P1028R3: SG14 `status_code` and standard `error` object

Document #:	P1028R3
Date:	2020-01-12
Project:	Programming Language C++ Library Evolution Working Group Library Evolution Working Group Incubator

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], a library-only implementation of [P0709]. The proposed objects have received extensive field testing in existing code bases, and have been found to work very well.

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 in production use for a year now, and has been shipping in Boost from v1.70 onwards as part of Outcome.Experimental.

This proposal is a much richer and more powerful framework than `<system_error>`. 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*.

Main change since R2:

- Despite six months elapsing since the last revision of this paper, and this library shipping in Boost and seeing widespread usage, no changes have been found necessary to the reference library, apart from bug fixes. However seeing as [P0709] *Zero-overhead deterministic exceptions: Throwing values* will likely not be making C++ 23, and C++ 23 targeting papers such as [P1883] `file_handle` *and* `mapped_file_handle` require deterministic exceptions, R3 of this paper adds proposing a port of Outcome's `result<T>` type to the standard library.
- Added `get()`, `get_if()` and `get_id()` support for accessing the underlying type of erased status code pointers.

Changes not made:

- A senior member of the C++ community reached out personally to me with deep concern about the semantic matching nature of the comparison operators, they wanted them to be made identity-based. That design choice passed SG14 without issue, and no user of the reference library has reported issues (some have emailed to ask about rare corner cases, this resulted in improvements to documentation).
- Quite a few people want `string_ref` to be hived out into its own paper, because it is generally useful. I welcome somebody who is not me to take on that work.

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 [P0939] *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.

There is an optional dependency on a core language enhancement adding bare minimum viable support for relocating objects [P1029] *move* = ***bitcopies***, which has passed EWG-I and is now before EWG.

That paper proposes a new choice of default for move constructors, `= bitcopies`. It is used by the programmer to guarantee to the compiler that move constructors equal the bit copy of source object's storage over the destination object's storage, followed by bit copy of a constexpr default constructed instance of the object over the source object's storage. By using `= bitcopies`, the programmer also guarantees that the destructor, when called on a default constructed instance, has no visible side effects.

It would be proposed that `status_code<erased<T>>` would be move bitcopying. This means it would be transportable in CPU registers, if the compiler felt that more optimal.

3 Proposed Design

3.1 `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.
  */
```

```
21  class string_ref
22  {
23  public:
24    //! The value type
25    using value_type = const char;
26    //! The size type
27    using size_type = size_t;
28    //! The pointer type
29    using pointer = const char *;
30    //! The const pointer type
31    using const_pointer = const char *;
32    //! The iterator type
33    using iterator = const char *;
34    //! The const iterator type
35    using const_iterator = const char *;
36
37  protected:
38    //! The operation occurring
39    enum class _thunk_op
40    {
41      copy,
42      move,
43      destruct
44    };
45    //! The prototype of the handler function. Copies can throw, moves and destructs cannot.
46    using _thunk_spec = void (*)(string_ref *dest, const string_ref *src, _thunk_op op);
47
48    //! Pointers to beginning and end of character range
49    pointer _begin{}, _end{};
50
51    //! Three ‘void*‘ of state
52    void *_state[3]{};  // at least the size of a shared_ptr
53
54    //! Handler for when operations occur
55    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    string_ref(const string_ref &o);
66    //! Move construct the derived implementation.
67    string_ref(string_ref &&o) noexcept;
68    //! Copy assignment
69    string_ref &operator=(const string_ref &o);
70    //! Move assignment
71    string_ref &operator=(string_ref &&o) noexcept;
72    //! Destruction
73    ~string_ref();
74
75    //! Returns whether the reference is empty or not
76    [[nodiscard]] bool empty() const noexcept;
```

