P1028R5: SG14 `status_code` and standard `error` object

Document #:	P1028R5
Date:	2023-05-11
Project:	Programming Language C++ Library Evolution Working Group

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

A proposed wording 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. In the Issaquah 2023 meeting, LEWG requested IS wording for this proposal targeting the C++ 26 standard release.

You can find the arguments about design rationale in R4 (`https://wg21.link/P1028R4`). From R5 onwards, this has been condensed into a set of design goals and change tracking log.

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 four 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*.

Contents

1 Design goals

These were originally set by SG14, and have to date survived LEWG discussion of this paper. You can read their original paper [P0824] *Summary of SG14 discussion on* `<system_error>` for more detail.

1. **Do not cause `#include <string>`.**

   Summary of P0824 rationale: No need to be dragging in string handling, the allocator machinery et al. This proposal is designed to work well on Freestanding.

2. **All constexpr sourcing, construction and destruction.**

   Summary of P0824 rationale: `std::error_category` is impossible to optimise out, and thus introduces unnecessary runtime overhead every time a `std::error_code` is constructed.

3. **Header only libraries can now safely define custom code categories.**

   Summary of P0824 rationale: `std::error_category` is completely broken in header only libraries, and can never work reliably at all.

4. **No more `if(!ec)...`**

   Summary of P0824 rationale: That boolean test is unreliable/confusing and doesn't do what most developers think it does.

5. **No more filtering codes returned by systems APIs.**

   Summary of P0824 rationale: Stop special casing/working around all bits zero error code values.

6. **All comparisons between codes are now semantic, not literal.**

   Summary of P0824 rationale: `std::error_code` comparison semantics are inconsistently confusing even for the expert.

7. **`std::error_condition` is removed entirely (i.e. not modelled in this proposal, though we retain 100% compatibility with it).**

   Summary of P0824 rationale: `std::error_code` comparison semantics are inconsistently confusing even for the expert.

8. **`status_code`'s value type is set by its domain.**

   Summary of P0824 rationale: `std::error_code`'s value type is hard coded to an `int` which is limiting. `status_code` can carry any value of any arbitrary type `T`.

9. **`status_code<DomainType>` is type erasable.**

   Summary of P0824 rationale: If a `status_code`'s arbitrary value type `T` can be safely bit copied (e.g. is trivially copyable), we allow conversion to an `erased_status_code`.

10. **More than one 'system' error coding domain: `system_code`.**

Summary of P0824 rationale: Platforms have multiple system error coding schemas, and this ought to be modelled into standard C++ instead of claiming that there is only one 'system' coding domain (`std::system_category`).

11. **`std::errc` gets its own code domain `generic_code`, eliminating modelling `std::error_condition`.**

    Summary of P0824 rationale: By giving `std::errc` its own domain, we regularise semantic comparisons between unknown status code types into a consistent logic with predictable meanings and outcomes.

2 Change tracking log for LWG since R4

The WG21 tracker for this paper can be found at https://github.com/cplusplus/papers/issues/405.

- R4 => R5:
  - `string_ref` => 'implementation defined type' as per LEWG request.
  - `status_code_ptr` => `nested_status_code` as per LEWG request.
  - `make_nested_status_code()` now takes a STL allocator, as per LEWG request.
  - `erased<T>` has been removed as per LEWG request. A new type alias `erased_status_code<T>` alias to the appropriate tagged specialisation of `status_code`.

3 Delta from N4928

The following normative wording delta is against https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4928.pdf. For existing normative wording, green text is wording to be added, red text is wording to be removed, black text is generally notes to LEWG which shall be removed if the paper is sent to LWG. For the entirely new sections, assume it as if it were all green.

In 17.3.2 [version.syn] paragraph 2:

```
#define __cpp_lib_system_status 202305L //also in <system_status>
```

In 19.5.2 [system.error.syn]:

```
enum class errc {
 unknown = -1,
 success = 0,
address_family_not_supported, //EAFNOSUPPORT
```

[*Note:* It is suggested that the following sections be appended after 19.6 [stacktrace] as section 19.7 – end note]

System status support [sysstatus]

General [sysstatus.general]

Subclause 19.7 describes components that the standard library and C++ programs may use to report error, warning, informational and success conditions originating from code behind an opaque ABI boundary, which includes the operating system. It is a fully compatible superset of the components within 19.5 [syserr], and is intended to completely replace those in newly written code.

Header <system_status> [system.status.syn]

Class status_code_domain [system.status.code_domain]

An implementation of abstract base class status_code_domain defines the coding and interpretation of status codes of that domain.

status_code_domain is trivially copyable, and can be constructed and destructed at compile-time.

```
namespace std {
  template <class DomainType> class status_code;
  class _generic_code_domain;
  using generic_code = status_code<_generic_code_domain>;

  class status_code_domain {
    template <class DomainType> friend class status_code;
    template <class StatusCode> friend class indirecting_domain;
  public:
    using unique_id_type = /* implementation defined unsigned, at least 64 bits */;

    class string_ref;                    // implementation defined
    class atomic_refcounted_string_ref;  // implementation defined, refines string_ref

    struct payload_info_t {
      size_t payload_size{0};     // The payload size in bytes
      size_t total_size{0};       // The total status code size in bytes (includes domain pointer and
          mixins state)
      size_t total_alignment{0};  // 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);
    };
  private:
    const unique_id_type _id;  // exposition only

  protected:
    constexpr explicit status_code_domain(unique_id_type id) noexcept;
    template<size_t N>
    constexpr explicit status_code_domain(const char (&uuid)[N]) noexcept;
```

```
    template<size_t N>
    constexpr explicit status_code_domain(const char* uuid, _uuid_size<N>) noexcept;

    // All trivial
    status_code_domain(const status_code_domain &) = default;
    status_code_domain(status_code_domain &&) = default;
    status_code_domain &operator=(const status_code_domain &) = default;
    status_code_domain &operator=(status_code_domain &&) = default;
    ~status_code_domain() = default;

  public:
    constexpr bool operator==(const status_code_domain &o) const noexcept;
    constexpr bool operator!=(const status_code_domain &o) const noexcept;
    constexpr bool operator<(const status_code_domain &o) const noexcept;

    constexpr unique_id_type id() const noexcept;
    constexpr virtual string_ref name() const noexcept = 0;
    constexpr virtual payload_info_t payload_info() const noexcept = 0;

  protected:
    constexpr virtual bool _do_failure(const status_code<void> &code) const noexcept = 0;
    constexpr virtual bool _do_equivalent(const status_code<void> &code1, const status_code<void> &
        code2) const noexcept = 0;
    constexpr virtual generic_code _generic_code(const status_code<void> &code) const noexcept = 0;
    constexpr virtual string_ref _do_message(const status_code<void> &code) const noexcept = 0;
    [[noreturn]] constexpr virtual void _do_throw_exception(const status_code<void> &code) const = 0;
    constexpr virtual bool _do_erased_copy(status_code<void> &dst, const status_code<void> &src,
        payload_info_t dstinfo);
  };
}
```

`string_ref` is an implementation defined type with the public interface of a `string_view` which refers to a string stored somewhere which lasts the lifetime of the program.
`atomic_refcounted_string_ref` is an implementation defined type refining `string_ref` which maintains a thread-safe reference counted shared state containing the string, like a `shared_ptr`. Upon the reference count reaching zero, the resources backing the string are deallocated.

