Document Number: P1459R0
Date: 2019-01-20
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 18 - Diagnostics library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 18 (Diagnostics)

As a drive-by fix, I have removed a bunch of empty descriptions of the form: "Effects: Constructs an object of class `Foo`." Also (Thanks, Tim!) I have added a missing "Returns:" clause to [syserr.errcode.nonmembers] (18.5.3.5)

The entire clause is reproduced here, but the changes are confined to a few sections:

— logic.error 18.2.2
— domain.error 18.2.3
— invalid.argument 18.2.4
— length.error 18.2.5
— out.of.range 18.2.6
— runtime.error 18.2.7
— range.error 18.2.8
— overflow.error 18.2.9
— underflow.error 18.2.10
— syserr.errcat.virtuals 18.5.2.2
— syserr.errcat.nonvirtuals 18.5.2.3
— syserr.errcode.constructors 18.5.3.2
— syserr.errcode.modifiers 18.5.3.3
— syserr.errcondition.constructors 18.5.4.2
— syserr.errcondition.modifiers 18.5.4.3
— syserr.syserr.members 18.5.7.2

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter18 diagnostics.tex
```

18 Diagnostics library [diagnostics]

18.1 General [diagnostics.general]

1 This Clause describes components that C++ programs may use to detect and report error conditions.

2 The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 38.

Table 38 — Diagnostics library summary

	Subclause	Header(s)
18.2	Exception classes	`<stdexcept>`
18.3	Assertions	`<cassert>`
18.4	Error numbers	`<cerrno>`
18.5	System error support	`<system_error>`

18.2 Exception classes [std.exceptions]

1 The C++ standard library provides classes to be used to report certain errors (**??**) in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: *logic* errors and *runtime* errors.

2 The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

3 By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept>` defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

18.2.1 Header `<stdexcept>` synopsis [stdexcept.syn]

```
namespace std {
  class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
  class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

18.2.2 Class `logic_error` [logic.error]

```
namespace std {
  class logic_error : public exception {
  public:
    explicit logic_error(const string& what_arg);
    explicit logic_error(const char* what_arg);
  };
}
```

1 The class `logic_error` defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

```
logic_error(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `logic_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

`logic_error(const char* what_arg);`

4 ~~*Effects:* Constructs an object of class `logic_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.3 Class `domain_error` [domain.error]

```
namespace std {
  class domain_error : public logic_error {
  public:
    explicit domain_error(const string& what_arg);
    explicit domain_error(const char* what_arg);
  };
}
```

1 The class `domain_error` defines the type of objects thrown as exceptions by the implementation to report domain errors.

`domain_error(const string& what_arg);`

2 ~~*Effects:* Constructs an object of class `domain_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

`domain_error(const char* what_arg);`

4 ~~*Effects:* Constructs an object of class `domain_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.4 Class `invalid_argument` [invalid.argument]

```
namespace std {
  class invalid_argument : public logic_error {
  public:
    explicit invalid_argument(const string& what_arg);
    explicit invalid_argument(const char* what_arg);
  };
}
```

1 The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

`invalid_argument(const string& what_arg);`

2 ~~*Effects:* Constructs an object of class `invalid_argument`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

`invalid_argument(const char* what_arg);`

4 ~~*Effects:* Constructs an object of class `invalid_argument`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.5 Class `length_error` [length.error]

```
namespace std {
  class length_error : public logic_error {
  public:
    explicit length_error(const string& what_arg);
    explicit length_error(const char* what_arg);
  };
}
```

1 The class `length_error` defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

`length_error(const string& what_arg);`

2 ~~*Effects:* Constructs an object of class `length_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
length_error(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `length_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.6 Class `out_of_range` [out.of.range]

```
namespace std {
  class out_of_range : public logic_error {
  public:
    explicit out_of_range(const string& what_arg);
    explicit out_of_range(const char* what_arg);
  };
}
```

1 The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```
out_of_range(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `out_of_range`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
out_of_range(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `out_of_range`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.7 Class `runtime_error` [runtime.error]

