
63                                  void *state2 = nullptr, _thunk_spec thunk = nullptr) noexcept;
64    //! Copy construct the derived implementation.
65    constexpr string_ref(const string_ref &o);
66    //! Move construct the derived implementation.
67    constexpr string_ref(string_ref &&o) noexcept;
68    //! Copy assignment
69    constexpr string_ref &operator=(const string_ref &o);
70    //! Move assignment
71    constexpr string_ref &operator=(string_ref &&o) noexcept;
72    //! Destruction
73    constexpr ~string_ref();
74
75    //! Returns whether the reference is empty or not
76    [[nodiscard]] constexpr bool empty() const noexcept;
77    //! Returns the size of the string
78    constexpr size_type size() const noexcept;
79    //! Returns a null terminated C string
80    constexpr const_pointer c_str() const noexcept;
81    //! Returns a null terminated C string
82    constexpr const_pointer data() const noexcept;
83    //! Returns the beginning of the string
84    constexpr iterator begin() noexcept;
85    //! Returns the beginning of the string
86    constexpr const_iterator begin() const noexcept;
87    //! Returns the beginning of the string
88    constexpr const_iterator cbegin() const noexcept;
89    //! Returns the end of the string
90    constexpr iterator end() noexcept;
91    //! Returns the end of the string
92    constexpr const_iterator end() const noexcept;
93    //! Returns the end of the string
94    constexpr const_iterator cend() const noexcept;
95  };
96
97  /*! A reference counted, threadsafe reference to a message string.
98  */
99  class atomic_refcounted_string_ref : public string_ref
100 {
101   struct _allocated_msg
102   {
103     mutable std::atomic<unsigned> count;
104   };
105   _allocated_msg *&_msg() noexcept;
106   const _allocated_msg *_msg() const noexcept;
107
108   static void _refcounted_string_thunk(string_ref *_dest, const string_ref *_src, _thunk_op op)
          noexcept;
109
110 public:
```

```
    //! Construct from a C string literal allocated using ‘malloc()‘.
    explicit atomic_refcounted_string_ref(const char *str, size_type len = static_cast<size_type>(-1),
                                          void *state1 = nullptr, void *state2 = nullptr) noexcept;
  };

private:
  unique_id_type _id;

protected:
  /*! Use [https://www.random.org/cgi-bin/randbyte?nbytes=8&format=h](https://www.random.org/cgi-bin/
      randbyte?nbytes=8&format=h) to get a random 64 bit id.
  Do NOT make up your own value. Do NOT use zero.
  */
  constexpr explicit status_code_domain(unique_id_type id) noexcept;
  //! No public copying at type erased level
  status_code_domain(const status_code_domain &) = default;
  //! No public moving at type erased level
  status_code_domain(status_code_domain &&) = default;
  //! No public assignment at type erased level
  status_code_domain &operator=(const status_code_domain &) = default;
  //! No public assignment at type erased level
  status_code_domain &operator=(status_code_domain &&) = default;
  //! No public destruction at type erased level
  ~status_code_domain() = default;

public:
  //! True if the unique ids match.
  constexpr bool operator==(const status_code_domain &o) const noexcept;
  //! True if the unique ids do not match.
  constexpr bool operator!=(const status_code_domain &o) const noexcept;
  //! True if this unique is lower than the other’s unique id.
  constexpr bool operator<(const status_code_domain &o) const noexcept;

  //! Returns the unique id used to identify identical category instances.
  constexpr unique_id_type id() const noexcept;
  //! Name of this category.
  constexpr virtual string_ref name() const noexcept = 0;
  //! Information about the payload of the code for this domain
  struct payload_info_t
  {
    size_t payload_size;     //!< The payload size in bytes
    size_t total_size;       //!< The total status code size in bytes (includes domain pointer and
        mixins state)
    size_t total_alignment;  //!< The total status code alignment in bytes

    payload_info_t() = default;
    constexpr payload_info_t(size_t _payload_size, size_t _total_size, size_t _total_alignment);
  };
  //! Information about this domain’s payload
  constexpr virtual payload_info_t payload_info() const noexcept = 0;

protected:
  //! True if code means failure.
  constexpr virtual bool _do_failure(const status_code<void> &code) const noexcept = 0;
  //! True if code is (potentially non-transitively) equivalent to another code in another domain.
  constexpr virtual bool _do_equivalent(const status_code<void> &code1, const status_code<void> &code2
```

```
      ) const noexcept = 0;
  //! Returns the generic code closest to this code, if any.
  constexpr virtual generic_code _generic_code(const status_code<void> &code) const noexcept = 0;
  //! Return a reference to a string textually representing a code.
  constexpr virtual string_ref _do_message(const status_code<void> &code) const noexcept = 0;
  //! Throw a code as a C++ exception.
  [[noreturn]] constexpr virtual void _do_throw_exception(const status_code<void> &code) const = 0;
};
```

4.2 Traits

```
//! Namespace for user injected mixins
namespace mixins
{
  template <class Base, class T> struct mixin : public Base
  {
    using Base::Base;
  };
}

/*! A tag for an erased value type for `status_code<D>`.
Available only if `ErasedType` satisfies `traits::is_move_relocating<ErasedType>::value`.
*/
template <class ErasedType>
requires(traits::is_move_relocating<ErasedType>::value)
struct erased
{
  using value_type = ErasedType;
};

template <class Enum> struct quick_status_code_from_enum;

//! Trait returning true if the type is a status code.
template <class T> struct is_status_code;
template <class T> static constexpr bool is_status_code_v;
```

4.3 status_code<void>

```
/*! A type erased lightweight status code reflecting empty, success, or failure.
Differs from `status_code<erased<>>` by being always available irrespective of
the domain's value type, but cannot be copied, moved, nor destructed. Thus one
always passes this around by const lvalue reference.
*/
template <> class status_code<void>
{
  template <class T> friend class status_code;

public:
  //! The type of the domain.
  using domain_type = void;
  //! The type of the status code.
```

```
  using value_type = void;
  //! The type of a reference to a message string.
  using string_ref = typename status_code_domain::string_ref;

protected:
  const status_code_domain *_domain{nullptr};

protected:
  //! No default construction at type erased level
  status_code() = default;
  //! No public copying at type erased level
  status_code(const status_code &) = default;
  //! No public moving at type erased level
  status_code(status_code &&) = default;
  //! No public assignment at type erased level
  status_code &operator=(const status_code &) = default;
  //! No public assignment at type erased level
  status_code &operator=(status_code &&) = default;
  //! No public destruction at type erased level
  ~status_code() = default;

  //! Used to construct a non-empty type erased status code
  constexpr explicit status_code(const status_code_domain *v) noexcept;

public:
  //! Return the status code domain.
  constexpr const status_code_domain &domain() const noexcept;
  //! True if the status code is empty.
  [[nodiscard]] constexpr bool empty() const noexcept;

  //! Return a reference to a string textually representing a code.
  constexpr string_ref message() const noexcept;
  //! True if code means success.
  constexpr bool success() const noexcept;
  //! True if code means failure.
  constexpr bool failure() const noexcept;

  /*! True if code is strictly (and potentially non-transitively) semantically equivalent to
  another code in another domain.

  Note that usually non-semantic i.e. pure value comparison is used when the other
  status code has the same domain. As `equivalent()` will try mapping to generic code,
  this usually captures when two codes have the same semantic meaning in `equivalent()`.
  */
  template <class T> constexpr bool strictly_equivalent(const status_code<T> &o) const noexcept;

  /*! True if code is equivalent, by any means, to another code in another domain
  (guaranteed transitive).

  Firstly `strictly_equivalent()` is run in both directions. If neither succeeds, each domain
  is asked for the equivalent generic code and those are compared.
  */
  template <class T> constexpr bool equivalent(const status_code<T> &o) const noexcept;

  //! Throw a code as a C++ exception.
  [[noreturn]] constexpr void throw_exception() const;
```

```
};
```

4.4 detail::status_code_storage<DomainType>

It is highly unusual for items in a detail namespace to be proposed for standardisation. However it was felt that until a judgement is taken on mixins, it was best to retain the structure of the reference library implementation.

```
namespace detail
{
  template <class DomainType> struct get_domain_value_type
  {
    using domain_type = DomainType;
    using value_type = typename domain_type::value_type;
  };
  template <class ErasedType> struct get_domain_value_type<erased<ErasedType>>
  {
    using domain_type = status_code_domain;
    using value_type = ErasedType;
  };
  template <class DomainType> class status_code_storage : public status_code<void>
  {
    static_assert(!std::is_void<DomainType>::value, "status_code_storage<void> should never occur!");
  public:
    //! The type of the domain.
    using domain_type = typename get_domain_value_type<DomainType>::domain_type;
    //! The type of the status code.
    using value_type = typename get_domain_value_type<DomainType>::value_type;
    //! The type of a reference to a message string.
    using string_ref = typename domain_type::string_ref;

    //! Return the status code domain.
    constexpr const domain_type &domain() const noexcept;

    //! Reset the code to empty.
    constexpr void clear() noexcept;

    //! Return a reference to the `value_type`.
    constexpr value_type &value() & noexcept;
    //! Return a reference to the `value_type`.
    constexpr value_type &&value() && noexcept;
    //! Return a reference to the `value_type`.
    constexpr const value_type &value() const &noexcept;
    //! Return a reference to the `value_type`.
    constexpr const value_type &&value() const &&noexcept;

  protected:
    status_code_storage() = default;
    status_code_storage(const status_code_storage &) = default;
    status_code_storage(status_code_storage &&) = default;
    status_code_storage &operator=(const status_code_storage &) = default;
    status_code_storage &operator=(status_code_storage &&) = default;
    ~status_code_storage() = default;
```

```
    value_type _value{};
    struct _value_type_constructor { };
    template <class... Args>
    constexpr status_code_storage(_value_type_constructor /*unused*/, const status_code_domain *v,
        Args &&... args);
  };
}  // namespace detail
```

4.5 status_code<DomainType>

```
/*! A lightweight, typed, status code reflecting empty, success, or failure.
This is the main workhorse of the system_error2 library.