```
    //! Returns the size of the string
    size_type size() const noexcept;
    //! Returns a null terminated C string
    const_pointer c_str() const noexcept;
    //! Returns a null terminated C string
    const_pointer data() const noexcept;
    //! Returns the beginning of the string
    iterator begin() noexcept;
    //! Returns the beginning of the string
    const_iterator begin() const noexcept;
    //! Returns the beginning of the string
    const_iterator cbegin() const noexcept;
    //! Returns the end of the string
    iterator end() noexcept;
    //! Returns the end of the string
    const_iterator end() const noexcept;
    //! Returns the end of the string
    const_iterator cend() const noexcept;
  };

  /*! A reference counted, threadsafe reference to a message string.
  */
  class atomic_refcounted_string_ref : public string_ref
  {
    struct _allocated_msg
    {
      mutable std::atomic<unsigned> count;
    };
    _allocated_msg *&_msg() noexcept;
    const _allocated_msg *_msg() const noexcept;

    static void _refcounted_string_thunk(string_ref *_dest, const string_ref *_src, _thunk_op op)
        noexcept;

  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.
  virtual string_ref name() const noexcept = 0;

protected:
  //! True if code means failure.
  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.
  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.
  virtual generic_code _generic_code(const status_code<void> &code) const noexcept = 0;
  //! Return a reference to a string textually representing a code.
  virtual string_ref _do_message(const status_code<void> &code) const noexcept = 0;
  //! Throw a code as a C++ exception.
  [[noreturn]] virtual void _do_throw_exception(const status_code<void> &code) const = 0;
};
```

3.2 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;
};

//! 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;
```

3.3 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.
  string_ref message() const noexcept;
  //! True if code means success.
  bool success() const noexcept;
  //! True if code means failure.
  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> 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> inline bool equivalent(const status_code<T> &o) const noexcept;

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

3.4 `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>
  {
  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
```

3.5 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 MakeStatusCodeOutType = decltype(make_status_code(std::declval<T>(), std::declval<
                Args>()...))> // ADL enable
  requires(!std::is_same<typename std::decay<T>::type, status_code>::value // not copy/move of self
           && is_status_code<MakeStatusCodeOutType>::value // ADL makes a status code
           && std::is_constructible<status_code, MakeStatusCodeOutType>::value // ADLed status code is
                compatible
  )
  constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(),
      std::declval<Args>()...)));

  //! 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::type_erasure_is_safe<ErasedType, value_type>::value)
  constexpr explicit status_code(const status_code<erased<ErasedType>> &v) noexcept(std::
      is_nothrow_copy_constructible<value_type>::value);

  //! Assignment from a `value_type`.
  constexpr status_code &operator=(const value_type &v) noexcept(std::is_nothrow_copy_assignable<
      value_type>::value);

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

3.6 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;
  ~status_code();

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

```
39
40    //! Implicit copy construction from any other status code if its value type is trivially copyable
41    //! and it would fit into our storage
42    template <class DomainType>
43    requires(std::is_trivially_copyable<typename DomainType::value_type>::value
44             && detail::type_erasure_is_safe<value_type, typename DomainType::value_type>::value)
45    constexpr status_code(const status_code<DomainType> &v) noexcept;
46
47    //! Implicit move construction from any other status code if its value type is trivially copyable
48    //! or move relocating and it would fit into our storage
49    template <class DomainType>
50    requires(detail::type_erasure_is_safe<value_type, typename DomainType::value_type>::value)
51    constexpr status_code(status_code<DomainType> &&v) noexcept;
52
53    //! Implicit construction from any type where an ADL discovered `make_status_code(T, Args ...)`
54    //! returns a `status_code`.
55    template <class T, class... Args,
56              class MakeStatusCodeOutType = decltype(make_status_code(std::declval<T>(), std::declval<
                  Args>()...))> // ADL enable
57    requires(!std::is_same<typename std::decay<T>::type, status_code>::value // not copy/move of self
58             && !std::is_same<typename std::decay<T>::type, value_type>::value // not copy/move of value
                  type
59             && is_status_code<MakeStatusCodeOutType>::value // ADL makes a status code
60             && std::is_constructible<status_code, MakeStatusCodeOutType>::value // ADLed status code is
                  compatible
61    )
62    constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(),
          std::declval<Args>()...)));
63  };
64
65  //! True if the status code's are semantically equal via `equivalent()`.
66  template <class DomainType1, class DomainType2> inline bool operator==(const status_code<DomainType1>
      &a, const status_code<DomainType2> &b) noexcept;
67  //! True if the status code's are not semantically equal via `equivalent()`.
68  template <class DomainType1, class DomainType2> inline bool operator!=(const status_code<DomainType1>
      &a, const status_code<DomainType2> &b) noexcept;
69  //! True if the status code's are semantically equal via `equivalent()` to the generic code.
70  template <class DomainType1> inline bool operator==(const status_code<DomainType1> &a, errc b)
      noexcept;
71  //! True if the status code's are semantically equal via `equivalent()` to the generic code.
72  template <class DomainType1> inline bool operator==(errc a, const status_code<DomainType1> &b)
      noexcept;
73  //! True if the status code's are not semantically equal via `equivalent()` to the generic code.
74  template <class DomainType1> inline bool operator!=(const status_code<DomainType1> &a, errc b)
      noexcept;
75  //! True if the status code's are not semantically equal via `equivalent()` to the generic code.
76  template <class DomainType1> inline bool operator!=(errc a, const status_code<DomainType1> &b)
      noexcept;
```

3.7 Exception types