```
namespace std {
  class runtime_error : public exception {
  public:
    explicit runtime_error(const string& what_arg);
    explicit runtime_error(const char* what_arg);
  };
}
```

1 The class `runtime_error` defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

```
runtime_error(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `runtime_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
runtime_error(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `runtime_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.8 Class `range_error` [range.error]

```
namespace std {
  class range_error : public runtime_error {
  public:
    explicit range_error(const string& what_arg);
    explicit range_error(const char* what_arg);
  };
}
```

1 The class `range_error` defines the type of objects thrown as exceptions to report range errors in internal computations.

```
range_error(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `range_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
range_error(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `range_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.9 Class `overflow_error` [overflow.error]

```
namespace std {
  class overflow_error : public runtime_error {
  public:
    explicit overflow_error(const string& what_arg);
    explicit overflow_error(const char* what_arg);
  };
}
```

1 The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

```
overflow_error(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `overflow_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
overflow_error(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `overflow_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.2.10 Class `underflow_error` [underflow.error]

```
namespace std {
  class underflow_error : public runtime_error {
  public:
    explicit underflow_error(const string& what_arg);
    explicit underflow_error(const char* what_arg);
  };
}
```

1 The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```
underflow_error(const string& what_arg);
```

2 ~~*Effects:* Constructs an object of class `underflow_error`.~~

3 *Ensures:* `strcmp(what(), what_arg.c_str()) == 0`.

```
underflow_error(const char* what_arg);
```

4 ~~*Effects:* Constructs an object of class `underflow_error`.~~

5 *Ensures:* `strcmp(what(), what_arg) == 0`.

18.3 Assertions [assertions]

1 The header `<cassert>` provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

18.3.1 Header `<cassert>` synopsis [cassert.syn]

```
#define assert(E) see below
```

1 The contents are the same as the C standard library header `<assert.h>`, except that a macro named `static_assert` is not defined.

See also: ISO C 7.2

18.3.2 The `assert` macro [assertions.assert]

1 An expression `assert(E)` is a constant subexpression (??), if

(1.1) — `NDEBUG` is defined at the point where `assert` is last defined or redefined, or

(1.2) — `E` contextually converted to `bool` (**??**) is a constant subexpression that evaluates to the value `true`.

18.4 Error numbers [errno]

1 The contents of the header `<cerrno>` are the same as the POSIX header `<errno.h>`, except that `errno` shall be defined as a macro. [*Note*: The intent is to remain in close alignment with the POSIX standard. — *end note*] A separate `errno` value shall be provided for each thread.

18.4.1 Header `<cerrno>` synopsis [cerrno.syn]

```
#define errno see below

#define E2BIG see below
#define EACCES see below
#define EADDRINUSE see below
#define EADDRNOTAVAIL see below
#define EAFNOSUPPORT see below
#define EAGAIN see below
#define EALREADY see below
#define EBADF see below
#define EBADMSG see below
#define EBUSY see below
#define ECANCELED see below
#define ECHILD see below
#define ECONNABORTED see below
#define ECONNREFUSED see below
#define ECONNRESET see below
#define EDEADLK see below
#define EDESTADDRREQ see below
#define EDOM see below
#define EEXIST see below
#define EFAULT see below
#define EFBIG see below
#define EHOSTUNREACH see below
#define EIDRM see below
#define EILSEQ see below
#define EINPROGRESS see below
#define EINTR see below
#define EINVAL see below
#define EIO see below
#define EISCONN see below
#define EISDIR see below
#define ELOOP see below
#define EMFILE see below
#define EMLINK see below
#define EMSGSIZE see below
#define ENAMETOOLONG see below
#define ENETDOWN see below
#define ENETRESET see below
#define ENETUNREACH see below
#define ENFILE see below
#define ENOBUFS see below
#define ENODATA see below
#define ENODEV see below
#define ENOENT see below
#define ENOEXEC see below
#define ENOLCK see below
#define ENOLINK see below
#define ENOMEM see below
#define ENOMSG see below
#define ENOPROTOOPT see below
#define ENOSPC see below
#define ENOSR see below
#define ENOSTR see below
#define ENOSYS see below
```

