P1028R0: SG14 `status_code` and standard `error` object for P0709 Zero-overhead deterministic exceptions

Document #:	P1028R0
Date:	2018-05-06
Project:	Programming Language C++ Library Evolution Working Group
Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A proposal for the replacement, in new code, of the system header `<system_error>` with a substantially refactored and lighter weight design, which meets modern C++ design and implementation. This paper received the following vote at the May 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 reference implementation has been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64.

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 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:

1. P1029 *SG14* `[[move_relocates]]` https://wg21.link/P1029.

   This proposes a new C++ attribute `[[move_relocates]]` which lets the compiler optimise such attributed moves as aggressively as trivially copyable types. If approved, this would enable a large increase in the variety of types directly transportable in the proposed `error` object, specifically the ability to transport `std::exception_ptr` instances directly, a highly desirable feature for improving efficiency of legacy C++ exceptions support under P0709.

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;

template <class T>
struct is_status_code
{
    static constexpr bool const value;
};

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>;
```

```
class status_code_domain
{
  template <class DomainType> friend class status_code;

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

  /*! (Potentially thread safe) Reference to a message string.
  Be aware that you cannot add payload to implementations of this class.
  You get exactly the ‘void *[3]‘ array to keep state, this is usually
  sufficient for a ‘std::shared_ptr<>‘ or a ‘std::string‘.
  You can install a handler to be called when this object is copied,
  moved and destructed. This takes the form of a C function pointer.
  */
  class string_ref
  {
  public:
    //! The value type
    using value_type = const char;
    //! The size type
    using size_type = size_t;
    //! The pointer type
    using pointer = const char *;
    //! The const pointer type
    using const_pointer = const char *;
    //! The iterator type
    using iterator = const char *;
```

```
    //! The const iterator type
    using const_iterator = const char *;

  protected:
    //! The operation occurring
    enum class _thunk_op
    {
      copy,
      move,
      destruct
    };
    //! The prototype of the handler function. Copies can throw, moves and destructs cannot.
    using _thunk_spec = void (*)(string_ref *dest, const string_ref *src, _thunk_op op);
    //! Pointers to beginning and end of character range
    pointer _begin{}, _end{};
    //! Three `void*` of state
    void *_state[3]{};  // at least the size of a shared_ptr
    //! Handler for when operations occur
    const _thunk_spec _thunk{nullptr};

    constexpr explicit string_ref(_thunk_spec thunk) noexcept;

  public:
    //! Construct from a C string literal
    constexpr explicit string_ref(const char *str, size_type len = static_cast<size_type>(-1),
                                  void *state0 = nullptr, void *state1 = nullptr,
                                  void *state2 = nullptr, _thunk_spec thunk = nullptr) noexcept;
    //! Copy construct the derived implementation.
    string_ref(const string_ref &o);
    //! Move construct the derived implementation.
    string_ref(string_ref &&o) noexcept;
    //! Copy assignment
    string_ref &operator=(const string_ref &o);
    //! Move assignment
    string_ref &operator=(string_ref &&o) noexcept;
    //! Destruction
    ~string_ref();

    //! Returns whether the reference is empty or not
    bool empty() const noexcept;
    //! Returns the size of the string
    size_type size() const noexcept;
    //! Returns a null terminated C string
    value_type *c_str() 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
    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 _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 _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 _message(const status_code<void> &code) const noexcept = 0;
  //! Throw a code as a C++ exception.
  virtual void _throw_exception(const status_code<void> &code) const = 0;
};
```

3.2 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 = status_code_domain;
  //! The type of the status code.
  using value_type = void;
  //! The type of a reference to a message string.
  using string_ref = typename domain_type::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.
  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.
  void throw_exception() const;
};
```

3.3 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.
*/
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 status_code<void>
{
  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;

protected:
  value_type _value{};

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;

  //! Implicit construction from any type where an ADL discovered
  //! `make_status_code(T, Args ...)` returns a `status_code`.
  template <class T, class... Args>
  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>
  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);

  // Replace the type erased implementations with type aware implementations for better codegen
  //! Return the status code domain.
  constexpr const domain_type &domain() const noexcept;
  //! Return a reference to a string textually representing a code.
```

```cpp
  string_ref message() 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;
};
```

3.4 status_code<erased<INTEGRAL_TYPE>>

```cpp
template <class ErasedType>
requires(std::is_integral<ErasedType>::value)
struct erased
{
    using value_type = ErasedType;
};

/*! Type erased status_code, but copyable/movable/destructible unlike ‘status_code<void>‘.
Available only if ‘erased<>‘ is available, which is when the domain’s type is trivially
copyable, and if the size of the domain’s typed error code is less than or equal to
this erased error code.