`payload_info_t` describes metadata about the payload which shall be carried by values of `status_code` with this implementation of domain.

This is a minimal example of a valid implementation of `status_code_domain`:

```
// The payload for the status_code, which has representation of int
enum class arithmetic_errc : int {
  success = 0,
  divide_by_zero,
  integer_divide_overflows,
  not_integer_division
};

// Forward declare the domain for the payload type, and its status code
class _arithmetic_errc_domain;
using arithmetic_errc_code = status_code<_arithmetic_errc_domain>;

// Only final domain types should be marked final!
```

```
class _arithmetic_errc_domain final : public status_code_domain {
  using _base = status_code_domain;

public:
  // Typedef value_type to the desired payload type
  using value_type = arithmetic_errc;

  // Best form is a constructor taking an overridable unique id
  // (this is more composable later). The ID chosen MUST be VERY
  // random. Do NOT invent your own ids! Always use a truly random
  // source such as https://www.random.org/cgi-bin/randbyte?nbytes=8&format=h
  // to avoid statistical collision.
  constexpr explicit _arithmetic_errc_domain
    typename _base::unique_id_type id = 0x290f170194f0c6c7) noexcept
      : _base(id) {}
  // Every domain must provide a static constexpr factory function
  // for itself. As domains are trivially copyable, this is very cheap.
  static inline constexpr const _arithmetic_errc_domain &get();

  // You must supply the name of the domain
  constexpr virtual _base::string_ref name() const noexcept override {
    static string_ref v("arithmetic error domain");
    return v;
  }
  // You must supply metadata about the payload. This implementation
  // is usually good for 99.9% of payload types and can be copy and pasted.
  constexpr virtual payload_info_t payload_info() const noexcept override {
    return {
      sizeof(value_type),
      sizeof(status_code_domain *) + sizeof(value_type),
      (alignof(value_type) > alignof(status_code_domain *))
        ? alignof(value_type)
        : alignof(status_code_domain *)};
  }

protected:
  constexpr virtual bool _do_failure(const status_code<void> &code) const noexcept override
  {
    assert(code.domain() == *this);
    const auto &c1 = static_cast<const arithmetic_errc_error &>(code);
    return c1.value() != arithmetic_errc::success;
  }
  constexpr virtual bool _do_equivalent(const status_code<void> &code1,
                                        const status_code<void> &code2) const noexcept override {
    // Our status codes are never equivalent to any other in comparisons
    return false;
  }
  constexpr virtual generic_code
    _generic_code(const status_code<void> &code) const noexcept override {
    // Our status codes do not map onto generic codes
    return {};
  }
  constexpr virtual _base::string_ref
    _do_message(const status_code<void> &code) const noexcept override {
    assert(code.domain() == *this);
    const auto &c1 = static_cast<const arithmetic_errc_error &>(code);
```

```
    switch(c1.value())
    {
    case arithmetic_errc::success:
      return _base::string_ref("success");
    case arithmetic_errc::divide_by_zero:
      return _base::string_ref("divide by zero");
    case arithmetic_errc::integer_divide_overflows:
      return _base::string_ref("integer divide overflows");
    case arithmetic_errc::not_integer_division:
      return _base::string_ref("not integer division");
    }
    return _base::string_ref("unknown");
  }
  [[noreturn]] constexpr virtual void
    _do_throw_exception(const status_code<void> &code) const override {
    // Attempting to convert us into a C++ exception throw is not allowed
    abort();
  }
};

// Factory implementation
constexpr _arithmetic_errc_domain arithmetic_errc_domain;
inline constexpr const _arithmetic_errc_domain &_arithmetic_errc_domain::get() {
  return arithmetic_errc_domain;
}

// OPTIONAL: Tell status code via ADL customisation point about the available
// implicit conversion such that status code specialisation ‘arithmetic_errc_code‘
// will implicitly construct from all values of type ‘arithmetic_errc‘
inline arithmetic_errc_code make_status_code(arithmetic_errc e) {
  return arithmetic_errc_code(in_place, e);
}
```

Construction and assignment [system.status.code_domain.cons]

```
    constexpr explicit status_code_domain(unique_id_type id) noexcept;
```

Effects: Constructs an object of class `status_code_domain` whose unique identifier shall be `id`.

Ensures: `id()== id`.

```
    template<size_t N>
    constexpr explicit status_code_domain(const char (&uuid)[N]) noexcept;
    template<size_t N>
    constexpr explicit status_code_domain(const char* uuid, _uuid_size<N>) noexcept;
```

Effects: Constructs an object of class `status_code_domain` whose unique identifier shall be parsed from a hexadecimal character string of the format `{430f1201-94fc-06c7-430f-120194fc06c7}` or `430f1201-94fc-06c7-430f-120194fc06c7`

Ensures: `id()` is a value derived from parsing the hexadecimal string[1].

Observers [system.status.code_domain.observers]

```
constexpr bool operator==(const status_code_domain &o) const noexcept;
constexpr bool operator!=(const status_code_domain &o) const noexcept;
constexpr bool operator<(const status_code_domain &o) const noexcept;
```

Effects: Compares and orders status code domains by their unique identifier.

```
constexpr unique_id_type id() const noexcept;
```

Returns: The unique identifier of the status code domain.

```
constexpr virtual string_ref name() const noexcept = 0;
```

Returns: The string representation of the status code domain.

```
constexpr virtual payload_info_t payload_info() const noexcept = 0;
```

Returns: Metadata about the payload carried by status codes of this domain.

```
constexpr virtual bool _do_failure(const status_code<void> &code) const noexcept = 0;
```

Returns: True if the status code's value represents a failure.

```
constexpr virtual bool _do_equivalent(const status_code<void> &code1, const status_code<void> &
    code2) const noexcept = 0;
```

Returns: True if the first status code's value is semantically equivalent to the second status code's value. Semantic equivalence is defined as:

1. If both status codes have equivalent domains to this domain, semantic equivalence is whether the values are equal.
2. Optionally, if the second code's domain is the generic code domain, semantic equivalence is whether this domain considers the first code's value as being equivalent.