```
#define ENOTCONN see below
#define ENOTDIR see below
#define ENOTEMPTY see below
#define ENOTRECOVERABLE see below
#define ENOTSOCK see below
#define ENOTSUP see below
#define ENOTTY see below
#define ENXIO see below
#define EOPNOTSUPP see below
#define EOVERFLOW see below
#define EOWNERDEAD see below
#define EPERM see below
#define EPIPE see below
#define EPROTO see below
#define EPROTONOSUPPORT see below
#define EPROTOTYPE see below
#define ERANGE see below
#define EROFS see below
#define ESPIPE see below
#define ESRCH see below
#define ETIME see below
#define ETIMEDOUT see below
#define ETXTBSY see below
#define EWOULDBLOCK see below
#define EXDEV see below
```

1 The meaning of the macros in this header is defined by the POSIX standard.

See also: ISO C 7.5

18.5 System error support [syserr]

1 This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

2 Components described in this subclause shall not change the value of `errno` (18.4). Implementations should leave the error states provided by other libraries unchanged.

18.5.1 Header <system_error> synopsis [system_error.syn]

```
namespace std {
  class error_category;
  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;

  class error_code;
  class error_condition;
  class system_error;

  template<class T>
    struct is_error_code_enum : public false_type {};

  template<class T>
    struct is_error_condition_enum : public false_type {};

  enum class errc {
    address_family_not_supported,       // EAFNOSUPPORT
    address_in_use,                     // EADDRINUSE
    address_not_available,              // EADDRNOTAVAIL
    already_connected,                  // EISCONN
    argument_list_too_long,             // E2BIG
    argument_out_of_domain,             // EDOM
    bad_address,                        // EFAULT
    bad_file_descriptor,                // EBADF
    bad_message,                        // EBADMSG
    broken_pipe,                        // EPIPE
```

```
    connection_aborted,                 // ECONNABORTED
    connection_already_in_progress,     // EALREADY
    connection_refused,                 // ECONNREFUSED
    connection_reset,                   // ECONNRESET
    cross_device_link,                  // EXDEV
    destination_address_required,       // EDESTADDRREQ
    device_or_resource_busy,            // EBUSY
    directory_not_empty,                // ENOTEMPTY
    executable_format_error,            // ENOEXEC
    file_exists,                        // EEXIST
    file_too_large,                     // EFBIG
    filename_too_long,                  // ENAMETOOLONG
    function_not_supported,             // ENOSYS
    host_unreachable,                   // EHOSTUNREACH
    identifier_removed,                 // EIDRM
    illegal_byte_sequence,              // EILSEQ
    inappropriate_io_control_operation, // ENOTTY
    interrupted,                        // EINTR
    invalid_argument,                   // EINVAL
    invalid_seek,                       // ESPIPE
    io_error,                           // EIO
    is_a_directory,                     // EISDIR
    message_size,                       // EMSGSIZE
    network_down,                       // ENETDOWN
    network_reset,                      // ENETRESET
    network_unreachable,                // ENETUNREACH
    no_buffer_space,                    // ENOBUFS
    no_child_process,                   // ECHILD
    no_link,                            // ENOLINK
    no_lock_available,                  // ENOLCK
    no_message_available,               // ENODATA
    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_canceled,                 // ECANCELED
    operation_in_progress,              // EINPROGRESS
    operation_not_permitted,            // EPERM
    operation_not_supported,            // EOPNOTSUPP
    operation_would_block,              // EWOULDBLOCK
    owner_dead,                         // EOWNERDEAD
    permission_denied,                  // EACCES
    protocol_error,                     // EPROTO
    protocol_not_supported,             // EPROTONOSUPPORT
    read_only_file_system,              // EROFS
    resource_deadlock_would_occur,      // EDEADLK
    resource_unavailable_try_again,     // EAGAIN
    result_out_of_range,                // ERANGE
    state_not_recoverable,              // ENOTRECOVERABLE
    stream_timeout,                     // ETIME
    text_file_busy,                     // ETXTBSY
    timed_out,                          // ETIMEDOUT
    too_many_files_open_in_system,      // ENFILE
    too_many_files_open,                // EMFILE
```