An ADL discovered helper function `make_status_code(T, Args...)` is looked up by one
of the constructors. If it is found, and it generates a status code compatible with this
status code, implicit construction is made available.

You may mix in custom member functions and member function overrides by injecting a specialisation of
`mixins::mixin<Base, YourDomainType>`. Your mixin must inherit from `Base`.
*/
template <class DomainType>
requires(
  (!std::is_default_constructible<typename DomainType::value_type>::value
    || std::is_nothrow_default_constructible<typename DomainType::value_type>::value)
  && (!std::is_move_constructible<typename DomainType::value_type>::value
    || std::is_nothrow_move_constructible<typename DomainType::value_type>::value)
  && std::is_nothrow_destructible<typename DomainType::value_type>::value
)
class status_code : public mixins::mixin<detail::status_code_storage<DomainType>, DomainType>
{
  template <class T> friend class status_code;

public:
  //! The type of the domain.
  using domain_type = DomainType;
  //! The type of the status code.
  using value_type = typename domain_type::value_type;
  //! The type of a reference to a message string.
  using string_ref = typename domain_type::string_ref;

public:
  //! Default construction to empty
  status_code() = default;
  //! Copy constructor
  status_code(const status_code &) = default;
  //! Move constructor
  status_code(status_code &&) = default;
  //! Copy assignment
  status_code &operator=(const status_code &) = default;
  //! Move assignment
  status_code &operator=(status_code &&) = default;
  ~status_code() = default;

```

```
  //! Return a copy of the code.
  constexpr status_code clone() const;

  //! Implicit construction from any type where an ADL discovered
  //! `make_status_code(T, Args ...)` returns a `status_code`.
  template <class T, class... Args, class MakeStatusCodeResult =
    typename detail::safe_get_make_status_code_result<T, Args...>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
  requires(!std::is_same<typename std::decay<T>::type, status_code>::value // not copy/move of self
           && !std::is_same<typename std::decay<T>::type, in_place_t>::value     // not in_place_t
           && is_status_code<MakeStatusCodeResult>::value                        // ADL makes a status
                code
           && std::is_constructible<status_code, MakeStatusCodeResult>::value))  // ADLed status code
               is compatible
  )
  constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(),
      std::declval<Args>()...)));

  //! Implicit construction from any `quick_status_code_from_enum<Enum>` enumerated type.
  template<class Enum, class QuickStatusCodeType
    = typename quick_status_code_from_enum<Enum>::code_type>         // Enumeration has been activated
  requires(std::is_constructible<status_code, QuickStatusCodeType>::value)      // Its status code is
      compatible
  constexpr status_code(Enum &&v) noexcept(std::is_nothrow_constructible<status_code,
      QuickStatusCodeType>::value);

  //! Explicit in-place construction.
  template <class... Args>
  constexpr explicit status_code(in_place_t /*unused */, Args &&... args) noexcept(std::
      is_nothrow_constructible<value_type, Args &&...>::value);

  //! Explicit in-place construction from initialiser list.
  template <class T, class... Args>
  constexpr explicit status_code(in_place_t /*unused */, std::initializer_list<T> il, Args &&... args)
       noexcept(std::is_nothrow_constructible<value_type, std::initializer_list<T>, Args &&...>::
      value);

  //! Explicit copy construction from a `value_type`.
  constexpr explicit status_code(const value_type &v) noexcept(std::is_nothrow_copy_constructible<
      value_type>::value);

  //! Explicit move construction from a `value_type`.
  constexpr explicit status_code(value_type &&v) noexcept(std::is_nothrow_move_constructible<
      value_type>::value);

  /*! Explicit construction from an erased status code. Available only if
  `value_type` is trivially destructible and `sizeof(status_code) <= sizeof(status_code<erased<>>)`.
  Does not check if domains are equal.
  */
  template <class ErasedType>
  requires(detail::domain_value_type_erasure_is_safe<domain_type, erased<ErasedType>>::value)
  constexpr explicit status_code(const status_code<erased<ErasedType>> &v) noexcept(std::
      is_nothrow_copy_constructible<value_type>::value);

  //! Return a reference to a string textually representing a code.
  constexpr string_ref message() const noexcept;
```

```
};
```

4.6 status_code<erased<TRIVIALLY_COPYABLE_OR_MOVE_BITCOPYING_TYPE>>

```
/*! Type erased, move-only status_code, unlike ‘status_code<void>‘ which cannot be moved nor
destroyed. Available only if ‘erased<>‘ is available, which is when the domain’s type is trivially
copyable or is move bitcopying, and if the size of the domain’s typed error code is less than
or equal to this erased error code. Copy construction is disabled, but if you want a copy call
‘.clone()‘.

An ADL discovered helper function ‘make_status_code(T, Args...)‘ is looked up by one of the
constructors. If it is found, and it generates a status code compatible with this status code,
implicit construction is made available.
*/
template <class ErasedType> class status_code<erased<ErasedType>>
  : public mixins::mixin<detail::status_code_storage<erased<ErasedType>>, erased<ErasedType>>
{
  template <class T> friend class status_code;

public:
  //! The type of the domain (void, as it is erased).
  using domain_type = void;
  //! The type of the erased status code.
  using value_type = ErasedType;
  //! The type of a reference to a message string.
  using string_ref = typename _status_code<void>::string_ref;

public:
  //! Default construction to empty
  status_code() = default;
  //! Copy constructor
  status_code(const status_code &) = delete;
  //! Move constructor
  status_code(status_code &&) = default;
  //! Copy assignment
  status_code &operator=(const status_code &) = delete;
  //! Move assignment
  status_code &operator=(status_code &&) = default;
  constexpr ~status_code();

  //! Return a copy of the erased code by asking the domain to perform the erased copy.
  constexpr status_code clone() const;

  //! Implicit copy construction from any other status code if its value type is
  //! trivially copyable, it would fit into our storage, and it is not an erased
  //! status code.
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value
          && !detail::is_erased_status_code<status_code<typename std::decay<DomainType>::type>>::value
              )
  constexpr status_code(const status_code<DomainType> &v) noexcept;

  //! Implicit move construction from any other status code if its value type is trivially copyable
  //! or move relocating and it would fit into our storage
```

```
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value)
  constexpr status_code(status_code<DomainType> &&v) noexcept;

  //! Implicit construction from any type where an ADL discovered `make_status_code(T, Args ...)`
  //! returns a `status_code`.
  template <class T, class... Args,
            class MakeStatusCodeResult =
    typename detail::safe_get_make_status_code_result<T, Args...>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
  requires(!std::is_same<typename std::decay<T>::type, status_code>::value // not copy/move of self
          && !std::is_same<typename std::decay<T>::type, value_type>::value     // not copy/move of
              value type
          && is_status_code<MakeStatusCodeResult>::value                        // ADL makes a status
              code
          && std::is_constructible<status_code, MakeStatusCodeResult>::value))  // ADLed status code
              is compatible
  )
  constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(),
      std::declval<Args>()...)));

  //! Implicit construction from any `quick_status_code_from_enum<Enum>` enumerated type.
  template<class Enum,
           class QuickStatusCodeType = typename quick_status_code_from_enum<Enum>::code_type)
               // Enumeration has been activated
  requires(std::is_constructible<status_code, QuickStatusCodeType>::value)      // Its status code is
      compatible
  constexpr status_code(Enum &&v) noexcept(std::is_nothrow_constructible<status_code,
      QuickStatusCodeType>::value);

  //! Explicit copy construction from an unknown status code. Note that this will throw
  //! an exception if its value type is not trivially copyable or would not
  //! fit into our storage or the source domain's `_do_erased_copy()` refused the copy.
  //! This function is not present if C++ exceptions are globally disabled.
  explicit constexpr status_code(const status_code<void> &v);

  //! Tagged copy construction from an unknown status code. Note that this will be empty
  //! if its value type is not trivially copyable or would not fit into our
  //! storage or the source domain's `_do_erased_copy()` refused the copy.
  constexpr status_code(std::nothrow_t, const status_code<void> &v) noexcept
};
```

4.7 Status code comparisons

```
//! True if the status code's are semantically equal via `equivalent()`.
template <class DomainType1, class DomainType2>
constexpr bool operator==(const status_code<DomainType1> &a, const status_code<DomainType2> &b)
    noexcept;