[1]The reference implementation uses 64 bit integers for the unique identifier, as SG14 decided that was sufficient and 128-bit integers will not become portable until C23 is merged into the C++ IS. The reference implementation XOR combines the first and latter halves of the 128-bit UUID string into a 64 bit unique identifier. If 128-bit integers have become supported in the IS using standard code which works in compile time by the time library implementers implement this proposal, it would be encouraged to use an unsigned 128-bit integer for the unique id type.

3. Optionally, this domain may consider values in the second code as equivalent for any other domains of its choosing.

```
    constexpr virtual generic_code _generic_code(const status_code<void> &code) const noexcept = 0;
```

Returns: The generic code most closely mapping onto the status code's value.

```
    constexpr virtual string_ref _do_message(const status_code<void> &code) const noexcept = 0;
```

Returns: The string representation of the status code's value.

```
    [[noreturn]] constexpr virtual void _do_throw_exception(const status_code<void> &code) const = 0;
```

Throws: Throws an exception most closely representing the status code's value.

```
    constexpr virtual bool _do_erased_copy(status_code<void> &dst, const status_code<void> &src,
        payload_info_t dstinfo);
```

Returns: True if `dst` was successfully configured.

Effects: Using payload metadata from `dstinfo` to configure `dst` correctly, set `dst` to have a value corresponding that from `src`. `dst` is permitted to be empty (and therefore have no domain set). The default implementation retrieves the source payload metadata, and returns false having done nothing if the destination's `total_size` is smaller than the source's `total_size`. If the destination is sufficiently large, `memcpy` is used to copy the representation bits from `src` to `dst`.

Traits [system.status.traits]

```
namespace std::system_code {
  template <class Base, class DomainType> struct mixin : public Base {
    using Base::Base;
  };
}
```

The `system_code::mixin` template is a customisation point. Users can specialise the template for their status code domain implementation type to cause the contents of the mixin type to be inherited into status codes of that domain. Mixin types may include state, or be stateless.

```
namespace std {
  template <T> struct is_status_code_erasable {
    static constexpr bool value = is_trivially_copyable_v<T>;
  };
  template <class T> static constexpr bool is_status_code_erasable_v;
  template <class T> concept status_code_erasable = is_status_code_erasable_v<T>;
}
```

This is a trait indicating whether a type is safe for erasure into a trivially copyable type of equal or larger size when stored within a `status_code`. Additional specialisations may be added for user-defined types.

```
namespace std {
  template <class ErasedType>
  using status_code_erased_tag_type = /* implementation defined */;
}
```

This is a convenience type alias making available the implementation defined tag type used to specialise erased editions of status code.

Constraints: `ErasedType` satisfies requirement `TriviallyCopyable`[2].

```
namespace std {
  template <class T> struct is_status_code;
  template <class T> static constexpr bool is_status_code_v;
}
```

Trait types used to detect `status_code` specialisations.

Quick status codes from enumeration [system.status.quick_status_code_from_enum]

For most users who only wish to wrap an enumeration into a status code domain, there is a convenience facility which generates a custom status code domain from your enumeration.

```
namespace std {
  template <class Enum> class _quick_status_code_from_enum_domain;
  template <class Enum> using quick_status_code_from_enum_code = status_code<
      _quick_status_code_from_enum_domain<Enum>>;

  template <class Enum> struct quick_status_code_from_enum_defaults {
    using code_type = quick_status_code_from_enum_code<Enum>;

    struct mapping {
      using enumeration_type = Enum;

      const Enum value;
      const char *message;
      const std::initializer_list<errc> code_mappings;
    };

    template <class Base> struct mixin : Base {
      using Base::Base;
    };
  };

  // Specialised by user enumeration types
```

[2] If a trivial relocation or move bitcopying implementation reaches the IS at some future point, trivially relocatable or move bitcopying objects are probably also safe to use here.

```
  template <class T> struct quick_status_code_from_enum;
}
```

An example of use also based around `arithmetic_errc` from the earlier example:

```
template <>
struct std::quick_status_code_from_enum<arithmetic_errc> :
  std::quick_status_code_from_enum_defaults<arithmetic_errc> {

  // Text name of the enum
  static constexpr const auto domain_name = "arithmetic error domain";

  // 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()
  {
    // Format is: { enum value, "string representation", { list of errc mappings ... } }
    static const std::initializer_list<mapping<AnotherCode>> v = {
    {arithmetic_errc::success, "success", {std::errc::success}},
    {arithmetic_errc::divide_by_zero, "divide by zero", {std::errc::argument_out_of_domain}},
    {arithmetic_errc::integer_divide_overflows, "integer divide overflows", {std::errc::
        result_out_of_range}},
    {arithmetic_errc::not_integer_division, "not integer division", {std::errc::invalid_argument}},
    };
    return v;
  }

  // Completely optional definition of mixin for the status code
  // synthesised from 'arithmetic_errc'. It can be omitted, and the
  // empty mixin from the defaults base class will be used.
  template <class Base> struct mixin : Base {
    using Base::Base;
    constexpr int custom_method() const { return 42; }
  };
};
```

Class `status_code<void>` [system.status.code.void]

There are two forms of type erased status code, this is the first. This form is always available irrespective of the domain's value type, but cannot be directly copied, moved, nor destructed. Thus one always passes this around by const lvalue reference.

```
namespace std {
  template <> class status_code<void> {
    template <class T> friend class status_code;
  public:
    using domain_type = void;
    using value_type = void;
    using string_ref = typename status_code_domain::string_ref;
  protected:
    const status_code_domain *_domain{nullptr};  // exposition only
```

```
10
11    constexpr explicit status_code(const status_code_domain *v) noexcept;
12    status_code() = default;
13    status_code(const status_code &) = default;
14    status_code(status_code &&) = default;
15    status_code &operator=(const status_code &) = default;
16    status_code &operator=(status_code &&) = default;
17    ~status_code() = default;
18  public:
19    constexpr const status_code_domain &domain() const noexcept;
20    [[nodiscard]] constexpr bool empty() const noexcept;
21
22    constexpr string_ref message() const noexcept;
23    constexpr bool success() const noexcept;
24    constexpr bool failure() const noexcept;
25
26    template <class T> constexpr bool strictly_equivalent(const status_code<T> &o) const noexcept;
27    template <class T> constexpr bool equivalent(const status_code<T> &o) const noexcept;
28
29    [[noreturn]] constexpr void throw_exception() const;
30  };
31 }
```

Construction and assignment [system.status.code.void.cons]

```
1    constexpr explicit status_code(const status_code_domain *v) noexcept;
```

Effects: Constructs an object of class `status_code<void>` whose domain reference shall be `*v`.

Ensures: `domain()== *v`.