```
    too_many_links,                     // EMLINK
    too_many_symbolic_link_levels,      // ELOOP
    value_too_large,                    // EOVERFLOW
    wrong_protocol_type,                // EPROTOTYPE
  };

  template<> struct is_error_condition_enum<errc> : true_type {};

  // 18.5.3.5, non-member functions
  error_code make_error_code(errc e) noexcept;

  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const error_code& ec);

  // 18.5.4.5, non-member functions
  error_condition make_error_condition(errc e) noexcept;

  // 18.5.5, comparison functions
  bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;
  bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
  bool operator< (const error_code& lhs, const error_code& rhs) noexcept;
  bool operator< (const error_condition& lhs, const error_condition& rhs) noexcept;

  // 18.5.6, hash support
  template<class T> struct hash;
  template<> struct hash<error_code>;
  template<> struct hash<error_condition>;

  // 18.5, system error support
  template<class T>
    inline constexpr bool is_error_code_enum_v = is_error_code_enum<T>::value;
  template<class T>
    inline constexpr bool is_error_condition_enum_v = is_error_condition_enum<T>::value;
}
```

1 The value of each `enum errc` constant shall be the same as the value of the `<cerrno>` macro shown in the above synopsis. Whether or not the `<system_error>` implementation exposes the `<cerrno>` macros is unspecified.

2 The `is_error_code_enum` and `is_error_condition_enum` may be specialized for program-defined types to indicate that such types are eligible for `class error_code` and `class error_condition` automatic conversions, respectively.

18.5.2 Class `error_category` [syserr.errcat]

18.5.2.1 Overview [syserr.errcat.overview]

1 The class `error_category` serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from `error_category` to support categories of errors in addition to those defined in this document. Such classes shall behave as specified in this subclause 18.5.2. [*Note*: `error_category` objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom `error_category` types should create a single object of each such type. — *end note*]

```
namespace std {
  class error_category {
  public:
    constexpr error_category() noexcept;
    virtual ~error_category();
```

```
    error_category(const error_category&) = delete;
    error_category& operator=(const error_category&) = delete;
    virtual const char* name() const noexcept = 0;
    virtual error_condition default_error_condition(int ev) const noexcept;
    virtual bool equivalent(int code, const error_condition& condition) const noexcept;
    virtual bool equivalent(const error_code& code, int condition) const noexcept;
    virtual string message(int ev) const = 0;

    bool operator==(const error_category& rhs) const noexcept;
    bool operator!=(const error_category& rhs) const noexcept;
    bool operator< (const error_category& rhs) const noexcept;
  };

  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;
}
```

18.5.2.2 Virtual members [syserr.errcat.virtuals]

`virtual ~error_category();`

1 *Effects:* Destroys an object of class `error_category`.

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

2 *Returns:* A string naming the error category.

`virtual error_condition default_error_condition(int ev) const noexcept;`

3 *Returns:* `error_condition(ev, *this)`.

`virtual bool equivalent(int code, const error_condition& condition) const noexcept;`

4 *Returns:* `default_error_condition(code) == condition`.

`virtual bool equivalent(const error_code& code, int condition) const noexcept;`

5 *Returns:* `*this == code.category() && code.value() == condition`.

`virtual string message(int ev) const = 0;`

6 *Returns:* A string that describes the error condition denoted by `ev`.

18.5.2.3 Non-virtual members [syserr.errcat.nonvirtuals]

`constexpr error_category() noexcept;`

1 *Effects:* Constructs an object of class `error_category`.

`bool operator==(const error_category& rhs) const noexcept;`

2 *Returns:* `this == &rhs`.

`bool operator!=(const error_category& rhs) const noexcept;`

3 *Returns:* `!(*this == rhs)`.

`bool operator<(const error_category& rhs) const noexcept;`

4 *Returns:* `less<const error_category*>()(this, &rhs)`.

[*Note*: `less` (??) provides a total ordering for pointers. — *end note*]

18.5.2.4 Program-defined classes derived from `error_category` [syserr.errcat.derived]

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

1 *Returns:* A string naming the error category.

`virtual error_condition default_error_condition(int ev) const noexcept;`

2 *Returns:* An object of type `error_condition` that corresponds to `ev`.