//! True if the status code's are not semantically equal via `equivalent()`.
template <class DomainType1, class DomainType2>
constexpr bool operator!=(const status_code<DomainType1> &a, const status_code<DomainType2> &b)
    noexcept;

```

```
//! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a status code
constexpr bool operator==(const status_code<DomainType1> &a, const T &b);

//! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a status code
constexpr bool operator==(const T &a, const status_code<DomainType1> &b);

//! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a status code
constexpr bool operator!=(const status_code<DomainType1> &a, const T &b);

//! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a status code
constexpr bool operator!=(const T &a, const status_code<DomainType1> &b);

//! True if the status code's are semantically equal via 'equivalent()' to 'quick_status_code_from_enum<T>::code_type(b)'.
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator==(const status_code<DomainType1> &a, const T &b);

//! True if the status code's are semantically equal via 'equivalent()' to 'quick_status_code_from_enum<T>::code_type(a)'.
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator==(const T &a, const status_code<DomainType1> &b);

//! True if the status code's are not semantically equal via 'equivalent()' to 'quick_status_code_from_enum<T>::code_type(b)'.
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const status_code<DomainType1> &a, const T &b);

//! True if the status code's are not semantically equal via 'equivalent()' to 'quick_status_code_from_enum<T>::code_type(a)'.
template <class DomainType1, class T, class QuickStatusCodeType
```

```
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const T &a, const status_code<DomainType1> &b);
```

4.8 Exception types

```
/*! Exception type representing a thrown status_code
*/
template <class DomainType> class status_error;

/*! The erased type edition of status_error.
*/
template <> class status_error<void> : public std::exception
{
protected:
  //! Constructs an instance. Not publicly available.
  status_error() = default;
  //! Copy constructor. Not publicly available
  status_error(const status_error &) = default;
  //! Move constructor. Not publicly available
  status_error(status_error &&) = default;
  //! Copy assignment. Not publicly available
  status_error &operator=(const status_error &) = default;
  //! Move assignment. Not publicly available
  status_error &operator=(status_error &&) = default;
  //! Destructor. Not publicly available.
  ~status_error() override = default;

  virtual const status_code<void> &do_code() const noexcept = 0;

public:
  //! The type of the status domain
  using domain_type = void;
  //! The type of the status code
  using status_code_type = status_code<void>;

public:
  //! The erased status code which generated this exception instance.
  const status_code<void> &code() const noexcept { return do_code(); }
};

/*! Exception type representing a thrown status_code
*/
template <class DomainType> class status_error : public status_error<void>
{
  status_code<DomainType> _code;
  typename DomainType::string_ref _msgref;

  virtual const status_code<void> &_do_code() const noexcept override final { return _code; }

public:
  //! The type of the status domain
  using domain_type = DomainType;
  //! The type of the status code
```

```
  using status_code_type = status_code<DomainType>;

  //! Constructs an instance
  explicit status_error(status_code<DomainType> code);

  //! Return an explanatory string
  virtual const char *what() const noexcept override;

  //! Returns a reference to the code
  const status_code_type &code() const &;
  //! Returns a reference to the code
  status_code_type &code() &;
  //! Returns a reference to the code
  const status_code_type &&code() const &&;
  //! Returns a reference to the code
  status_code_type &&code() &&;
};
```

4.9 Generic error coding

```
//! The generic error coding (POSIX)
enum class errc : int
{
  success = 0,    //!< This is new over std::errc
  unknown = -1,   //!< This is new over std::errc

  address_family_not_supported = EAFNOSUPPORT,
  address_in_use = EADDRINUSE,
  address_not_available = EADDRNOTAVAIL,
  already_connected = EISCONN,
  argument_list_too_long = E2BIG,
  argument_out_of_domain = EDOM,
  bad_address = EFAULT,
  bad_file_descriptor = EBADF,
  bad_message = EBADMSG,
  broken_pipe = EPIPE,
  connection_aborted = ECONNABORTED,
  connection_already_in_progress = EALREADY,
  connection_refused = ECONNREFUSED,
  connection_reset = ECONNRESET,
  cross_device_link = EXDEV,
  destination_address_required = EDESTADDRREQ,
  device_or_resource_busy = EBUSY,
  directory_not_empty = ENOTEMPTY,
  executable_format_error = ENOEXEC,
  file_exists = EEXIST,
  file_too_large = EFBIG,
  filename_too_long = ENAMETOOLONG,
  function_not_supported = ENOSYS,
  host_unreachable = EHOSTUNREACH,
  identifier_removed = EIDRM,
  illegal_byte_sequence = EILSEQ,
  inappropriate_io_control_operation = ENOTTY,
  interrupted = EINTR,
```

```
  invalid_argument = EINVAL,
  invalid_seek = ESPIPE,
  io_error = EIO,
  is_a_directory = EISDIR,
  message_size = EMSGSIZE,
  network_down = ENETDOWN,
  network_reset = ENETRESET,
  network_unreachable = ENETUNREACH,
  no_buffer_space = ENOBUFS,
  no_child_process = ECHILD,
  no_link = ENOLINK,
  no_lock_available = ENOLCK,
  no_message = ENOMSG,
  no_protocol_option = ENOPROTOOPT,
  no_space_on_device = ENOSPC,
  no_stream_resources = ENOSR,
  no_such_device_or_address = ENXIO,
  no_such_device = ENODEV,
  no_such_file_or_directory = ENOENT,
  no_such_process = ESRCH,
  not_a_directory = ENOTDIR,
  not_a_socket = ENOTSOCK,
  not_a_stream = ENOSTR,
  not_connected = ENOTCONN,
  not_enough_memory = ENOMEM,
  not_supported = ENOTSUP,
  operation_cancelled = ECANCELED,
  operation_in_progress = EINPROGRESS,
  operation_not_permitted = EPERM,
  operation_not_supported = EOPNOTSUPP,
  operation_would_block = EWOULDBLOCK,
  owner_dead = EOWNERDEAD,
  permission_denied = EACCES,
  protocol_error = EPROTO,
  protocol_not_supported = EPROTONOSUPPORT,
  read_only_file_system = EROFS,
  resource_deadlock_would_occur = EDEADLK,
  resource_unavailable_try_again = EAGAIN,
  result_out_of_range = ERANGE,
  state_not_recoverable = ENOTRECOVERABLE,
  stream_timeout = ETIME,
  text_file_busy = ETXTBSY,
  timed_out = ETIMEDOUT,
  too_many_files_open_in_system = ENFILE,
  too_many_files_open = EMFILE,
  too_many_links = EMLINK,
  too_many_symbolic_link_levels = ELOOP,
  value_too_large = EOVERFLOW,
  wrong_protocol_type = EPROTOTYPE
};

//! A specialisation of `status_error` for the generic code domain.
using generic_error = status_error<_generic_code_domain>;
//! A constexpr source variable for the generic code domain, which is that of `errc`
//! (POSIX). Returned by `_generic_code_domain::get()`.
constexpr _generic_code_domain generic_code_domain;
```

```
// Enable implicit construction of generic_code from errc
constexpr inline generic_code make_status_code(errc c) noexcept;
```

4.10 errored_status_code<DomainType>

```
/*! A ‘status_code‘ which is always a failure. The closest equivalent to
‘std::error_code‘, except it cannot be modified, and is templated.

Differences from ‘status_code‘:

- Never successful (this contract is checked on construction, if fails then it
terminates the process).
- Is immutable.
*/
template <class DomainType> class errored_status_code : public status_code<DomainType>
{
  using _base = status_code<DomainType>;
  using _base::clear;
  using _base::success;

public:
  //! The type of the errored error code.
  using typename _base::value_type;
  //! The type of a reference to a message string.
  using typename _base::string_ref;

  //! Default constructor.
  errored_status_code() = default;
  //! Copy constructor.
  errored_status_code(const errored_status_code &) = default;
  //! Move constructor.
  errored_status_code(errored_status_code &&) = default;
  //! Copy assignment.
  errored_status_code &operator=(const errored_status_code &) = default;
  //! Move assignment.
  errored_status_code &operator=(errored_status_code &&) = default;
  ~errored_status_code() = default;

  //! Explicitly construct from any similar status code
  constexpr explicit errored_status_code(const _base &o)
    noexcept(std::is_nothrow_copy_constructible<_base>::value)
    [[expects: o.failure() == true]];
  //! Explicitly construct from any similar status code
  constexpr explicit errored_status_code(_base &&o)
    noexcept(std::is_nothrow_move_constructible<_base>::value)
    [[expects: o.failure() == true]];

  //! Implicit construction from any type where an ADL discovered
  //! ‘make_status_code(T, Args ...)‘ returns a ‘status_code‘.
  template <class T, class... Args, class MakeStatusCodeResult
    = typename detail::safe_get_make_status_code_result<T, Args...>::type>  // Safe ADL lookup of
       make_status_code(), returns void if not found
  requires(!std::is_same<typename std::decay<T>::type, errored_status_code>::value // not copy/move of
       self
```