Observers [system.status.code.void.observers]

```
1    constexpr const status_code_domain &domain() const noexcept;
```

Returns: A constant lvalue reference to the domain of this status code.

```
1    [[nodiscard]] constexpr bool empty() const noexcept;
```

Returns: True if this status code was default constructed.

```
1    constexpr string_ref message() const noexcept;
```

Returns: If not empty, `domain()._do_message(*this)`, otherwise the string `"(empty)"`.

```
1    constexpr bool success() const noexcept;
```

Returns: If not empty, `! domain()._do_failure(*this)`, otherwise false.

```
constexpr bool failure() const noexcept;
```

Returns: If not empty, `domain()._do_failure(*this)`, otherwise false.

```
template <class T> constexpr bool strictly_equivalent(const status_code<T> &o) const noexcept;
```

Returns: If both empty, true; if neither empty, `domain()._do_equivalent(*this, o)`; otherwise false.

```
template <class T> constexpr bool equivalent(const status_code<T> &o) const noexcept;
```

Returns: If both empty, true; if one empty and the other not, false; otherwise the following algorithm is performed:

1. If `domain()._do_equivalent(*this, o)`, return true.
2. If `domain()._do_equivalent(o, *this)`, return true.
3. If `o.domain()._generic_code(o)!= errc::unknown` and `domain()._do_equivalent(*this, o.domain()._generic_code(o))`, return true.
4. If `o.domain()._generic_code(o)!= errc::unknown` and `domain()._do_equivalent(o.domain()._generic_code(o), *this)`, return true.
5. Otherwise, return false;

```
[[noreturn]] constexpr void throw_exception() const;
```

Throws: If not empty, `domain()._do_throw_exception(*this)`, otherwise implementation defined.

Class `status_code<DomainType>` [system.status.code]

This is a typed status code whose payload type is defined by the status code's domain type. The customisation point `system_code::mixin` may inject additional state and/or member functions.

It is required to inherit publicly from `status_code<void>` and `std::system_code::mixin</*implementation defined base type */, DomainType>`.

An ADL discovered customisation point `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 from `T, Args...` is made available.

```
namespace std {
  template <class DomainType>
  requires(
    (!is_default_constructible_v<typename DomainType::value_type>
      || is_nothrow_default_constructible_v<typename DomainType::value_type>)
    && (!is_move_constructible_v<typename DomainType::value_type>
      || is_nothrow_move_constructible_v<typename DomainType::value_type>)
    && is_nothrow_destructible_v<typename DomainType::value_type>
  )
  class status_code : public /* implementation defined */ {
    template <class T> friend class status_code;
  public:
    using domain_type = DomainType;
    using value_type = typename domain_type::value_type;
    using string_ref = typename domain_type::string_ref;
  protected:
    value_type _value;  // exposition only
  public:
    status_code() = default;
    status_code(const status_code &) = default;
    status_code(status_code &&) = default;
    status_code &operator=(const status_code &) = default;
    status_code &operator=(status_code &&) = default;
    ~status_code() = default;
    void swap(status_code &) noexcept(is_nothrow_swappable_v<status_code>);

    template <class T, class... Args,
      class MakeStatusCodeResult = /* safe ADL lookup of make_status_code() */>
    requires(!is_same_v<decay_t<T>, status_code>                    // not copy/move of self
           && !is_same<decay_t<T>, in_place_t>                      // not in_place_t
           && is_status_code_v<MakeStatusCodeResult>                // ADL makes a status code
           && is_constructible_v<status_code, MakeStatusCodeResult>)  // ADLed status code is
                compatible
    )
    constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(declval<T>(),
        declval<Args>()...)));

    template<class Enum, class QuickStatusCodeType
      = typename quick_status_code_from_enum<Enum>::code_type>      // Enumeration has been activated
    requires(is_constructible_v<status_code, QuickStatusCodeType>)  // Its status code is compatible
    constexpr status_code(Enum &&v) noexcept(is_nothrow_constructible_v<status_code,
        QuickStatusCodeType>::value);

    template <class... Args>
    constexpr explicit status_code(in_place_t /*unused */, Args &&... args) noexcept(
        is_nothrow_constructible_v<value_type, Args &&...>);

    template <class T, class... Args>
    constexpr explicit status_code(in_place_t /*unused */, initializer_list<T> il, Args &&... args)
        noexcept(is_nothrow_constructible_v<value_type, initializer_list<T>, Args &&...>);

    constexpr explicit status_code(const value_type &v) noexcept(is_nothrow_copy_constructible_v<
        value_type>);

    constexpr explicit status_code(value_type &&v) noexcept(is_nothrow_move_constructible<value_type>)
```

```
        ;

    template <class ErasedType>
    requires(/* if construction from erased type is safe */)
    constexpr explicit status_code(const erased_status_code<ErasedType> &v) noexcept(
        is_nothrow_copy_constructible<value_type>);

    constexpr const domain_type &domain() const noexcept;
    constexpr string_ref message() const noexcept;

    constexpr value_type &value() & noexcept;
    constexpr value_type &&value() && noexcept;
    constexpr const value_type &value() const& noexcept;
    constexpr const value_type &&value() const&& noexcept;

    constexpr status_code clone() const;
    constexpr void clear() noexcept;
  };
}
```

Construction and assignment [system.status.code.cons]

```
    status_code() = default;
```

Effects: Constructs an empty `status_code<DomainType>`.

Ensures: `empty()== true`.

```
    status_code(const status_code &) = default;
    status_code(status_code &&) = default;
    status_code &operator=(const status_code &) = default;
    status_code &operator=(status_code &&) = default;
    ~status_code() = default;
    void swap(status_code &) noexcept(is_nothrow_swappable_v<status_code>);
```

Effects: Status code's copy and move constructors and special member functions replicate the availability and noexcept of those of its `value_type`.

Ensures: If not empty, `domain()== DomainType::get()`.

```
    template <class T, class... Args,
      class MakeStatusCodeResult = /* safe ADL lookup of make_status_code() */>
    requires(!is_same_v<decay_t<T>, status_code>                      // not copy/move of self
            && !is_same<decay_t<T>, in_place_t>                       // not in_place_t
            && is_status_code_v<MakeStatusCodeResult>                 // ADL makes a status code
            && is_constructible_v<status_code, MakeStatusCodeResult>) // ADLed status code is
                compatible
    )
    constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(declval<T>(),
        declval<Args>()...)));
```

Effects: Implicitly constructs an object of class `status_code<DomainType>` from the value yielded by the ADL customisation point `make_status_code(T, Args...)`.

Constraints: `T` cannot be `status_code`; `T` cannot be `in_place_t`; `make_status_code(T, Args...)` must yield a status code from which this status code can be constructed.