```
virtual bool equivalent(int code, const error_condition& condition) const noexcept;
```

3 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

```
virtual bool equivalent(const error_code& code, int condition) const noexcept;
```

4 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

18.5.2.5 Error category objects [syserr.errcat.objects]

```
const error_category& generic_category() noexcept;
```

1 *Returns:* A reference to an object of a type derived from class `error_category`. All calls to this function shall return references to the same object.

2 *Remarks:* The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"generic"`.

```
const error_category& system_category() noexcept;
```

3 *Returns:* A reference to an object of a type derived from class `error_category`. All calls to this function shall return references to the same object.

4 *Remarks:* The object's `equivalent` virtual functions shall behave as specified for class `error_category`. The object's `name` virtual function shall return a pointer to the string `"system"`. The object's `default_error_condition` virtual function shall behave as follows:

If the argument `ev` corresponds to a POSIX `errno` value `posv`, the function shall return `error_condition(posv, generic_category())`. Otherwise, the function shall return `error_condition(ev, system_category())`. What constitutes correspondence for any given operating system is unspecified. [*Note*: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX `errno` value. Thus implementations are given latitude in determining correspondence. — *end note*]

18.5.3 Class `error_code` [syserr.errcode]

18.5.3.1 Overview [syserr.errcode.overview]

1 The class `error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [*Note*: Class `error_code` is an adjunct to error reporting by exception. — *end note*]

```
namespace std {
  class error_code {
  public:
    // 18.5.3.2, constructors
    error_code() noexcept;
    error_code(int val, const error_category& cat) noexcept;
    template<class ErrorCodeEnum>
      error_code(ErrorCodeEnum e) noexcept;

    // 18.5.3.3, modifiers
    void assign(int val, const error_category& cat) noexcept;
    template<class ErrorCodeEnum>
      error_code& operator=(ErrorCodeEnum e) noexcept;
    void clear() noexcept;

    // 18.5.3.4, observers
    int value() const noexcept;
    const error_category& category() const noexcept;
    error_condition default_error_condition() const noexcept;
    string message() const;
    explicit operator bool() const noexcept;
```

```
  private:
    int val_;                   // exposition only
    const error_category* cat_; // exposition only
  };

  // 18.5.3.5, non-member functions
  error_code make_error_code(errc e) noexcept;

  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const error_code& ec);
}
```

18.5.3.2 Constructors [syserr.errcode.constructors]

`error_code() noexcept;`

1 ~~*Effects:* Constructs an object of type `error_code`.~~

2 *Ensures:* `val_ == 0` and `cat_ == &system_category()`.

`error_code(int val, const error_category& cat) noexcept;`

3 ~~*Effects:* Constructs an object of type `error_code`.~~

4 *Ensures:* `val_ == val` and `cat_ == &cat`.

```
template<class ErrorCodeEnum>
  error_code(ErrorCodeEnum e) noexcept;
```

5 ~~*Effects:* Constructs an object of type `error_code`.~~

6 *Constraints:* `is_error_code_enum_v<ErrorCodeEnum>` is `true`.

7 *Ensures:* `*this == make_error_code(e)`.

8 ~~*Remarks:* This constructor shall not participate in overload resolution unless `is_error_code_enum_v<ErrorCodeEnum>` is `true`.~~

18.5.3.3 Modifiers [syserr.errcode.modifiers]

`void assign(int val, const error_category& cat) noexcept;`

1 *Ensures:* `val_ == val` and `cat_ == &cat`.