```
48          && !std::is_same<typename std::decay<T>::type, in_place_t>::value           // not in_place_t
49          && is_status_code<MakeStatusCodeResult>::value                              // ADL makes a status code
50          && std::is_constructible<errored_status_code, MakeStatusCodeResult>::value  // ADLed status code is compatible
51  )
52  constexpr errored_status_code(T &&v, Args &&... args)
53    noexcept(noexcept(make_status_code(std::declval<T>(), std::declval<Args>()...)))
54    [[expects: make_status_code(std::forward<T>(v) /* unsafe? */, std::forward<Args>(args)...).failure() == true]];
55
56  //! Implicit construction from any `quick_status_code_from_enum<Enum>` enumerated type.
57  template<class Enum, class QuickStatusCodeType
58    = typename quick_status_code_from_enum<Enum>::code_type>              // Enumeration has been activated
59  requires(std::is_constructible<errored_status_code, QuickStatusCodeType>::value)    // Its status code is compatible
60  constexpr errored_status_code(Enum &&v)
61    noexcept(std::is_nothrow_constructible<errored_status_code, QuickStatusCodeType>::value)
62    [[expects: errored_status_code(QuickStatusCodeType(static_cast<Enum &&>(v))).failure() == true]];
63
64  //! Explicit in-place construction.
65  template <class... Args>
66  constexpr explicit errored_status_code(in_place_t /*unused */, Args &&... args)
67    noexcept(std::is_nothrow_constructible<value_type, Args &&...>::value)
68    [[expects: _base(std::forward<Args>(args)... /* unsafe? */).failure() == true]];
69
70  //! Explicit in-place construction from initialiser list.
71  template <class T, class... Args>
72  constexpr explicit errored_status_code(in_place_t /*unused */, std::initializer_list<T> il, Args &&... args)
73    noexcept(std::is_nothrow_constructible<value_type, std::initializer_list<T>, Args &&...>::value)
74    [[expects: _base(il, std::forward<Args>(args)... /* unsafe? */).failure() == true]];
75
76  //! Explicit copy construction from a `value_type`.
77  constexpr explicit errored_status_code(const value_type &v)
78    noexcept(std::is_nothrow_copy_constructible<value_type>::value)
79    [[expects: _base(v).failure() == true]];
80
81  //! Explicit move construction from a `value_type`.
82  constexpr explicit errored_status_code(value_type &&v)
83    noexcept(std::is_nothrow_move_constructible<value_type>::value)
84    [[expects: _base(std::move(v) /* unsafe? */.failure() == true]];
85
86  /*! Explicit construction from an erased status code. Available only if
87  `value_type` is trivially destructible and `sizeof(status_code) <= sizeof(status_code<erased<>>)`.
88  Does not check if domains are equal.
89  */
90  template <class ErasedType>
91  requires(detail::domain_value_type_erasure_is_safe<domain_type, erased<ErasedType>>::value)
92  constexpr explicit errored_status_code(const status_code<erased<ErasedType>> &v)
93    noexcept(std::is_nothrow_copy_constructible<value_type>::value)
94    [[expects: v.failure() == true]];
95
96  //! Always false (including at compile time), as errored status codes are never successful.
```

```
  constexpr bool success() const noexcept { return false; }
  //! Return a const reference to the `value_type`.
  constexpr const value_type &value() const &noexcept;
};
```

4.11 errored_status_code<erased<TRIVIALLY_COPYABLE_OR_MOVE_BITCOPYING_TYPE>>

```
template <class ErasedType> class errored_status_code<erased<ErasedType>>
  : public status_code<erased<ErasedType>>
{
  using _base = status_code<erased<ErasedType>>;
  using _base::success;

public:
  using domain_type = typename _base::domain_type;
  using value_type = typename _base::value_type;
  using string_ref = typename _base::string_ref;

  //! Default construction to empty
  errored_status_code() = default;
  //! Copy constructor
  errored_status_code(const errored_status_code &) = default;
  //! Move constructor
  errored_status_code(errored_status_code &&) = default;
  //! Copy assignment
  errored_status_code &operator=(const errored_status_code &) = default;
  //! Move assignment
  errored_status_code &operator=(errored_status_code &&) = default;
  ~errored_status_code() = default;

  //! Explicitly construct from any similarly erased status code
  constexpr explicit errored_status_code(const _base &o)
    noexcept(std::is_nothrow_copy_constructible<_base>::value)
    [[expects: o.failure() == true]];
  //! Explicitly construct from any similarly erased status code
  constexpr explicit errored_status_code(_base &&o)
    noexcept(std::is_nothrow_move_constructible<_base>::value)
    [[expects: o.failure() == true]];

  //! Implicit copy construction from any other status code if its value type is
  //! trivially copyable, it would fit into our storage, and it is not an erased
  //! status code.
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value
          && !detail::is_erased_status_code<status_code<typename std::decay<DomainType>::type>>::value
              )
  constexpr errored_status_code(const status_code<DomainType> &v) noexcept
    [[expects: v.failure() == true]];

  //! Implicit copy construction from any other status code if its value type is
  //! trivially copyable, it would fit into our storage, and it is not an erased
  //! status code.
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value
```

```
        && !detail::is_erased_status_code<status_code<typename std::decay<DomainType>::type>>::value)
  constexpr errored_status_code(const errored_status_code<DomainType> &v) noexcept;

  //! Implicit move construction from any other status code if its value type is trivially copyable
  //! or move relocating and it would fit into our storage
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value)
  constexpr errored_status_code(status_code<DomainType> &&v) noexcept
    [[expects: v.failure() == true]];

  //! Implicit move construction from any other status code if its value type is trivially copyable
  //! or move relocating and it would fit into our storage
  template <class DomainType>
  requires(detail::domain_value_type_erasure_is_safe<erased<ErasedType>, DomainType>::value)
  constexpr errored_status_code(errored_status_code<DomainType> &&v) noexcept;

  //! Implicit construction from any type where an ADL discovered `make_status_code(T, Args ...)`
  //! returns a `status_code`.
  template <class T, class... Args,
            class MakeStatusCodeResult =
    typename detail::safe_get_make_status_code_result<T, Args...>::type>  // Safe ADL lookup of make_status_code(), returns void if not found
  requires(!std::is_same<typename std::decay<T>::type, errored_status_code>::value // not copy/move of self
           && !std::is_same<typename std::decay<T>::type, value_type>::value        // not copy/move of value type
           && is_status_code<MakeStatusCodeResult>::value                           // ADL makes a status code
           && std::is_constructible<errored_status_code, MakeStatusCodeResult>::value))  // ADLed status code is compatible
  constexpr errored_status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(), std::declval<Args>()...))) [[expects: make_status_code(std::forward<T>(v) /* unsafe? */, std::forward<Args>(args)...).failure() == true]];

  //! Implicit construction from any `quick_status_code_from_enum<Enum>` enumerated type.
  template<class Enum,
           class QuickStatusCodeType = typename quick_status_code_from_enum<Enum>::code_type)
               // Enumeration has been activated
  requires(std::is_constructible<errored_status_code, QuickStatusCodeType>::value)      // Its status code is compatible
  constexpr errored_status_code(Enum &&v) noexcept(std::is_nothrow_constructible<status_code, QuickStatusCodeType>::value);

  //! Explicit copy construction from an unknown status code. Note that this will throw
  //! an exception if its value type is not trivially copyable or would not
  //! fit into our storage or the source domain's `_do_erased_copy()` refused the copy.
  //! This function is not present if C++ exceptions are globally disabled.
  explicit constexpr errored_status_code(const status_code<void> &v);

  // errored_status_code(std::nothrow_t, const status_code<void> &v) is deliberately omitted,
  // as empty errored_status_code's are not possible due to contract violation. One can
  // use status_code's nothrow constructor, do a runtime check for emptiness, then implicitly
  // construct an errored_status_code from that.

```

```
  //! Always false (including at compile time), as errored status codes are never successful.
  constexpr bool success() const noexcept { return false; }
  //! Return the erased 'value_type' by value.
  constexpr value_type value() const noexcept;
};
```

4.12 Errored status code comparisons

```
//! True if the status code's are semantically equal via 'equivalent()'.
template <class DomainType1, class DomainType2>
constexpr bool operator==(const status_code<DomainType1> &a, const status_code<DomainType2> &b)
    noexcept;
template <class DomainType1, class DomainType2>
constexpr bool operator==(const status_code<DomainType1> &a, const errored_status_code<DomainType2> &b
    ) noexcept;
template <class DomainType1, class DomainType2>
constexpr bool operator==(const errored_status_code<DomainType1> &a, const status_code<DomainType2> &b
    ) noexcept;

//! True if the status code's are not semantically equal via 'equivalent()'.
template <class DomainType1, class DomainType2>
constexpr bool operator!=(const status_code<DomainType1> &a, const status_code<DomainType2> &b)
    noexcept;
template <class DomainType1, class DomainType2>
constexpr bool operator!=(const status_code<DomainType1> &a, const errored_status_code<DomainType2> &b
    ) noexcept;
template <class DomainType1, class DomainType2>
constexpr bool operator!=(const errored_status_code<DomainType1> &a, const status_code<DomainType2> &b
    ) noexcept;

//! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
      make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                // ADL makes a
    status code
constexpr bool operator==(const status_code<DomainType1> &a, const T &b);
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
      make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                // ADL makes a
    status code
constexpr bool operator==(const errored_status_code<DomainType1> &a, const T &b);

//! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
      make_status_code(), returns void if not found
requires(is_status_code<MakeStatusCodeResult>::value)                                // ADL makes a
    status code
constexpr bool operator==(const T &a, const status_code<DomainType1> &b);
template<class DomainType1, class T, class MakeStatusCodeResult
  = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
      make_status_code(), returns void if not found
```

```
34  requires(is_status_code<MakeStatusCodeResult>::value)                                  // ADL makes a
        status code
35  constexpr bool operator==(const T &a, const errored_status_code<DomainType1> &b);
36
37  //! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)'.
38  template<class DomainType1, class T, class MakeStatusCodeResult
39    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
40  requires(is_status_code<MakeStatusCodeResult>::value)                                  // ADL makes a
        status code
41  constexpr bool operator!=(const status_code<DomainType1> &a, const T &b);
42  template<class DomainType1, class T, class MakeStatusCodeResult
43    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
44  requires(is_status_code<MakeStatusCodeResult>::value)                                  // ADL makes a
        status code
45  constexpr bool operator!=(const errored_status_code<DomainType1> &a, const T &b);
46
47  //! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)'.
48  template<class DomainType1, class T, class MakeStatusCodeResult
49    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
50  requires(is_status_code<MakeStatusCodeResult>::value)                                  // ADL makes a
        status code
51  constexpr bool operator!=(const T &a, const status_code<DomainType1> &b);
52  template<class DomainType1, class T, class MakeStatusCodeResult
53    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
        make_status_code(), returns void if not found
54  requires(is_status_code<MakeStatusCodeResult>::value)                                  // ADL makes a
        status code
55  constexpr bool operator!=(const T &a, const errored_status_code<DomainType1> &b);
56
57  //! True if the status code's are semantically equal via 'equivalent()' to '
        quick_status_code_from_enum<T>::code_type(b)'.
58  template <class DomainType1, class T, class QuickStatusCodeType
59    = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
60  >
61  constexpr bool operator==(const status_code<DomainType1> &a, const T &b);
62  template <class DomainType1, class T, class QuickStatusCodeType
63    = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
64  >
65  constexpr bool operator==(const errored_status_code<DomainType1> &a, const T &b);
66
67  //! True if the status code's are semantically equal via 'equivalent()' to '
        quick_status_code_from_enum<T>::code_type(a)'.
68  template <class DomainType1, class T, class QuickStatusCodeType
69    = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
70  >
71  constexpr bool operator==(const T &a, const status_code<DomainType1> &b);
72  template <class DomainType1, class T, class QuickStatusCodeType
73    = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
74  >
75  constexpr bool operator==(const T &a, const errored_status_code<DomainType1> &b);
76
77  //! True if the status code's are not semantically equal via 'equivalent()' to '
        quick_status_code_from_enum<T>::code_type(b)'.
```

```cpp
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const status_code<DomainType1> &a, const T &b);
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const errored_status_code<DomainType1> &a, const T &b);

//! True if the status code's are not semantically equal via 'equivalent()' to '
    quick_status_code_from_enum<T>::code_type(a)'.
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const T &a, const status_code<DomainType1> &b);
template <class DomainType1, class T, class QuickStatusCodeType
  = typename quick_status_code_from_enum<T>::code_type  // Enumeration has been activated
>
constexpr bool operator!=(const T &a, const errored_status_code<DomainType1> &b);
```

4.13 Quick declaration of a new status code domain

```cpp
//! A status code wrapping 'Enum' generated from 'quick_status_code_from_enum'.
template <class Enum>
using quick_status_code_from_enum_code = status_code<_quick_status_code_from_enum_domain<Enum>>;

//! Defaults for an implementation of 'quick_status_code_from_enum<Enum>'
template <class Enum>
struct quick_status_code_from_enum_defaults
{
  //! The type of the resulting code
  using code_type = quick_status_code_from_enum_code<Enum>;
  //! Used within 'quick_status_code_from_enum' to define a mapping of enumeration value with its
      status code
  struct mapping
  {
    //! The enumeration type
    using enumeration_type = Enum;

    //! The value being mapped
    const Enum value;
    //! A string representation for this enumeration value
    const char *message;
    //! A list of 'errc' equivalents for this enumeration value
    const std::initializer_list<errc> code_mappings;
  };
  //! Used within 'quick_status_code_from_enum' to define mixins for the status code wrapping 'Enum'
  template <class Base> struct mixin : Base
  {
    using Base::Base;
  };
};

template <class Enum>
```

```
constexpr _quick_status_code_from_enum_domain<Enum> quick_status_code_from_enum_domain = {};

namespace mixins
{
  template <class Base, class Enum> struct mixin<Base, _quick_status_code_from_enum_domain<Enum>> :
      public quick_status_code_from_enum<Enum>::template mixin<Base>
  {
    using quick_status_code_from_enum<Enum>::template mixin<Base>::mixin;
  };
}  // namespace mixins
```

4.14 OS specific and common codes

```
// Lets you retrieve the 'std::error_category' from a 'std_error_code'
namespace mixins
{
  template <class Base> struct mixin<Base, _std_error_code_domain> : public Base
  {
    using Base::Base;

    //! Implicit constructor from a 'std::error_code'
    inline mixin(std::error_code ec);

    //! Returns the error code category
    inline const std::error_category &category() const noexcept;
  };
}  // namespace mixins

//! A wrapper of 'std::error_code'. This passes through the original
//! error code's category during comparisons. This can't be an
//! 'errored_status_code', as 'std::error_code' can be null.
using std_error_code = status_code<_error_code_domain<std::error_code, detail::make_std_categories>>;


// Provides static function 'posix_code::current()'
namespace mixins
{
  template <class Base> struct mixin<Base, _posix_code_domain> : public Base
  {
    using Base::Base;

    //! Returns a 'posix_code' for the current value of 'errno'.
    static posix_code current() noexcept;
  };
}  // namespace mixins

//! A POSIX error code, those returned by 'errno'.
using posix_code = status_code<_posix_code_domain>;
//! A specialisation of 'errored_status_code' for the POSIX error code domain.
using posix_error = errored_status_code<_posix_code_domain>;


// Lets you question a 'http_status_code' for its characteristics
namespace mixins
```

```
{
  template <class Base> struct mixin<Base, _http_status_code_domain> : public Base
  {
    using Base::Base;

    //! True if the HTTP status code is informational
    inline bool is_http_informational() const noexcept;
    //! True if the HTTP status code is successful
    inline bool is_http_success() const noexcept;
    //! True if the HTTP status code is redirection
    inline bool is_http_redirection() const noexcept;
    //! True if the HTTP status code is client error
    inline bool is_http_client_error() const noexcept;
    //! True if the HTTP status code is server error
    inline bool is_http_server_error() const noexcept;
  };
}  // namespace mixins

//! A HTTP status code.
using http_status_code = status_code<_http_status_code_domain>;
//! A HTTP status code.
using http_status_error = errored_status_code<_http_status_code_domain>;


//! A getaddrinfo error code, those returned by `getaddrinfo()`.
using getaddrinfo_code = status_code<_getaddrinfo_code_domain>;
//! A specialisation of `errored_status_code` for the getaddrinfo code domain.
using getaddrinfo_error = errored_status_code<_getaddrinfo_code_domain>;


#ifdef _WIN32
// Provides static function `win32_code::current()`
namespace mixins
{
  template <class Base> struct mixin<Base, _win32_code_domain> : public Base
  {
    using Base::Base;

    //! Returns a `win32_code` for the current value of `GetLastError()`.
    static inline win32_code current() noexcept;
  };
}  // namespace mixins

//! (Windows only) A Win32 error code, those returned by `GetLastError()`.
using win32_code = status_code<_win32_code_domain>;
//! (Windows only) A specialisation of `errored_status_code` for the Win32 error code domain.
using win32_error = errored_status_code<_win32_code_domain>;


//! (Windows only) A NT error code, those returned by NT kernel functions.
using nt_code = status_code<_nt_code_domain>;
//! (Windows only) A specialisation of `errored_status_code` for the NT error code domain.
using nt_error = errored_status_code<_nt_code_domain>;


/*! (Windows only) A COM error code. Note semantic equivalence testing is only
```

```
implemented for ‘FACILITY_WIN32‘ and ‘FACILITY_NT_BIT‘. As you can see at
[https://blogs.msdn.microsoft.com/eldar/2007/04/03/a-lot-of-hresult-codes/](https://blogs.msdn.
    microsoft.com/eldar/2007/04/03/a-lot-of-hresult-codes/),
there are an awful lot of COM error codes, and keeping mapping tables for all of
them would be impractical (for the Win32 and NT facilities, we actually reuse the
mapping tables in ‘win32_code‘ and ‘nt_code‘). You can, of course, inherit your
own COM code domain from this one and override the ‘_equivalent()‘ function
to add semantic equivalence testing for whichever extra COM codes that your
application specifically needs.
*/
using com_code = status_code<_com_code_domain>;
//! (Windows only) A specialisation of ‘errored_status_code‘ for the COM error code domain.
using com_error = errored_status_code<_com_code_domain>;
#endif  // _WIN32
```

4.15 Erased system code, and proposed `std::error` object

```
/*! An erased-mutable status code suitably large for all the system codes
which can be returned on this system.