Ensures: `domain()== DomainType::get()`.

```
    template<class Enum, class QuickStatusCodeType
      = typename quick_status_code_from_enum<Enum>::code_type>      // Enumeration has been activated
    requires(is_constructible_v<status_code, QuickStatusCodeType>)  // Its status code is compatible
    constexpr status_code(Enum &&v) noexcept(is_nothrow_constructible_v<status_code,
        QuickStatusCodeType>::value);
```

Effects: Implicitly constructs an object of class `status_code<DomainType>` from the enumeration value.

Constraints: `Enum` must have a `quick_status_code_from_enum` specialisation; `quick_status_code_from_enum<Enum>::code_type` must be a type from which this status code can be constructed.

Ensures: `domain()== DomainType::get()`; `value()== v`.

```
    template <class... Args>
    constexpr explicit status_code(in_place_t /*unused */, Args &&... args) noexcept(
        is_nothrow_constructible_v<value_type, Args &&...>);
```

Effects: Explicitly constructs an object of class `status_code<DomainType>` whose `value()` is emplacement constructed from `Args...`.

Ensures: `domain()== DomainType::get()`; `value()== value_type(Args...)`.

```
    template <class T, class... Args>
    constexpr explicit status_code(in_place_t /*unused */, initializer_list<T> il, Args &&... args)
        noexcept(is_nothrow_constructible_v<value_type, initializer_list<T>, Args &&...>);
```

Effects: Explicitly constructs an object of class `status_code<DomainType>` whose `value()` is emplacement constructed from `il, Args...`.

Ensures: `domain()== DomainType::get()`; `value()== value_type(il, Args...)`.

```
    constexpr explicit status_code(const value_type &v) noexcept(is_nothrow_copy_constructible_v<
        value_type>);
```

Effects: Explicitly constructs an object of class `status_code<DomainType>` whose `value()` is copy constructed from `v`.

Ensures: `domain()== DomainType::get()`; `value()== v`.

```
    constexpr explicit status_code(value_type &&v) noexcept(is_nothrow_move_constructible<value_type>)
        ;
```

Effects: Explicitly constructs an object of class `status_code<DomainType>` whose `value()` is move constructed from `v`.

Ensures: `domain()== DomainType::get()`; `value()== v`.

```
    template <class ErasedType>
    requires(/* if construction from erased type is safe */)
    constexpr explicit status_code(const erased_status_code<ErasedType> &v);
```

Expects: `domain()== v.domain()`.

Effects: Explicitly constructs an object of class `status_code<DomainType>` whose `value()` is bit copied from the erased value type of the erased status code `v`.

Constraints: `value_type` must be `StatusCodeErasable` and `ErasedType` and any mixin must be trivially copyable; `v`'s size must be smaller or equal to `*this`'s size.

Ensures: `domain()== DomainType::get()`.

Observers [system.status.code.observers]

```
    constexpr const domain_type &domain() const noexcept;
```

Returns: A constant lvalue reference to the domain of this status code.

Remarks: This overrides `status_code<void>::domain()` to return the implementation domain type.

```
    constexpr string_ref message() const noexcept;
```

Returns: If not empty, `domain()._do_message(*this)`, otherwise the string `"(empty)"`.

Remarks: This overrides `status_code<void>::message()` to return the implementation domain type's `string_ref` rather than the base type.

```
    constexpr value_type &value() & noexcept;
    constexpr value_type &&value() && noexcept;
    constexpr const value_type &value() const& noexcept;
    constexpr const value_type &&value() const&& noexcept;
```

Returns: A reference to the payload value of this status code.

```
    constexpr status_code clone() const;
```

Expects: `value_type` is copy constructible.

Returns: A copy of this status code.

Modifiers [system.status.code.modifiers]

```
    constexpr void clear() noexcept;
```

Effects: If not empty, destroy the payload value and set the status code's value to empty.

Ensures: `empty()== true`.

Typedef `erased_status_code<ErasedType>` [system.status.erased.code]

Constraints: `ErasedType` satisfies requirement `TriviallyCopyable`[3].

There are two forms of type erased status code, this is a type alias to the second. This form is move-only, retaining the domain of its source, but erasing the payload value into a trivially copyable storage type if `DomainType::value_type` is `StatusCodeErasable`. The customisation point `system_code::mixin` may inject additional state and/or member functions.

It is required to inherit publicly from `status_code<void>` and `system_code::mixin</*implementation defined base type */, status_code_erased_tag_type<ErasedType>>`.

```
namespace std {
  template <class ErasedType>
  using erased_status_code = status_code</* implementation defined */>;

  template <class ErasedType>
  class status_code</* implementation defined */>
    : public /* implementation defined */ {
    template <class T> friend class status_code;
  public:
    using domain_type = void;
    using value_type = ErasedType;
    using string_ref = typename status_code<void>::string_ref;
  protected:
    value_type _value;  // exposition only
  public:
    status_code() = default;
    status_code(const status_code &) = delete;
    status_code(status_code &&) = default;
    status_code &operator=(const status_code &) = delete;
```

[3] If a trivial relocation or move bitcopying implementation reaches the IS at some future point, trivially relocatable or move bitcopying objects are probably also safe to use here.

```
    status_code &operator=(status_code &&) = default;
    constexpr ~status_code();

    template <class DomainType>
    requires(/* domain value type erasure is safe */
            && !is_erased_status_code_v<status_code<DomainType>>)
    constexpr status_code(const status_code<DomainType> &v) noexcept;

    template <class DomainType>
    requires(/* domain value type erasure is safe */)
    constexpr status_code(status_code<DomainType> &&v) noexcept;

    template <class T, class... Args,
              class MakeStatusCodeResult =/* safe ADL lookup of make_status_code() */>
    requires(!is_same_v<decay_t<T>, status_code>                        // not copy/move of self
            && !is_same<decay_t<T>, value_type>                         // not in_place_t
            && is_status_code_v<MakeStatusCodeResult>                   // ADL makes a status code
            && is_constructible_v<status_code, MakeStatusCodeResult>))  // ADLed status code is
                compatible
    )
    constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(declval<T>(),
        declval<Args>()...)));

    template<class Enum,
            class QuickStatusCodeType = typename quick_status_code_from_enum<Enum>::code_type)
                        // Enumeration has been activated
    requires(is_constructible_v<status_code, QuickStatusCodeType>)      // Its status code is
        compatible
    constexpr status_code(Enum &&v) noexcept(is_nothrow_constructible_v<status_code,
        QuickStatusCodeType>);

    explicit constexpr status_code(const status_code<void> &v);

    constexpr status_code(nothrow_t, const status_code<void> &v) noexcept

    constexpr status_code clone() const;
    constexpr void clear() noexcept;
  };
}
```

Construction and assignment [system.status.erased.code.cons]

```
    status_code() = default;
```

Effects: Constructs an empty status code.

Ensures: `empty()== true`.

```
    status_code(const status_code &) = delete;
    status_code(status_code &&) = default;
    status_code &operator=(const status_code &) = delete;
    status_code &operator=(status_code &&) = default;
```

Effects: Erased status codes are move-only.