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 status_code<void>
{
  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;

protected:
  value_type _value{};

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

```cpp
  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;

  //! 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>
  constexpr status_code(const 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>
  constexpr status_code(T &&v, Args &&... args) noexcept(noexcept(make_status_code(std::declval<T>(),
      std::declval<Args>()...)));
  //! Reset the code to empty.
  constexpr void clear() noexcept;
  //! Return the erased ‘value_type‘ by value.
  constexpr 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 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 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 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 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 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 status_code<DomainType1> &b)
    noexcept;
```

3.5 Exception types

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

/*! The erased type edition of status_error.
*/
template <> class status_error<void> : public std::exception
{
protected:
  //! Constructs an instance. Not publicly available.
  status_error() = default;
```

```
  //! Copy constructor. Not publicly available
  status_error(const status_error &) = default;
  //! Move constructor. Not publicly available
  status_error(status_error &&) = default;
  //! Copy assignment. Not publicly available
  status_error &operator=(const status_error &) = default;
  //! Move assignment. Not publicly available
  status_error &operator=(status_error &&) = default;
  //! Destructor. Not publicly available.
  ~status_error() = default;

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

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

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

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

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

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

3.6 Generic code

```
//! The generic error coding (POSIX)
enum class errc : int
{
  success = 0,
  unknown = -1,
```

```
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,
  protcol_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;
```

3.7 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>;

#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

/*! 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‘ 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>>;
```

3.8 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: Throwing values](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)
- No default construction.
- No empty state possible.
- Is immutable.

As with ‘system_code‘, it remains guaranteed to be two CPU registers in size,
and trivially copyable.
*/
class error : public system_code
{
  using system_code::clear;
  using system_code::empty;
  using system_code::success;
  using system_code::failure;

```

```
public:
  //! The type of the erased error code.
  using system_code::value_type;
  //! The type of a reference to a message string.
  using system_code::string_ref;

  //! Default construction not permitted.
  error() = delete;
  //! Copy constructor.
  error(const error &) = default;
  //! Move constructor.
  error(error &&) = default;
  //! Copy assignment.
  error &operator=(const error &) = default;
  //! Move assignment.
  error &operator=(error &&) = default;
  ~error() = default;

  /*! Implicit copy construction from any other status code if its value type is
  trivially copyable and it would fit into our storage.

  The input is checked to ensure it is a failure, if not then
  ‘SYSTEM_ERROR2_FATAL()‘ is called which by default calls ‘std::terminate()‘.
  */
  template <class DomainType>
  error(const status_code<DomainType> &v) noexcept;

  /*! Implicit construction from any type where an ADL discovered
  ‘make_status_code(T &&)‘ returns a ‘status_code‘ whose value type is
  trivially copyable and it would fit into our storage.

  The input is checked to ensure it is a failure, if not then
‘SYSTEM_ERROR2_FATAL()‘ is called which by default calls ‘std::terminate()‘.
  */
  template <class T>
  error(T &&v) noexcept(noexcept(make_status_code(std::declval<T>())));
};

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

4 Design decisions, guidelines and rationale

4.1 Do not #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 just that. If it cannot be constexpr, it is literal or trivial to the maximum extent possible. Empirical testing has found excellent effects on the density of assembler generated, with recent GCCs and clangs, almost all of the time now 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.

[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.

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

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.

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 mostly true on POSIX, but not 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` is now represented as type alias `generic_code`

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 Acknowledgements

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

8 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
SG14 [[move_relocates]]
https://wg21.link/P1029

[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