```
template<class ErrorCodeEnum>
  error_code& operator=(ErrorCodeEnum e) noexcept;
```

2 *Constraints:* `is_error_code_enum_v<ErrorCodeEnum>` is `true`.

3 *Ensures:* `*this == make_error_code(e)`.

4 *Returns:* `*this`.

5 ~~*Remarks:* This operator shall not participate in overload resolution unless `is_error_code_enum_v<ErrorCodeEnum>` is `true`.~~

`void clear() noexcept;`

6 *Ensures:* `value() == 0` and `category() == system_category()`.

18.5.3.4 Observers [syserr.errcode.observers]

`int value() const noexcept;`

1 *Returns:* `val_`.

`const error_category& category() const noexcept;`

2 *Returns:* `*cat_`.

`error_condition default_error_condition() const noexcept;`

3 *Returns:* `category().default_error_condition(value())`.

```
string message() const;
```

4 *Returns:* `category().message(value())`.

```
explicit operator bool() const noexcept;
```

5 *Returns:* `value() != 0`.

18.5.3.5 Non-member functions [syserr.errcode.nonmembers]

```
error_code make_error_code(errc e) noexcept;
```

1 *Returns:* `error_code(static_cast<int>(e), generic_category())`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const error_code& ec);
```

2 *Effects:* As if by: `os << ec.category().name() << ':' << ec.value();`

3 *Returns:* `os`.

18.5.4 Class `error_condition` [syserr.errcondition]

18.5.4.1 Overview [syserr.errcondition.overview]

1 The class `error_condition` describes an object used to hold values identifying error conditions. [*Note*: `error_condition` values are portable abstractions, while `error_code` values (18.5.3) are implementation specific. — *end note*]

```
namespace std {
  class error_condition {
  public:
    // 18.5.4.2, constructors
    error_condition() noexcept;
    error_condition(int val, const error_category& cat) noexcept;
    template<class ErrorConditionEnum>
      error_condition(ErrorConditionEnum e) noexcept;

    // 18.5.4.3, modifiers
    void assign(int val, const error_category& cat) noexcept;
    template<class ErrorConditionEnum>
      error_condition& operator=(ErrorConditionEnum e) noexcept;
    void clear() noexcept;

    // 18.5.4.4, observers
    int value() const noexcept;
    const error_category& category() const noexcept;
    string message() const;
    explicit operator bool() const noexcept;

  private:
    int val_;                       // exposition only
    const error_category* cat_; // exposition only
  };
}
```

18.5.4.2 Constructors [syserr.errcondition.constructors]

```
error_condition() noexcept;
```

1 ~~*Effects:* Constructs an object of type `error_condition`.~~

2 *Ensures:* `val_ == 0` and `cat_ == &generic_category()`.

```
error_condition(int val, const error_category& cat) noexcept;
```

3 ~~*Effects:* Constructs an object of type `error_condition`.~~

4 *Ensures:* `val_ == val` and `cat_ == &cat`.

```
template<class ErrorConditionEnum>
  error_condition(ErrorConditionEnum e) noexcept;
```

~~Effects:~~ ~~Constructs an object of type `error_condition`.~~

Constraints: `is_error_code_enum_v<ErrorCodeEnum>` is `true`.

Ensures: `*this == make_error_condition(e)`.

~~Remarks:~~ ~~This constructor shall not participate in overload resolution unless `is_error_condition_enum_v<ErrorConditionEnum>` is `true`.~~

18.5.4.3 Modifiers [syserr.errcondition.modifiers]

```
void assign(int val, const error_category& cat) noexcept;
```

Ensures: `val_ == val` and `cat_ == &cat`.

```
template<class ErrorConditionEnum>
  error_condition& operator=(ErrorConditionEnum e) noexcept;
```

Constraints: `is_error_code_enum_v<ErrorCodeEnum>` is `true`.

Ensures: `*this == make_error_condition(e)`.

Returns: `*this`.

~~Remarks:~~ ~~This operator shall not participate in overload resolution unless `is_error_condition_enum_v<ErrorConditionEnum>` is `true`.~~

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

Ensures: `value() == 0` and `category() == generic_category()`.