Ensures: If the source is not empty, `domain()` becomes the source domain, and the source object becomes empty, otherwise results in an empty status code.

```
    template <class DomainType>
    requires(/* domain value type erasure is safe */
            && !is_erased_status_code_v<status_code<DomainType>>)
    constexpr status_code(const status_code<DomainType> &v) noexcept;
```

Effects: If the source is not empty, mplicitly constructs an object of alias `erased_status_code<ErasedType>` through bit copying of the value of `v`.

Constraints: `value_type` must be `StatusCodeErasable` and `ErasedType` and any mixin must be trivially copyable; `v`'s size must be smaller or equal to `*this`'s size; `status_code<DomainType>` cannot be an erased status code.

Ensures: If source was not empty, `domain()== v.domain()`, otherwise results in an empty status code.

```
    template <class DomainType>
    requires(/* domain value type erasure is safe */)
    constexpr status_code(status_code<DomainType> &&v) noexcept;
```

Effects: If the source is not empty, implicitly constructs an object of alias `erased_status_code<ErasedType>` through bit copying of the value of `v`, afterwards the source value is left in a state as if it had a move constructor performed upon it.

Constraints: `value_type` must be `StatusCodeErasable` and `ErasedType` and any mixin must be trivially copyable; `v`'s size must be smaller or equal to `*this`'s size.

Ensures: If not empty, `domain()` becomes the source domain, and the source object becomes empty, otherwise results in an empty status code.

```
    template <class T, class... Args,
      class MakeStatusCodeResult = /* safe ADL lookup of make_status_code() */>
    requires(!is_same_v<decay_t<T>, status_code>                      // not copy/move of self
            && !is_same<decay_t<T>, in_place_t>                       // not in_place_t
            && is_status_code_v<MakeStatusCodeResult>                 // ADL makes a status code
            && is_constructible_v<status_code, MakeStatusCodeResult>) // ADLed status code is
                compatible
    )
    constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(declval<T>(),
        declval<Args>()...)));
```

Effects: Implicitly constructs an object of the erased status code from the value yielded by the ADL customisation point `make_status_code(T, Args...)`.

Constraints: `T` cannot be `status_code`; `T` cannot be `in_place_t`; `make_status_code(T, Args...)` must yield a status code from which this status code can be constructed.

Ensures: `domain()== make_status_code(T, Args...).domain()`.

```
    template<class Enum, class QuickStatusCodeType
      = typename quick_status_code_from_enum<Enum>::code_type>      // Enumeration has been activated
    requires(is_constructible_v<status_code, QuickStatusCodeType>)  // Its status code is compatible
    constexpr status_code(Enum &&v) noexcept(is_nothrow_constructible_v<status_code,
        QuickStatusCodeType>::value);
```

Effects: Implicitly constructs an object of the erased status code from the enumeration value.

Constraints: `Enum` must have a `quick_status_code_from_enum` specialisation; `quick_status_code_from_enum<Enum>::code_type` must be a type from which this status code can be constructed.

Ensures: `domain()== make_status_code(T, Args...).domain()`.

```
    explicit constexpr status_code(const status_code<void> &v);
```

Effects: Explicitly constructs an object of the erased status code from the other kind of erased status code by invoking `v.domain()._do_erased_copy(*this, v, payload_info_t{sizeof(value_type), sizeof(status_code), alignof(status_code)})` to perform the copy.

Ensures: `domain()== v.domain()`.

Throws: An exception if `_do_erased_copy()` returns false.

> [*Note:* On implementations with exceptions globally disabled, one might consider omitting this overload entirely. – end note]

```
constexpr status_code(nothrow_t, const status_code<void> &v) noexcept
```

Effects: Constructs an object of the erased status code from the other kind of erased status code by invoking `v.domain()._do_erased_copy(*this, v, payload_info_t{sizeof(value_type), sizeof(status_code), alignof(status_code)})` to perform the copy. If that invocation returns false, the constructed object shall be empty.

Ensures: `domain()== v.domain()` if `_do_erased_copy()` returns true, empty otherwise.

Destructor [system.status.erased.code.cons]

```
    constexpr ~status_code();
```

Effects: If not empty, invokes `domain()._do_erased_destroy(*this, payload_info_t{sizeof(value_type), sizeof(status_code), alignof(status_code)})`.

Observers [system.status.erased.code.observers]

```
    constexpr status_code clone() const;
```

Returns: A copy of this erased status code, which if non-empty is obtained by invoking `domain()._do_erased_copy(ret, *this, domain().payload_info())`.

Throws: An exception if `_do_erased_copy()` returns false.

Modifiers [system.status.erased.code.modifiers]

```
    constexpr void clear() noexcept;
```

Effects: If not empty, destroy the payload value by invoking `domain()._do_erased_destroy(*this, payload_info_t{sizeof(value_type), sizeof(status_code), alignof(status_code)})`, and set the status code's value to empty.

Ensures: `empty()== true`.

Status code comparisons [system.status.code.comparisons]

```
namespace std {
  //! 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);
}
```

Exception types [system.status.code.excepts]

For status codes with no natural mapping to an obvious exception type equivalent to their meaning, class `status_error` provides exception types modelling `status_code`. This is a reasonable type to throw from implementations of `status_code_domain::throw_exception()` if nothing more appropriate is to hand.

Class `status_error<void>` [system.status.code.excepts.erased]

```
namespace std {
  template <class DomainType> class status_error;

  template <>
  class status_error<void> : public std::exception
  {
  protected:
    status_error() = default;
    status_error(const status_error &) = default;
    status_error(status_error &&) = default;
    status_error &operator=(const status_error &) = default;
    status_error &operator=(status_error &&) = default;
    ~status_error() override = default;

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

  public:
    using domain_type = void;
    using status_code_type = status_code<void>;

  public:
    const status_code<void> &code() const noexcept { return do_code(); }
  };
}
```

TODO

Class `status_error<DomainType>` [system.status.code.excepts.typed]

```
namespace std {
  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:
    using domain_type = DomainType;
    using status_code_type = status_code<DomainType>;

    explicit status_error(status_code<DomainType> code);

    virtual const char *what() const noexcept override;

    const status_code_type &code() const &;
    status_code_type &code() &;
    const status_code_type &&code() const &&;
    status_code_type &&code() &&;
  };
}
```

TODO

Generic error coding implementation [system.status.code.generic.impl]

```
namespace std {
  class _generic_code_domain;  // exposition only
  using generic_error = status_error<_generic_code_domain>;
  constexpr _generic_code_domain generic_code_domain;
  constexpr inline generic_code make_status_code(errc c) noexcept;
}
```

The implementation of alias `generic_code` declared before `status_code_domain`. `generic_code` is enabled for implicit construction from `errc` enumeration values.