For Windows, these might be:

    - ‘com_code‘ (‘HRESULT‘)  [you need to include "com_code.hpp" explicitly for this]
    - ‘nt_code‘ (‘LONG‘)
    - ‘win32_code‘ (‘DWORD‘)

For POSIX, ‘posix_code‘ and ‘getaddrinfo_code‘ is possible.

You are guaranteed that ‘system_code‘ can be transported by the compiler
in exactly two CPU registers.
*/
using system_code = status_code<erased<intptr_t>>;

/*! A utility function which returns the closest matching system_code to a supplied
exception ptr.
*/
inline system_code system_code_from_exception(std::exception_ptr &&ep = std::current_exception(),
  system_code not_matched = generic_code(errc::resource_unavailable_try_again)) noexcept;


/*! An errored ‘system_code‘ which is always a failure. The closest equivalent to
‘std::error_code‘, except it cannot be null and cannot be modified.

This refines ‘system_code‘ into an ‘error‘ object meeting the requirements of
[P0709 Zero-overhead deterministic exceptions](https://wg21.link/P0709).

Differences from ‘system_code‘:

- Always a failure (this is checked at construction, and if not the case,
the program is terminated as this is a logic error)
- Is immutable.

As with ‘system_code‘, it remains guaranteed to be two CPU registers in size,
and trivially copyable.
```

```
*/
using error = errored_status_code<erased<system_code::value_type>>;
```

4.16 iostream printing support

```
/*! Print the status code to a ‘std::ostream &‘.
Requires that ‘DomainType::value_type‘ implements an ‘operator<<‘ overload for ‘std::ostream‘.
*/
template <class DomainType>
requires(std::is_same<std::ostream, typename std::decay<decltype(std::declval<std::ostream>() << std::
    declval<typename status_code<DomainType>::value_type>())>::type>::value)
std::ostream &operator<<(std::ostream &s, const status_code<DomainType> &v);

/*! Print a status code domain’s ‘string_ref‘ to a ‘std::ostream &‘.
 */
std::ostream &operator<<(std::ostream &s, const status_code_domain::string_ref &v);

/*! Print the erased status code to a ‘std::ostream &‘.
*/
template <class ErasedType>
std::ostream &operator<<(std::ostream &s, const status_code<erased<ErasedType>> &v);

/*! Print the generic code to a ‘std::ostream &‘.
*/
std::ostream &operator<<(std::ostream &s, const generic_code &v);
```

4.17 status code ptr

```
/*! Make an erased status code which indirects to a dynamically allocated status code.
This is useful for shoehorning a rich status code with large value type into a small
erased status code like ‘system_code‘, with which the status code generated by this
function is compatible. Note that this function can throw due to ‘bad_alloc‘.
*/
template <class T>
requires(is_status_code<T>::value)
status_code<erased<typename std::add_pointer<typename std::decay<T>::type>::type>>
    make_status_code_ptr(T &&v);


/*! If a status code refers to a ‘status_code_ptr‘ which indirects to a status
code of type ‘StatusCode‘, return a pointer to that ‘StatusCode‘. Otherwise return null.
*/
template <class StatusCode, class U>
requires(is_status_code<StatusCode>::value>
StatusCode *get_if(status_code<erased<U>> *v) noexcept;

//! \overload Const overload
template <class StatusCode, class U>
requires(is_status_code<StatusCode>::value>
const StatusCode *get_if(const status_code<erased<U>> *v) noexcept;

```

```
23
24  /*! If a status code refers to a `status_code_ptr` which indirects to a status
25  code of type `StatusCode`, return a reference to that `StatusCode`. Otherwise throw
26  `bad_status_ptr_access`.
27  */
28  template <class StatusCode, class U>
29  requires(is_status_code<StatusCode>::value>
30  StatusCode &get(status_code<erased<U>> &v);
31
32  //! \overload Const overload
33  template <class StatusCode, class U>
34  requires(is_status_code<StatusCode>::value>
35  const StatusCode &get(const status_code<erased<U>> &v);
36
37
38  /*! If a status code refers to a `status_code_ptr`, return the id of the erased
39  status code's domain. Otherwise return a meaningless number.
40  */
41  template <class U>
42  typename status_code_domain::unique_id_type get_id(const status_code<erased<U>> &v) noexcept;
```

5 Design decisions, guidelines and rationale

These are copied from [P0824] *Summary of SG14 discussion on* `<system_error>` for your information.

5.1 Do not cause #include `<string>`

`<system_error>`, on all the major STL implementations, includes `<string>` as `std::error_code::message()`, amongst other facilities, returns a `std::string`. `std::string`, in turn, drags in the STL allocator machinery and a fair few algorithms and other headers.

Bringing in so much extra stuff is a showstopper for the use of `std::error_code` in the global APIs of very large C++ code bases due to the effects on build and link times. As much as C++ Modules may, or may not, fix this some day, adopting `std::error_code` – which is highly desirable to large C++ code bases which globally disable C++ exceptions such as games – is made impossible. Said users end up having to locally reinvent a near clone of `std::error_code`, but one which doesn't use `std::string`, which is unfortunate.

Moreover, because `<stdexcept>` must include `<system_error>`, and many otherwise very simple STL facilities such as `<array>`, `<complex>`, `<iterator>` or `<optional>` must include `<stdexcept>`, we end up dragging in `<string>` and the STL allocator machinery when including those otherwise simple and lightweight STL headers for no good purpose other than that `std::error_code::message()` returns a `std::string`! That deprives very large C++ code bases of being able to use `std::optional<T>` and other such vocabulary types in their global headers.

Hence, this implicit dependency of `<system_error>` on `<string>` contravenes [P2000]'s admonition *'Note that the cost of compilation is among the loudest reasonable complaints about C++ from its*

users'

It also breaks the request *'make C++ easier to use and more effective for large and small embedded systems'* by making a swathe of C++ library headers not [P0829] *Freestanding C++* compatible.

It is trivially easy to fix: stop using `std::string` to return textual representation of codes. This proposed design uses a `string_ref` instead, this is a potentially reference counted handle to a string. It is extremely lightweight, freestanding C++ compatible, and drags in no unnecessary headers.

5.2 All constexpr sourcing, construction and destruction

`<system_error>` was designed before constexpr entered the language, and many operations which ought to be constexpr for such a simple and low-level facility are not. Simple things like the `std::error_code` constructor is not constexpr, bigger things like `std::error_category` are not constexpr, and far more importantly the global source of error code categories is not constexpr, forcing the compiler to emit a magic static initialisation fence, which introduces significant added code bloat as magic fences cannot be elided by the optimiser.

The proposed replacement makes everything which can be constexpr be just that. If it cannot be constexpr, it is literal or trivial to the maximum extent possible. Empirical testing in real world code bases has found excellent effects on the density of assembler generated, with recent GCCs and clangs, almost all of the time the code generated with the replacement design is as optimal as a human assembler writer might write.

5.3 Header only libraries can now safely define custom code categories

Something probably unanticipated at the time of the design of `<system_error>` is that bespoke `std::error_category` implementations are unsafe in header only libraries. This has caused significant, and usually unpleasant, surprise in the C++ user base.

The problem stems from the comparison of `std::error_category` implementations which is *required* by the C++ standard to be a comparison of address of instance. When comparing an error code to an error condition, the `std::error_category::equivalent()` implementation compares the input error code's category against a list of error code categories known to it in order to decide upon equivalence. This is by address of instance.

Header only libraries must use Meyer singletons to implement the source of the custom `std::error_category` implementation i.e.