```
1  /*! Exception type representing a thrown status_code
2  */
3  template <class DomainType> class status_error;
```

```
4
5  /*! The erased type edition of status_error.
6  */
7  template <> class status_error<void> : public std::exception
8  {
9  protected:
10   //! Constructs an instance. Not publicly available.
11   status_error() = default;
12   //! Copy constructor. Not publicly available
13   status_error(const status_error &) = default;
14   //! Move constructor. Not publicly available
15   status_error(status_error &&) = default;
16   //! Copy assignment. Not publicly available
17   status_error &operator=(const status_error &) = default;
18   //! Move assignment. Not publicly available
19   status_error &operator=(status_error &&) = default;
20   //! Destructor. Not publicly available.
21   ~status_error() override = default;
22
23 public:
24   //! The type of the status domain
25   using domain_type = void;
26   //! The type of the status code
27   using status_code_type = status_code<void>;
28 };
29
30 /*! Exception type representing a thrown status_code
31 */
32 template <class DomainType> class status_error : public status_error<void>
33 {
34   status_code<DomainType> _code;
35   typename DomainType::string_ref _msgref;
36
37 public:
38   //! The type of the status domain
39   using domain_type = DomainType;
40   //! The type of the status code
41   using status_code_type = status_code<DomainType>;
42
43   //! Constructs an instance
44   explicit status_error(status_code<DomainType> code);
45
46   //! Return an explanatory string
47   virtual const char *what() const noexcept override;
48
49   //! Returns a reference to the code
50   const status_code_type &code() const &;
51   //! Returns a reference to the code
52   status_code_type &code() &;
53   //! Returns a reference to the code
54   const status_code_type &&code() const &&;
55   //! Returns a reference to the code
56   status_code_type &&code() &&;
57 };
```

3.8 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;
```

3.9 `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 MakeStatusCodeOutType = decltype(make_status_code(std::declval<T>(), std::declval<
                Args>()...))> // ADL enable
  requires(!std::is_same<typename std::decay<T>::type, errored_status_code>::value // not copy/move of
       self
           && is_status_code<MakeStatusCodeOutType>::value // ADL makes a status code
           && std::is_constructible<errored_status_code, MakeStatusCodeOutType>::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]];

  //! Explicit in-place construction.
  template <class... Args>
  constexpr explicit errored_status_code(in_place_t /*unused */, Args &&... args) noexcept(std::
      is_nothrow_constructible<value_type, Args &&...>::value)  [[expects: _base(std::forward<Args>(
      args)... /* unsafe? */).failure() == true]];

  //! Explicit in-place construction from initialiser list.
  template <class T, class... Args>
  constexpr explicit errored_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) [[expects: _base(il, std::forward<Args>(args)... /* unsafe? */).failure() ==
      true]];