Class `errored_status_code<DomainType>` [system.status.errored.code]

This refines `status_code<DomainType>` to include a precondition that the value always indicates a failure.

> [*Note:* If Contracts are in the IS by the time this is standardised, it is explicitly intended that construction of the `errored_*` types with a non-failure value is a contract violation. – end note]

```
namespace std {
  template <class DomainType>
  class errored_status_code : public status_code<DomainType>
  {
    using _base::clear;    // disabled in this refinement
    using _base::success;  // disabled in this refinement

  public:
    //! The type of the errored error code.
    using typename status_code<DomainType>::value_type;
    //! The type of a reference to a message string.
    using typename status_code<DomainType>::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
        && !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<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<errored_status_code, QuickStatusCodeType>::value)
      [[expects: errored_status_code(QuickStatusCodeType(static_cast<Enum &&>(v))).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::domain_value_type_erasure_is_safe<domain_type, erased<ErasedType>>::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]];

    //! 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;
  };
}
```

TODO

Typedef `erased_errored_status_code<ErasedType>` [system.status.erased.errored.code]

This refines `erased_status_code<ErasedType>` to include a precondition that the value always indicates a failure.

> [*Note:* If Contracts are in the IS by the time this is standardised, it is explicitly intended that construction of the `errored_*` types with a non-failure value is a contract violation. – end note]

```
namespace std {
  template <class ErasedType>
  using erased_errored_status_code = errored_status_code</* implementation defined */>;

  template <class ErasedType>
  class errored_status_code</* implementation defined */>
    : public erased_status_code</* implementation defined */>
  {
    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;
  };
}
```

TODO

Errored status code comparisons [system.status.errored.code.comparisons]

```
namespace std {
  //! 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>
```

```
12  constexpr bool operator!=(const status_code<DomainType1> &a, const status_code<DomainType2> &b)
        noexcept;
13  template <class DomainType1, class DomainType2>
14  constexpr bool operator!=(const status_code<DomainType1> &a, const errored_status_code<DomainType2>
        &b) noexcept;
15  template <class DomainType1, class DomainType2>
16  constexpr bool operator!=(const errored_status_code<DomainType1> &a, const status_code<DomainType2>
        &b) noexcept;
17
18  //! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
19  template<class DomainType1, class T, class MakeStatusCodeResult
20    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
21  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
22  constexpr bool operator==(const status_code<DomainType1> &a, const T &b);
23  template<class DomainType1, class T, class MakeStatusCodeResult
24    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
25  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
26  constexpr bool operator==(const errored_status_code<DomainType1> &a, const T &b);
27
28  //! True if the status code's are semantically equal via 'equivalent()' to 'make_status_code(T)'.
29  template<class DomainType1, class T, class MakeStatusCodeResult
30    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
31  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
32  constexpr bool operator==(const T &a, const status_code<DomainType1> &b);
33  template<class DomainType1, class T, class MakeStatusCodeResult
34    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
35  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
36  constexpr bool operator==(const T &a, const errored_status_code<DomainType1> &b);
37
38  //! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)
        '.
39  template<class DomainType1, class T, class MakeStatusCodeResult
40    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
41  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
42  constexpr bool operator!=(const status_code<DomainType1> &a, const T &b);
43  template<class DomainType1, class T, class MakeStatusCodeResult
44    = typename detail::safe_get_make_status_code_result<const T &>::type>  // Safe ADL lookup of
          make_status_code(), returns void if not found
45  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a
        status code
46  constexpr bool operator!=(const errored_status_code<DomainType1> &a, const T &b);
47
48  //! True if the status code's are not semantically equal via 'equivalent()' to 'make_status_code(T)
        '.
49  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
  requires(is_status_code<MakeStatusCodeResult>::value)                                 // ADL makes a status code
  constexpr bool operator!=(const T &a, const errored_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);
  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 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);

  //! 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);
  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);
}
```

TODO

Quick declaration of a new status code domain implementation [system.status.code.quick.impl

```
namespace std {
  template <class Enum>
  class _quick_status_code_from_enum_domain<Enum>;  // exposition only
  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
}
```

The implementation of the coding domain for each enumeration `Enum`. As `Enum` is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(Enum)` is enabled.

`error_category` compatibility [system.status.code.error_category]

Status codes will implicitly construct from an `error_code`, using a status code domain which wraps the error code's category and delegates decisions to it in a backwards compatible way. The alias to the typed form of status codes representing an `error_code` is `system_error_code`. As `error_code` 's value type is `int` which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(int)` is enabled.

```
namespace std {
  class _std_error_code_domain;  // exposition only
  namespace mixins {
    template <class Base>
    struct mixin<Base, _std_error_code_domain> : public Base {
      using Base::Base;
      inline mixin(std::error_code ec);

      // Returns the original category of the original error code
      inline const std::error_category &category() const noexcept;
    };
  }  // namespace mixins
  using system_error_code = status_code</* implementation defined */>;
}
```

POSIX coding [system.status.code.posix_code]

POSIX codes are whatever superset of POSIX <cerrno> the operating system implements. For convenience, a mixin `posix_code::current()` is provided which returns a status code with the posix code domain containing the current value of POSIX `errno`. As POSIX `errno`'s type is `int` which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(int)` is enabled.

If the implementation does not implement any form of POSIX support, this section is omitted.

```
namespace std {
  class _posix_code_domain;  // exposition only
  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

using posix_code = status_code<_posix_code_domain>;
using posix_error = errored_status_code<_posix_code_domain>;
```

HTTP status coding [system.status.code.http_status_code]

HTTP status codes are defined by the IETF RFC 9110 standard, and can represent success, informational, and failure codes. As HTTP status code's type is `int` which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(int)` is enabled.

If the implementation does not implement any form of network support, this section is omitted.

```
namespace std {
  class _http_status_code_domain;  // exposition only
  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

```

```
  using http_status_code = status_code<_http_status_code_domain>;
  using http_status_error = errored_status_code<_http_status_code_domain>;
```

IP address resolution coding [system.status.code.getaddrinfo_code]

Retrieving information about an internet address is very widely supported across implementations and uses a function returning its own error coding which is nearly identical everywhere. As `getaddrinfo()`'s coding type is `int` which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(int)` is enabled.

If the implementation does not implement any form of network support, this section is omitted.

```
namespace std {
  class _getaddrinfo_code_domain;  // exposition only
  //! 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>;
}
```

Microsoft Windows coding [system.status.code.microsoft]

Microsoft Windows has multiple system error code domains, the most common three of which are encoded here:

1. Win32 error codes, type is `DWORD` which is `unsigned int`. These can reflect either success or cause of failure.
2. NT kernel status codes, type is `NTSTATUS` which is `long`. These can reflect success values, information values, warning values, and error values.
3. COM result codes, type is `HRESULT` which is `long`. These can reflect values of application defined meaning, and codes can be from multiple unknown applications, however they do have a single universal bit which indicates whether a value means success or failure.