```
inline const my_custom_error_category &custom_category()
{
  static my_custom_error_category v;
  return v;
}
```

Ordinarily speaking, the linker would choose one of these inline function implementations, and thus `my_custom_error_category` gets exactly one instance, and thus one address in the final executable. All would therefore seem good.

Problems begin when a user uses the header only library inside a shared library. Now there is a single instance of the inline function *per shared library*, not per final executable. It is not uncommon for users to use more than one shared library, and thus multiple instances of the inline function come into existence. You now get the unpleasant situation where there are multiple singletons in the process, each with a different address, despite being the same error code category. Comparisons between error codes and categories thus subtly break in a somewhat chance based, hard to debug, way[1].

Those bitten by this 'feature' tend to be quite bitter about it. This author is one of those embittered. He has met others who have been similarly bitten through the use of ASIO and the Boost C++ Libraries. It's a niche problem, but one which consumes many days of very frustrating debugging for the uninitiated.

The proposed design makes error category sources all-constexpr as well as error code construction. This is incompatible with singletons, so the proposed design does away with the need for singleton sources entirely in favour of stateless code domains with a static random unique 64-bit id, of which there can be arbitrarily many instantiated at once, and thus the proposed design is safe for use in header only libraries.

In case there is concern of collision in a totally random unique 64 bit id, here are the number of random 64-bit numbers needed in the same process space for various probabilities of collision (note that 10e15 is the number of bits which a hard drive guarantees to return without mistake):

Probability of collision	10e-15	10e-12	10e-9	10e-6	10e-3 (0.1%)	10e-2 (1%)
Random 64-bit numbers needed	190	6100	190,000	6,100,000	190,000,000	610,000,000

5.4 No more `if(!ec)...`

`std::error_code` provides a boolean test. The correct definition for the meaning of the boolean test is 'is the value in this error code all bits zero, ignoring the category?'. It does **not** mean 'is there no error?'.

This may seem like an anodyne distinction, but it causes real confusion. During a discussion on the Boost C++ Libraries list regarding this issue, multiple opinions emerged over whether this was ambiguous, whether it would result in bugs, whether it was serious, whether programmers who wrote the code assuming the latter were the ones at fault, or whether it was the meaning of the boolean test. No resolution was found.

All this suggests to SG14 that there is unhelpful ambiguity which we believe can never lead to better quality software, so we have removed the boolean test in the proposed design. Developers must now be clear as to exactly what they mean: `if(ec.success())...`, `if(ec.failure())...` and so on.

[1] Do inline variables help? Unfortunately not. They suffer from the same problem of instance duplication when used in shared libraries. This is because standard C++ code has no awareness of shared libraries.

5.5 No more filtering codes returned by system APIs

Because `std::error_code` treats all bits zero values specially, and its boolean test does not consider category at all, when constructing error codes after a syscall, one must inevitably add some logic which performs a local check of whether the system returned code is a failure or not, and only then follow the error path.

This is fine for a lot of use cases, but many platforms, and indeed third party libraries, like to return success-with-information or success-with-warning codes. The current `<system_error>` does not address the possibility of multiple success codes being possible, nor that there is any success code other than all bits zero.

It also forces the program code which constructs the system code into an error code to be aware of implementation details of the source of the code in order to decide whether it is a failure or not. That is usually the case, but is not always the case. For where it is not the case, forcing this on users breaks clean encapsulation.

The proposed redesign accepts unfiltered and unmodified codes from any source. The category – called a *domain* in this proposal – interprets codes of any form of success or failure. Users can always safely construct a `status_code` (in this proposal, not [P0262]'s `status_value`) without knowing anything about the implementation details of its source. No one value is treated specially from any other.

5.6 All comparisons between codes are now semantic, not literal

Even some members of WG21 get the distinction between `std::error_code` and `std::error_condition` incorrect. That is because they appear to be almost the same thing, the same design, same categories, with only a vague documentation that one is to be used for system-specific codes and the other for non-system-specific codes.

This leads to an unnecessarily steep learning curve for the uninitiated, confusion amongst programmers reading code, incorrect choice of `std::error_condition` when `std::error_code` was meant, surprise when comparisons between codes and conditions are semantic not literal, and more of that general ambiguity and confusion we mentioned earlier.

The simple solution is to do away with all literal comparison entirely. Comparisons of `status_code` are **always** semantic. If the user really does want a literal comparison, they can manually compare domain and values by hand. Almost all of the time they actually want semantic comparison, and thus `operator ==`'s non-regular semantic comparison is exactly right.

5.7 `std::error_condition` is removed entirely

As comparisons are now always semantic between `status_code`'s, there is no longer any need for a distinction between `std::error_code` and `std::error_condition`. We therefore simplify the situation by removing any notion of `std::error_condition` altogether.

5.8 `status_code`'s value type is set by its domain

`std::error_code` hard codes its value to an `int`, which is problematic for third party error coding schemes which use a `long`, or even an `unsigned int`. `status_code<DomainType>` sets its `value_type` to be `DomainType::value_type`. Thus if you define your own domain type, its value type can be any type you like, including a structure or class.

This enables *payload* to be transmitted with your status code e.g. if the status code represents a failure in the filesystem, the payload might contain the path of a relevant file. It might contain the stack backtrace of where a failure or warning occurred, a `std::exception_ptr` instance, or anything else you might like.

We make great use of this domain definable value type facility to wrap up all possible `std::error_code`'s into status codes via a code domain whose value type is a `std::error_code`. This enables complete participation of any existing error code scheme within the proposed status code scheme.

5.9 `status_code<DomainType>` is type erasable

`status_code<DomainType>` can be type erased into a `status_code<void>` which is an immutable, unrelocatable, uncopyable type suitable for passing around by const lvalue reference only. This allows non-templated code to work with arbitrary, unknown, `status_code<DomainType>` instances. One may no longer retrieve their value obviously, but one can still query them for whether they represent success or failure, or for a textual message representing their value, and so on.

If, and only if, `DomainType::value_type` and some type `U` are `TriviallyCopyable` and the size of `DomainType::value_type` is less than or equal to size of `U`, an additional type erasure facility becomes available, that of `status_code<erased<U>>`. Unlike `status_code<void>`, this type erased form is copyable which is safe as `DomainType::value_type` and `U` are `TriviallyCopyable`, and are therefore both copyable as if via `memcpy()`.

This latter form of type erasure is particularly powerful. It allows one to define some global `status_code<erased<U>>` which is common to all code: `status_code<erased<intptr_t>>` would be a very portable choice[2]. Individual components may work in terms of `status_code<LocalErrorType>`, but all public facing APIs may return only the global `status_code<erased<intptr_t>>`. This facility thus allows any arbitrary `LocalErrorType` to be returned, unmodified, *with value semantics* through code which has no awareness of it. The only conditions are that `LocalErrorType` is trivially copyable, and is not bigger than the erased `intptr_t` type.

5.10 More than one 'system' error coding domain: `system_code`

`std::system_category` assumes that there is only one 'system' error coding, something not even true on POSIX (note that POSIX's error coding is always a subset of the POSIX implementation's

[2] Why? On x64 with SysV calling convention, a trivially copyable object no more than two CPU registers of size will be returned from functions via CPU registers, saving quite a few CPU cycles. AArch64 will return trivially copyable objects of up to 64 bytes via CPU registers!

error coding), let alone elsewhere, especially on Microsoft Windows where at least four primary system error coding schemes exist: (i) POSIX `errno` (ii) Win32 `GetLastError()` (iii) NT kernel `NTSTATUS` (iv) COM/WinRT/DirectX `HRESULT`.

The proposed library makes use of the `status_code<erased<U>>` facility described in the previous section to define a type alias `system_code` to a type erased status code sufficiently large enough to carry any of the system error codings on the current platform. This allows code to use the precise error code domain for the system failure in question, and to return it type erased in a form perfectly usable by external code, which need neither know nor care that the failure stemmed originally from COM, or Win32, or POSIX. All that matters is that the status code semantically compares true to say `std::errc::no_such_file_or_directory`.

5.11 `std::errc` gets its own code domain `generic_code`, eliminating `std::error_condition`

Similar, but orthogonal, to `system_code` is `generic_code` which has a value type of the strongly typed enum `std::errc`. Codes in the generic code domain become the 'portable error codes' formerly represented by `std::error_condition` in that they act as semantic comparator of last resort.

Generic codes allow one to write code which semantically compares success or failure to the standard failure reasons defined by POSIX. This allows one to write portable code which works independent of platform and implementation.

6 Technical specifications

No Technical Specifications are involved in this proposal.

7 Frequently asked questions

7.1 Implied in this design is that code domains must do nothing in their constructor and destructors, as multiple instances are permitted and both must be trivial and constexpr. How then can dynamic per-domain initialisation be performed e.g. setting up at run time a table of localised message strings?

The simplest is to use statically initialised local variables, though be aware that it is always legal to use status code from within static initialisation and finalisation, so you need to lazily construct any tables on first use and never deallocate. Slightly more complex is to use the domain's `string_ref` instances to keep a reference count of the use of the code domain, when all `string_ref` instances are destroyed, it is safe to deallocate any per-domain data.

7.2 Move only `std::error`?

For some reason people object to move-only `error`, which is the erased form of status code able to transport payloads of unknown type. This design decision needs to therefore be explained.

C++ is a strongly typed language, this allows its optimiser to make strong assumptions about code e.g. it can be copied. Erasing the type reduces the assumptions which can be made by the compiler, requiring decisions to be taken by run-time code instead. This is exactly the case with erased status code, which must be move only in case the erased type they transport is move only e.g. if the payload were a unique ptr, it needs to be move only. And a move only erased status code can transport copyable types just fine, but the compiler can't know that from the outside.

There is a `.clone()` member function on erased status codes. This asks the code's domain to clone the erased type, as it knows what that type and its payload is. It may refuse by throwing an exception. Because of the type erasure, this can never be a trivial operation, which is exactly why copy construction is disabled by default and you must explicitly opt into the cost of the clone via `.clone()`.

8 Addendum: Boost's `result<T>` type

I wish to state that I would **greatly** prefer if [P0709] *Zero overhead deterministic exceptions* were standardised instead of `result<T>`. It is superior in every way, and much less verbose and fiddly to work with, than functions returning Result types. We also get the ability to have constructors fail deterministically, instead of the usual two-stage static initialisation tricks, or having to use free functions as constructors.

But equally we cannot hold up standards proposals which require deterministic exceptions until they enter the language, which may be never. So I reluctantly propose a subset of `result<T>` derived from (Boost.)Outcome and Boost.System[3], hardcoded around `E` being always an `error`.

`result<T>` looks annoyingly different to `optional<T>` and `expected<T, error>` in that it has a `variant`-modelled constructor interface, and clearly separated wide and narrow contract observers like variant does. This is quite different to the `optional`-modelled constructor interface of Expected, which has mixed wide and narrow contract observers. Part of the difference stems from Result being designed after much use experience of Optional by the Boost developer community. Part of the difference stems from Variant by then entering the standard. And part of the difference is because of the hard coding of the alternative type to `error`. All this led to the eventual choosing of the presented design, instead of Expected's design, which occurred after one of the most voluminous peer reviews Boost has ever undertaken.

Outcome's result type never has a valueless state, however Boost.System's result *does* have a valueles state. It may, or may not, use union storage (Outcome's current implementation uses struct storage when `E` is `error`, as it has lower impact on compile times). Because type erased `error` is move only, `result<T>` is always move only (if you want a copy, call `.clone()`). Result has a bitcopying

[3]Since R3 of this paper, Boost.System has reimplemented this paper's proposed `result<T>`. See `https://www.boost.org/doc/libs/1_80_0/libs/system/doc/html/system.html#ref_resultt_e` .

move constructor if T is trivially copyable or move bitcopying (see [P1029] *move = bitcopies*). Comparisons deliberately exclude anything except equality, this prevents the perplexing surprise which otherwise occur with types which implicitly construct from values they can transport.