  //! Explicit copy construction from a `value_type`.
```

```
  constexpr explicit errored_status_code(const value_type &v) noexcept(std::
      is_nothrow_copy_constructible<value_type>::value) [[expects: _base(v).failure() == true]];

  //! Explicit move construction from a `value_type`.
  constexpr explicit errored_status_code(value_type &&v) noexcept(std::is_nothrow_move_constructible<
      value_type>::value) [[expects: _base(std::move(v) /* unsafe? */).failure() == true]];

  /*! 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::type_erasure_is_safe<ErasedType, value_type>::value)
  constexpr explicit errored_status_code(const status_code<erased<ErasedType>> &v) noexcept(std::
      is_nothrow_copy_constructible<value_type>::value) [[expects: v.failure() == true]];

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

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

3.10 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>>;
public:
  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
  //! and it would fit into our storage
  template <class DomainType>
  requires(detail::type_erasure_is_safe<value_type, typename DomainType::value_type>::value)
  constexpr errored_status_code(const errored_status_code<DomainType> &v) noexcept [[expects: v.
      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 MakeStatusCodeOutType = decltype(make_status_code(std::declval<T>(), std::declval<
                Args>()...))> // ADL enable
  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<MakeStatusCodeOutType>::value // ADL makes a status code
           && std::is_constructible<errored_status_code, MakeStatusCodeOutType>::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]];

  //! Return the erased `value_type` by value.
  constexpr value_type value() const noexcept;
};
```

3.11 OS specific codes, and erased system code

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

//! A wrapper of ‘std::error_code‘.
using std_error_code = status_code<_error_code_domain<std::error_code, detail::make_std_categories>>;

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

#ifdef _WIN32
//! (Windows only) A Win32 error code, those returned by ‘GetLastError()‘.
using win32_code = status_code<_win32_code_domain>;
//! (Windows only) A specialisation of ‘status_error‘ for the Win32 error code domain.
using win32_error = status_error<_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 ‘status_error‘ for the NT error code domain.
using nt_error = status_error<_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 ‘status_error‘ for the COM error code domain.
using com_error = status_error<_com_code_domain>;
#endif  // _WIN32

/*! 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;
```

3.12 Proposed `std::error` object

```
/*! An erased `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>>;
```

3.13 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)
inline std::ostream &operator<<(std::ostream &s, const status_code<DomainType> &v);

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

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

3.14 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)
inline 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>
inline StatusCode *get_if(status_code<erased<U>> *v) noexcept;

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


/*! If a status code refers to a `status_code_ptr` which indirects to a status
code of type `StatusCode`, return a reference to that `StatusCode`. Otherwise throw
`bad_status_ptr_access`.
*/
template <class StatusCode, class U>
requires(is_status_code<StatusCode>::value>
inline StatusCode &get(status_code<erased<U>> &v);

//! \overload Const overload
template <class StatusCode, class U>
requires(is_status_code<StatusCode>::value>
inline const StatusCode &get(const status_code<erased<U>> &v);


/*! If a status code refers to a `status_code_ptr`, return the id of the erased
status code's domain. Otherwise return a meaningless number.
*/
template <class U>
inline typename status_code_domain::unique_id_type get_id(const status_code<erased<U>> &v) noexcept;
```

4 Design decisions, guidelines and rationale

4.1 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 [P0939]'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.

4.2 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.

4.3 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

[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.

4.4 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.

4.5 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.

4.6 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.

4.7 `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.

4.8 `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.

4.9 `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.

4.10 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 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`.

4.11 `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.

5 Technical specifications

No Technical Specifications are involved in this proposal.

[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!

6 Frequently asked questions

6.1 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 Addendum: Outcome'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>` from (Boost.)Outcome, 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. It may, or may not, use union storage (Outcome's current implementation uses struct storage when `E` is `error`). Because type erased `error` is move only, `result<T>` is always move only (if you want a copy, call `.clone()`). Result has a bitcopying 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.