All of these representation types are trivially copyable types, so erasure into `erased_status_code` with an erased type larger or equal to `sizeof(representation_type)` is enabled.

If the implementation does not implement any specific form of an error coding listed here, that specific coding is omitted.

```
namespace std {
  class _win32_code_domain;  // exposition only
  class _nt_code_domain;     // exposition only
  class _com_code_domain;    // exposition only
  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>;
}
```

Erased system code and `error` object [system.status.code.aliases]

It is very common to erase system codes and enumerations into a single erased status code type which appears in public interfaces. These type aliases are guaranteed to be sufficiently large to erase any status code constructed using default arguments where those arguments have defaults defined in header <system_status>.

```
namespace std {
  using system_code = erased_status_code</* implementation defined, usually an intptr_t on most
      platforms */>;
  using error = erased_errored_status_code<system_code::value_type>;
}
```

> [*Note:* These are an obvious source of potential future ABI breakage, so the erased type ought to be chosen carefully as it can never be changed again without breaking the ABI of everything which returns system codes. – end note]

Exception ptr compatibility [system.status.code.exception_ptr]

```
namespace std {
  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;
}
```

Returns: If when rethrown `ep`'s exception type would be caught by a standard exception type with a natural mapping onto a status code defined in header `<system_status>`, and if relevant that exception's value has a natural mapping onto a value within a status code defined in header `<system_status>`, return the closest status code with (if relevant) the closest value to the exception. If unable to match, return `not_matched`.

> [*Note:* For example, `bad_alloc` (which is valueless) and `system_error` (which has a value) with `.code()== errc::not_enough_memory` would both return `make_status_code (errc::not_enough_memory)` which returns a `generic_code` which will then implicitly erase itself into `system_code`. Comparisons of that returned system code to `errc:: not_enough_memory` will thereafter be true. – end note]

```
namespace std {
  class _std_exception_ptr_domain;  // exposition only
  namespace mixins {
    template <class Base>
    struct mixin<Base, _std_exception_ptr_domain> : public Base {
      using Base::Base;

      // Returns the original exception_ptr
      std::exception_ptr ptr() const noexcept;
    };
  }  // namespace mixins
  using exception_ptr_code = status_code<_std_exception_ptr_domain>;
}
```

Status codes will implicitly construct from an `exception_ptr`, using a status code domain which wraps the exception ptr and delegates decisions to it such as whether it is semantically equivalent to another status code. The alias to the typed form of status codes representing an `exception_ptr` is `exception_ptr_code`. As `exception_ptr_code`'s value type is a pointer which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(void *)` is enabled.

iostream support [system.status.code.iostream]

```
namespace std {
  ostream &operator<<(ostream &s, const status_code_domain::string_ref &v);
}
```

Effects: Equivalent to `return s << string_view(v);`.

```
namespace std {
```

```
  template <class DomainType>
  ostream &operator<<(ostream &s, const status_code<DomainType> &v);
}
```

Expects: That `DomainType::value_type` implements `ostream &operator<<(ostream &, const value_type &)`.

Effects: If empty, equivalent to `return s << "(empty)";`, otherwise equivalent to `return s << v.domain().name()<< ": "<< v.value();`.

```
namespace std {
  template <class DomainType>
  ostream &operator<<(ostream &s, const status_code<DomainType> &v);
}
```

Expects: That `DomainType::value_type` is not a valid expression or does not implement `ostream &operator<<(ostream &, const value_type &)`.

Effects: If empty, equivalent to `return s << "(empty)";`, otherwise equivalent to `return s << v.domain().name()<< ": "<< v.message();`.

Nested status codes [system.status.code.nested]

Nested status codes are status codes which indirect to a dynamically allocated status code, thus enabling a small copyable owning reference to a larger or non-moveable status code in memory. As nested status code's `value_type` is a pointer to the indirected status code which is a trivially copyable type, erasure into `erased_status_code` with an erased type larger or equal to `sizeof(void*)` is enabled. As `erased_status_code` is move-only, expensive dynamic memory allocation copies are not possible after erasure.

```
namespace std {
  template <class T, class Alloc = allocator<decay_t<T>>>
  requires(is_status_code_v<T>)
  erased_status_code<add_pointer_t<decay_t<T>>> make_nested_status_code(T &&v, Alloc alloc = {});
}
```

Expects: `T` is a status code.

Returns: An erased status code with value type of `T*` pointing to a dynamically allocated status code of type `T` which is allocated and move constructed from `v` using allocator `Alloc`.

Throws: Anything which the allocator may throw.

Ensures: That the member functions of the object referenced by `domain()` invoke the corresponding function in `value()->domain()` for `*value()` for these functions: `name()`, `payload_info()`, `_do_failure()`, `_do_equivalent()`, `_generic_code()`, `_do_message()`, `_do_throw_exception()`; that the domain's `_do_erased_copy()` and `_do_erased_destroy()` invoke the supplied allocator to

perform a new dynamic memory allocation and copy construction, and a destruction and dynamic memory release respectively.

```
namespace std {
  template <class StatusCode, class U>
  requires(is_status_code_v<StatusCode>>
  StatusCode *get_if(status_code<U> *v) noexcept;

  template <class StatusCode, class U>
  requires(is_status_code_v<StatusCode>>
  const StatusCode *get_if(const status_code<U> *v) noexcept;
}
```

Expects: `StatusCode` is a status code.

Returns: If `v` is a pointer to a nested status code which indirects to a status code of type `StatusCode`, return a pointer to that status code. Otherwise return null.

```
namespace std {
  template <class StatusCode, class U>
  requires(is_status_code_v<StatusCode>>
  StatusCode &get(status_code<U> &v);

  template <class StatusCode, class U>
  requires(is_status_code_v<StatusCode>>
  const StatusCode &get(const status_code<U> &v);
}
```

Expects: `StatusCode` is a status code.

Returns: If `v` is a reference to a nested status code which indirects to a status code of type `StatusCode`, return a reference to that status code.

Throws: If `v` is not a reference to a nested status code, or the nested status code does not indirect to a status code of type `StatusCode`, throws an exception of `bad_nested_status_code_access`.

```
namespace std {
  template <class U>
  typename status_code_domain::unique_id_type get_id(const status_code<U> &v) noexcept;
}
```

Returns: If `v` is a reference to a nested status code, returns the id of the indirected to domain. Otherwise return a meaningless number.

4 Acknowledgements

This paper would not have happened in a timely fashion without my current client MayStreet London Stock Exchange Group permitting me a few work hours per month to write this revision. My thanks to them for enabling this paper revision.

5 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