```
/*! \brief Exception type representing the failure to retrieve an error.
 */
class bad_result_access;

/*! \class result
\brief A `result<T>` type with its error type hardcoded to `error`.

This may, or may not, have union storage. It must never have a valueless state.
*/
template <class T>
requires(!std::is_reference_v<T> && !std::is_array_v<T> && !std::is_same_v<T, error>)
class result
{
public:
  //! The value type
  using value_type = T;
  //! The error type
  using error_type = error;

  //! Used to rebind result types
  template <class U> using rebind = result<U>;

public:
  //! Default constructor is disabled
  result() = delete;
  //! Copy constructor is disabled
  result(const result &) = delete;
  //! Move constructor. See P1029 move = bitcopies.
  result(result &&) = bitcopies(auto);
  //! Copy assignment is disabled
  result &operator=(const result &) = delete;
  //! Move assignment
  result &operator=(result &&) = default;
  //! Destructor
  ~result() = default;

  //! Implicit result converting move constructor
  template <class U>
  requires(std::is_convertible_v<U, T>)
  constexpr result(result<U> &&o) noexcept(std::is_nothrow_constructible_v<T, U>);

  //! Implicit result converting copy constructor
  template <class U>
  requires(std::is_convertible_v<U, T>)
  constexpr result(const result<U> &o) noexcept(std::is_nothrow_constructible_v<T, U>);

  //! Explicit result converting move constructor
  template <class U>
  requires(std::is_constructible_v<T, U>)
  constexpr explicit result(result<U> &&o) noexcept(std::is_nothrow_constructible_v<T, U>);

  //! Explicit result converting copy constructor
```

```
  template <class U>
  requires(std::is_constructible_v<T, U>)
  constexpr explicit result(const result<U> &o) noexcept(std::is_nothrow_constructible_v<T, U>);

  /* Slightly wider constructors than std::variant<error, T> would have. */

  //! Implicit in-place converting value constructor
  template<class Arg1, class... Args>
  requires(!(std::is_constructible_v<value_type, Arg1, Args...>
             && std::is_constructible_v<error_type, Arg1, Args...>)  //
    &&std::is_constructible_v<value_type, Arg1, Args...>)
  constexpr variant(Arg1 &&arg1, Args &&... args) noexcept(...);

  //! Implicit in-place converting error constructor
  template<class Arg1, class... Args>
  requires(!(std::is_constructible_v<value_type, Arg1, Args...>
             && std::is_constructible_v<error_type, Arg1, Args...>)  //
    &&std::is_constructible_v<error_type, Arg1, Args...>)
  constexpr variant(Arg1 &&arg1, Args &&... args) noexcept(...);

  //! Explicit in-place constructor for either value or error
  template< class Arg, class... Args >
  constexpr explicit result(std::in_place_type_t<Arg>, Args&&... args) noexcept(...);

  //! Explicit in-place constructor for either value or error
  template< class Arg, class... Args >
  constexpr explicit result(std::in_place_type_t<Arg>, Args&&... args) noexcept(...);

  //! Explicit in-place constructor for either value or error
  template< class Arg1, class Arg2, class... Args >
  constexpr explicit result(std::in_place_type_t<Arg1>,
                            std::initializer_list<Arg2> il, Args&&... args) noexcept(...);

  //! Implicit construction from any type where an ADL discovered `make_status_code(T, Args ...)`
  //! returns a `status_code`.
  template <class U, class... Args>
  requires(/* safe ADL lookup of make_status_code*/)
  constexpr result(U &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<U>(), std::
      declval<Args>()...)));

  //! Swap with another result
  constexpr void swap(result &o) noexcept(...);

  //! Clone the result
  constexpr result clone() const;

  //! True if result has a value
  constexpr bool has_value() const noexcept;

  //! True if result has a value
  explicit operator bool() const noexcept;

  //! True if result has an error
  constexpr bool has_error() const noexcept;

  //! Accesses the value if one exists, else calls `.error().throw_exception()`.
```

```
  constexpr value_type_if_enabled &value() &;

  //! Accesses the value if one exists, else calls `.error().throw_exception()`.
  constexpr const value_type_if_enabled &value() const &;

  //! Accesses the value if one exists, else calls `.error().throw_exception()`.
  constexpr value_type_if_enabled &&value() &&;

  //! Accesses the value if one exists, else calls `.error().throw_exception()`.
  constexpr const value_type_if_enabled &&value() const &&;

  //! Accesses the error if one exists, else throws `bad_result_access`.
  constexpr error_type &error() &;

  //! Accesses the error if one exists, else throws `bad_result_access`.
  constexpr const error_type &error() const &;

  //! Accesses the error if one exists, else throws `bad_result_access`.
  constexpr error_type &&error() &&;

  //! Accesses the error if one exists, else throws `bad_result_access`.
  constexpr const error_type &&error() const &&;


  //! Accesses the value, being UB if none exists
  constexpr value_type_if_enabled &assume_value() & noexcept;

  //! Accesses the error, being UB if none exists
  constexpr const value_type_if_enabled &assume_value() const &noexcept;

  //! Accesses the error, being UB if none exists
  constexpr value_type_if_enabled &&assume_value() && noexcept;

  //! Accesses the error, being UB if none exists
  constexpr const value_type_if_enabled &&assume_value() const &&noexcept;


  //! Accesses the error, being UB if none exists
  constexpr error_type &assume_error() & noexcept;

  //! Accesses the error, being UB if none exists
  constexpr const error_type &assume_error() const &noexcept;

  //! Accesses the error, being UB if none exists
  constexpr error_type &&assume_error() && noexcept;

  //! Accesses the error, being UB if none exists
  constexpr const error_type &&assume_error() const &&noexcept;
};

//! True if the two results compare equal. Available only if T can be compared to U.
template <class T, class U>
requires(...)
constexpr inline bool operator==(const result<T> &a, const result<U> &b) noexcept;

//! True if the two results compare unequal. Available only if T can be compared to U.
```

```
template <class T, class U>
requires(...)
constexpr inline bool operator!=(const result<T> &a, const result<U> &b) noexcept;
```

9 Acknowledgements

Thanks to Ben Craig, Arthur O'Dwyer and Vicente J. Botet Escriba for their comments and feedback.

Thanks to Jesse Towner for walking through the code and pointing out lots of small mistakes, as well as contributing a patchset fixing memory leaks, improving constexpr type erasure, and other implementation improvements.

10 References

[N2066] Beman Dawes,
TR2 Diagnostics Enhancements
https://wg21.link/N2066

[P0262] Lawrence Crowl, Chris Mysen,
A Class for Status and Optional Value
https://wg21.link/P0262

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[P0824] O'Dwyer, Bay, Holmes, Wong, Douglas,
Summary of SG14 discussion on <*system_error*>
https://wg21.link/P0824

[P0829] Ben Craig,
Freestanding proposal
https://wg21.link/P0829

[P1029] Douglas, Niall
move = bitcopies
https://wg21.link/P1029

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031

[P1095] Douglas, Niall
Zero overhead deterministic failure – A unified mechanism for C and C++
https://wg21.link/P1095

[P1144] O'Dwyer, Arthur
Object relocation in terms of move plus destroy
https://wg21.link/P1144

[P1195] Dimov, Peter
Making `<system_error>` *constexpr*
https://wg21.link/P1195

[P1196] Dimov, Peter
Value-based `std::error_category` *comparison*
https://wg21.link/P1196

[P1197] Dimov, Peter
A non-allocating overload of `error_category::message()`
https://wg21.link/P1197

[P1198] Dimov, Peter
Adding `error_category::failed()`
https://wg21.link/P1198

[P1631] Douglas, Niall and Steagall, Bob
Object detachment and attachment
https://wg21.link/P1631

[P1883] Douglas, Niall
`file_handle` *and* `mapped_file_handle`
https://wg21.link/P1883

[P2000] H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
http://wg21.link/P2000

[P2170] Salvia, Charles
Feedback on implementing the proposed `std::error` *type*
https://wg21.link/P2170

[1] *Boost.Outcome*
Douglas, Niall and others
https://ned14.github.io/outcome/

[2] *stl-header-heft github analysis project*
Douglas, Niall
https://github.com/ned14/stl-header-heft