```
1 /*! \brief Exception type representing the failure to retrieve an error.
2  */
3 class bad_result_access;
```

```
4
5  /*! \class result
6  \brief A ‘result<T>‘ type with its error type hardcoded to ‘error‘.
7
8  This may, or may not, have union storage. It must never have a valueless state.
9  */
10 template <class T>
11 requires(!std::is_reference_v<T> && !std::is_array_v<T> && !std::is_same_v<T, error>)
12 class result
13 {
14 public:
15   //! The value type
16   using value_type = T;
17   //! The error type
18   using error_type = error;
19
20   //! Used to rebind result types
21   template <class U> using rebind = result<U>;
22
23 public:
24   //! Default constructor is disabled
25   result() = delete;
26   //! Copy constructor is disabled
27   result(const result &) = delete;
28   //! Move constructor. See P1029 move = bitcopies.
29   result(result &&) = bitcopies(auto);
30   //! Copy assignment is disabled
31   result &operator=(const result &) = delete;
32   //! Move assignment
33   result &operator=(result &&) = default;
34   //! Destructor
35   ~result() = default;
36
37   //! Implicit result converting move constructor
38   template <class U>
39   requires(std::is_convertible_v<U, T>)
40   constexpr result(result<U> &&o) noexcept(std::is_nothrow_constructible_v<T, U>);
41
42   //! Implicit result converting copy constructor
43   template <class U>
44   requires(std::is_convertible_v<U, T>)
45   constexpr result(const result<U> &o) noexcept(std::is_nothrow_constructible_v<T, U>);
46
47   //! Explicit result converting move constructor
48   template <class U>
49   requires(std::is_constructible_v<T, U>)
50   constexpr explicit result(result<U> &&o) noexcept(std::is_nothrow_constructible_v<T, U>);
51
52   //! Explicit result converting copy constructor
53   template <class U>
54   requires(std::is_constructible_v<T, U>)
55   constexpr explicit result(const result<U> &o) noexcept(std::is_nothrow_constructible_v<T, U>);
56
57   /* Slightly wider constructors than std::variant<error, T> would have. */
58
59   //! 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() &&;
```

```
115
116   //! Accesses the value if one exists, else calls `.error().throw_exception()`.
117   constexpr const value_type_if_enabled &&value() const &&;
118
119   //! Accesses the error if one exists, else throws `bad_result_access`.
120   constexpr error_type &error() &;
121
122   //! Accesses the error if one exists, else throws `bad_result_access`.
123   constexpr const error_type &error() const &;
124
125   //! Accesses the error if one exists, else throws `bad_result_access`.
126   constexpr error_type &&error() &&;
127
128   //! Accesses the error if one exists, else throws `bad_result_access`.
129   constexpr const error_type &&error() const &&;
130
131
132   //! Accesses the value, being UB if none exists
133   constexpr value_type_if_enabled &assume_value() & noexcept;
134
135   //! Accesses the error, being UB if none exists
136   constexpr const value_type_if_enabled &assume_value() const &noexcept;
137
138   //! Accesses the error, being UB if none exists
139   constexpr value_type_if_enabled &&assume_value() && noexcept;
140
141   //! Accesses the error, being UB if none exists
142   constexpr const value_type_if_enabled &&assume_value() const &&noexcept;
143
144
145   //! Accesses the error, being UB if none exists
146   constexpr error_type &assume_error() & noexcept;
147
148   //! Accesses the error, being UB if none exists
149   constexpr const error_type &assume_error() const &noexcept;
150
151   //! Accesses the error, being UB if none exists
152   constexpr error_type &&assume_error() && noexcept;
153
154   //! Accesses the error, being UB if none exists
155   constexpr const error_type &&assume_error() const &&noexcept;
156 };
157
158 //! True if the two results compare equal. Available only if T can be compared to U.
159 template <class T, class U>
160 requires(...)
161 constexpr inline bool operator==(const result<T> &a, const result<U> &b) noexcept;
162
163 //! True if the two results compare unequal. Available only if T can be compared to U.
164 template <class T, class U>
165 requires(...)
166 constexpr inline bool operator!=(const result<T> &a, const result<U> &b) noexcept;
```

8 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.

9 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

[P0939] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
http://wg21.link/P0939

[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

[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