18.5.4.4 Observers [syserr.errcondition.observers]

```
int value() const noexcept;
```

Returns: `val_`.

```
const error_category& category() const noexcept;
```

Returns: `*cat_`.

```
string message() const;
```

Returns: `category().message(value())`.

```
explicit operator bool() const noexcept;
```

Returns: `value() != 0`.

18.5.4.5 Non-member functions [syserr.errcondition.nonmembers]

```
error_condition make_error_condition(errc e) noexcept;
```

Returns: `error_condition(static_cast<int>(e), generic_category())`.

18.5.5 Comparison functions [syserr.compare]

```
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
```

Returns:

```
lhs.category() == rhs.category() && lhs.value() == rhs.value()
```

```
bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;
```

Returns:

```
lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value())
```

```
bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;
```

Returns:

```
rhs.category().equivalent(rhs.value(), lhs) || lhs.category().equivalent(rhs, lhs.value())
```

```
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
```

4 *Returns:*

```
lhs.category() == rhs.category() && lhs.value() == rhs.value()
```

```
bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
```

5 *Returns:* `!(lhs == rhs)`.

```
bool operator<(const error_code& lhs, const error_code& rhs) noexcept;
```

6 *Returns:*

```
lhs.category() < rhs.category() ||
(lhs.category() == rhs.category() && lhs.value() < rhs.value())
```

```
bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
```

7 *Returns:*

```
lhs.category() < rhs.category() ||
(lhs.category() == rhs.category() && lhs.value() < rhs.value())
```

18.5.6 System error hash support [syserr.hash]

```
template<> struct hash<error_code>;
template<> struct hash<error_condition>;
```

1 The specializations are enabled (??).

18.5.7 Class `system_error` [syserr.syserr]

18.5.7.1 Overview [syserr.syserr.overview]

1 The class `system_error` describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

2 [*Note*: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type `bad_alloc` (??) rather than `system_error`. — *end note*]

```
namespace std {
  class system_error : public runtime_error {
  public:
    system_error(error_code ec, const string& what_arg);
    system_error(error_code ec, const char* what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat, const string& what_arg);
    system_error(int ev, const error_category& ecat, const char* what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const noexcept;
    const char* what() const noexcept override;
  };
}
```

18.5.7.2 Members [syserr.syserr.members]

```
system_error(error_code ec, const string& what_arg);
```

1 ~~*Effects:* Constructs an object of class `system_error`.~~

2 *Ensures:* `code() == ec` and `string(what()).find(what_arg) != string::npos`.

```
system_error(error_code ec, const char* what_arg);
```

3 ~~*Effects:* Constructs an object of class `system_error`.~~

4 *Ensures:* `code() == ec` and `string(what()).find(what_arg) != string::npos`.

`system_error(error_code ec);`

~~Effects:~~ ~~Constructs an object of class `system_error`.~~

Ensures: `code() == ec`.

`system_error(int ev, const error_category& ecat, const string& what_arg);`

~~Effects:~~ ~~Constructs an object of class `system_error`.~~

Ensures: `code() == error_code(ev, ecat)` and `string(what()).find(what_arg) != string::npos`.

`system_error(int ev, const error_category& ecat, const char* what_arg);`

~~Effects:~~ ~~Constructs an object of class `system_error`.~~

Ensures: `code() == error_code(ev, ecat)` and `string(what()).find(what_arg) != string::npos`.

`system_error(int ev, const error_category& ecat);`

~~Effects:~~ ~~Constructs an object of class `system_error`.~~

Ensures: `code() == error_code(ev, ecat)`.

`const error_code& code() const noexcept;`

Returns: `ec` or `error_code(ev, ecat)`, from the constructor, as appropriate.

`const char* what() const noexcept override;`

Returns: An NTBS incorporating the arguments supplied in the constructor.

[*Note*: The returned NTBS might be the contents of `what_arg + ":  " + code.message()`. — *end note*]